Efficient Compilation for the Linear Language CLASS

Ricardo Gomes de Oliveira Caeiro Antunes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Luís Caires

November 2025

Acknowledgments

I would like to thank my dissertation supervisor, Prof. Luís Caires, for trusting me with the opportunity to pursue this project, for his invaluable guidance and for his willingness and enthusiasm in spending a great deal of time discussing ideas and providing feedback.

I am deeply grateful to my girlfriend, for her encouragement and support throughout this journey, and for the rubber duck debugging sessions. I am also thankful to my parents, my grandmother, and my sister, for supporting me, believing in me, and motivating me to pursue my goals.

Abstract

In this thesis we provide a comprehensive study of compilation techniques for high-level linear-typed (multiparadigm) languages, focusing on CLASS, a proof-of-concept general purpose programming language based on linear/session types that supports many realistic concurrent programming idioms, while guaranteeing memory safety and absence of deadlocks by typing. While the principles and meta-theory behind linear logic based session languages are well known, associated implementation techniques for such languages, in particular compilation to native code, are still poorly understood. In this work, we leverage the sequential execution strategy of the Linear Session Abstract Machine (SAM) to propose a novel multi-stage compilation scheme from CLASS to IR to C for a specially designed intermediate language IR, implement a prototype compiler based on it, and validate it in terms of coverage, correctness and performance. Moreover, the developed compiler covers all of CLASS linear logic primitives, including affine sessions and shared state. The performance of the compiled programs is compared with the original CLASS interpreter, the SAM interpreter, and with equivalent programs written in Haskell. The results show that the compiled programs are orders of magnitude faster than the interpreted ones, and, in some cases, outperform equivalent Haskell programs compiled with GHC. To the best of our knowledge this is the first work demonstrating the efficient compilation of linear session basic languages to machine code.

Keywords

Session types, Linear Logic, Compilation scheme, Intermediate representation, CLASS, Linear Session Abstract Machine (SAM)

Resumo

Nesta tese, apresentamos um estudo abrangente sobre técnicas de compilação para linguagens de programação de alto nível com sistemas de tipos lineares, com foco em CLASS, uma linguagem de programação baseada em tipos de sessão, com suporte para vários padrões realistas de programação concorrente, garantindo simultaneamente segurança de memória e ausência de deadlocks através do sistema de tipos. Embora os princípios e a meta-teoria subjacentes às linguagens de sessão baseadas em lógica linear sejam bem conhecidos, as técnicas de implementação associadas a essas linguagens, em particular, a compilação para código nativo, permanecem ainda pouco compreendidas. Neste trabalho, aproveitamos a estratégia de execução sequencial da Linear Session Abstract Machine (SAM) para propor um novo esquema de compilação em múltiplas etapas, de CLASS para IR, e depois para C, através de uma linguagem intermédia IR especialmente concebida. Validamos ainda o esquema proposto em termos de cobertura, correção e desempenho, tendo implementado um compilador protótipo baseado neste esquema. O compilador desenvolvido cobre toda a funcionalidade de CLASS, incluindo sessões afins e estado partilhado. O desempenho dos programas compilados é comparado com o do intepretador original de CLASS, com o interpretador da SAM, e com programas equivalentes escritos em Haskell. Os resultados demonstram que os programas compilados são ordens de magnitude mais rápidos do que os interpretados e, em alguns casos, superam programas equivalentes compilados com o GHC. Tanto quanto sabemos, este é o primeiro trabalho a demonstrar a compilação eficiente de linguagens à base de sessões lineares para código de máquina.

Palavras Chave

Tipos de sessão, Lógica linear, Esquema de compilação, Representação intermédia, CLASS, SAM

Contents

List of Figures

List of Tables

Acronyms

IR	Intermediate Representation
CLASS	Classic Linear Logic with Affine Shared State
SAM	Linear Session Abstract Machine
GCC	GNU Compiler Collection
AST	Abstract Syntax Tree
RSS	Resident Set Size
FFI	Foreign Function Interface
CCS	Calculus of Communicating Systems

1

Introduction

Contents

The main aim of this thesis is to provide a comprehensive study and development of compilation techniques for high-level linear-typed (multiparadigm) languages, focusing on CLASS, a session typed language strongly motivated by a propositions-as-types interpretation of classical linear logic introduced by Rocha and Caires [1–3]. While the principles and meta-theory behind such linear logic based session languages are well known [4,5], associated implementation techniques for such languages, in particular compilation to native code, are still poorly understood. In this work we thus propose the design of a multi-stage compilation scheme from CLASS to C, implement a prototype compiler based on it, and validate it in terms of coverage, correctness and performance.

1.1 Linear Types in Programming Languages

Type systems are a powerful tool which can be used to ensure the absence of some runtime errors, which would otherwise require extensive testing or program analysis. Essentially, type systems form a set of rules which limit the programs that can be written in a given language. Most traditional compiled imperative languages, such as C and Java, have type systems that avoid some common programming errors such as passing a string to a function that expects an integer. Although programs in a weakly or dynamically typed language such as Python or JavaScript would compile and run, they would likely fail at runtime, whereas in a strongly typed language, the compiler would catch the error at compile time. Strongly typed languages can additionally make use of their types to provide more information about the behavior of the code to the compiler, opening up new optimization paths.

More recently, substructural types have been proposed to statically ensure even stronger properties of the code, namely resource-usage properties related to memory manipulation, and protocol compliance, namely via linear types and session types [6–8]. In fact, linear and affine types are becoming more and more relevant in practice, as witnessed by the adoption of programming languages such as Rust [9] (for general systems programming), Move [10, 11] (for smart contracts), and Linear Haskell [12], among others [13–17] (for general purpose programming). For example, Rust uses an affine type system to enforce memory safety, absence of data races and dangling pointers, which are a common source of bugs in C and C++ programs, without the need for a garbage collector. However, Rust still does not guarantee that concurrent programs are deadlock-free, as it does not have a type system that enforces communication protocols. Session types have been used to enforce correctness in communication protocols, namely in languages based on process calculus, which model concurrent systems [18–20]. By coupling session types with linear logic [21, 22], it has been possible to additionally guarantee deadlock freedom.

Classic Linear Logic with Affine Shared State (CLASS) [1–3] is a proof-of-concept general purpose linear programming language based on linear/session types. CLASS supports many realistic concurrent programming idioms, and is the first language to simultaneously guarantee memory safety and the absence of dead-locks. However, the CLASS interpreter implemented by Rocha and Caires [23, 24] served as a proof-of-concept for the language, and was not at all optimized for performance, and thus, was not suitable for real-world applications — such fully concurrent implementations spawn threads for all concurrent processes, based on the “default” concurrent interpretation of session-based programs (e.g., [16, 25]).

As a first step, the Linear Session Abstract Machine (SAM) was introduced by Caires and Toninho [26] to provide a formal model for sequentially executing the processes of session-typed languages, allowing for orders of magnitude better runtime performance. However, although the SAM design opens the way for the mechanical sequential execution of CLASS programs, it is still a formal abstract machine model, in the spirit of [27, 28]. Therefore, the prototype implementation of the SAM [29] was of course expected to still introduce a heavy performance overhead, compared to a potential, still to be developed, version of the language implementation based on compilation to native code.

During this work, a compilation scheme was designed and implemented which translates CLASS programs into efficient C code, while maintaining all of the safety and liveness guarantees of CLASS. This approach allows a two orders of magnitude faster execution of CLASS programs than the SAM interpreter, as the compiler can make use of the extensive knowledge of the compiled programs. Moreover, the developed compiler supports all of CLASS, including affine sessions and shared state. To make working with CLASS more comfortable, a Visual Studio Code extension was developed to provide syntax highlighting for CLASS programs[1].

The base CLASS language was slightly extended, with features such as an input extraction construct, which was necessary to measure the real-world performance of the compiled programs. The performance of the compiled programs was compared with the original CLASS interpreter, the SAM interpreter, and with equivalent programs written in Haskell. The results show that the compiled programs are orders of magnitude faster than the interpreted ones, and, in some cases, outperform equivalent Haskell programs compiled with GHC.

1.2 A Glimpse into CLASS Compilation

CLASS programs are linearly typed session-based programs. Programs are constructed by composing processes which communicate through channels. Channels are first-class citizens of the language, replacing variables from traditional imperative languages. Communication through channels is regulated by the channel's session type, which describes the communication protocol followed by the channel. The linearity of the type system ensures that all communication is carried out exactly once, preventing leakage of resources, and guaranteeing memory safety.

An example CLASS program is shown in Figure 1.1a. This program defines a process `main` which will be executed when the program is run. Within it, the **cut** process creates a new channel x through which both sides of the cut communicate. On the right hand side of the cut, x is typed by **colint**, and on the left hand side of the cut, by its dual type, **lint**. Semantically, the left hand side of the cut must produce an integer, which the right hand side must then consume. In this case, the left hand side produces the integer 1, through the **let** process, and the right hand side consumes it by printing it to the standard output, through the **print** process. The **print** process requires a continuation, which in this case is the unit process **()**, which represents end of computation.

On the first prototype of the developed compiler, the compilation was performed in a single step, directly translating CLASS programs into C code. However, as the supported feature set was expanded, and optimizations were implemented, the complexity of the compiler increased significantly. Thus, an Intermediate Representation (IR) for session-based languages was developed to simplify the compilation

[1] `https://marketplace.visualstudio.com/items?itemName=classlang.class`

```
proc main() {
    cut {
        let x 1
        |x : colint|
        print(x); ()
    }
};;
```

(a) An example CLASS program

```
main:
    exit points: 2
    types: none
    continuations: c0
    data d0: <[int]> (c0)
entry:
    initContinuation(c0, cut_0, d0)
    writeExpression(c0, 1)
    finish(c0, exit point)
cut_0:
    print(move(d0, int))
    popTask(exit point)
```

(b) Generated IR program

Figure 1.1: An example CLASS program which prints the integer 1 and the base IR program generated from it

process. This IR is based on the SAM execution model, sequentially executing the processes of the CLASS program with buffered communication. The final compilation scheme is divided into three main phases: the translation from CLASS to IR, optimization of the IR, and generation of C code from the optimized IR.

The unoptimized IR generated from the example CLASS program is shown in Figure 1.1b. The process `main` is translated into an IR process of the same name, consisting of a header, and a set of blocks of instructions. This header defines multiple properties necessary for the execution of the process. In this example, the number of exit points, i.e., the number of exit instructions that must run for the process to finish, is 2, one for each side of the cut. Additionally, the process is defined as having no type parameters, a continuation c_0, which is used to implement session x, and a data section d_0, used to store the integer written to x.

The `entry` block of the IR process is generated from the **cut** process. It starts by initializing c_0, indicating its writing location to be d_0, and setting its return address to block `cut_0`. The writing side is always executed first, so the next instructions are generated from the **let** process. A write of the integer 1 to the continuation c_0 (and thus to d_0) is generated, along with a `finishContinuation` instruction, which passes control to the return address `cut_0` of c_0. The `cut_0` block is generated from the right hand side of the cut. **print** is translated to a `print` instruction, which reads the integer from d_0 and prints it. Finally, **()** is translated to a `popTask` instruction, which, in this case, ends the program.

Before the C generation phase, a set of optimizations are applied to the IR program to improve its performance. The final optimized version of the IR is shown in Figure 1.2a. Using information obtained from a symbolic execution of the IR program, the continuation c_0 can be optimized away entirely, leaving only the data variable d_0. In this case, since `finishSession` must pass control to the `cut_0` block, it could be removed and the blocks concatenated. Furthermore, the writing location of c_0 is known to be d_0. Thus `writeExpression` can write directly to it.

```
main:
    exit points: 1
    types: none
    continuations: none
    data d0: <[int]>
entry:
    writeExpression(d0, 1)
    print(move(d0, int))
    popTask(exit point)
```

(a) Optimized IR program

```
main_entry:
    /* writeExpression(d0, 1) */
    *(int*)(env + 0) = 1;
    /* print(move(d0, int)) */
    printf("%d", *(int*)(env + 0));
    /* popTask(exit point) */
    managed_free(env);
```

(b) Generated C code

Figure 1.2: Final optimized IR program and a segment of the C code generated from it

Finally, the optimized IR program is compiled into C code. A segment of the generated code is shown in Figure 1.2b. Due to the continuation-based nature of the SAM execution model, processes are not represented as functions, but instead as code blocks within a single function, with control being transferred between them through the use of `goto` statements. Stack frames are replaced by heap-allocated environments, linked together to form a spaghetti stack, and which contain all local variables and state used by the process. A set of variables are used as registers to keep the current execution state, such as a pointer `env` to the active environment.

The `main_entry` label represents the `entry` block of the IR `main` process. At the start of the program, an environment `env` is allocated and control is passed to this label. The `writeExpression` instruction is compiled into a simple assignment to the location of d_0 in the environment, which in this case is at offset 0. The `print` instruction is compiled into a call to the C `printf` function, reading the integer from the same location. Finally, the `popTask` instruction is compiled into code which frees the current environment, and, in this case, ends the program.

During this work, several alternative compilation schemes were considered and partially explored. Particularly, a different IR model was considered and fully implemented, but ultimately abandoned in favor of the current scheme due to its inferior performance. This alternative implementation is available in the project's repository[2], through the tag `old-impl`.

On this alternative model, each session was represented as a heap-allocated object, separate from process environments. This meant allocating and deallocating sessions on the heap very frequently, which introduced a significant overhead. The IR was much similar to the source language CLASS, containing, for example, instructions for pushing and popping data from buffered communication channels, instead of direct reads and writes. This shifted a lot of complexity to the C generation phase, and additionally, made it harder to implement multiple optimizations.

[2]https://github.com/RiscadoA/class

1.2.1 Structure of the Document

The second chapter focuses on the theoretical background and on prior work related to this thesis, introducing the CLASS language and the SAM execution model. The third chapter describes the base compilation scheme in detail. The fourth chapter lists the various transformations made to the base compilation scheme in order to improve the performance of the generated programs. The fifth chapter describes the implementation of the compiler and its test suite, and discusses the performance results of the developed benchmarks. Finally, the sixth chapter gives some concluding remarks and discusses possible future work. Two appendices are also included, one with the full benchmark results, and another with the specification of the IR syntax and full instruction set.

2

Background and Related Work

Contents

In this chapter, the theoretical foundations of linear logic, process calculi and session types are introduced, and then the programming language CLASS [1] and the abstract machine SAM [26], on which this work builds upon, are presented. Finally, some work following the same line of research or with the same goal but in different contexts is discussed.

2.1 Linear Logic

Introduced by Girard [21] in the 1980s, linear logic is a logic where propositions are consumed exactly once, as opposed to classical logic where propositions may be used any number of times. This property makes linear logic ideal for modeling resources on real-world systems, enabling various optimizations, such as in-place updates in functional languages [30]. More importantly for this work, the propositions-as-types correspondence of linear logic has proven useful for modeling concurrent systems [22].

Connectives of classic linear logic are divided into three groups: multiplicative, additive, and exponential. Multiplicative connectives are $\otimes$ (tensor), $⅋$ (par), 1 and $\bot$. Additive connectives are $\&$ (with), $\oplus$ (plus), $\top$ and 0. Exponential connectives are $!$ (of course) and $?$ (why not).

Here, a definition of linear logic as a sequent calculus [31] is presented, where a sequent consists of a list of propositions separated by commas, with a turnstile $\vdash$ separating the antecedent from the succedent. In linear logic, each connective has a dual connective, represented by $\overline{A}$ and defined as

$$\begin{array}{llll} \overline{A \otimes B} \equiv \overline{A} ⅋ \overline{B} & \overline{A \& B} \equiv \overline{A} \oplus \overline{B} & \overline{1} \equiv \bot & \overline{!A} \equiv ?\overline{A} \\ \overline{A ⅋ B} \equiv \overline{A} \otimes \overline{B} & \overline{A \oplus B} \equiv \overline{A} \& \overline{B} & \overline{\bot} \equiv 1 & \overline{?A} \equiv !\overline{A} \end{array}$$

As such, there is no need for propositions to be placed before the turnstile, as they can be replaced by their dual on the succedent.

Proofs of sequents can be constructed through inference rules. In linear logic, weakening and contraction rules are not present, as they would allow for the duplication and discarding of resources. The only structural rule that is defined is the exchange rule, as the order of propositions is not important in linear logic. The initial and cut rules are also defined:

$$\frac{}{\vdash A, \overline{A}} \qquad\qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, \overline{A}}{\vdash \Gamma, \Delta}$$

The cut rule allows for the composition of two proofs, and the initial rule allows for the introduction of a proposition and its dual without any premises.

The multiplicative conjunction $\otimes$ represents the tensor product of two propositions, and is equivalent to logical conjunction in classical logic. Multiplicative disjunction $⅋$ is the par product of two propositions and is equivalent to logical disjunction in classical logic. The unit of $\otimes$ is 1 and the unit of $⅋$ is $\bot$. The reduction rules for $\otimes$ and $⅋$ are defined below:

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \qquad\qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A ⅋ B}$$

Additive conjunction $\&$ and additive disjunction $\oplus$ are, respectively, the with product of two propositions, and the plus product of two propositions. Their reduction rules can be seen below.

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \& B} \qquad\qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \qquad\qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}$$

The unit of $\&$ is $\top$ and the unit of $\oplus$ is 0. Once again, $\&$ and $\oplus$ are equivalent to the logical conjunction and disjunction in classical logic, respectively. They differ from $\otimes$ and $⅋$ in that the context in the multiplicative connectives is split in two, while in the additive connectives the context in the premises is the same as the context in the conclusion.

Finally, the exponential connectives allow weakening and contracting of propositions in contexts in a controlled manner, through the rules shown below.

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?A} \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

Since its inception, linear logic has been seen as a good candidate for modeling the foundations of concurrent stateful systems [6]. In fact, through the works of Abramsky [31], it has been found that classical linear logic can be interpreted computationally, and since then, this connection has been developed further by multiple authors [32]. To exemplify, the cut rule can be interpreted as the composition of two processes, the initial rule as the creation of processes, and the additive and multiplicative rules as communication between processes. Classical linear logic can extended with a mix rule (shown below), which enables modeling parallel composition of processes.

$$\frac{\vdash \Gamma \qquad \vdash \Delta}{\vdash \Gamma, \Delta}$$

There are many variations of linear logic. One particularly relevant to this project is dual intuitionistic linear logic (DILL) [33], which separates the linear and intuitionistic parts of the contexts, such that a general sequent is of the form $\Gamma; \Delta \vdash A$, where Γ is the intuitionistic part, Δ is the linear part, and A is the succedent. In the intuitionistic part, propositions can be used any number of times, while in the linear part of the context, propositions are consumed exactly once. This distinction allows for a more faithful modeling of servers in concurrent systems.

2.2 Process Calculus

Process calculi are formal languages for modeling concurrent systems, where processes interact by sending and receiving messages. One of the most well-known process calculi is Calculus of Communicating Systems (CCS) [18], which introduced the notion of parallel composition, restriction and communication.

In the 90s, Milner et al [19, 20] introduced π-calculus, a process calculus that allows for mobility of processes — channels, through which processes communicate, can themselves be sent and received. It unifies data with communication channels, treating them as first-class citizens. This allows for more complex communication patterns, where the topology of the network can change dynamically. π-calculus can be used to encode various computation paradigms, such as λ-calculus and Turing machines, and thus, is a universal model of computation.

Processes can be seen as a generalization of functions, where the input and output can be sent over a channel - a function call can be seen as sending the input over a channel, and then receiving the output over the same channel. Additionally, coroutine constructs and generator expressions common in modern languages such as Python can also be easily modeled as a process calculus, where calls to *yield* statements can be seen as sending values over a channel. Channels in π-calculus can also be replicated, which makes it possible to model services that can be accessed by multiple clients, such as web servers.

Below, a simple π-calculus example is presented. A channel x is created, and two processes are composed in parallel. The first process sends a message over x and then terminates. The second process receives a message y over x, prints it, and then terminates.

$$\begin{aligned}(\nu x)\,(\,&\overline{x}\langle\texttt{"Hello World!"}\rangle.0\\ |\,&x(y).\texttt{print}(y).0\,)\end{aligned}$$

2.3 Session Types

Session types, first introduced by Honda [34], and expanded upon by Honda et al. [7], are a type discipline for communication-based programming that ensures that the communication between two parties is well-typed. Essentially, session types are a type discipline for process calculi, such as π-calculus, where operations on channels must follow the protocol defined by the session type. While in typed λ-calculus types are associated with terms, which denote functions and values, session types are associated with channel names.

Operations on session types include sending and receiving, branching and selection and recursion. Session types have been used to model and verify communication protocols, enforcing their correctness [35]. They have been implemented in various programming languages, either as a language feature, such as in Scribble [36] and Effpi [37], or as a library, such as in Rust [25] and Haskell [38].

More recently, a direct correspondence between session types and linear logic has been established [4, 5, 22, 39, 40], where session types can be interpreted as linear propositions, and vice-versa. The connectives of linear logic are enough to describe the communication patterns of finite sessions.

The multiplicative connectives of linear logic can be used to represent the sending and receiving of channels. For example, a session type $A \otimes B$ describes a channel that first sends a channel name of type A, and then behaves as a channel of type B. The dual of $A \otimes B$ is $\overline{A} \parr \overline{B}$, which describes a channel that first receives a channel name of type $\overline{A}$, and then behaves as a channel of type $\overline{B}$, matching the behavior that the other endpoint of the channel must follow. The additive connectives can be interpreted as offering and choice. A session type $A \,\&\, B$ describes a channel that offers a choice between the channel following type A or type B. The dual of $A \,\&\, B$ is $\overline{A} \oplus \overline{B}$, which describes a channel that chooses between the channel following type $\overline{A}$ or type $\overline{B}$. Finally, the exponential connective $!A$ of linear logic can be interpreted as a session type whose channels may be shared and which serve channels of type A.

2.4 CLASS

CLASS (Classic Linear Logic with Affine Shared State) is an experimental language for session-based programs [1–3]. It expands on the correspondence between session types and linear logic, adding affine types and shared state. In CLASS, well-typed programs are guaranteed to never deadlock on communication or reference cell acquisition, and to not leak any memory. Additionally, programs always terminate, while still allowing for higher-order polymorphic functional code and even recursion.

2.4.1 μCLL

This subsection presents the pure fragment of CLASS, dubbed μCLL. As the name suggests, μCLL is related, via a propositions-as-types correspondence [4], to classical linear logic, extended with mix and inductive/coinductive types.

As usual in session-typed based programming languages, in μCLL processes communicate through sessions $x, y, z, \ldots$. Each session has two endpoints (also referred to as channels), and each endpoint is associated with a session type, which describes the communication protocol that the endpoint must follow. $x : A$ denotes that the channel x obeys type A. These types describe the direction, order and content of the messages exchanged. For example,

$$x : \overline{\mathsf{Nat}} \otimes \mathsf{Bool} \otimes \mathbf{1}$$

describes a session x that must first receive a natural number, then send a boolean, and finally close, after which no more messages can be exchanged. The dual connective $\overline{A}$ is used to represent the dual of a type A, and is used to describe the type of the other endpoint of the session. The dual of the type before, for example, is

$$x : \mathsf{Nat} \parr \overline{\mathsf{Bool}} \parr \bot$$

which describes a session x that must first send a natural number, then receive a boolean, and finally wait for the other endpoint to close.

Session types can be seen as dynamic entities which evolve as the communication progresses. After sending a natural number, the session type of the endpoint x in the previous example would become

$$x : \overline{\mathsf{Bool}} \parr \bot$$

The most fundamental process composition operation in μCLL is the **cut** operator

$$\textbf{cut}\,\{P\,|x : A|\,Q\}$$

which composes two processes P and Q that run concurrently and communicate through a session x, where the endpoint x in Q obeys type A, and the endpoint x in P obeys type $\overline{A}$. This composition corresponds to the cut rule of linear logic, i.e., no session other than x appears simultaneously in both P and Q. It thus restricts interactions between P and Q to the session x, which is essential to achieve desirable metatheoretical properties such as deadlock freedom.

Table 2.1: Actions and their respective types and descriptions in μCLL

Type	Action	Description
$\mathbf{1}$	**close** x	Close x
$\bot$	**wait** $x; P$	Wait for x to close, then run P
$A \,\&\, B$	**case** x **of** $\{$|**#inl** : P_1 |**#inr** : $P_2\}$	Receive a choice c through x, then run P_c
$A \oplus B$	**#c** $x; P$	Send choice c through x, then run P
$A \otimes B$	**send** $x(y.\,P); Q$	Send y through x, then run Q
$A \parr B$	**recv** $x(y); Q$	Receive y through x, then run Q
$!A$	**!** $x(y); P$	Replicate a session y interacted with by P on x
$?A$	**?** $x; P$	Make x unrestricted, then run P
	call $x(y); P$	Invokes x with argument y and runs P
$\exists X.A$	**sendty** $x(B); P$	Send a type B through x, then run P
$\forall X.A$	**recvty** $x(X); P$	Receive a type X through x, then run P
$\mu X.A$	**unfold**$_\mu$ $x; P$	Unfold x, then run P
$\nu X.A$	**unfold**$_\nu$ $x; P$	Unfold x, then run P
	corec $X(z, \vec{w});\ P\ [x, \vec{y}]$	Corecursive definition with body P

One other way of composing processes is through the **par** construct, which allows for parallel composition of processes. The process

$$\textbf{par}\,\{P \,||\, Q\}$$

runs P and Q concurrently, without any communication between them. This composition corresponds to the mix rule of linear logic, i.e., no session appears simultaneously in both P and Q. Additionally, there is the **fwd** construct, which allows forwarding messages between sessions, acting as a bridge between two sessions. For example

$$\textbf{fwd}\, x\, y$$

redirects messages sent to x to y, and vice-versa, where x and y are session endpoints with dual types.

Table 2.1 presents the actions that can be performed on session types in μCLL.

2.4.2 Type System

The types of μCLL are defined by the following BNF grammar:

$$\begin{array}{rll} A, B ::= & X \text{ (type variable)} & \mid \overline{X} \text{ (dual type variable)} \\ & \mid \mathbf{1} \text{ (one)} & \mid \bot \text{ (bottom)} \\ & \mid A \otimes B \text{ (tensor)} & \mid A \parr B \text{ (par)} \\ & \mid A \,\&\, B \text{ (with)} & \mid A \oplus B \text{ (plus)} \\ & \mid\, !A \text{ (of course)} & \mid\, ?A \text{ (why not)} \\ & \mid \exists X.A \text{ (exists)} & \mid \forall X.A \text{ (forall)} \\ & \mid \mu X.A \text{ (mu)} & \mid \nu X.A \text{ (nu)} \end{array}$$

Types are formed from type variables (X, Y, ...), units ($\mathbf{1}$ and $\bot$), multiplicatives ($\otimes$ and $\parr$), additives ($\&$ and $\oplus$), exponentials (! and ?), existential and universal second-order type quantifiers ($\exists$ and $\forall$), and inductive types (μ and ν). Types $\exists X.A$, $\forall X.A$, $\mu X.A$ and $\nu X.A$ all bind type variables X in their bodies A. Additionally, types are equipped with a duality operator $\overline{A}$ corresponding to linear logic negation and which captures the symmetry behavior in interaction. The duality operator is defined as follows:

$$\begin{array}{lll} \overline{\mathbf{1}} \equiv \bot & \overline{A \otimes B} \equiv \overline{A} \parr \overline{B} & \overline{A \oplus B} \equiv \overline{A} \,\&\, \overline{B} \\ \overline{!A} \equiv ?\overline{A} & \overline{\exists X.A} \equiv \forall X.\overline{A} & \overline{\mu X.A} \equiv \nu X.\overline{A} \end{array}$$

Typing contexts are defined as a finite partial assignments from names to types, denoted by

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Delta} \; ; \; \underbrace{y_1 : B_1, \ldots, y_m : B_m}_{\Gamma}$$

where Δ represents the linear part of the context and Γ the unrestricted (exponential) part. The empty context is written as $\emptyset$. The disjoint union of two contexts is written as Δ_1, Δ_2 (separated by a comma).

Below, some of the typing rules for μCLL are presented. The inaction process, **()**, for example, types with an empty linear context, and any unrestricted context. The parallel composition of two processes P and Q, **par** $\{P \,||\, Q\}$, types with the union of the disjoint linear contexts of P and Q, and any unrestricted context. The cut composition of two processes P and Q, **cut** $\{P \,|x : A|\, Q\}$, types with the disjoint union of the linear contexts of P and Q with the session x having type A on the linear context of P and the dual of A on the linear context of Q, and any unrestricted context. The full set of typing rules for μCLL can be found in [3].

$$\frac{}{\textbf{()} \vdash \emptyset; \Gamma}\ \text{(Inaction)} \qquad \frac{P \vdash \Delta_1; \Gamma \qquad Q \vdash \Delta_2; \Gamma}{\textbf{par}\ \{P \,||\, Q\} \vdash \Delta_1, \Delta_2; \Gamma}\ \text{(Par)}$$

$$\frac{P \vdash \Delta_1, x : A; \Gamma \qquad Q \vdash \Delta_2, x : \overline{A}; \Gamma}{\textbf{cut}\ \{P \,|x : A|\, Q\} \vdash \Delta_1, \Delta_2; \Gamma}\ \text{(Cut)}$$

2.4.3 Syntax of CLASS

The syntax of CLASS used in this work differs slightly from μCLL. Additionally, while in μCLL the typing contexts are separated into linear and unrestricted parts, in CLASS there is one extra division: the affine part. Process constructs and types for affine sessions, reference cells and non-determinism are also added. The types of CLASS follow the correspondence to μCLL presented in Table 2.2, and are defined by the BNF grammar in Figure 2.1.

The **lint**, **colint**, **lbool**, **colbool**, **lstring**, **colstring**, **affine**, **coaffine**, **state**, **usage**, **statel** and **usagel** types are not present in μCLL and are used for basic values, defining affine sessions and reference cells.

A CLASS program is a collection of type and process definitions, described by the BNF grammar

$$\begin{array}{rll} \text{Program} ::= & \text{ProcDef Program} & \mid \text{ProcDef} \\ & \mid \text{TypeDef Program} & \mid \text{TypeDef} \end{array}$$

$$\begin{array}{llll}
A, B ::= X \text{ (type variable)} & \mid \textbf{close} & \mid \textbf{lint} & \mid \textbf{!}A \\
\quad\quad \mid {\sim}X \text{ (dual type variable)} & \mid \textbf{wait} & \mid \textbf{colint} & \mid \textbf{?}A \\
\quad\quad \mid \textbf{choice of}\,\{\mid \textbf{\#inl} : A \mid \textbf{\#inr} : B\} & \mid \textbf{lbool} & \mid \textbf{lstring} & \mid \textbf{affine}\ A \\
\quad\quad \mid \textbf{offer of}\,\{\mid \textbf{\#inl} : A \mid \textbf{\#inr} : B\} & \mid \textbf{colbool} & \mid \textbf{colstring} & \mid \textbf{coaffine}\ A \\
\quad\quad \mid \textbf{send}\ A; B & \mid \textbf{sendty}\ X; A & \mid \textbf{state}\ A & \mid \textbf{statel}\ A \\
\quad\quad \mid \textbf{recv}\ A; B & \mid \textbf{recvty}\ X; A & \mid \textbf{usage}\ A & \mid \textbf{usagel}\ A \\
\quad\quad \mid \textbf{rec}\ X;\ A & \mid \textbf{corec}\ X;\ A & &
\end{array}$$

Figure 2.1: CLASS Type Grammar

Table 2.2: Mapping μCLL types to CLASS types

μCLL Type	CLASS Type	μCLL Type	CLASS Type
$\mathbf{1}$	**close**	$!A$	**!**A
$\bot$	**wait**	$?A$	**?**A
$A \& B$	**offer of** $\{\mid$**#inl** $: A \mid$**#inr** $: B\}$	$\exists X.A$	**sendty** $X; A$
$A \oplus B$	**choice of** $\{\mid$**#inl** $: A \mid$**#inr** $: B\}$	$\forall X.A$	**recvty** $X; A$
$A \otimes B$	**send** $A; B$	$\mu X.A$	**rec** $X;\ A$
$A \parr B$	**recv** $A; B$	$\nu X.A$	**corec** $X;\ A$

A type definition associates a name X to a session type A, and can either be a simple type definition, or a recursive type definition. Optionally, type definitions may also receive type variables as parameters, which are bound in A. Type definitions follow the BNF grammar

$$\begin{array}{lll}
\text{TypeDef} ::= & \textbf{type}\ X\,\{A\};; & \mid \textbf{rec type}\ X\,\{A\};; \\
& \mid \textbf{type}\ X(X_1,\ \ldots,\ X_n)\,\{A\};; & \mid \textbf{rec type}\ X(X_1,\ \ldots,\ X_n)\,\{A\};;
\end{array}$$

A process definition associates a name `proc_id` to a process P, which receives a list of channels $x_1, \ldots, x_n$ to be bound in P, with types $A_1, \ldots, A_n$, respectively. As with type definitions, process definitions may also receive type parameters. Process definitions follow the BNF grammar

$$\begin{array}{ll}
\text{ProcDef} ::= & \textbf{proc}\ \texttt{proc\_id}(x_1 : A_1,\ \ldots,\ x_n : A_n)\,\{P\};; \\
& \mid \textbf{proc}\ \texttt{proc\_id}\langle X_1,\ \ldots,\ X_n\rangle(x_1 : A_1,\ \ldots,\ x_n : A_n)\,\{P\};; \\
& \mid \textbf{rec proc}\ \texttt{proc\_id}(x_1 : A_1,\ \ldots,\ x_n : A_n)\,\{P\};; \\
& \mid \textbf{rec proc}\ \texttt{proc\_id}\langle X_1,\ \ldots,\ X_n\rangle(x_1 : A_1,\ \ldots,\ x_n : A_n)\,\{P\};;
\end{array}$$

Processes in CLASS, defined by the BNF grammar in Figure 2.2, have a direct correspondence to the process composition constructs in μCLL and the actions shown in Table 2.1, with the exception of the process definition calling, affine, shared state, scan, print, let and if constructs. Note that in CLASS the sum/choice type is expanded to allow for an arbitrary number of choices, as opposed to μCLL, where the sum type is binary. This simplifies the syntax of the language, and all arbitrary sized sums can be trivially encoded as binary sums. Additionally, CLASS adds scan and print constructs for input and

$$
\begin{array}{llll}
P, Q ::= & \textbf{cut}\,\{P\,|x : A|\,Q\} & & |\ \textbf{par}\,\{P\,||\,Q\} \\
& |\ \textbf{fwd}\,x\,y & & |\ \textbf{()} \\
& |\ \textbf{close}\,x & & |\ \textbf{wait}\,x; P \\
& |\ \textbf{case}\,x\,\textbf{of}\,\{|\#\textbf{c}_1 : P_1\ \ldots\ |\#\textbf{c}_n : P_n\} & & |\ \#\textbf{c}\ x; P \\
& |\ \textbf{send}\,x(y.\,P); Q & & |\ \textbf{recv}\,x(y); Q \\
& |\ \textbf{!}\,x(y); P & & |\ \textbf{?}\,x; P \\
& |\ \textbf{sendty}\,x(B); P & & |\ \textbf{recvty}\,x(X); P \\
& |\ \textbf{unfold}_\mu\,x; P & & |\ \textbf{unfold}_\nu\,x; P \\
& |\ \texttt{proc\_id}(x_1, \ldots, x_n) & & |\ \texttt{proc\_id}\langle X_1, \ldots, X_n\rangle(x_1, \ldots, x_m) \\
& |\ \textbf{affine}\,x; P & & |\ \textbf{discard}\,x \\
& |\ \textbf{use}\,x; P & & |\ \textbf{cell}\,x(y.\,P) \\
& |\ \textbf{release}\,x & & |\ \textbf{take}\,x(y); P \\
& |\ \textbf{put}\,x(y.\,P); Q & & |\ \textbf{scan}(x) \\
& |\ \textbf{print}(E); P & & |\ \textbf{println}(E); P \\
& |\ \textbf{let}\,x\,E & & |\ \textbf{if}\,E\,\{P\}\,\textbf{else}\,\{Q\} \\
& |\ \textbf{call}\,x(y); & &
\end{array}
$$

Figure 2.2: CLASS process grammar, where x and y are channels and E is an expression.

output, support for basic types such as integers, booleans and strings through the let and if constructs, and the process definition call construct, similar to a function call in an imperative language.

With the introduction of basic types, along with the print, if and let processes, there is a need to define expressions in CLASS. Expressions E are defined as follows, where i is an integer literal, s is a string literal, and x is a channel name:

$$
\begin{array}{lllll}
E, E_1, E_2 ::= & E_1 + E_2 & |\ E_1 - E_2 & |\ E_1 * E_2 & |\ E_1 / E_2 \\
& |\ E_1 \% E_2 & |\ -E & |\ (E) & |\ x \\
& |\ \texttt{true} & |\ \texttt{false} & |\ s & |\ i \\
& |\ E_1\ \texttt{and}\ E_2 & |\ E_1\ \texttt{or}\ E_2 & |\ \texttt{not}\ E & |\ E_1 == E_2 \\
& |\ E_1 < E_2 & |\ E_1 > E_2 & |\ E_1 <= E_2 & |\ E_1 >= E_2
\end{array}
$$

The affine constructs, **affine** $x; P$, **use** $x; P$ and **discard** x, are used to define affine sessions. The process **affine** $x; P$ is typed such that only co-affine or exponential names can be used in P, and x is an affine session. The process **use** $x; P$ marks the co-affine session x as used in P, and the process **discard** x discards the affine session x.

Finally, the constructs **cell**, **put**, **take**, **release** and **share** build on top of the affine constructs to model shared mutable state through reference cells. The process **cell** $x(y.\,P)$ creates a new cell on x containing the affine session y, used in P. The process **take** $x(y); P$ locks the cell x, binding its content to y, and runs P. The process **put** $x(y.\,P); Q$ places a new affine session y, used in P, in the locked cell x, unlocks it, and runs Q. The process **release** x drops the cell x, and, finally, the process **share** $x\,\{P\,||\,Q\}$ runs P and Q concurrently, both sharing access to the cell x.

```
proc main() {
    cut {
        wait x; ()
         |x : close|
        close x
    }
};;
```

(a) Opening a channel through cut

```
proc main() {
    cut {
        send x(y. println(y); ());
        println(x); ()
         |x : recv lint; lbool|
        recv x(z); par {let x 1 || let z true}
    }
};;
```

(b) Sending and receiving channels

```
proc main() {
    cut {
        case x of {|#inl : close x|#inr : wait x; ()}
         |x : choice of {|#inl : wait |#inr : close}|
        #inl x; wait x; ()
    }
};;
```

(c) Branching through choice types

```
proc main() {
    cut {
        wait x; ()
         |x : close|
        cut {fwd x y |y : close| close y}
    }
};;
```

(d) Forwarding channels

Figure 2.3: CLASS examples illustrating the basic constructs of the language

2.4.4 Example Programs

Some example programs in CLASS are now presented, starting with a simple program which does nothing but close a session (Figure 2.3a). The `main` process definition never has arguments of any kind. In this program, a session x is initialized through a **cut**. Its type annotation x : **close** indicates that on the right-hand side of the **cut** the only action that can (and must) be performed is **close**. On the left-hand side, the dual is true - that is, the actions on x must match the dual type of **close**, which is **wait**. The **wait** action has a continuation, which in this case is set to **()** (empty), indicating that the process has terminated.

In Figure 2.3b a program is presented which sends a session y of type **colint** through a session x, which then continues with type **colbool**. The sent session y has one of its endpoints defined by the closure **println**(y); **()**, which prints the integer received through y and terminates. On the right-hand side of the cut, session y is received and bound to z. The type system forces an integer and a boolean, to be produced on x and z, respectively. Since **let** has no continuation, the **par** construct is used to compose the two **let** processes.

The program shown in Figure 2.3c exemplifies the use of choice types in CLASS. On the left-hand side of the cut, a choice between two actions is offered. Thus, the right-hand side must choose one of

```
rec type Nat {
    choice of {
        |#zero : wait
        |#succ : Nat
    }
};;

rec proc printNat(x : ~Nat) {
    unfold_ν x;
    case x of {
        |#zero : println("0"); close x
        |#succ : print("S"); printNat(x)
    }
};;
```

```
proc main() {
    cut {
        printNat(x)
        |x : Nat|
        unfold_μ x; #succ x;
        unfold_μ x; #succ x;
        unfold_μ x; #zero x;
        wait x; ()
    }
};;
```

Figure 2.4: Inductive natural number definition in CLASS

them, choosing, in this case, the **#inl** label. The right-hand-side must then follow the contract specified by the type **wait**. If the right-hand side had chosen the **#inr** label, the type would be **close**.

The forward construct **fwd** is used to link two sessions together, making them behave as if they were the same session. in Figure 2.3d a program similar to the one in Figure 2.3a is presented, where instead of directly closing the session x on the right-hand side, a new session y is initialized with a new cut, and all data sent to x are forwarded to y.

When writing complex programs in CLASS, type and process definitions are used to simplify the code. Additionally, CLASS supports defining inductive types and processes. Figure 2.4 presents a inductive definition of natural numbers in CLASS, and a recursive process that prints a natural number. The unfold actions **unfold**$_\mu\, x;$ and **unfold**$_\nu\, x;$ are used to unfold the inductive type Nat, i.e., transform Nat into **choice of** $\{$|**#zero** : **wait** |**#succ** : $Nat\}$. Unfold actions are added implicitly, but they are explicit in the example for clarity. This example prints `S S 0`, representing the number 2.

In Figure 2.5, a simple demonstration of polymorphism in CLASS is shown. The type definition `List` is a polymorphic type that receives a type variable X, and is defined as a choice between two labels, one for the empty list and another for a list with a head of type X and a tail of type $List\ X$. The inductive process definition `concatList` operates on lists of a generic type X: it receives two lists x and y of type $List\ X$, and concatenates them into a new list z of type $List\ X$. Unfolds have been omitted for brevity, but they are implicit in the example.

The concatenation is done by matching on the first list x. If it is **#nil**, then the list y is forwarded into z without modification. If it is **#cons**, then the head a is extracted from x, and inserted it into z, by choosing the **#cons** label on z and sending a to z. Finally, the process calls itself recursively with the same type X, the tail of x, the same list y, and the new list z.

```
rec proc concatList⟨X⟩(
    x : ~List(X), y : ~List(X), z : List(X)) {
    case x of {
        |#nil : fwd y z
        |#cons :
            recv x(a);
            #cons z;
            send z(b. fwd a b);
            concatList⟨X⟩(x, y, z)
    }
};;
```

```
rec type List(X) {
    choice of {
        |#nil : wait
        |#cons : send X; List
    }
};;
```

Figure 2.5: Polymorphic list concatenation in CLASS

```
proc main() {
    cut {
        affine x; let x "Hello, Affine!"
        |x : coaffine colstring|
        discard x
    }
};;
```

(a) Discarding an affine session

```
proc main() {
    cut {
        cell x(1)
        |x : usage colint|
        share x {
            take x(a); println(a); put x(2); release x
            ||
            take x(b); println(b); put x(3); release x
        }
    }
};;
```

(b) Sharing a mutable integer on two threads

Figure 2.6: Affine sessions and shared state in CLASS

An example of an affine session is shown in Figure 2.6a. The **affine** process on the left-hand-side of the **cut** produces an affine string on x. Since x is affine, it can either be used once or discarded. In this case, it is discarded through the **discard** process.

The last example, shown in Figure 2.6b, demonstrates the use of shared mutable state. A cell x is created, holding an integer initialized to 1. The cell x is then shared between two threads, on the right-hand-side of the **cut**. Each thread takes the integer from the cell, prints it, puts a new integer in the cell, and then releases the cell. When the **take** process is executed, the cell is locked, preventing the other thread from accessing it until the **put** process is executed, which unlocks the cell. In this case, the output of the program is non-deterministic, and can be either `1 2` or `1 3`, depending on which thread executes first.

2.5 Linear Session Abstract Machine

In this section, an overview of the Linear Session Abstract Machine (SAM) is given. The SAM, introduced by Caires and Toninho in 2024 [26], is a virtual machine for executing session processes typed by (classical) linear logic. It provides an "efficient deterministic execution model for the implicitly sequential session-typed program idioms". Essentially, its goal is to provide a performant execution model for the sequential parts of session-typed programs, which make up the majority of the code in practice, while still seamlessly integrating with the concurrent parts. A proof-of-concept implementation of the SAM [29] was integrated as an alternative backend for CLASS [23, 24].

The core components of the SAM are session references x, y, values $V ::= \checkmark \mid \textbf{\#c} \mid \text{clos}(x, P)$, queues $q ::= \text{nil} \mid V \mid V@q$, session records $R ::= x\langle q, P\rangle y$ and the heap $H ::= x \mapsto R$, where P is a process. Values V can either be the close token $\checkmark$, a label $\#c$, or a closure $\text{clos}(x, P)$, where x is a session reference and P is a process. Queues q are sequences of values, and session records R are tuples of session references x, y, a queue q and a process P. The heap H maps session references to session records.

2.5.1 Reduction Rules

The Linear SAM is a state transition system, where the state is a tuple (P, H), with the heap H and a process P. The transition relation is defined by a set of rules, some of which are now presented.

The **cut** rule, defined below, inserts a new session record in the heap, associated to the session reference x, for executing the continuation Q of the process P, assuming, without loss of generality, that A is a positive type, i.e., that P will push values into the session queue, and Q will later pop them.

$$(\textbf{cut}\,\{P\,|x : A^+|\,Q\}, H) \Mapsto (P, H[x : A^+\langle \text{nil}, \{y/x\}Q\rangle y : \overline{A^+}])$$

Essentially, the rule transitions to a state where P will be executed immediately, and where Q is delayed to a later time. If A is a negative type, the same rule is applied but with P and Q swapped.

The **close** rule, defined below, pushes the close token $\checkmark$ into the session queue, and then transitions to the continuation P previously stored in the session record.

$$(\textbf{close}\,x, H[x : \textbf{close}\langle q, P\rangle y : B]) \Mapsto (P, H[x : \emptyset\langle q@\checkmark, \textbf{()}\rangle y : B])$$

P will eventually pop the close token from the queue and free the session record. Note that the queue q might not be empty - other positive actions might have preceded the close action and pushed values into the queue. The close token is appended to the end of the queue, and thus will be the last value to be popped.

The **wait** rule, shown below, pops the close token $\checkmark$ from the session queue, frees the session record, and transitions to the continuation of the **wait** action.

$$(\textbf{wait}\,y; P, H[x : \emptyset\langle \checkmark, \textbf{()}\rangle y : \textbf{wait}]) \Mapsto (P, H)$$

The choice rule, shown below, given an action **#c** $x; P$, pushes the label **#c** into the session queue and changes the process to P.

$$(\textbf{#c}\ x; P, H[x : \textbf{choice of}\,\{|\textbf{#c} : A,\ \dots\}\langle q, Q\rangle y : B]) \mapsto (P, H[x : A\langle q@\textbf{#c}, Q\rangle y : B])^{wr}$$

Note that in this rule, a write-to-read adjustment is performed, denoted by $(P, H[x : A\langle q, Q\rangle y : B])^{wr}$. This operation swaps P with Q, if and only if the continuation type A is a negative type. This operation ensures that the next action will be executed by the positive endpoint of the session. In this specific rule, the write-to-read adjustment is performed because the choice action is a positive action, which means values are being pushed for the other endpoint to pop later on. If the continuation process P on this endpoint will next perform a read action on x, i.e., A is negative, then, for that action to be executed, the other endpoint must first perform the corresponding write action.

The offer rule, shown below, performs the dual operation of the choice rule - it pops a label from the session queue, and transitions to the continuation of the **case of** action corresponding to the label.

$$(\textbf{case}\, x\, \textbf{of}\,\{|\textbf{#c} : P_c, \dots\}, H[x : A\langle \textbf{#c}@q, Q\rangle y : \textbf{offer of}\,\{|\textbf{#c} : B_c, \dots\}]) \mapsto (P_c, H[x : A\langle q, Q\rangle y : B_c]^{rw})$$

The rule performs a read-to-write adjustment, denoted by $[x : A\langle q, Q\rangle y : B]^{rw}$. This adjustment swaps the two endpoints of the session record, $x : A$ and $y : B$, if and only if q is empty. Since processes are well-typed, then, if q is empty, P should have no more reads to perform. This adjustment then allows P to start writing to the session.

Finally, the **fwd** rule is shown below, assuming, without loss of generality, that B^+ is a positive type.

$$(\textbf{fwd}\, x\, y, H[z : A\langle q_1, Q\rangle x : \overline{B^+}][y : B^+\langle q_2, P\rangle w : C]) \mapsto (P, H[z : A\langle q_2@q_1, Q\rangle w : C])$$

The rule merges two session records, $(z : A\langle q_1, Q\rangle x : \overline{B^+})$ and $(y : B^+\langle q_2, Q\rangle w : C)$, into a single session record with continuation P and queue $q_2@q_1$. If B^+ is a negative type, x and y can be swapped, producing the equivalent process **fwd** $y\,x$. Since x is the read endpoint of the session and y is the write endpoint, then Q must have written through z the contents of q_1. and the current process will have written through y the contents of q_2. Thus, P is waiting to read the contents of $q_2@q_1$.

2.5.2 Example Execution

In this section two simple examples of process execution are presented. These examples iteratively apply the reduction rules shown in the previous section. The type annotations are omitted for brevity. Consider the process P shown below:

$$P \equiv \textbf{cut}\,\{\textbf{wait}\, x; \textbf{()}\,|x|\,\textbf{close}\, x\}$$

The execution of P is shown below. The initial state of the SAM is $(P, \emptyset)$. First, the **cut** rule is applied, which allocates a new session record in the heap, and transitions to **close**, as it corresponds to the positive type **close**. Then, the **close** rule is applied, which pushes the close token ✓ into the queue, and transitions to the session record continuation **wait**. Finally, the **wait** rule is applied, which pops the close token ✓ from the queue, and transitions to the continuation $\textbf{()}$, reaching the final state $(\textbf{()}, \emptyset)$.

$$\begin{aligned}(P, \emptyset) &\equiv (\textbf{cut}\,\{\textbf{wait}\,x; \textbf{()}\ |x|\ \textbf{close}\,x\}, \emptyset)\\ &\mapsto (\textbf{close}\,x, x\langle nil, \textbf{wait}\,y; \textbf{()}\rangle y)\\ &\mapsto (\textbf{wait}\,y; \textbf{()}, x\langle \checkmark, \textbf{()}\rangle y)\\ &\mapsto (\textbf{()}, \emptyset)\end{aligned}$$

The second example is the process Q whose definition and execution is shown below. First, the **cut** rule is applied, which inserts a new session record in the heap. Then, the choice rule is applied, which pushes the label **#inl** into the queue of the session record, after which, a write-to-read adjustment and a transition to the continuation **case of** is performed. Then, the **case of** reduction rule is applied, which pops the label **#inl** from the queue, and transitions to the continuation matching the label. Next, a read-to-write adjustment is performed, which means the session endpoints are swapped. With the endpoints swapped, the **close** and **wait** rules are applied as in the previous example and the final state $(\textbf{()}, \emptyset)$ is finally reached.

$$\begin{aligned}Q &\equiv \textbf{cut}\,\{\textbf{\#inl}\ x; \textbf{wait}\,x; \textbf{()}\ |x|\ \textbf{case}\,x\ \textbf{of}\,\{|\textbf{\#inl} : \textbf{close}\,x|\textbf{\#inr} : \textbf{wait}\,x; \textbf{()}\}\}\\ (Q, \emptyset) &\mapsto (\textbf{\#inl}\ x; \textbf{wait}\,x; \textbf{()}, x\langle nil, \textbf{case}\,y\ \textbf{of}\,\{|\textbf{\#inl} : \textbf{close}\,y|\textbf{\#inr} : \textbf{wait}\,y; \textbf{()}\}\rangle y)\\ &\mapsto (\textbf{wait}\,x; \textbf{()}, x\langle \textbf{\#inl}, \textbf{case}\,y\ \textbf{of}\,\{|\textbf{\#inl} : \textbf{close}\,y|\textbf{\#inr} : \textbf{wait}\,y; \textbf{()}\}\rangle y)^{wr}\\ &\equiv (\textbf{case}\,y\ \textbf{of}\,\{|\textbf{\#inl} : \textbf{close}\,y|\textbf{\#inr} : \textbf{wait}\,y; \textbf{()}\}, x\langle \textbf{\#inl}, \textbf{wait}\,x; \textbf{()}\rangle y)\\ &\mapsto (\textbf{close}\,y, x\langle nil, \textbf{wait}\,x; \textbf{()}\rangle y)^{rw}\\ &\equiv (\textbf{close}\,y, y\langle nil, \textbf{wait}\,x; \textbf{()}\rangle x)\\ &\mapsto (\textbf{wait}\,x; \textbf{()}, x\langle \checkmark, \textbf{()}\rangle y)\\ &\mapsto (\textbf{()}, \emptyset)\end{aligned}$$

2.6 Alternative Applications

In this section, some works [12, 25, 41–43] are presented which apply linear logic and/or session types through different approaches. These works are not directly related to CLASS or to the Linear SAM, nor is this work based on them, but they are interesting to mention as they show alternative ways of solving the same problems this thesis addresses.

2.6.1 L^3

L^3 is a linear functional language developed by Ahmed et al. [41]. It is based on linear logic, and its main feature is the ability to perform *strong updates* on memory cells - that is, to change the value and type of a cell, without invalidating existing pointers to it. This is achieved by using linear types to track *capabilities*, which are permissions to access a memory location as a certain type. To perform a strong update, a capability of the original type for that location must be consumed, and in return, a capability of the new type is produced. Since capabilities are linear, there is no risk of another part of the program still holding a capability to the old type.

The restrictions placed by the type system also ensure that any program written in L^3 terminates: while cyclic graphs can be created through pointers, the linearity of the capabilities themselves prevents any infinite loops. Although L^3 is not a session-typed language, it is interesting to note that strong updates are in a way a feature present in session typed languages, where the type of a session changes during execution, after each action is performed.

2.6.2 Linear Haskell

Linear Haskell [12] is an extension of Haskell that introduces linear types to the language in a backwards-compatible way, allowing the programmer to use linear types in their program without changing the existing code. It does so by introducing a new kind of function arrow, that is, a new type of function, called a linear function, which must consume its argument exactly once.

Session types have been implemented in Linear Haskell by Kokke et al. [42] in the form of a small library called *Priority Sesh*. This library allows the programmer to easily define session types and integrate them into their programs. The library also gives the possibility of the user annotating session types with priorities, which can then be used to prove that the program is deadlock-free.

One big disadvantage of the Linear Haskell approach is that it is not able to fully exploit the linearity of the types for compiler optimizations, due to the core compiler being non-linear. In contrast, our approach should in theory be able to do so, as CLASS is a session-typed language from the ground up.

2.6.3 Rust

Rust is a systems programming language known for its strong type system. Although it has no linear types, it supports affine types through its ownership system, which forbids variables from being used after they have been moved. References allow the programmer to share data between different parts of the program, but they must follow the strict borrowing rules of the language - there is no limit on the number of immutable references, but there can only be one mutable reference to a given piece of data at a time, and only if there are no immutable references to it.

These rules are enforced at compile time, and the Rust compiler is able to catch many common programming errors, such as use-after-free, data races, and null pointer dereferences, which makes it a very safe language. However, Rust lacks the expressiveness of linear types, which could be used to enforce more complex invariants in the program, such as the order in which functions must be called, the number of times a function must exactly be called, avoiding resources to be dropped or enforcing absence of deadlocks. It also has no built-in support for session types, and therefore, no built-in support for enforcing communication protocols.

Nonetheless, non-linear, i.e., affine session types have been implemented in Rust in the form of libraries by several authors, such as by Jespersen et al. [25] and Lagaillardie et al. [43].

2.6.4 Move

Move [10, 11] is a resource-oriented programming language first introduced for the Libra (now Diem) blockchain project. Its type system draws inspiration from linear logic by introducing the concept of *resources* — values that cannot be duplicated or discarded implicitly. This design prevents digital assets from being accidentally copied or lost. These guarantees are enforced through Move's type system and its bytecode verifier, which ensure at compile time that all resources are either moved, borrowed, or explicitly destroyed in accordance with well-defined global invariants.

In contrast to CLASS, which employs linear and session types to ensure communication safety and correct process interactions, Move's type system is tailored to the domain of asset management and ownership. Nevertheless, the language demonstrates the practical value of linear types for enforcing real-world safety properties at compile time, and it suggests that similar principles could be successfully applied to other domains.

3

Compiling Session-Based Languages

Contents

This chapter presents the base compilation scheme developed in this thesis. This scheme generates C code that executes in a sequential and deterministic manner, closely resembling the execution of the SAM, implementing continuation-based coroutining. Each process call is associated to block of data called an environment, similar to a stack frame of traditional imperative languages. This environment stores the continuations of each session, as well as any data received through them. As in the SAM, when processes finish writing to a session, control passes to the session's other endpoint, which will read the data and continue execution.

At a high level, the compilation scheme is divided into four phases. First, the source CLASS program is parsed and type-checked. This phase is not discussed in this report, as its implementation has been reused from the previous work [2, 26] on the CLASS and SAM interpreters. Then, the program is compiled to an Intermediate Representation (IR) that abstracts some details of the target language.

Next, the IR is analyzed and optimized by a series of passes. Finally, the optimized IR is translated to C code, which is then compiled to native code using the GNU Compiler Collection (GCC).

In the first section of this chapter, the IR's constructs and its abstract machine are defined. In the second section, the compilation of CLASS to the IR is described. Finally, in the third section, an overview is given of how the IR is translated to C code.

3.1 Abstractions for the Intermediate Representation

One of the main contributions of this work is the design and implementation of the first Intermediate Representation (IR) for session-based languages which can be translated to efficient C code. Multiple abstractions were developed to model the execution of session-based programs in a low-level imperative language. This section starts by both describing the structure of IR programs and introducing the abstract machine which executes them. Then, an example execution is covered step by step. Finally, an overview is given on various aspects of the IR which were not covered in the example. The full syntax and instruction set of the IR are provided in Appendix B.

The IR is an imperative stateful language, inspired by the execution model of the SAM, sequentially executing session-based programs through buffered communication. The concept of sessions is split into two parts: *continuations*, which model the flow of control between two endpoints of a session, and *data sections*, which store data received in sessions. A continuation can be seen as a representation of a session endpoint which stores the necessary information to communicate with and pass control to the opposite endpoint.

A continuation consists of a return address, a writing location, and a reference to another continuation. The return address is a code address, which is jumped to when passing control to the opposite endpoint. The writing location is a data location where data to be read by the opposite endpoint should be written to. Finally, the continuation reference points to to the continuation which the opposite endpoint will use to pass control back to this point. The purpose of this reference will become clear in the example execution covered by the next section.

Data sections and continuations make the representation of sessions much closer to variables and function calls in C-like languages. For example, for a given session endpoint x : **lint**, a continuation is stored which holds the return address to jump to after producing an integer to x. In the other endpoint of x, y : **colint**, a data section is stored to hold the integer which will be written to x.

3.1.1 Programs and the IR Abstract Machine

An IR program consists of a set of named process definitions, each with a header and a body. The header declares properties and resources used by the process, such as continuations, data sections

```
proc main() {
    cut {
        p(x)
        |x: recv colint; lstring|
        recv x(y);
        let x "Res: " + y
    }
};;
```

```
proc p(x: send lint; colstring) {
    send x(z. let z 17);
    println(x);
    ()
};;
```

$\Downarrow$

```
main:
    exit points: 2
    continuations: c0 c1
    data d0: <[cont]>
    data d1: <[int]>
entry:
    initContinuation(c0, rhs, d0)
    callProcess(p, exit point, c2 <- c0)
rhs:
    bindContinuation(d0, c1, d1)
    continue(c1, ret)
ret:
    writeExpression(c0, "Res: " +
                        move(d1, int))
    finish(c0, exit point)
```

```
p:
    exit points: 2
    continuations: c2 c3
    data d2: <[string]> (c2)
    data d3: <[]>
entry:
    initContinuation(c3, send, d3)
    writeContinuation(c2, c3)
    continue(c2, ret)
ret:
    printLine(move(d2, string))
    popTask(exit point)
send:
    writeExpression(c3, 17)
    finish(c3, exit point)
```

Figure 3.1: An example IR program (bottom) translated from a simple CLASS program (top) with two processes.

and any type parameters, while the body is a set of named instruction blocks, each consisting of a list of instructions to be executed sequentially. A full specification of the IR's syntax is provided in Listing B.1.

During execution, whenever a process is called, an *environment* is created. An *environment* is a data structure which holds the state of the process, similar to a stack frame in languages such as C. An environment holds the states of the continuations and data sections described in the header of its process. Additionally, the environment also stores the number of remaining exit points of the process. An exit point is an instruction through which control leaves the scope of the current environment, without a guarantee of coming back. When all exit points of a called process have been reached, its environment can be safely destroyed. Essentially, exit points are used as a reference counting mechanism for environments.

The execution of an IR program is described by an abstract machine with state $\langle E, T, C\rangle$, where E is the environment of the process being executed, T is the stack of pending tasks, and C is the program counter. The stack T of pending tasks is used to model the **par** process of CLASS. Each task stores a code address and an environment, on which execution should resume when the task is popped from the stack.

To introduce the IR and its abstractions, the execution of an example program will be covered step by step on the next section.

3.1.2 Example Execution

The example CLASS program shown in Figure 3.1 consists of two process definitions, `main` and `p`. When executed, the two processes communicate through session x. Process `p` sends the integer 17 over x, which, on `main`, is received and used to produce a string. This string is sent back to `p`, which prints it to the standard output and terminates.

The IR program translated from the source CLASS program is also shown in Figure 3.1. It consists of two processes with the same names as in the source program. Before delving into the inner workings of the execution, a brief overview of the execution trace is given. The execution starts at the `entry` block of the `main` process. The steps of the execution are as follows:

Step 1: `initContinuation(c0, rhs, d0)` — initializes continuation c_0 in `main`.

Step 2: `callProcess(p, exit point, c2 <- c0)` — calls process `p`, passing c_0 as an argument.

Step 3: `initContinuation(c3, send, d3)` — initializes continuation c_3 in `p`.

Step 4: `writeContinuation(c2, c3)` — writes c_3 to the writing location of c_2, which is d_0.

Step 5: `continue(c2, ret)` — yields control to c_2, jumping to `rhs` in `main`.

Step 6: `bindContinuation(d0, c1, d1)` — reads c_3 from d_0, binding it to c_1 and pointing c_3 to d_1.

Step 7: `continue(c1, ret)` — yields control to c_1, jumping to `send` in `p`.

Step 8: `writeExpression(c3, 17)` — writes 17 to the writing location of c_3, which is d_1.

Step 9: `finish(c3, exit point)` — jumps to `ret` .in `main`.

Step 10: `writeExpression(c0, "Res: " + move(d1, int))` — reads 17 from d_1 and writes the string `"Res: 17"` to the writing location of c_0, which is d_2.

Step 11: `finish(c0, exit point)` — jumps to `ret` in `p`.

Step 12: `printLine(move(d2, string))` — reads `"Res: 17"` from d_2 and prints it.

Step 13: `popTask(exit point)` — finishes execution.

Now, each step of the execution will be covered in detail, along with the changes to the abstract machine's state. Each environment is shown as a table, with the process name at the top, followed by the number of exit points, and the states of its continuations and data sections. Changes between steps are highlighted in orange.

When the program starts, the `main` process is called, and a new environment is created for it from the information declared in the header of `main`. This initial environment is shown below.

(Initial)

main
exit points: 2
continuation c0: [uninit]
data d0: [uninit]
continuation c1: [uninit]
data d1: [uninit]

(Step 1)

main
exit points: 2
continuation c0: main:rhs, d0, c0
data d0: [uninit]
continuation c1: [uninit]
data d1: [uninit]

In **Step 1**, `initContinuation(c0, rhs, d0)` is executed. This instruction initializes c_0, setting its return address to the block `rhs`, its writing location to d_0, and its remote continuation reference to itself. This reference links c_0 to itself, as the other endpoint of the session, which is the right-hand-side of the **cut** in the source program, is within the same environment and uses the same continuation c_0.

In **Step 2**, `callProcess(p, exit point, c2 <- c0)` is executed, moving the program counter to the `entry` block of `p`, and creating a new environment for `p`. Since c_0 was passed as an argument, it is copied to c_2 in `p`. The continuation referenced by c_0, which is c_0 itself, is linked to c_2. Both environments now hold opposite endpoints of the same session x, represented by c_0 and c_2, which are linked to each other. The writing location of c_0 is changed to d_2, allowing data written by `main` to be read by `p`. Finally, since the instruction is marked as an exit point, the remaining exit-point count of `main` is decremented. The new state of the environments is shown below.

(Step 2)

main
exit points: 1
continuation c0: main:rhs, d2, c2
data d0: [uninit]
continuation c1: [uninit]
data d1: [uninit]

p
exit points: 2
continuation c2: main:rhs, d0, c0
data d2: [uninit]
continuation c3: [uninit]
data d3: [uninit]

In **Step 3**, `initContinuation(c3, send, d3)` is executed, initializing c_3 similarly to how c_0 was initialized in step 1. In **Step 4**, `writeContinuation(c2, c3)` is executed, writing a reference to the just initialized c_3 to the writing location of c_2, which is d_0. The new state is shown below.

(Step 3, 4)

main
exit points: 1
continuation c0: main:rhs, d2, c2
data d0: c1
continuation c1: [uninit]
data d1: [uninit]

p
exit points: 2
continuation c2: main:rhs, d0, c0
data d2: [uninit]
continuation c3: p:send, d3, c3
data d3: [uninit]

In **Step 5**, `continue(c2, ret)` is executed. The program counter jumps to the code address stored in c_2, which is the `rhs` block in `main`. Additionally, the continuation referenced by c_2, which is c_0, has its return address updated to point to the `ret` block in `p`, allowing control to return to `p` after `main` finishes executing. In **Step 6**, `bindContinuation(d0, c1, d1)` is executed. This instruction reads the continuation reference stored in d_0, which is c_3, and binds it to c_1. The writing location of c_3 is updated to point to d_1, allowing data written by `p` to be read by `main`.

(Step 5, 6)

main
exit points: 1
continuation c0: p:ret, d2, c2
data d0: [uninit]
continuation c1: p:send, d3, c3
data d1: [uninit]

p
exit points: 2
continuation c2: main:rhs, d0, c0
data d2: [uninit]
continuation c3: p:send, d1, c1
data d3: [uninit]

In **Step 7**, `continue(c1, ret)` is executed, passing control back to the code address stored in c_1, which is the `send` block in `p`. The continuation referenced by c_1, which is c_3, has its return address set to the `ret` block in `main`, allowing control to return to `main` later on. **Step 8** consists in the execution of `writeExpression(c3, 17)`, which writes the integer 17 to the writing location of c_3, which is d_1.

(Step 7, 8)

main
exit points: 1
continuation c0: p:ret, d2, c2
data d0: [uninit]
continuation c1: p:send, d3, c3
data d1: 17

p
exit points: 2
continuation c2: main:rhs, d0, c0
data d3: [uninit]
continuation c3: main:ret, d1, c1
data d3: [uninit]

In **Step 9**, `finish(c3, exit point)` is executed, causing a jump to the code address stored in c_3, which is the `ret` block of `main`. Both c_3 and the continuation it references, c_1, are considered uninitialized and will not be used again. Since the instruction is marked as an exit point, the remaining exit-point count of `p` is decremented. In **Step 10**, `writeExpression(c0, "Res: " + move(d1, int))` is executed. This instruction reads the integer 17 from d_1, constructs the string `"Res: 17"`, and writes it to the writing location of c_0, which is d_2.

(Step 9, 10)

main
exit points: 1
continuation c0: p:ret, d2, c2
data d0: [uninit]
continuation c1: [uninit]
data d1: [uninit]

p
exit points: 1
continuation c2: main:rhs, d0, c0
data d2: "Res: 17"
continuation c3: [uninit]
data d3: [uninit]

In **Step 11**, `finish(c0, exit point)` is executed, passing control to the `ret` block of `p`. The exit-point count of `main` is decremented, as before, reaching 0. This signals that the environment of `main` will not be used again, and thus, can be destroyed. In **Step 12**, `printLine(move(d2, string))` is executed, printing the string `"Res: 17"` to the standard output.

(Step 11, 12)

p
`exit points: 1`
`continuation c2: [uninit]`
`data d2: [uninit]`
`continuation c3: [uninit]`
`data d3: [uninit]`

Finally, in **Step 13**, `popTask(exit point)` is executed, which pops the next task from the task stack and jumps to it. In this example, the task stack was never modified, and thus contains only the program termination task. Since the instruction is marked as an exit instruction, the exit-point count of `p` reaches 0, and it is destroyed, ensuring no memory is leaked.

In the next subsections, various aspects of the IR which were not covered in the example execution are covered, along with some smaller examples.

3.1.3 Data Representation

IR programs are completely platform agnostic, and thus, data types and addresses are described through their composition of *slots*, instead of concrete byte sizes and offsets. A slot S is a unit of data in the IR, and is defined as

$$S ::= \text{int} \mid \text{bool} \mid \text{string} \mid \text{tag} \mid \text{cont} \mid \text{cell}\,[T] \mid \text{exponential} \mid \text{type} \mid t_i$$

where int, bool and string represent the basic data types, tag represents a CLASS choice, cont a continuation reference, cell $[T]$ a reference to a shared state cell with contents described by the slot tree T (introduced below), exponential an exponential reference, type a stored type parameter, and t_i a type parameter declared in the current process header.

In order to model branching data types (such as choices from CLASS), data layouts are described through trees of slots, instead of plain sequences. A slot tree T is defined recursively as

$$T ::= [] \;\mid\; [S_1; \ldots; S_n; T] \;\mid\; \text{tag}\,[T_1 \mid \ldots \mid T_n]$$

where $[]$ is the empty slot tree, $[S_1; \ldots; S_n; T]$ is a sequence of a slots $S_1; \ldots; S_n$ followed by a slot tree T, and tag $[T_1 \mid \ldots \mid T_n]$ is a tag slot followed by a choice between the slot trees T_1 to T_n. For example, $[\text{int}; \text{tag}\,[[\text{bool}] \mid [\text{string}]]]$ describes an integer followed by a tag, which, if 0, is followed by a boolean, or if 1, is followed by a string.

Data sections are described within process headers by a sum of slot trees describing the possible layouts of data that can be stored on them. For example, `data d0:` $<[\text{int}; \text{bool}] + [\text{string}]>$ describes a data section d_0 which can either store an int followed by a bool, or a string.

3.1.3.A Addressing Data

A *data location* is denoted as $b.o$, where b is a base address and o an offset. A base address can be either a local data section (e.g., d_0), a writing location given by a continuation (e.g., c_0), or a cell location (e.g., $c(d_0)$). An offset is defined by two slot trees, written as $P{\sim}F$, P describing the data being skipped (named *past*), and F describing the data that may come next (named *future*). The future tree is necessary to determine the alignment of the data being accessed. For example, $[\text{int}; \text{bool}] \sim [\text{string}]$ is an offset which skips an integer followed by a boolean, and expects a string to come next. When the past slot tree is empty, the future slot tree is unnecessary, and the offset is simply written as 0.

To exemplify, the location $d_0.\,[\text{int}] \sim [\text{int}]$ refers to the second integer stored in d_0, while the location $c_0.\,[\text{bool}] \sim [\text{string}]$ refers to a string stored in the writing location of the continuation c_0, after a boolean. When the offset is zero, it is omitted for brevity. For example, d_0 as a data location is equivalent to $d_0.0$.

3.1.4 Moving Data

The `moveSlots` instruction allows moving data from one location to another. It takes as arguments a destination location, a source location and a slot tree describing the data to be moved. To exemplify, `moveSlots(c1,` $d_0.\,[\texttt{bool}] \sim [\texttt{string}; \texttt{int}]\,, [\texttt{string}, \texttt{int}])$ takes a string followed by an integer located after a boolean at the data section d_0, to the writing location of c_1. This instruction is, for example, when implementing forwarding of sessions in CLASS.

The `cloneSlots` instruction is used to clone data from one location to another, taking the same arguments. This instruction is only defined for cloneable slots, which are int, bool, string, tag and exponential. In addition to copying the memory, it also increments reference counts for exponentials and allocates new strings as necessary. It is used to model the exponential part of CLASS.

3.1.5 Dropping Data

To model the affine and exponential parts of CLASS, it is necessary to be able to drop some types of data if they are no longer needed. To this end, two instructions are provided: `dropSlots` and `deferDrop`.

The `dropSlots` instruction takes as arguments a data location and a slot tree, and immediately drops the passed data. The cont slot requires special care. The remote endpoint of the stored continuation is expected to follow the affine protocol, defined as follows: when the continuation is to be dropped, a tag with value 0 is written to indicate the drop, and control is yielded to the continuation. Control should then be returned, at which point the continuation is considered dropped.

The `deferDrop` instruction takes as argument a *drop bit identifier*, and sets the respective bit. Drop bits are declared in the process header, and indicate a data location and a slot tree to be dropped when the process terminates, if the bit is set. Drop bits are identified by an `e` followed by a number (e.g., `e0`).

Below is an example of dropping data directly and through a drop bit:

```
main:
    data d0: <[string; string]>
    drop e0: [string] at d0.[string]~[string]
entry:
    ... // something that initializes both strings at d0
    deferDrop(e0) // the second string will be dropped on environment cleanup
    dropSlots(d0, [string]) // the first string is dropped immediately
```

3.1.6 Expressions

Expressions are tree-like structure which compute basic values (int, bool or string) without side effects. IR expressions are defined exactly as the expressions of CLASS, with the difference that session names are replaced by `move(d, S)` and `clone(d, S)` expressions. Move and clone expressions read a slot S from a data location d, with the difference that move consumes the data (cannot be read again) while clone leaves the data intact. For example, an integer clone is equivalent to a move, as integers are trivially copyable, but a string clone must allocate a new string and copy the contents.

While expressions do not occur as instructions, they are used as arguments to some instructions, such as `writeExpression`, `branchExpr` and `print`. An example of an expression is `(move(d0, int) + 3) + clone(d1, string)`. This expression reads an integer from d_0, adds 3 to it, then reads a cloned string from d_1 and finally concatenates the two.

3.1.7 Input and Output

Output is modeled through the `print` and `printLine` instructions. They take a single argument expression, which is evaluated and printed to the standard output. The `printLine` instruction additionally appends a new line after printing the value. Input is modeled through the `writeScan` instruction, which reads values from the standard input. It takes as arguments a data location to store the read value to and a slot describing the type of data to be read. Below is an example program, which reads two integers and prints their sum.

```
main:
    data d0: <[int; int]>
entry:
    writeScan(d0, int)
    writeScan(d0.[int]~[int], int)
    printLine(move(d0, int) + move(d0.[int]~[int], int))
```

3.1.8 Forwarding Continuations

To model the forwarding of sessions in CLASS, it is necessary to be able to link two continuations together, detaching them from the active environment. This is done through the `forward` instruction.

This instruction takes two continuations as arguments, links their remote endpoints together, and passes control to the continuation of the second argument. Below is an example of forwarding two continuations:

```
entry:
    initContinuation(c0, lhs, d0) // 1
    initContinuation(c1, rhs, d1) // 2
    forward(c0, c1)  // 3
lhs:
    ...                           // 5
rhs:
    finish(c1)                    // 4
```

In this example two continuations c_0 and c_1 are initialized as before. The `forward(c0, c1)` instruction is then executed, which makes c_0 the remote continuation of c_1 and vice-versa. Control then passes to c_1, and thus, to the block `rhs`. The instruction `finish(c1)` is finally executed, which as before, passes control to c_1, and thus to the block `lhs`, as c_1's code location was set to c_0's when forwarded.

3.1.9 Tags and Branching

Branching is done through the `branchTag` and `branchExpr` instructions. The first instruction branches on a tag stored a data location, similarly to a switch statement in C, and the second instruction branches on the result of a boolean expression, similarly to an if statement in C. Each possible branch is defined by a code location (block label) and an integer indicating how many exit points are behind that branch. This number is used to adjust the exit-point count of the environment when branching.

Tags are written to data locations through the `writeTag` instruction, which takes as arguments a data location and a constant tag value. Below is an example of writing a tag and then branching on it.

```
main:
    data d0: <tag[[int] | [string]]>
entry:
    writeTag(d0, 1)
    branchTag(d0, if_int:1, if_string:1)
if_int: // ...
if_string: // ...
```

3.1.10 Tasks

As mentioned before, tasks are a control flow mechanism completely separate from environments and continuations, and are used to model the **par** construct from CLASS. The abstract machine maintains a stack of tasks, where each task stores a continuation to be executed when the current task is finished.

Tasks can be pushed to the stack through the `pushTask` instruction, which takes a code location as argument. To finish the current task and jump to the continuation of the topmost task, the `popTask` instruction is used.

When execution begins, the task stack is initialized with a single task responsible for terminating the program. Thus, the last instruction to be executed in a program is always `popTask`.

3.1.11 Concurrency

New threads of execution can be launched through the `launchThread` instruction, which takes as arguments a code location to start executing at. Each thread functions as a separate instance of the abstract machine, with its own stack of tasks and active environment. When a thread is launched, it inherits the current thread's active environment.

Accesses to the environment are not synchronized except for reference and exit-point counting, which are done atomically. When generating IR programs from CLASS, its type system guarantees that no two threads will access the same data concurrently other than through cells, which are synchronized.

3.1.12 Cells

Cells are used to model the shared state part of CLASS. They're mutable memory locations which can be safely shared between multiple threads of execution. Cells are referenced through cell $[T]$ slots, and maintain a mutex to ensure safe concurrent access, as well as a reference count to guarantee proper deallocation.

A new cell can be created through the `writeCell` instruction, which takes as arguments a data location to write the cell $[T]$ slot to, and a slot tree T describing the data that will be stored in the cell. The reference count of a cell can be incremented and decremented through the `acquireCell` and `releaseCell` instructions, respectively. Finally, the mutex of a cell can be locked and unlocked through the `lockCell` and `unlockCell` instructions, respectively. The data stored in a cell can be accessed through any data location, by using a cell location as the base address, as explained in Section 3.1.3.A.

Below is an example of creating a cell, writing data to it, acquiring and releasing references to it, and finally deallocating it:

```
main:
    data d0: <cell[int]>
entry:
    writeCell(d0, [int]) // create a cell with space for an integer
    lockCell(c(d0)) // lock the cell's mutex
    writeExpression(c(d0), 42) // write 42 to the cell
    unlockCell(c(d0)) // unlock the cell's mutex
    acquireCell(c(d0)) // increase reference count to 2
    releaseCell(c(d0)) // decrease reference count to 1
    releaseCell(c(d0)) // decrease reference count to 0, deallocating the cell
```

3.1.13 Polymorphism

Processes in the IR may be type parametrized. These type parameters are identified by names which start with `t` followed by a number, such as t_0 and t_1, and are assigned when the process is called. Within the process, types parameters may be used directly as slots in slot trees (e.g.: $[t_2; \text{int}; t_0]$).

To model the CLASS **sendty** and **recvty** constructs, it is necessary to send type information dynamically through sessions. To achieve this, the `writeType` instruction takes as arguments a target data location and a slot tree describing the data layout of the type. This instruction writes the necessary type information to the data location, producing a type slot. For example, `writeType(d0, [int; t1])` writes information to d_0 describing the slot tree $[\text{int}; t_1]$. The written type slot can then later be used as a type parameter when calling a process, as explained in the next section.

3.1.14 Process Calls

On the example execution shown earlier, a process call instruction was shown, but only a continuation was passed as argument. In order to model the full CLASS process call semantics, it is necessary to also be able to pass both type parameters and data arguments to the new process.

The `callProcess` instruction thus takes as arguments the name of the process to invoke, a list of type parameters, and a list of continuation and data arguments to initialize the new process environment.

A type parameter is described by the target type parameter identifier on the new process, and by either a data location containing a type slot, or by a slot tree directly. For example, `t0 <- [int]` initializes the type parameter t_0 of the new process to be a type which stores an integer, while `t1 <- d3.[int]~[type]` initializes t_1 to the type parameter stored after an integer at d_3.

A continuation argument is described by the target continuation identifier on the new process, by either a source continuation identifier or a data location containing a cont slot, and finally, by a data offset to apply to the writing location of the continuation. For example, `c0 <- c1+[int]~[int]` initializes the continuation parameter c_0 of the new process to be the same as continuation c_1 of the current process, but with its writing location pointing at $c_1.[\text{int}] \sim [\text{int}]$. If the offset is zero, it is omitted for brevity.

Finally, a data argument is described by the target data identifier on the new process, by a source data location, by a slot tree describing the data, and by a flag indicating whether to clone the data or not. For example, `d0 =[string] d1` writes to d_0 a clone of the string stored at d_1, while `d1 <-[string] d2` moves to d_1 the string stored at d_2 without cloning it.

An example of calling a process with type, session and data arguments is shown below.

```
foo:
    types: t0
    continuations: c0
    data d0: <[t0]>
entry:
    moveSlots(c0, d0, [t0]) // 4
    finish(c0)              // 5

main:
    continuations: c0
    data d0: <[int]>
entry:
    initContinuation(c0, d0, rhs)                          // 1
    writeExpression(c0, 42)                                // 2
    callProcess(foo, t0 <- [int], c0 <- c0, d0 <-[int] d0) // 3
rhs:
    printLine(move(d0, int)) // 6
```

First, the continuation c_0 is initialized, with its writing location being d_0 and code location being the block `rhs`. Then, the integer `42` is written to c_0 (which is d_0). Next, the process `foo` is called, with its type parameter t_0 being set to an integer, its continuation c_0 being set to the local c_0, and its data section d_0 being initialized by a move of the data section d_0. Inside `foo`, the integer stored at its data section d_0 (received from `main`) is moved to the writing location of its continuation c_0 (which is still d_0 from the `main` process), and then, the session is finished, passing control back to the block `rhs` of the calling process. Finally, the integer stored at d_0 is printed.

3.1.15 Exponentials

Exponentials in CLASS can be modeled through the exponential slot, which stores the necessary data to dynamically call a process closure, along with any captured data arguments. Exponentials are created through the `writeExponential` instruction, which takes the same arguments as `callProcess`, with the exception that continuation arguments are not allowed, and that passed data must be cloneable. Additionally, the referenced process is expected to follow the exponential process convention.

The convention for exponential processes is simply that they must take a single continuation argument, c_0, which will be used to communicate with the caller. This continuation is initialized by the `callExponential` instruction, which invokes the process. This instruction takes as arguments a data location containing an exponential slot, a continuation identifier to bind the exponential process continuation to, and a data location to set as the writing location on the exponential process.

One important detail is that exponentials are reference counted data, and the exponential slot must be dropped when no longer needed, or at the end of the process, if it was not moved. This is done through the `dropSlots` or `deferDrop` instructions, as explained in the Section 3.1.5 section. Drops of exponentials decrement this reference count, while clones increment it. Below is an example of creating and calling an exponential.

```
foo:
    types: t0
    continuations: c0
    data d0: <[t0]>
entry:
    moveSlots(c0, d0, [t0]) // 6, 10
    finish(c0)                // 7, 11

main:
    continuations: c0 c1
    data d0: <[string]>
    data d1: <[exponential]>
    data d2: <[string]>
    data d3: <[string]>
    drop e0: [exponential] at d1
entry:
    writeExpression(d0, "Hello, World!")                        // 1
    writeExponential(d1, foo, t0 <- [string], d0 <-[string] d0) // 2
    deferDrop(e0)                 // 3
    callExponential(d1, c0, d2) // 4
    callExponential(d1, c1, d3) // 4
    continue(c0, rhs0)            // 5
rhs0:
    printLine(move(d2, string)) // 8
    continue(c1, rhs1)            // 9
rhs1:
    printLine(move(d3, string)) // 12
```

In this example, the process `foo` is a simple polymorphic process with a type parameter t_0, a continuation c_0 and a data section d_0. It moves data of type [string] from d_0 to the writing location of c_0, and then finishes on c_0, passing control back to the caller.

The `main` process writes a new string to d_0, and then creates a new exponential at d_1 for the process `foo`, moving the just written string to it. This string will be cloned every time the exponential is called, and dropped when the exponential is freed. The exponential will be dropped at the end of the process through the drop bit `e0`, which is set by the `deferDrop(e0)` instruction.

The exponential is then called twice, first binding its argument continuation to c_0 and its writing location to d_2, and then binding its continuation to c_1 and its writing location to d_3. Each call will clone the string stored at d_0 and pass it to the exponential process. Control is not immediately passed to the exponential process. Instead, it is passed when the argument sessions, either c_0 or c_1, are continued, as their continuations were set by the `callExponential` instruction to be the entry block of the exponential process.

Finally, within the exponential process, the string is moved from d_0 to the remote data of c_0, which is either d_2 or d_3 of the caller, depending on which call was continued. Finally, the continuation is finished, passing control back to the caller, which prints the received string.

3.2 Translating CLASS to the IR

This section defines the base (unoptimized) translation from CLASS to the IR defined in the previous section. It starts by defining notation used to represent and update the state of the compiler. Then, it defines how session types are mapped to the slot trees introduced in Section 3.1.3. Next, an overview of the instruction translation scheme is presented, and some important concepts are defined. Finally, the translation from CLASS processes to IR programs is incrementally defined, starting from simple processes and adding more constructs until the whole language is covered.

Well-typed CLASS processes are given as input, along with the required type information of free names in all processes. The notation $x : A$ is used to indicate that the type checker infers session endpoint x to be a free name of type A on some process.

3.2.1 Compiler State

Compilation of CLASS processes into IR programs always takes place within the context of a single IR process. As CLASS processes are compiled into IR instructions, new information is added to the IR process header, along with the new instructions and blocks. Changes to the header of the IR process are denoted by the actions shown below.

$$c \leftarrow \mathsf{new_c}() \quad (3.1) \qquad d \leftarrow \mathsf{new_d}(T) \quad (3.2)$$

$$e \leftarrow \mathsf{new_e}(L,\, S) \quad (3.3) \qquad l \leftarrow \mathsf{new_b}() \quad (3.4)$$

Action (3.1) adds a new continuation to the process header, identified by c. Action (3.2) creates a new data section with data layout T identified by d. Action (3.3) creates a new drop bit pertaining to data location L and slots S, identified by e. Action (3.4) generates a unique label l for a new instruction block within the process body.

The compiler must keep track of information regarding the type parameters and the state of each session endpoint within the process being compiled. This state is referred to as the environment, and is represented by a tuple (θ, Δ), where θ is the type environment and Δ is the session environment.

3.2.1.A Type Environments

A type environment θ is a mapping from type variables to tuples (p, t), where p is the polarity of the type variable ($+$ or $-$) and t is its identifier (e.g, t_0, t_1) within the header of the process being compiled. $\theta_p(X)$ and $\theta_t(X)$ are defined as the polarity and identifier of type variable X in the type environment θ, respectively.

Positive polarity types type session endpoints which are used to write data, while a type with negative polarity types endpoints which are used to read data. The function $\theta_p(A)$ is further extended to all session types A through pattern matching, as shown in Figure 3.2. For the base case of type variables, the polarity is looked up in the type environment.

Note that due to affine types being implemented as a choice between using the data or dropping it, i.e., **affine** $A \equiv$ **offer of** $\{$|**#use** $: A$ |**#discard** : **close**$\}$, the polarity of **affine** A is negative, while the polarity of **coaffine** A is positive.

$$
\begin{array}{llll}
\theta_p(\textbf{lint}) = + & \theta_p(\textbf{colint}) = - & \theta_p(\textbf{lbool}) = + & \theta_p(\textbf{colbool}) = - \\
\theta_p(\textbf{lstring}) = + & \theta_p(\textbf{colstring}) = - & \theta_p(\textbf{close}) = + & \theta_p(\textbf{wait}) = - \\
\theta_p(\textbf{choice of}\,\{\dots\}) = + & \theta_p(\textbf{offer of}\,\{\dots\}) = - & \theta_p(\textbf{send}\,A; B) = + & \theta_p(\textbf{recv}\,A; B) = - \\
\theta_p(\textbf{!}A) = + & \theta_p(\textbf{?}A) = - & \theta_p(\textbf{sendty}\,X; A) = + & \theta_p(\textbf{recvty}\,X; A) = - \\
\theta_p(\textbf{coaffine}\,A) = + & \theta_p(\textbf{affine}\,A) = - & \theta_p(\textbf{state}\,A) = + & \theta_p(\textbf{usage}\,A) = - \\
\theta_p(\textbf{rec}\,X;\ A) = + & \theta_p(\textbf{corec}\,X;\ A) = - & \theta_p(\textbf{statel}\,A) = + & \theta_p(\textbf{usagel}\,A) = -
\end{array}
$$

Figure 3.2: Polarity of session types

3.2.1.B Session Environments

A session environment Δ is a mapping from session endpoint names to session states. Each session state is a tuple (c, d, o), where c is the continuation identifier associated with the session endpoint (or 0 if there is no continuation), d is the identifier of the data section from which data is read (or 0 if there is no data section), and o is the data offset to apply to the location of the next write or read operation on the session. The data offset is represented as a slot tree, as explained in Section 3.1.3.A.

Given a session environment Δ, $\Delta[x \mapsto (c, d, o)]$ is defined as the updated mapping where session endpoint x is mapped to the new session state (c, d, o). Additionally $\Delta_c(x)$, $\Delta_d(x)$ and $\Delta_o(x)$ are defined as the continuation identifier, local data identifier and data offset associated with session x in the session environment Δ, respectively.

During development, in the unfinished compiler, session data offsets were maintained as runtime state, being updated whenever a read or write operation was performed. This approach was later abandoned in favor of the current version, where offsets are known at compile time, greatly simplifying the generated code and reducing runtime overhead.

3.2.1.C Writing and Reading Locations

For convenience, $\Delta_r(x)$ and $\Delta_w(x)$ are denoted as the current data location from which data of session endpoint x is to be read, and the data location into which data for session x is to be written, respectively. Both take into account the offset, and are defined as

$$\Delta_r(x) = \Delta_d(x).\Delta_o(x) \qquad\qquad \Delta_w(x) = \Delta_c(x).\Delta_o(x)$$

3.2.1.D Advancing Session Offsets

When more than one slot is read from or written to a session endpoint in succession, the session's offset must be advanced accordingly. To this end, the concatenation of offsets must be defined. Given two offsets $o_1 = P_1{\sim}F_1$ and $o_2 = P_2{\sim}F_2$, their concatenation is defined as $o_1 \oplus o_2 = (P_1 \oplus P_2){\sim}F_2$, where $P_1 \oplus P_2$ is the concatenation of the two slot trees. In other words, when two offsets are concatenated, the resulting offset has concatenates both past slot trees, and keeps the future slot tree of the second offset. Slot tree concatenation is defined as replacing every leaf of the first slot tree with the second tree.

To simplify the definitions in the next sections, $\Delta[x \overset{p,\theta}{\oplus} (S,\ A)]$ denotes the session environment Δ where the offset of session x is either reset, or advanced by the necessary offset to skip over the slots S. The decision is made based on whether the polarity p (either $+$ or $-$) matches the polarity of the type A in the type environment θ. If they match, the new offset is computed as $\Delta_o(x) \oplus (S{\sim}\mathcal{S}(\theta,\ A))$, where S is a slot tree describing the data which was just read or written, and $\mathcal{S}(\theta,\ A)$ is the slot tree corresponding to type A, which will be defined in the next section. Otherwise the new offset is 0. Formally:

$$\Delta[x \overset{p,\theta}{\oplus} (S,\ A)] = \begin{cases} \Delta[x \mapsto (\Delta_c(x), \Delta_d(x), \Delta_o(x) \oplus (S{\sim}\mathcal{S}(\theta,\ A)))] & \text{if } p = \theta_p(A) \\ \Delta[x \mapsto (\Delta_c(x), \Delta_d(x), 0)] & \text{otherwise} \end{cases}$$

3.2.2 Mapping Session Types to Slot Trees

To translate CLASS processes to IR programs, we need to define which data is sent and received over sessions. As explained in Section 3.1.3, in the IR data types are represented through slot trees. Thus, we need to define how session types are mapped to slot trees.

We start by defining this mapping for session types A which are strictly positive. Let $\mathcal{S}(\theta,\ A)$ be the slot tree corresponding to the data exchanged over a session type A within type environment θ. This function is defined through pattern matching on session types:

$$\begin{array}{ll}
\mathcal{S}(\theta,\ \textbf{close}) = [] & \mathcal{S}(\theta,\ \textbf{lint}) = [\text{int}] \\
\mathcal{S}(\theta,\ \textbf{lbool}) = [\text{bool}] & \mathcal{S}(\theta,\ \textbf{lstring}) = [\text{string}] \\
\mathcal{S}(\theta,\ \textbf{send}\ A; B) = [\text{cont}; \mathcal{S}(\theta,\ B)] & \mathcal{S}(\theta,\ \textbf{!}A) = [\text{exponential}] \\
\mathcal{S}(\theta,\ \textbf{sendty}\ X; A) = [\text{type}; \text{cont}; \text{tag}] & \mathcal{S}(\theta,\ \textbf{coaffine}\ A) = \text{tag}\,[[]\ \mid \mathcal{S}(\theta,\ A)] \\
\mathcal{S}(\theta,\ \textbf{state}\ A) = [\text{cell}\,[\text{cont}]] & \mathcal{S}(\theta,\ \textbf{statel}\ A) = [\text{cell}\,[\text{cont}]] \\
\mathcal{S}(\theta,\ \textbf{choice of}\ \{|\textbf{\#a} : A\ \ldots\}) = \text{tag}\,[\mathcal{S}(\theta,\ A) \mid\ \ldots] & \mathcal{S}(\theta,\ X) = [\theta_t(X)]\ \text{s.t. } \theta_p(X) = + \\
\mathcal{S}(\theta,\ A) = []\ \text{if } \theta_p(A) = - &
\end{array}$$

To exemplify, consider the session type **send lint**; **send close**; **choice of** $\{|$**#a** : **lstring** $|$**#b** : **lbool**$\}$. Its slot tree is [cont; cont; tag [[string] | [bool]]], as first a continuation is sent, then another continuation is sent, and finally, a tag is sent followed by either a string or a boolean, depending on the sent tag.

To extend this concept to all session types, we must consider that session types may have multiple segments of data exchange, separated by changes in polarity. We can separate a session type into its segments by replacing every continuation of a different polarity with the unit type of that polarity (**close** or **wait**) on the current segment and starting a new segment with the continuation.

For example, we can split **send lint**; **recv lstring**; **close** into three segments: first **send lint**; **close**, then **recv lstring**; **wait** and finally **close**. Each segment is either strictly positive or strictly negative. Strictly negative segments can be turned into strictly positive segments by computing their dual types.

Thus, we can map each segment to a slot tree through the $\mathcal{S}(\theta,\, A)$ function defined above.

The slot tree corresponding to the first segment of a session type A is designated as its *active slot tree* (denoted simply by $\mathcal{S}(\theta,\, A)$), and the slot trees corresponding to the remaining segments as its *remainder slot trees*. Additionally, we denote positive segments as *remote slot trees* and negative segments as *local slot trees*, as positive segments correspond to data which is sent (and thus stored remotely), while negative segments correspond to data which is received (and thus stored locally). The local set of slot trees of a session type A is denoted by $\mathcal{S}_L(\theta,\, A)$, and the total set of slot trees by $\mathcal{S}_T(\theta,\, A)$. Sets of slot trees are termed *layouts*, as they describe the full set of data that may be exchanged over a session.

3.2.3 Instruction Translation

The IR instructions translated from a CLASS process P on a given environment (θ, Δ) are denoted by $\mathcal{I}_P(\theta,\, \Delta,\, P)$. This function returns a sequence of IR instructions, optionally followed by blocks with more instructions, and is defined incrementally in the following sections.

The translated code implements the write-to-read adjustment of the SAM through the placement of `continue` instructions whenever the polarity of a session changes from positive to negative. To simplify the definition of the translation rules, a pseudo-instruction `continueIfNegative` is defined as

$$\texttt{continueIfNegative}(\theta, \Delta, x, A) \equiv \begin{cases} \quad \texttt{\% label} \leftarrow \mathsf{new_b}() & \\ \quad \texttt{continue}(\Delta_c(x), \texttt{label}) & \text{if } \theta_p(A) = - \\ \texttt{label:} & \\ \hline \quad \texttt{\% empty} & \text{otherwise} \end{cases}$$

which means that if and only if $\theta_p(A) = -$, `continueIfNegative` is replaced by a `continue` instruction, and the next instructions are placed in a new block. Otherwise, `continueIfNegative` is simply removed from the instruction sequence.

3.2.4 Expression Translation

Apart from translating CLASS processes to instructions, CLASS expressions must also be translated to IR expressions. This is necessary for when translating processes such as **print**, **let** and **if**. To achieve this, the function $\mathcal{I}_E(\Delta,\, E)$ is defined, which takes a CLASS expression E and a session environment Δ, and returns an equivalent IR expression. This function is trivially defined recursively through pattern matching on CLASS expressions, which are identical to IR expressions, except for session names x, for which the translation is defined as

$$\mathcal{I}_E(\Delta,\, x) = \texttt{move}(\Delta_r(x), S)$$

where S is the slot corresponding to the type of the session endpoint x, which is guaranteed to be a basic type (e.g., int for **colint**, string for **colstring**).

3.2.5 Exit Point Counting

Another important aspect of the compilation is counting the number of exit points of the translated IR code for a given CLASS process. This is necessary to ensure that the translated code properly deallocates the process environment when and only when there is no more code left to execute within it.

The number of exit points in a process P is denoted by $\mathcal{E}(\theta,\, P)$, and is defined by pattern matching on processes. It will be incrementally defined along with the instruction translation function in the following sections.

3.2.6 Process Definitions

A CLASS program is a set of process definitions, each identified by a unique name. For each process definition, a set of resulting IR processes is generated, one for each possible combination of polarities of its type parameters. This is necessary to ensure that the polarities of type parameters are always known at compile time, as they greatly affect the control flow of the translated code.

For each resulting IR process, an environment (θ, Δ) is initialized. The type environment θ maps each type parameter name to a tuple (p, t) such that p is the polarity of the type parameter in the current combination, and t is a unique identifier for the type parameter, which is inserted into the process header.

The session environment Δ maps each argument session name to a session state $(c, d, 0)$. If the argument is linear, $c \leftarrow \text{new}_{\text{c}}()$. Otherwise, $c = 0$, as exponential sessions do not have an associated continuation. In both cases, $d \leftarrow \text{new}_{\text{d}}(\mathcal{S}_L(\theta,\, A))$, where A is the type of the argument session. The data layout of d reserves space for all the data that will be received on the session endpoint. The offset is initialized to 0 since no data has yet been read or written.

For each of the exponential arguments of the CLASS process definition, a new drop bit is added through $e \leftarrow \text{new}_{\text{e}}(L,\, S)$, where L is the location of the exponential data, and S is the slot tree describing the data. This bit e is marked as always set, guaranteeing that the received exponential data is dropped when the process call finishes.

Finally, we generate the instructions for the process body P on the `entry` block through $\mathcal{I}_P(\theta,\, \Delta,\, P)$, and set the number of exit points of the IR process to $\mathcal{E}(\theta,\, P)$.

3.2.7 Cut, Parallel and Empty

The compilation definitions for the **cut** $\{P \,|x : A|\, Q\}$ process on an environment (θ, Δ) are shown below. It is assumed without loss of generality that $\theta_p(A) = +$, and thus that P will be writing to x before Q

executes; if A were negative, P and Q could simply be swapped. Both the continuation c and the data section d are created to implement the session x. The layout $\mathcal{S}_T(\theta,\, A)$ is given to d, guaranteeing that enough space is allocated for all data received by both endpoints of x.

$$
\begin{aligned}
&\mathcal{E}(\theta,\, \textbf{cut}\,\{P\,|x : A|\,Q\}) \equiv \mathcal{E}(\theta,\, P) + \mathcal{E}(\theta,\, Q)\\
&\mathcal{I}_P(\theta,\, \Delta,\, \textbf{cut}\,\{P\,|x : A|\,Q\}) \equiv\\
&\qquad \%\ c \leftarrow \mathsf{new}_{\mathsf{c}}(), d \leftarrow \mathsf{new}_{\mathsf{d}}(\mathcal{S}_T(\theta,\, A)), \mathtt{rhs} \leftarrow \mathsf{new}_{\mathsf{b}}()\\
&\qquad \mathtt{initializeContinuation}(c, \mathtt{rhs}, d)\\
&\qquad \mathcal{I}_P(\theta,\, \Delta[x \mapsto (c, d, 0)],\, P)\\
&\quad \mathtt{rhs:}\\
&\qquad \mathcal{I}_P(\theta,\, \Delta[x \mapsto (c, d, 0)],\, Q)
\end{aligned}
$$

The instructions initialize the continuation c with a new block `rhs` as the return address and d as a writing location. Both P and Q are recursively translated using a new environment where the name x is bound to $(c, d, 0)$. P is placed on the original block and Q on the new block `rhs`. The total number of exit points is the sum of the number of exit points of both P and Q.

The **par** $\{P \,||\, Q\}$ process, unlike the **cut** process, does not link the two processes P and Q in any way. The SAM implements this construct by executing Q only after P finishes. The compilation scheme achieves this through the task stack mechanism. Q is compiled into a new block `rhs`. A task for `rhs` is pushed through the `pushTask` instruction. Eventually, when P finishes execution, `popTask` executes, and the program counter jumps to `rhs`, executing Q.

$$
\begin{aligned}
&\mathcal{E}(\theta,\, \textbf{par}\,\{P \,||\, Q\}) \equiv \mathcal{E}(\theta,\, P) + \mathcal{E}(\theta,\, Q)\\
&\mathcal{I}_P(\theta,\, \Delta,\, \textbf{par}\,\{P \,||\, Q\}) \equiv\\
&\qquad \mathtt{pushTask}(rhs) \quad \%\ \mathtt{rhs} \leftarrow \mathsf{new}_{\mathsf{b}}()\\
&\qquad \mathcal{I}_P(\theta,\, \Delta,\, P)\\
&\quad \mathtt{rhs:}\\
&\qquad \mathcal{I}_P(\theta,\, \Delta,\, Q)
\end{aligned}
$$

The **()** process represents the end of a computation. Control must be passed to the next task in the task stack, potentially terminating the execution. This is achieved through the `popTask` instruction. Because control might not return to the current process call, `popTask` is marked as an exit point.

$$
\begin{aligned}
&\mathcal{E}(\theta,\, \textbf{()}) \equiv 1\\
&\mathcal{I}_P(\theta,\, \Delta,\, \textbf{()}) \equiv\\
&\qquad \mathtt{popTask(exit\ point)}
\end{aligned}
$$

3.2.8 Close and Wait

The **close** x process indicates that communication over a session x is finished. As in the SAM, control is passed to the other endpoint of x, never to return. This jump is performed with the `finish` instruction, taking as its argument the continuation of session x, obtained from Δ. As in the previous case, this instruction is marked as an exit point.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{close}\,x) \equiv 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{close}\,x) \equiv\\
&\qquad \texttt{finish}(\Delta_c(x), \texttt{exit point})
\end{aligned}$$

The **wait** process is a no-op during compilation: $\mathcal{E}(\theta,\ \textbf{wait}\,x; P) \equiv \mathcal{E}(\theta,\ P)$, $\mathcal{I}_P(\theta,\ \Delta,\ \textbf{wait}\,x; P) \equiv \mathcal{I}_P(\theta,\ \Delta,\ P)$. No instructions are generated for the wait action, since the act of waiting for the other side of the channel is already performed implicitly by the SAM execution scheme.

3.2.9 Let, Scan and Print

The **let** $x\,E$ process is similar to the **close** process, except that before passing control to the other endpoint of session x, an expression E is evaluated, and its result is sent over the session. The result is written to the writing location of the continuation of x, given by $\Delta_w(x)$.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{let}\,x\,E) \equiv 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{let}\,x\,E) \equiv\\
&\qquad \texttt{writeExpression}(\Delta_w(x), \mathcal{I}_E(\Delta,\ E))\\
&\qquad \texttt{finish}(\Delta_c(x), \texttt{exit point})
\end{aligned}$$

The **scan**(x) process is nearly identical, except that instead of computing an expression, it uses the `writeScan` instruction to read from the standard input.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{scan}(x)) \equiv 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{scan}(x)) \equiv\\
&\qquad \texttt{writeScan}(\Delta_w(x), \mathcal{S}(\theta,\ \tau)) \quad \%\ \text{where } x : \tau\\
&\qquad \texttt{finish}(\Delta_c(x), \texttt{exit point})
\end{aligned}$$

To compile the **print**$(E); P$ and **println**$(E); P$ processes, a single instruction is generated to print the result of expression E to the standard output. Compilation then recurses on P.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{print}(E); P) \equiv \mathcal{E}(\theta,\ P)\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{print}(E); P) \equiv\\
&\qquad \texttt{print}(\mathcal{I}_E(\Delta,\ E))\\
&\qquad \mathcal{I}_P(\theta,\ \Delta,\ P)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{println}(E); P) \equiv \mathcal{E}(\theta,\ P)\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{println}(E); P) \equiv\\
&\qquad \texttt{printLine}(\mathcal{I}_E(\Delta,\ E))\\
&\qquad \mathcal{I}_P(\theta,\ \Delta,\ P)
\end{aligned}$$

3.2.10 Choice and Branching

The choice process **#a** $x; P$ selects the branch labeled `a` on the offer process at the other endpoint of session x, and then continues as P. This selection is implemented by writing a tag that indicates the chosen branch to the writing location of x, $\Delta_w(x)$. This tag is represented by an integer, given by $i(\#a)$, where $i(\#a)$ is the index of the tag `a` in the alphabetically sorted list of tags in the choice type of x.

Since choice is a writing action, a write-to-read adjustment might need to be performed — as in the SAM — if the next action on x is a reading action. This is the case if the remainder type of x in P is negative. To implement the write-to-read adjustment, the pseudo-instruction `continueIfNegative` introduced earlier is used.

Finally, instructions for the remainder process P are generated within an environment where the offset of channel x is advanced to skip over the written tag. As covered in Section 3.2.1.D, the type A of x in P is used to determine whether the offset is advanced or reset to zero. In this case, the offset advances by [tag] only if the next action on x is also a write, i.e., if $\theta_p(A) = +$.

$$\begin{aligned}
&\mathcal{E}(\theta,\, \textbf{\#a}\ x; P) \equiv \mathcal{E}(\theta,\, P)\\
&\mathcal{I}_P(\theta,\, \Delta,\, \textbf{\#a}\ x; P) \equiv\\
&\qquad \texttt{writeTag}(\Delta_w(x), i(\#a))\\
&\qquad \texttt{continueIfNegative}(\theta, \Delta, x, A) \quad \text{\% where } x : A \text{ in } P\\
&\qquad \mathcal{I}_P(\theta,\, \Delta[x \overset{+,\theta}{\oplus} ([\text{tag}]\,,\, A)],\, P)
\end{aligned}$$

The compilation of a **case** x **of** $\{\dots\}$ process is shown below. The compiler generates code that reads a tag from the local data of session x, jumps to the corresponding label and then recursively generates code for each possible branch, each preceded by the respective label. The labels are generated by their alphabetical order to match the tags written by the `writeTag` instruction. As before, the offset of channel x is advanced to skip over the read tag, but in this case only if the next action on x is a read, i.e., if $\theta_p(A) = -$ for the type A of x in each branch.

$$\begin{aligned}
&\mathcal{E}(\theta,\, \textbf{case}\, x\, \textbf{of}\, \{|\textbf{\#a} : A,\, \dots,\, |\textbf{\#z} : Z\}) \equiv \max(\mathcal{E}(\theta,\, A),\, \dots,\, \mathcal{E}(\theta,\, Z))\\
&\mathcal{I}_P(\theta,\, \Delta,\, \textbf{case}\, x\, \textbf{of}\, \{|\textbf{\#a} : A,\, \dots,\, |\textbf{\#z} : Z\}) \equiv\\
&\qquad \texttt{branchTag}(\Delta_r(x), \texttt{a} : \mathcal{E}(\theta,\, A),\, \dots,\, \texttt{z} : \mathcal{E}(\theta,\, Z))\\
&\quad \texttt{a:}\\
&\qquad \mathcal{I}_P(\theta,\, \Delta[x \overset{-,\theta}{\oplus} ([\text{tag}]\,,\, \tau_A)],\, A) \quad \text{\% where } x : \tau_A \text{ in } A\\
&\qquad \dots\\
&\quad \texttt{z:}\\
&\qquad \mathcal{I}_P(\theta,\, \Delta[x \overset{-,\theta}{\oplus} ([\text{tag}]\,,\, \tau_Z)],\, Z) \quad \text{\% where } x : \tau_Z \text{ in } Z
\end{aligned}$$

The number of exit points of branching processes is given by the maximum number of exit points across all branches. During execution, only one branch runs, and the current exit-point count is adjusted accordingly, as described in Section 3.1.9.

3.2.11 Send and Receive

The **send** $x(y.\,P); Q$ process creates a new session y, sending one of its endpoints over session x. The other endpoint of y is then used in process P, while process Q continues using session x. To model this behavior, the continuation c and data section d are created to implement session y, where A is the type of session y. The layout $\mathcal{S}_L(\theta,\, A)$ is given to d, guaranteeing that enough space is allocated for all data that will be received by the local endpoint of y.

The continuation c is initialized with return address `lhs` and writing location d. c is then written to the writing location of session x, $\Delta_w(x)$. As before, a write-to-read adjustment might be necessary and thus the `continueIfNegative` pseudo-instruction is used.

```
$\mathcal{E}(\theta,\, \textbf{send}\, x(y.\,P); Q) \equiv \mathcal{E}(\theta,\, P) + \mathcal{E}(\theta,\, Q)$
$\mathcal{I}_P(\theta,\, \Delta,\, \textbf{send}\, x(y.\,P); Q) \equiv$
    $\%\ c \leftarrow \mathsf{new_c}(), d \leftarrow \mathsf{new_d}(\mathcal{S}_L(\theta,\, A)), \texttt{lhs} \leftarrow \mathsf{new_b}()$
    $\texttt{initializeContinuation}(c, \texttt{lhs}, d)$
    $\texttt{writeContinuation}(\Delta_w(x), c)$
    $\texttt{continueIfNegative}(\theta, \Delta, x, A)$    $\%$ where $x : A$ in $Q$
    $\mathcal{I}_P(\theta,\, \Delta[x \overset{+,\theta}{\oplus} ([\mathsf{cont}]\,,\, A)],\, Q)$
  $\texttt{lhs:}$
    $\mathcal{I}_P(\theta,\, \Delta[y \mapsto (c, d, 0)],\, P)$
```

The **recv** $x(y); P$ process receives a session endpoint y over session x, which then is used in process P. As in the SAM, control must be passed to the other side of session endpoint y if its type is negative, ensuring that the writing side of the session runs first. A a data section d is created on which y's data will be received, along with a continuation c to bind y to.

```
$\mathcal{E}(\theta,\, \textbf{recv}\, x(y : A); P) \equiv \mathcal{E}(\theta,\, P)$
$\mathcal{I}_P(\theta,\, \Delta,\, \textbf{recv}\, x(y : A); P) \equiv$
    $\texttt{bindContinuation}(c, \Delta_r(x), d)$    $\%\ c \leftarrow \mathsf{new_c}(), d \leftarrow \mathsf{new_d}(\mathcal{S}_L(\theta,\, A))$
    $\texttt{continueIfNegative}(\theta, \Delta, y, A)$
    $\mathcal{I}_P(\theta,\, \Delta[x \overset{-,\theta}{\oplus} ([\mathsf{cont}]\,,\, B)][y \mapsto (c, d, 0)],\, P)$    $\%$ where $x : B$ in $P$
```

3.2.12 Forwarding

Given a process **fwd** $x\, y$, it can be assumed without loss of generality that x is of negative type and y of positive type; if this is not the case, x and y can be swapped. Forwarding x to y requires moving any unread data which has already been received on x to y, as in the SAM. This data is described by $\mathcal{S}(\theta,\, A)$, where $y : A$. For example, if $A = \textbf{lint}$, then $[\mathsf{int}]$ must be moved from $\Delta_r(x)$ to the $\Delta_w(y)$. Additionally, the continuations referenced by x and y are linked to each other through the `forward` instruction, ensuring that future communication happens directly between the other endpoints of x and y. The `forward` instruction passes control directly to the other endpoint of y, which will not return, and thus, is marked as an exit point.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{fwd}\,x\,y) \equiv 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{fwd}\,x\,y) \equiv\\
&\qquad \texttt{moveSlots}(\Delta_w(y), \Delta_r(x), \mathcal{S}(\theta,\ A))\\
&\qquad \texttt{forward}(\Delta_c(x), \Delta_c(y), \texttt{exit point})
\end{aligned}$$

3.2.13 Process Calls

Process calls are compiled as shown below. The identifier id of the IR process to call is obtained from the identifier of the CLASS process being called and the polarities of any passed type parameters, matching the scheme used to translate process definitions, covered in Section 3.2.6.

$$\begin{aligned}
&\mathcal{E}(\theta,\ \texttt{processId}\langle A_1,\ \dots,\ A_n\rangle(x_1,\ \dots,\ x_m;\ y_1,\ \dots, y_k)) \equiv 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \texttt{processId}\langle A_1,\ \dots,\ A_n\rangle(x_1,\ \dots,\ x_m;\ y_1,\ \dots, y_k)) \equiv\\
&\qquad \texttt{callProcess}(id, \texttt{exit point}, T, C, D)
\end{aligned}$$

The list of IR type parameters $T_1,\ \dots,\ T_n$ is generated from the CLASS type parameters $A_1,\ \dots,\ A_n$. Each T_i is the active slot tree of type, given by $\mathcal{S}(\theta,\ A_i)$.

Let Δ^* be the session environment used to compile the process definition being called, and x_i^* the name of the i-th linear argument of the process definition. The list of continuation arguments $C_1,\ \dots,\ C_m$ is generated from the linear arguments $x_1,\ \dots,\ x_m$. Each C_i is a tuple (c_t, c_s, o), where $c_t = \Delta_c^*(x_i^*)$ is target continuation, $c_s = \Delta_c(x_i)$ is the source continuation, and $o = \Delta_o(x_i)$ is the offset to apply to the writing location.

Finally, the list of data arguments $D_1,\ \dots,\ D_{m+k}$ is generated similarly to the continuation arguments, but with extra entries for the exponential arguments. Each D_i is a tuple (d_t, d_s, t, c). For each linear argument x_i, $d_t = \Delta_d^*(x_i^*)$ is the target data section, $d_s = \Delta_r(x)$, $t = \mathcal{S}(\theta,\ \tau)$ is the data slots to be moved, where $x_i : \tau$, and c is the flag indicating whether the argument data should be cloned or moved, which in this case is always `move`, as x_i is linear. For each exponential argument y_j, $d_t = \Delta_d^*(y_j^*)$, where y_j^* is the name of the j-th exponential argument of the process definition being called, $d_s = \Delta_r(y_j)$, $t = \mathcal{S}(\theta,\ \textbf{?}\tau)$ where $y_j : \tau$. In this case, c is always `clone`, as y_j is exponential.

3.2.14 Creating and Calling Exponentials

The right-hand-side P of a $\textbf{!}\,x(y); P$ process is compiled as a separate IR process, with a unique identifier id generated from the name of the process definition being compiled. This new process inherits the same type parameters as the current process. Session y is bound to a continuation $c \leftarrow \mathsf{new_c}()$ in the new process header, along with a data section $d \leftarrow \mathsf{new_d}(\mathcal{S}_L(\theta,\ A))$, where $y : A$. The number of exit points of the new process is set to $\mathcal{E}(\theta,\ P)$.

The type system of CLASS guarantees that free names in P other than y are exponentials. For each free name $z_i \neq y$, a data section $d_i \leftarrow \mathsf{new}_\mathsf{d}(\mathcal{S}_L(\theta,\, \mathbf{?}\tau))$ is created in the new process header, where $z_i : \tau$. The environment used to compile P is thus defined as

$$(\theta^*, \Delta^*) = (\theta, [y \mapsto (c, d, 0), z_1 \mapsto (0, d_1, 0),\, \ldots,\, z_m \mapsto (0, d_m, 0)])$$

On the original process, a `writeExponential` instruction is generated referencing the new process identifier, passing any necessary arguments. After writing the exponential, control is passed to the continuation of session x, as in a **close** process, and thus, the instruction is marked as an exit point.

$$\begin{aligned}
&\mathcal{E}(\theta,\, \mathbf{!}\,x(y); P) \equiv 1\\
&\mathcal{I}_P(\theta,\, \Delta,\, \mathbf{!}\,x(y); P) \equiv\\
&\qquad \mathtt{writeExponential}(\Delta_w(x), id, T, D)\\
&\qquad \mathtt{finish}(\Delta_c(x), \mathtt{exit\ point})
\end{aligned}$$

The list of type parameters $T_1,\, \ldots,\, T_n$ is generated from the polarities and identifier of each type parameter in the current type environment θ.

The list of data arguments $D_1,\, \ldots,\, D_m$ is generated from the free names in P, excluding y. Each D_i is a tuple (d_t, d_s, t, c), where $d_t = \Delta^*_d(z_i)$, $d_s = \Delta_r(z_i)$, $t = \mathcal{S}(\theta,\, \mathbf{?}\tau)$ where $z_i : \tau$ and c is the flag indicating whether the argument data should be cloned or moved, which in this case is always `clone`, as all free names in P are exponentials.

The compilation of a $\mathbf{?}\,x; P$ process is shown below. A new drop bit e is created. This drop bit e, indicates that the received slots $\mathcal{S}(\theta,\, A)$ at $\Delta_r(x)$ may need to be dropped when the process call ends. This bit is set by the `deferDrop` instruction. This mechanism ensures that if this process is behind a choice, the data is only dropped if this branch is taken and the data received.

$$\begin{aligned}
&\mathcal{E}(\theta,\, \mathbf{?}\,x; P) \equiv \mathcal{E}(\theta,\, P)\\
&\mathcal{I}_P(\theta,\, \Delta,\, \mathbf{?}\,x; P) \equiv\\
&\qquad \mathtt{deferDrop}(e) \quad \%\ e \leftarrow \mathsf{new}_\mathsf{e}(\Delta_r(x),\, \mathcal{S}(\theta,\, A)),\ \text{where } x : \mathbf{?}A\\
&\qquad \mathcal{I}_P(\theta,\, \Delta,\, P)
\end{aligned}$$

3.2.15 Unfolding Recursive Sessions

Unfold processes are compiled such that positive unfolds (**unfold**$_\mu\, x; P$) always pass control to the opposite endpoint of x, as in the SAM. The rationale behind this is that, given $x : A$, if A is negative, i.e., the next action on x is a read, control must be passed back to the other endpoint so that it can write the data it is read. If A is positive, then the next action on x is a write, and control must be passed to the other endpoint so that it can read any data which had already been written before the unfold.

Negative unfolds (**unfold**$_\nu\, x; P$) only pass control to their continuation if A is negative, as in this case, the positive endpoint must have already executed a positive unfold, and thus, given control to this side. Since the positive side must execute first, control must be surrendered back to the opposite endpoint.

Otherwise, if the remainder type is positive, we can continue executing immediately.

In both cases, the process finishes by recursing on the right-hand-side P of the unfold, with a new environment where x is bound to a new channel state equal to its previous state, but with offset 0, as a new segment of the session type starts. The compilation definitions of both processes are shown below:

```
$\mathcal{E}(\theta,\ \textbf{unfold}_\mu\, x; P) \equiv \mathcal{E}(\theta,\ P)$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{unfold}_\mu\, x; P) \equiv$
        continue($\Delta_c(x)$, cont)
    cont:
        $\mathcal{I}_P(\theta,\ \Delta[x \mapsto (\Delta_c(x), \Delta_d(x), 0)],\ P)$
```

```
$\mathcal{E}(\theta,\ \textbf{unfold}_\nu\, x; P) \equiv \mathcal{E}(\theta,\ P)$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{unfold}_\nu\, x; P) \equiv$
        % $x : A$ in $P$
        continueIfNegative($\theta, \Delta, x, A$)
        $\mathcal{I}_P(\theta,\ \Delta[x \mapsto (\Delta_c(x), \Delta_d(x), 0)],\ P)$
```

3.2.16 Receiving and Sending Types

Polymorphism in the IR is modeled through process type parameters. Thus, to compile **recvty** $x(X); P$, a new IR process implementing P must be called in which X is a type parameter. This mechanism is similar to the one used for exponentials, but in this case, the new process may have free linear names which must be passed as arguments. Instead of just one, two IR processes are generated: one for when X is negative, another for when X is positive, identified by id^- and id^+ respectively.

The information sent by **sendty** and received by **recvty** consists of three slots [type; cont; tag], where type holds the type parameter information for X, cont holds a continuation to be used for the remainder of the session, and tag indicates the polarity of X. Thus, the **recvty** process is compiled as a branch on the received tag, calling either id^- or id^+ depending on the tag value.

```
$\mathcal{E}(\theta,\ \textbf{recvty}\, x(X); P) \equiv 1$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{recvty}\, x(X); P) \equiv$
        branchTag($\Delta_r(x) \oplus$ [type; cont] $\sim$ [tag] , neg:1, pos:1)
    neg:
        callProcess($id^-$, exit point, $T, C, D$)
    pos:
        callProcess($id^+$, exit point, $T, C, D$)
```

Type parameters T are obtained from the current type environment θ, with an extra entry for X, read from $\Delta_r(x)$. Continuation arguments and data arguments C and D are generated in the same way as for process calls, from the free names in P and their state in Δ. The name x on the right-hand-side P has its continuation replaced by the continuation at $\Delta_r(x) \oplus$ [type] $\sim$ [cont; tag], and its offset reset to 0.

On the other hand, the **sendty** $x(A); P$ process is compiled in a more direct manner, as shown below. First, a new continuation c is initialized with `cont` as return address, and writing address pointing to a new data section d. The type information of A is written to x, along with c and a tag indicating A's polarity. The old continuation of x is finished, returning control to the other endpoint of x.

```
$\mathcal{E}(\theta,\ \textbf{sendty}\, x(A); P) \equiv \mathcal{E}(\theta,\ P)$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{sendty}\, x(A); P) \equiv$
    % $c \leftarrow \mathsf{new_c}(), \texttt{cont} \leftarrow \mathsf{new_b}(), d \leftarrow \mathsf{new_d}(\mathcal{S}_L(\theta,\ B))$, where $x : B$ in $P$
    initializeContinuation($c$, cont, $d$)
    writeType($\Delta_w(x), \mathcal{S}(\theta,\ A)$)
    writeContinuation($\Delta_w(x) \oplus [\mathsf{type}] \sim [\mathsf{cont}; \mathsf{tag}], c$)
    writeTag($\Delta_w(x) \oplus [\mathsf{type}; \mathsf{cont}] \sim [\mathsf{tag}], p$)   % where $p = 0$ if $\theta_p(A) = -$ or $1$ otherwise
    finish($\Delta_c(x)$)
  cont:
    $\mathcal{I}_P(\theta,\ \Delta[x \mapsto (c, d, 0)],\ P)$
```

3.2.17 Affine, Use and Discard

The compilation of the **affine** $x; P$ process, shown below, implements the affine protocol mentioned in Section 3.1.5: a tag indicating whether x is being used or discarded is received on x. If x is being used, the execution continues to P. If x is discarded, any used shared state cells and co-affine data free in P are also discarded and x is finished, returning control to the endpoint which discarded x.

```
$\mathcal{E}(\theta,\ \textbf{affine}\, x; P) \equiv \mathcal{E}(\theta,\ P)$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{affine}\, x; P) \equiv$
    branchTag($\Delta_d(x)$, use:$\mathcal{E}(\theta,\ P)$, discard:1)
  use:
    $\mathcal{I}_P(\theta,\ \Delta[x \overset{-,\theta}{\oplus} ([\mathsf{tag}],\ A)],\ P)$   % where $x : A$ in $P$
  discard:
    $\mathcal{I}_P(\theta,\ \Delta,\ \textbf{release}\, y_i)$   % for each free usage name $y_i$ in $P$
    $\mathcal{I}_P(\theta,\ \Delta,\ \textbf{discard}\, y_i)$   % for each free co-affine name $y_i$ in $P$
    finish($\Delta_c(x)$, exit point)
```

The process **discard** x writes a discard tag to the writing location of x and passes control to it. When control returns, the current task is finished, as the **discard** process has no remainder process. However, if this discard was generated as part of the discard branch of an affine process, the `popTask` instruction is not generated, as control must return to the continuation of the affine process.

```
$\mathcal{E}(\theta,\ \textbf{discard}\, x) \equiv 1$
$\mathcal{I}_P(\theta,\ \Delta,\ \textbf{discard}\, x) \equiv$
    writeTag($\Delta_w(x), 1$)   % indicating discard
    continue($\Delta_c(x)$, cont)
  cont:
    popTask(exit point)   % unless part of an affine discard branch
```

Finally, the **use** $x; P$ process is compiled as shown below. A tag is written to the remote data of channel x indicating that the session is being used. Then, if the continuation is negative, the control is passed to the other side of the session. Otherwise, the process continues executing immediately.

$$
\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{use}\, x; P) \equiv \mathcal{E}(\theta,\ P)\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{use}\, x; P) \equiv\\
&\qquad \texttt{writeTag}(\Delta_w(x), 0) \quad \text{\% indicating use}\\
&\qquad \texttt{continueIfNegative}(\theta, \Delta, x, A) \quad \text{\% where } x : A \text{ in } P\\
&\qquad \mathcal{I}_P(\theta,\ \Delta[x \overset{+,\theta}{\oplus} ([\text{tag}]\,,\ A)],\ P)
\end{aligned}
$$

3.2.18 Shared State

The process **cell** $x(y.\,P)$ writes a new shared cell to channel x, initialized with a new session y which is used in process P. Similarly to **let** $x\,E$, session x is finished after writing the cell. A continuation c and a data section d are created to implement session y. The written cell is initialized with space for a single cont slot, to which a reference to c is written. Continuation c is initialized with `cont` as its return address, and d as writing location.

$$
\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{cell}\, x(y.\,P)) \equiv \mathcal{E}(\theta,\ P) + 1\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{cell}\, x(y.\,P)) \equiv\\
&\qquad \text{\% } c \leftarrow \text{new}_\text{c}(), \texttt{cont} \leftarrow \text{new}_\text{b}(), d \leftarrow \text{new}_\text{d}(\mathcal{S}_L(\theta,\ A)) \text{ where } y : A \text{ in } P\\
&\qquad \texttt{writeCell}(\Delta_w(x), [\text{cont}])\\
&\qquad \texttt{initializeContinuation}(c, \texttt{cont}, d)\\
&\qquad \texttt{writeContinuation}(c(\Delta_w(x)), c)\\
&\qquad \texttt{finish}(\Delta_c(x), \texttt{exit point})\\
&\quad \texttt{cont:}\\
&\qquad \mathcal{I}_P(\theta,\ \Delta[y \mapsto (c, d, 0)],\ P)
\end{aligned}
$$

The process **take** $x(y); P$ operates on a shared cell received on x. It starts by locking the cell through the `lockCell` instruction. Then, the continuation stored in the cell is read and bound to a new continuation c. The writing location of c is set to a new data section d. Given $y : A$ in P, if A is negative, control must be passed to the other endpoint of y, such that data is written to d.

$$
\begin{aligned}
&\mathcal{E}(\theta,\ \textbf{take}\, x(y); P) \equiv \mathcal{E}(\theta,\ P)\\
&\mathcal{I}_P(\theta,\ \Delta,\ \textbf{take}\, x(y); P) \equiv\\
&\qquad \text{\% } c \leftarrow \text{new}_\text{c}(), d \leftarrow \text{new}_\text{d}(\mathcal{S}_L(\theta,\ A))\\
&\qquad \texttt{lockCell}(\Delta_r(x))\\
&\qquad \texttt{bindContinuation}(c, c(\Delta_r(x)), d)\\
&\qquad \texttt{continueIfNegative}(\theta, \Delta, y, A) \quad \text{\% where } y : A \text{ in } P\\
&\qquad \mathcal{I}_P(\theta,\ \Delta[y \mapsto (c, d, 0)],\ P)
\end{aligned}
$$

The process **put** $x(y.\,P); Q$, similarly to the **cell** process, initializes a new session y. In this case, the existing cell is written to, instead of creating a new cell. After writing to the cell, it is unlocked, and execution continues on process Q.

```
$\mathcal{E}(\theta,\, \textbf{put}\, x(y.\, P); Q) \equiv \mathcal{E}(\theta,\, P) + \mathcal{E}(\theta,\, Q)$
$\mathcal{I}_P(\theta,\, \Delta,\, \textbf{put}\, x(y.\, P); Q) \equiv$
    $\%\ c \leftarrow \mathsf{new_c}(), \texttt{lhs} \leftarrow \mathsf{new_b}(), d \leftarrow \mathsf{new_d}(\mathcal{S}_L(\theta,\, A))$ where $y : A$ in $P$
    $\texttt{initializeContinuation}(c, \texttt{lhs}, d)$
    $\texttt{writeContinuation}(c(\Delta_r(x)), c)$
    $\texttt{unlockCell}(\Delta_r(x))$
    $\mathcal{I}_P(\theta,\, \Delta,\, Q)$
  lhs:
    $\mathcal{I}_P(\theta,\, \Delta[y \mapsto (c, d, 0)],\, P)$
```

The process **release** x decrements the reference count of the cell at session x through `releaseCell`. The responsibility of correctly dropping the cell's contents when the reference count reaches zero is abstracted away by the IR. This process finishes the current task, similarly to **discard** x.

```
$\mathcal{E}(\theta,\, \textbf{release}\, x) \equiv 1$
$\mathcal{I}_P(\theta,\, \Delta,\, \textbf{release}\, x) \equiv$
    $\texttt{releaseCell}(\Delta_r(x))$
    $\texttt{popTask}(\texttt{exit point})$   % unless part of an affine discard branch
```

Finally, the **share** $x\, \{P \,||\, Q\}$ process shares a cell x between P and Q, which are executed concurrently. Since the cell is going to be shared, its reference count is incremented through `acquireCell`. If concurrency is enabled, a new thread is launched to execute the right-hand-side Q of the share process, as shown below. If concurrency is disabled, the right-hand-side is instead added as a new task to the task queue, which will be executed when the left-hand-side finishes.

```
$\mathcal{E}(\theta,\, \textbf{share}\, x\, \{P \,||\, Q\}) \equiv \mathcal{E}(\theta,\, P) + \mathcal{E}(\theta,\, Q)$
$\mathcal{I}_P(\theta,\, \Delta,\, \textbf{share}\, x\, \{P \,||\, Q\}) \equiv$
    $\texttt{acquireCell}(\Delta_r(x))$
    $\texttt{launchThread}(\texttt{rhs})$
    $\mathcal{I}_P(\theta,\, \Delta,\, P)$
  rhs:
    $\mathcal{I}_P(\theta,\, \Delta,\, Q)$
```

3.3 Translating the IR to C

The last phase of the compilation pipeline is the translation from the IR to C. This phase takes as input the IR produced by the compilation scheme described in the previous section, and produces a C program which implements the semantics of the IR. This C program can then be compiled with GCC to produce an executable which runs a computation equivalent to the original CLASS program. The translated code depends only on the standard C library and on `pthread`, if concurrency is enabled. Additionally, it makes extensive use of the GCC extension for computed gotos, used to efficiently implement continuations.

This section starts by describing how the slot trees are translated to concrete data structures in C. Then, an overview of the the C representation of IR environments is given. Next, the code layout of the translated C program is laid out. Finally, the implementation in C of the main instructions of the IR abstract machine is described.

3.3.1 Representing Slot Trees in C

In the IR abstract machine, the concept of data is abstracted away into slots, covered in Section 3.1.3. A slot tree describes the possible sequences of slots which may be stored in a data section. A set of slot trees describes the multiple possible slot trees which may be stored in a data section, only one of them being active at any given time.

When translating IR programs to C code, these slot trees must be translated to concrete, byte-addressed, memory layouts. To this end, a memory layout is defined as a tuple $L = (s, a)$, where s is the size in bytes of the layout, and a is the alignment in bytes the layout must have in memory. Both s and a are not constants, but C expressions which can be evaluated at runtime. This is necessary due to type parameters having layouts known only at runtime.

In order to satisfy the alignment requirements of the various C types used to implement the IR abstract machine, a C macro `ALIGN(s, a)` is defined, which rounds up the size `s` to the nearest multiple of the alignment `a`. This macro is used extensively when computing memory layouts. Additionally, a C macro `MAX(a, b)` is defined, which computes the maximum of two integers `a` and `b`.

Two memory layouts $L_1 = (s_1, a_1), L_2 = (s_2, a_2)$ can be concatenated to form a new layout $L = L_1 + L_2$, where $L = (\texttt{ALIGN}(s_1, s_2) + s_2,\ \texttt{MAX}(a_1, a_2))$, such that L_1 is placed first in memory, followed by L_2, with padding added to ensure that L_2 starts at an address which is a multiple of a_2. Additionally, the maximum of two layouts $L = L_1 \vee L_2$ is defined as $L = (\texttt{MAX}(s_1, s_2),\ \texttt{MAX}(a_1, a_2))$.

The memory layout of a slot S on a given target architecture $\mathcal{T}$ is denoted by $\mathcal{L}_\mathcal{T}(\sigma,\ S)$, defined as

$$
\begin{array}{ll}
\mathcal{L}_\mathcal{T}(\sigma,\ \textsf{int}) = \mathcal{T}(\texttt{int}) & \mathcal{L}_\mathcal{T}(\sigma,\ \textsf{bool}) = \mathcal{T}(\texttt{unsigned char}) \\
\mathcal{L}_\mathcal{T}(\sigma,\ \textsf{string}) = \mathcal{T}(\texttt{char*}) & \mathcal{L}_\mathcal{T}(\sigma,\ \textsf{cont}) = \mathcal{T}(\texttt{struct cont*}) \\
\mathcal{L}_\mathcal{T}(\sigma,\ \textsf{tag}) = \mathcal{T}(\texttt{unsigned char}) & \mathcal{L}_\mathcal{T}(\sigma,\ \textsf{type}) = \mathcal{T}(\texttt{struct type}) \\
\mathcal{L}_\mathcal{T}(\sigma,\ \textsf{exponential}) = \mathcal{T}(\texttt{struct exponential*}) & \mathcal{L}_\mathcal{T}(\sigma,\ \textsf{cell}\,[\ldots]) = \mathcal{T}(\texttt{struct cell*}) \\
\mathcal{L}_\mathcal{T}(\sigma,\ t_i) = \sigma(t_i) &
\end{array}
$$

where σ is a map between type parameters and their layouts. A target architecture $\mathcal{T}$ is a mapping from C types to their memory layouts, such that $\mathcal{T}(\texttt{type}) = (s, a)$ where s and a where, respectively, the size and alignment of `type` on the target architecture. A definition of σ will be introduced in the next section.

The definition of $\mathcal{L}_\mathcal{T}(\sigma,\ s)$ is extended below to support both slot trees, and sets of slot trees. The layout of a slot tree is defined as the concatenation of the layouts of each of its slots, and maximum across any branches, while the layout of a set of slot trees is defined as the maximum layout among all its trees.

$$
\begin{aligned}
&\mathcal{L}_{\mathcal{T}}(\sigma,\, []) = (0, 1) && \mathcal{L}_{\mathcal{T}}(\sigma,\, [S; T]) = \mathcal{L}_{\mathcal{T}}(\sigma,\, S) + \mathcal{L}_{\mathcal{T}}(\sigma,\, T) \\
&\mathcal{L}_{\mathcal{T}}(\sigma,\, \text{tag}\, [T_1 \mid \ldots \mid T_n]) = \bigvee_{i=1}^{n} \mathcal{L}_{\mathcal{T}}(\sigma,\, T_i) && \mathcal{L}_{\mathcal{T}}(\sigma,\, \{T_1, \ldots, T_n\}) = \bigvee_{i=1}^{n} \mathcal{L}_{\mathcal{T}}(\sigma,\, T_i)
\end{aligned}
$$

3.3.2 Environment Memory Layout

IR environments store all state related to a process invocation. In the translated C programs, environments are represented as heap allocated byte buffers. The environment layout $\mathcal{E}$ specifies at compile time how environment buffers will be indexed at runtime.

An environment layout $\mathcal{E}$ is a compile-time data structure which describes the byte offsets at which each entry of the environment can be found. This layout is computed from the header of the IR process being compiled. The address of the exit-point counter (a C integer or atomic integer if concurrency is enabled) is given by $\mathcal{E}_{\text{eps}}(\texttt{env})$ where `env` is a pointer to the environment. Drop bits are stored at $\mathcal{E}_{\text{e}}(\texttt{env})$, type information for type variable t_i is stored at $\mathcal{E}_{\text{ty}}(\texttt{env}, t_i)$, continuations c_i are stored at $\mathcal{E}_{\text{c}}(\texttt{env}, c_i)$, and data sections d_i at $\mathcal{E}_{\text{d}}(\texttt{env}, d_i)$.

The layout of the environment is computed trivially by concatenating the layouts of each of its entries, in the order they appear in the process header. The layout for data sections is the most noteworthy, as it depends on the stored data type. This layout is computed from $\mathcal{L}_{\mathcal{T}}(\sigma,\, S)$, where S is the set of slot trees of the data section, and $\sigma(t_i) = \texttt{access}(\mathcal{E}_{\text{ty}}(\texttt{env}, t_i))$ is a mapping between type parameters and their layouts, fetched from the environment itself.

Continuations are stored in environments as instances of the C type `struct cont`, shown below. Continuations store a return address to jump to (`loc`), a pointer to the environment to switch to before jumping (`env`), a pointer to the writing location (`write`), and a pointer to the continuation pertaining to the other endpoint of the session (`remote`).

```c
struct cont {
  void* loc;
  char* env;
  char* write;
  struct cont* remote;
};
```

Type information is stored in the environment as instances of the C type `struct type`, shown below. This structure holds the `size` and `alignment` of the data described by the type. This structure is also used to store the value of type slots in the data sections.

```c
struct type {
    int size;
    int alignment;
};
```

3.3.2.A Environment Cleanup

Whenever an instruction decrements the exit-point counter, it checks if it reached zero. If so, any drop bits set in the environment are processed, dropping their associated data. Then, the environment itself is freed. The following C snippet illustrates this cleanup code, where `env` points to the environment.

```c
if (atomic_fetch_sub(&(*(atomic_int*)(env)), 1) == 1) {
    if ((*(unsigned char*)(env + 4)) & (1 << 0)) {
        /* drop data associated to drop bit 0 */
    }
    /* other drop bit checks... */
    managed_free(env);
}
```

3.3.3 Execution and Registers

The `main` function of the generated C program initializes any required global state, such as the allocator, or profiling tools, and invokes a function named `executor` with a `NULL` argument. The `executor` function is the core of the IR abstract machine implementation. Each thread of execution runs its own instance of this function, receiving as argument a pointer to its entry task.

At the start of the `executor` function, registers used to implement the IR instructions are declared. The first two registers `task` and `env` respectively store pointers to the head of the task stack (introduced in the next subsection) and the current environment, which is state kept across IR instructions. The rest of the registers are used to store temporary values used within the execution of each instruction. After declaring the registers, the thread state is initialized.

```c
void executor(struct task* entry) {
    /* Registers */
    register struct task* task;
    register char* env;
    register void* tmp_ptr1;
    register void* tmp_ptr2;
    register int tmp_int;
    struct cont tmp_cont;
    /* Initialize thread state (...) */
```

After initializing the thread state, control is passed to the entry task (received as argument), which contains a continuation address and environment. If the entry task is `NULL`, the main process is called instead, as shown below. To call the main process, a new environment is allocated with the size computed from its header, and a jump is performed to its entry block.

```c
    if (entry == NULL) {
        /* callProcess(main) */
        tmp_ptr1 = managed_alloc(144);
        env = ((char*)(tmp_ptr1));
        goto main_entry;
    } else {
        env = entry->env;
        tmp_ptr1 = entry->loc;
        managed_free(entry);
        goto *tmp_ptr1;
    }
```

After these instructions, still within the `executor` function, for each block of each process a unique C label is generated, followed by the C code generated from the IR instructions in the block. Finally, a label named `end` is generated, which represents the end of the thread's execution.

```
main_entry:
    /* instructions of main's entry block ... */
foo_cut_lhs_0:
    /* instructions of foo's cut_lhs_0 block ... */
    /* other blocks ... */
end:
    return 0;
}
```

3.3.4 Tasks

Tasks are stored in a linked list local to each thread of execution, where each node is of the type shown below. The `task` register points to the head node of this stack. The `next` field points to the next task to be executed, while the `loc` and `env` fields store the code address and environment to switch to when executing the task.

```
struct task {
    struct task* next;
    void* loc;
    char* env;
};
```

The `task` register is initialized at the start of the `executor` function with a dummy task which jumps to the `end` label, ending the thread's execution. This dummy task is necessary to ensure that the task stack is never empty, simplifying the implementation of the `popTask` instruction.

```
task = managed_alloc(sizeof(struct task));
task->loc = &&end;
```

The two IR instructions for manipulating tasks, `pushTask` and `popTask`, manipulate the linked list of tasks pointed to by the `task` register. The first instruction, `pushTask`, allocates a new task, sets its continuation to the C label of the chosen block, and inserts it at the head of the linked list. The generated C code for this instruction is shown below.

```
/* pushTask(cont) */
tmp_ptr1 = task;
task = managed_alloc(sizeof(struct task));
```

```
task->next = ((struct task*)(tmp_ptr1));
task->loc = &&proc_cont; /* proc_cont is the C label of block 'cont' */
task->env = env;
```

The popTask instruction, whose generation is shown below, removes the first task from the linked list, changes the current environment and jumps to its continuation.

```
/* popTask() */
tmp_ptr1 = task;
tmp_ptr2 = task->loc;
env = task->env;
task = task->next;
managed_free(tmp_ptr1);
goto *tmp_ptr2;
```

3.3.5 Continuation Manipulation

The initContinuation instruction is compiled as shown below. The continuation at $\mathcal{E}_{\mathrm{c}}(\mathtt{env}, c_0)$ (in this example, $\mathtt{env} + 8$) is initialized with the given program label, the current environment, the address $\mathcal{E}_{\mathrm{d}}(\mathtt{env}, d_0)$ (in this case, $\mathtt{env} + 40$) as its writing location data, and a pointer to itself as its remote continuation.

```
/* initContinuation(c0, cut_lhs_0, d0) */
(*(struct cont*)(env + 8)) = (struct cont){
    .loc=&&main_cut_lhs_0,
    .env=env,
    .write=env + 40,
    .remote=env + 8
};
```

The compilation of the finish instruction is shown below. First, the remote continuation of the finished continuation is unlinked from this continuation by setting its remote field to itself. This is necessary to avoid use after free errors in instructions such as forward and callProcess. Then, the target location and environment is stored on the registers and the environment's exit-point counter is decremented atomically (when concurrency is disabled, a normal integer is used instead). If the counter reaches zero, the environment cleanup code described in Section 3.3.2.A is executed. Finally, the environment and code address are switched to the continuation values, and a jump is performed.

```
/* finish(c0, exit point) */
(struct cont*)(env + 8)->remote->remote = (struct cont*)(env + 8)->remote;
tmp_ptr1 = (*(struct cont*)(env + 8)).loc;
tmp_ptr2 = (*(struct cont*)(env + 8)).env;
if (atomic_fetch_sub((atomic_int*)(env), 1) == 1) {
    /* environment cleanup code */
}
env = ((char*)(tmp_ptr2));
goto *tmp_ptr1;
```

The `continue` instruction is compiled similarly to `finish`, as shown below, except that the exit-point counter is not decremented, and that the remote continuation must instead be updated so that it eventually jumps back to the correct label.

```
/* continue(c0, continue_12) */
tmp_ptr1 = (*(struct cont*)(env + 8)).loc;
(*(*(struct cont*)(env + 8)).remote).loc = &&main_continue_12;
env = (*(struct cont*)(env + 8)).env;
goto *tmp_ptr1;
```

The `forward` instruction first stores the continuation of its second argument in the temporary registers. Then, it sets the remotes of both argument continuations to point to each other, ensuring no references are left to the local continuations. A temporary variable is used to store the first argument continuation, as its value may be overwritten due to the possibility of a continuation being its own remote. Finally, the exit-point counter is decremented, and control is passed to the location and environment stored in the temporary registers.

```
/* forward(c0, c1, exit point) */
tmp_ptr1 = (*(struct cont*)(env + 40)).loc;
tmp_ptr2 = (*(struct cont*)(env + 40)).env;
tmp_cont = *(struct cont*)(env + 8);
*(*(struct cont*)(env + 8)).remote = *(struct cont*)(env + 40);
*(*(struct cont*)(env + 40)).remote = tmp_cont;
if (atomic_fetch_sub(*(atomic_int*)(env), 1) == 1) {
    /* environment cleanup code */
}
env = (char*)tmp_ptr2;
goto *tmp_ptr1;
```

3.3.6 Exponentials

Exponentials are reference counted, heap allocated structures, represented in C by the type shown below. Each exponential stores stores a template for the environments to be created when the exponential is called. The `entry_cont_offset` field stores the byte offset the argument continuation is stored at on the environment template, while the `entry_data_offset` field stores the byte offset the argument data is stored at. The `env_size` field stores the size in bytes of the environment template, while the `manager` field stores a code location, known as the exponential's manager, which is responsible for cloning and dropping the exponential's template environment. Finally, the flexible array member `env` stores the actual environment template data.

```
struct exponential {
    atomic_int ref_count;
    int entry_cont_offset;
    int entry_data_offset;
    int env_size;
    void* manager;
    char env[];
};
```

The `writeExponential` instruction is compiled as shown below. First, a new exponential structure is allocated with enough space to store the environment template, and its fields are initialized. The reference count is initialized to 1, the offsets and size are computed from the environment layout of the exponential process, and the manager is set to a new unique C label. The environment template is initialized based on the exponential process header. The argument continuation is initialized to point to the entry block of the exponential process, and any argument types or data are written.

```
/* writeExponential(d0, main_exp0) */
*(struct exponential**)(env + 64) = managed_alloc(60);
(*(struct exponential**)(env + 64))->ref_count = 1;
(*(struct exponential**)(env + 64))->entry_cont_offset = 0;
(*(struct exponential**)(env + 64))->entry_data_offset = 32;
(*(struct exponential**)(env + 64))->env_size = 36;
(*(struct exponential**)(env + 64))->manager = &&main_exp0_exp_manager;
tmp_ptr1 = (*(struct exponential**)(env + 64))->env;
(*(struct cont*)(((char*)(tmp_ptr1)) + 0)).loc = &&main_exp0_entry;
// (...) write argument types and data to environment tmp_ptr1 ...
```

The exponential manager is a block of code which when jumped to, branches on the value of `tmp_int`, which is set to 0 for cloning the exponential, and to 1 for dropping it. If cloning, the current environment is assumed to be the newly allocated environment for the exponential call, while `tmp_ptr1` is assumed to be the environment template stored in the exponential. If dropping, the current environment is assumed to be the environment template stored in the exponential. The cloning branch clones the argument data from the environment template into the new environment, while the dropping branch drops the argument data in the environment template. Finally, in both cases, control is passed to the argument continuation stored in the exponential environment.

```
main_exp0_exp_manager:
    if (tmp_int == 0) {
        /* (...) clone argument data from environment tmp_ptr1 to env */
    } else {
        /* (...) drop argument data in environment env */
    }
    tmp_ptr2 = (*(struct cont*)(env + 0)).loc;
    env = (*(struct cont*)(env + 0)).env;
    goto *tmp_ptr2;
```

The `callExponential` instruction, compiled as shown below, first allocates a new environment for the exponential call with the size stored in the exponential structure. This environment is initialized through a memory copy from the exponential's template environment, and is set as the environment of the produced continuation. Then, the produced continuation and data is connected to the argument continuation in the new environment, and finally the exponential manager is jumped to, with `tmp_int` set to 0 to indicate a clone operation. When finished, control is passed back to the label `main_exp_call_return_0`.

```
    /* callExponential(d0, c0, d1) */
#define exp (*(struct exponential**)(env + 64))
#define exp_env (*(struct cont*)(env + 0)).env
#define exp_session (*(struct cont*)(exp_env + exp->entry_cont_offset))
```

```c
    exp_env = managed_alloc(exp->env_size);
    memcpy(exp_env, exp->env, exp->env_size);
    (*(struct cont*)(env + 0)).loc = exp_session.loc;
    ((env + 0)).write = exp_env + exp->entry_data_offset;
    (*(struct cont*)(env + 0)).remote = &exp_session;
    exp_session.env = env;
    exp_session.write = env + 72;
    exp_session.remote = env;
    exp_session.loc = &&main_exp_call_return_0;
    tmp_ptr1 = exp->env;
    tmp_ptr2 = exp->manager;
    tmp_int = 0;
    env = exp_env;
    goto *tmp_ptr2;
main_exp_call_return_0:
```

When the reference count of an exponential reaches zero, the control is passed to its manager with `tmp_int` set to 1, indicating a drop operation, and the current environment set to the exponential's template environment. Control is then returned to the drop caller and the exponential structure is freed.

3.3.7 Data Manipulation

The `moveSlots`, `cloneSlots` and `dropSlots` instructions are compiled using a *slot traversal* utility function, which recursively generates C code to visit each slot address given a slot tree and a base address. This is necessary due to the complexity of slots trees, which may branch both due to the presence of tags (CLASS choices) and due to polymorphic types.

For example, to move, clone or drop a slot tree $T =$ tag [[string] | [exponential]], different code must be generated depending on whether the tag indicates that the first or second branch is active. Cloning a string slot implies allocating a new string and copying the original string, while cloning an exponential slot implies incrementing the reference count of the exponential value. Below is the generated C code for the `cloneSlots` instruction, operating on the slot tree T described above, assuming d_1 is at $\texttt{env} + 32$, both d_1.tag~string and d_1.tag~exponential are at $\texttt{env} + 40$, d_0 at $\texttt{env} + 48$ and both d_0.tag~string and d_0.tag~exponential are at $\texttt{env} + 56$.

```c
/* cloneSlots(d1, d0, tag[string | exponential]) */
*(unsigned char*)(env + 32) = *(unsigned char*)(env + 48);
switch (*(unsigned char*)(env + 48)) {
    case 0: /* string */
        *(char**)(env + 40) = *(char**)(env + 56);
        *(char**)(env + 40) = string_clone(*(char**)(env + 40));
        break;
    case 1: /* exponential */
        *(struct exponential**)(env + 40) = *(struct exponential**)(env + 56);
        /* ... increment reference count */
        break;
}
```

Moves of slots s are always generated as simple memory copies of size given by $\mathcal{L}_{\mathcal{T}}(\sigma, s)$. Clones and drops, on the other hand, are generated by pattern matching on the slot s, generating the appropriate code for each case. Cloning and dropping basic types such as int, bool and tag slots is a no-op, while cloning and dropping other slots may involve memory (de)allocation and reference count manipulation, as shown in the example above.

To clone or drop type parameter slots t_i, type information for each type parameter t_i is stored alongside the environment buffer, describing, at run-time, the slots which must be cloned or dropped within the type parameter. This type information is generated when a new environment is created, or when type information is stored using the `writeType` instruction.

3.3.8 Process Calls

Process calls are performed through the `callProcess` instruction. This instruction receives the identifier of the process to be called, a list of type parameters, a list of continuation arguments and a list of data arguments. The generated C code for this instruction is shown below.

```c
/* callProcess(foo, exit point, c0 <- c0) */
tmp_ptr1 = managed_alloc(40);
(*(atomic_int*)(((char*)(tmp_ptr1)))) = 1; // setup exit points
/* (...) pass type, continuation and data arguments */
if (atomic_fetch_sub(&(*(atomic_int*)(env)), 1) == 1) {
    /* environment cleanup code */
}
env = ((char*)(tmp_ptr1));
goto foo_entry;
```

Type parameters are passed by writing their information to the appropriate offsets in the new environment. Continuation arguments are passed by initializing the continuations in the new environment, and connecting them to the local continuations passed as arguments. This implies updating the remote of the argument continuations to point to the continuations in the new environment. Finally, data arguments are passed by simply moving or cloning the data from the current environment to the new environment, in the same way as the `moveSlots` and `cloneSlots` instructions.

Before jumping to the entry block of the called process, the exit-point counter of the current environment is decremented, and if it reaches zero, the current environment is cleaned up and freed, as described in Section 3.3.2.A. Control is only then passed to the entry block of the called process.

3.3.9 Cell Operations

Cells are implemented in C as heap-allocated instances of the type shown below. The first three fields, `mutex`, `cond_var` and `is_free`, are used to implement the locking mechanism of the cell. The `ref_count` field is an atomic integer which stores the reference count of the cell, while the flexible array member `data` stores the actual contents of the cell, similarly to a data section in an environment.

```c
struct cell {
    pthread_mutex_t mutex;
    pthread_cond_t cond_var;
    unsigned char is_free;
    atomic_int ref_count;
    char[] data;
};
```

The `writeCell` instruction is simply compiled as the allocation of a new `struct cell` with enough space to store its data section, and the initialization of its fields. The `acquireCell` and `releaseCell` instructions are compiled respectively as the atomic increment and decrement of the cell's reference count. If the reference count reaches zero, the cell's data is dropped, the cell's mutex and condition variable are destroyed, and finally the cell itself is freed.

The cell locking/unlocking operations cannot be implemented with a simple mutex lock/unlock pair, as it is possible for a locked cell to be passed to another thread, which must then be able to unlock it. This violates the requirement that `pthread` mutexes can only be unlocked by the thread which locked them. To workaround this, the `is_free` flag indicates if the cell is locked or not, and the mutex and condition variable are used solely to synchronize access to this flag. The `lockCell` instruction thus compiled as shown below. First, the cell's mutex is locked. Then, while the cell is not free, the thread waits on the cell's condition variable, atomically releasing the mutex while waiting, and re-acquiring it when woken up. When the cell is free, it is marked as not free, and the mutex is unlocked.

```c
/* lockCell(d0) */
pthread_mutex_lock(&((*(struct cell**)(env + 16))->mutex));
while (!(*(struct cell**)(env + 16))->is_free) {
    pthread_cond_wait(&(*(struct cell**)(env + 16)->cond_var),
                      &(*(struct cell**)(env + 16))->mutex);
}
(*(struct cell**)(env + 16))->is_free = 0;
pthread_mutex_unlock(&((*(struct cell**)(env + 16))->mutex));
```

The `unlockCell`, shown below, locks the cell's mutex, marks the cell as free, signals one thread waiting on the cell's condition variable (if any), and finally unlocks the mutex.

```c
/* unlockCell(d0) */
pthread_mutex_lock(&(*(struct cell**)(env + 16))->mutex);
(*(struct cell**)(env + 0))->is_free = 1;
pthread_cond_signal(&(*(struct cell**)(env + 16))->cond_var);
pthread_mutex_unlock(&(*(struct cell**)(env + 16))->mutex);
```

4 Optimizing the Generated Code

Contents

During the course of this project, several optimizations on various parts of the compiler were implemented to produce more efficient code. Code transformations such as *inlining* and polymorphic process *monomorphization* were implemented with great results, but are not described in this chapter, as they are well-known techniques. Instead, this chapter focuses on optimizations specific to the language and its intermediate representation.

The first section goes over the transformations made to the compiler to produce more efficient code when handling types with certain properties. The second section describes optimizations specific to the C code generation phase, focused on simplifying the counting of exit points and tail calls. Finally, the last section introduces a new compilation phase which applies a series of transformations to the generated IR code before passing it to the C code generation phase.

$$
\begin{array}{ll}
\mathcal{V}_\theta(\textbf{lint}) = \mathbf{guaranteed} & \mathcal{V}_\theta(\textbf{lbool}) = \mathbf{guaranteed} \\
\mathcal{V}_\theta(\textbf{lstring}) = \mathbf{guaranteed} & \mathcal{V}_\theta(\textbf{close}) = \mathbf{guaranteed} \\
\mathcal{V}_\theta(\textbf{send}\, A; B) = \mathcal{V}_\theta(B) & \mathcal{V}_\theta(\textbf{sendty}\, X; B) = \mathbf{guaranteed} \\
\mathcal{V}_\theta(\textbf{!}A) = \mathbf{guaranteed} & \mathcal{V}_\theta(\textbf{affine}\, A) = \mathbf{impossible} \\
\mathcal{V}_\theta(\textbf{state}\, A) = \mathbf{guaranteed} & \mathcal{V}_\theta(\textbf{statel}\, A) = \mathbf{guaranteed} \\
\mathcal{V}_\theta(\textbf{choice of}\, \{|\textbf{\#a} : A\ (\dots)\ |\textbf{\#z} : Z\}) = \bigwedge(\mathcal{V}_\theta(A), \dots, \mathcal{V}_\theta(Z)) & \mathcal{V}_\theta(X) = \mathbf{dependent}(\mathcal{V}(\theta_t(X))) \\
\mathcal{V}_\theta(A) = \mathbf{impossible} \text{ if } \theta_p(A) = - &
\end{array}
$$

Figure 4.1: Definition of $\mathcal{V}_\theta(A)$ for positive types

4.1 Making Use of Type Properties

The compiler presented in Chapter 3 does not take any advantage of the fact that many types in the language can be handled more efficiently than the general case. For example, the type **send lint**; **lint** is stored as a reference to a continuation (essentially a closure) followed by an integer. The closure producing the **lint** will eventually be executed, producing a single integer and immediately terminating. Instead of sending this continuation, the integer could be computed immediately and sent directly, saving memory and time, as the pointer indirection would be removed, along with some unnecessary jumps.

This section describes the implemented optimizations which take advantage of type properties such as these to generate more efficient code. In the first subsection, three type properties are defined, which determine which optimizations can be applied to them. Then, the second subsection describes the necessary compiler adaptations to classify whether a type has these properties. The last three subsections describe the transformations made using these properties.

4.1.1 Value, Droppable and Cloneable Types

In the optimizations described in this section, types are optimized based on whether they're *value*, *droppable* or *cloneable* types. These properties are referred to as *type flags*. A type flag of a given type may either be $\mathbf{guaranteed}$ to hold, $\mathbf{impossible}$ to hold, or be dependent on whether some type parameters have some type flags set, which is denoted as $\mathbf{dependent}(F_1, \dots, F_n)$, where each F_i is either $\mathcal{V}(t_j)$, $\mathcal{D}(t_j)$ or $\mathcal{C}(t_j)$ (value, droppable and cloneable flags, respectively), and t_j is a type variable. These requirements are referred to as type flag requirements. The union of two type flag requirements R_1 and R_2, denoted as $R_1 \wedge R_2$, is defined as follows:

$$R_1 \wedge R_2 = R_2 \wedge R_1 \qquad R \wedge \mathbf{impossible} = \mathbf{impossible} \qquad R \wedge \mathbf{guaranteed} = R$$

$$\mathbf{dependent}(F_1, \dots, F_n) \wedge \mathbf{dependent}(G_1, \dots, G_m) = \mathbf{dependent}(F_1, \dots, F_n, G_1, \dots, G_m)$$

$$\begin{array}{ll}
\mathcal{D}_\theta(\textbf{lint}) = \textbf{guaranteed} & \mathcal{D}_\theta(\textbf{lbool}) = \textbf{guaranteed} \\
\mathcal{D}_\theta(\textbf{lstring}) = \textbf{guaranteed} & \mathcal{D}_\theta(\textbf{close}) = \textbf{guaranteed} \\
\mathcal{D}_\theta(\textbf{send}\ A; B) = \mathcal{D}_\theta(A) \land \mathcal{D}_\theta(B) & \mathcal{D}_\theta(\textbf{sendty}\ X; B) = \textbf{impossible} \\
\mathcal{D}_\theta(\textbf{!}A) = \textbf{guaranteed} & \mathcal{D}_\theta(\textbf{affine}\ A) = \textbf{guaranteed} \\
\mathcal{D}_\theta(\textbf{state}\ A) = \textbf{guaranteed} & \mathcal{D}_\theta(\textbf{statel}\ A) = \textbf{impossible} \\
\mathcal{D}_\theta(\textbf{choice of}\ \{|\textbf{\#a} : A\ (\dots)\ |\textbf{\#z} : Z\}) = \bigwedge(\mathcal{D}_\theta(A), \dots, \mathcal{D}_\theta(Z)) & \mathcal{D}_\theta(X) = \textbf{dependent}(\mathcal{D}(\theta_t(X))) \\
\mathcal{D}_\theta(A) = \textbf{impossible} \text{ if } \theta_p(A) = - &
\end{array}$$

Figure 4.2: Definition of $\mathcal{D}_\theta(A)$ for positive types

$$\begin{array}{ll}
\mathcal{C}_\theta(\textbf{lint}) = \textbf{guaranteed} & \mathcal{C}_\theta(\textbf{lbool}) = \textbf{guaranteed} \\
\mathcal{C}_\theta(\textbf{lstring}) = \textbf{guaranteed} & \mathcal{C}_\theta(\textbf{close}) = \textbf{guaranteed} \\
\mathcal{C}_\theta(\textbf{send}\ A; B) = \textbf{impossible} & \mathcal{C}_\theta(\textbf{sendty}\ X; B) = \textbf{impossible} \\
\mathcal{C}_\theta(\textbf{!}A) = \textbf{guaranteed} & \mathcal{C}_\theta(\textbf{affine}\ A) = \textbf{impossible} \\
\mathcal{C}_\theta(\textbf{state}\ A) = \textbf{impossible} & \mathcal{C}_\theta(\textbf{statel}\ A) = \textbf{impossible} \\
\mathcal{C}_\theta(\textbf{choice of}\ \{|\textbf{\#a} : A\ (\dots)\ |\textbf{\#z} : Z\}) = \bigwedge(\mathcal{C}_\theta(A), \dots, \mathcal{C}_\theta(Z)) & \mathcal{C}_\theta(X) = \textbf{dependent}(\mathcal{C}(\theta_t(X))) \\
\mathcal{C}_\theta(A) = \textbf{impossible} \text{ if } \theta_p(A) = - &
\end{array}$$

Figure 4.3: Definition of $\mathcal{C}_\theta(A)$ for positive types

A type is said to be a value type if it has a strict polarity, being either strictly positive, or strictly negative. A channel of a negative value type will only be read from, and control will not be yielded to it. A channel of a positive value type will only be written to, and control will not return from it. For example, **lint** and **send** $A;$ **close** are both positive value types. A type A is only a value if and only if its type flag requisites are met, which are denoted as $\mathcal{V}_\theta(A)$, defined for positive types in Figure 4.1, where θ is the type environment introduced in Section 3.2.1.A. For negative types, $\mathcal{V}_\theta(A) = \mathcal{V}_\theta(\sim A)$.

A type is said to be droppable if a drop function is defined for it, effectively making it affine. A type A is droppable if and only if its type flag requisites are met, denoted as $\mathcal{D}_\theta(A)$ and defined for positive types in Figure 4.2. For negative types, $\mathcal{D}_\theta(A) = \mathcal{D}_\theta(\sim A)$. For example, **lint** can be dropped trivially, no actions required, and a channel of type **send** $A; B$ can only be dropped if both A and B can be dropped.

Finally, a type is said to be cloneable if a clone operation is defined for it. The type flag requisites for type A to be cloneable are denoted as $\mathcal{C}_\theta(A)$, and their definition for positive types is shown in Figure 4.3. For negative types, $\mathcal{C}_\theta(A) = \mathcal{C}_\theta(\sim A)$. For example, **!**$A$ is a cloneable type, as exponentials can be cloned by incrementing their reference count.

4.1.2 Adding Type Flags to the Compiler

To generate code which depends on type flags, it is necessary to add a new branching instruction `branchFlags`, with the exact same syntax as `branchExpr`, but instead of branching on a boolean expression, it branches on whether some flag requirements are met. `branchFlags(if(t0.value), value:1, not_value:1)`, for example, jumps to block `value` if the type parameter t_0 has the value flag set, or to block `not_value` otherwise. Most flag requisites are either $\mathbf{guaranteed}$ or $\mathbf{impossible}$, and thus can be determined at compile time, avoiding the branch.

To enable different memory representations for session types based on these flags, slot trees also need to be extended with a new definition $\mathsf{flags}\,\langle R\rangle\,[T_1 \mid T_2]$. This tree is similar to $\mathsf{tag}\,[\ldots]$, but instead of branching on a written tag, it branches on whether some flag requirements R are met - if R is met, T_1 is chosen, otherwise, T_2 is chosen.

To branch on the flags of a type parameter, the flags must be stored in environments, along with the rest of the type parameter information. This is done by adding a new integer field to the `type` C struct, which stores one bit for each of the type flags. Additionally, all instructions which pass around type information must be extended to support these type flags. For example, `writeType` must receive the flag requirements for each of the flags.

4.1.3 Sending Values instead of Continuations

The first implemented optimization takes advantage of the fact that a session of a positive value type A being sent (e.g., $\mathbf{send}\,A; B$) is a session which will have control returned to it exactly once, when it is received. The total number of operations can be reduced by immediately writing the data of the value instead of a continuation which would write that data. For example, instead of $\mathbf{send\,lint}; \mathbf{send\,lint}; \mathbf{lint}$ being represented as $[\mathsf{cont}; \mathsf{cont}; \mathsf{int}]$, it can be represented as $[\mathsf{int}; \mathsf{int}; \mathsf{int}]$.

First, the mapping from sessions types A to their respective slot trees introduced in Section 3.2.2 is updated such that, for the $\mathbf{send}\,A; B$ type, if the left-hand-side A type is a positive value type, the slots of the A are computed, and concatenated with the slots of the right-hand-side B. Otherwise, as before, a cont slot is simply added before the slots of B. Formally:

$$\mathcal{S}(\theta,\, \mathbf{send}\,A; B) = \mathsf{flags}\,\langle \mathcal{V}_\theta(A)\rangle\,[\mathcal{S}(\theta,\, A) \mid [\mathsf{cont}]] \oplus \mathcal{S}(\theta,\, B)$$

Then, the compilation of both $\mathbf{send}$ and $\mathbf{recv}$ processes must be modified to reflect this new structure. A `branchFlags` instruction is generated which branches on whether the type of the argument session being sent or received is a positive or negative value, respectively. If it is not a value, the implementation is the same as before.

On the sending side, if the argument is a positive value type, the argument's continuation is initialized so that it writes directly into the main continuation's writing location. Then, the send's argument code is immediately executed, writing the data to the main session, through the argument session. When it finishes, the right-hand-side of the send executes, writing data after the data that has just been written.

On the receiver side, instead of receiving a continuation, the data is simply moved from the main session's data section using `moveSlots` into the argument's data section.

4.1.4 Eagerly Computing Exponentials

The second implemented optimization greatly simplifies the representation of exponential types when their inner type is a cloneable value type. Instead of representing **!**A as an exponential slot, the value can simply be computed immediately and stored directly. Without this optimization, for example, storing the result of a complex computation as an exponential would require re-executing the entire computation every time the exponential is called. With this change, the computation is executed only once, and the result is cloned whenever a call is made. When the exponential is dropped, the result value is dropped.

This optimization changes the semantics of exponentials, as the computation is done immediately and only once, instead of once per exponential call. This transformation may end up worsening performance in cases where the exponential is never called. However, in practice, when exponentials are used, they are often called multiple times, and the performance gain in these cases is significant.

Once again, the mapping from session types to slot trees is modified. If the inner type A of a **!**A type is a cloneable and droppable type, the slots of A are computed and used directly. Otherwise, a single exponential slot is used, as before. Formally:

$$\mathcal{S}(\theta,\ \mathbf{!}A) = \text{flags}\ \langle \mathcal{C}_\theta(A) \wedge \mathcal{D}_\theta(A) \rangle\ [\mathcal{S}(\theta,\ A)\ |\ [\text{exponential}]]$$

This change turns, for example, the representation of **!!int** into just int, instead of an exponential which would produce another exponential which would finally produce an int.

The **!** $x(y); P$ process, just like the **send** and **recv** processes, is modified to generate a `branchFlags` instruction which branches on whether the inner type A is a cloneable and droppable type. If it is not, the implementation is the same as before. Otherwise, the inner process P is executed immediately, with the name y bound as an alias of x, and the data is written directly where the exponential would be stored.

On the **?** $x; P$ process, nothing changes. The drop of the slots corresponding to the exponential are deferred, as before. Previously these slots would always be a single exponential slot, but now they might be the slots of the inner type A, if A is a cloneable and droppable type.

Finally, the **call** $x(y);$ process is modified to simply generate a `cloneSlots` instruction whenever the inner type A is a cloneable and droppable type, instead of generating a `callExponential` instruction.

4.1.5 Eagerly Computing Affine Sessions

On the base compiler, sessions of type **affine** A are implemented as following the affine protocol, i.e., **offer of** $\{$|**#use** : A |**#discard** : **close**$\}$. Although this implementation saves time by lazily executing the affine code only if the session is used, it has the overhead of storing a continuation instead of the actual value, requiring multiple jumps and a tag to determine whether the session is used or discarded, even if the inner type A is a droppable type, e.g., **affine lint** or **affine !lint**. In these cases, the semantic could be changed so that the value is computed immediately, and dropped if the session is discarded.

This of course, does not always improve performance. While exponentials are very likely to be used multiple times, affine channels are often used to prevent computing a value which is never used. With this transformation, time may be wasted performing an unnecessary expensive computation.

The biggest benefit of this change appears on the shared state types, such as **state** A. On the base implementation, this type is represented as a cell [cont] slot, due to cell contents being affine, which effectively means all cells store continuations instead of concrete values. With this optimization, if the inner type A of a **state** A type is a droppable type, such as another cell, an exponential, or a basic type such as **lint**, the cell can store the value directly, as a cell $[A]$ slot. The drawbacks for cells are not as big as for affine types, as most values stored in cells, in practice, end up being used - only the last value written to a cell is discarded.

One difficult aspect of this optimization is that, unlike the previous two optimizations, the polarity of the affected **affine** A type ends up being dependent on whether the inner type A is a droppable type or not. If A is not droppable, the polarity is negative, as in the base implementation. However if A is a positive droppable type, the polarity becomes positive, as the **affine** process will write the value immediately. This problem was tacked by defining the polarity $\theta_p(\textbf{affine}\,A)$ as always being positive, and negative for the **coaffine** A type. In order to implement the write-to-read adjustment for **affine** A types, the compilation of the **affine** process is modified to always generate a `continue` instruction for non-droppable inner types before branching on the use-discard choice.

The mapping from session types to slot trees is modified as follows:

$$\mathcal{S}(\theta,\, \textbf{affine}\,A) = \text{flags}\,\langle \mathcal{D}_\theta(A)\rangle\,[\mathcal{S}(\theta,\,A) \mid []]$$
$$\mathcal{S}(\theta,\, \textbf{coaffine}\,A) = \text{flags}\,\langle \mathcal{D}_\theta(A)\rangle\,[[] \mid \text{tag}\,[[] \mid \mathcal{S}(\theta,\,A)]]$$

The compilation of the **affine** $x; P$ process is modified to generate a `branchFlags` instruction which branches on whether the inner type A is a positive droppable type. If it is not, the implementation is the same as before, with the addition of a `continue` instruction, as mentioned above. Otherwise, the inner process P is executed immediately, writing the value directly to the affine session.

The **use** $x; P$ process, where x : **coaffine** A is modified to also branch on whether A is a negative droppable type. If it is not, the implementation is the same as before. Otherwise, the instructions does nothing. The same goes for the **discard** x process, but if A is a negative droppable type, a `dropSlots` instruction is generated to drop the value immediately.

Finally, the **cell** $x(y.\,P)$ and **put** $x(y.\,P); Q$ processes are modified to branch on whether the inner type A (where $y : A$) is a positive droppable type. If not, the implementation is unchanged. Otherwise, the process P is executed immediately, with session y initialized such that it writes directly to the cell's storage. The **take** $x(y); P$ process is also modified, such that if A (where $y : A$) is a negative droppable type, the value is simply moved from cell x to a new data section for y, instead of receiving a continuation.

4.2 Optimizing Single Exit Point Processes and Tail Calls

In the base version of the compiler, every exit point instruction decrements the exit-point counter of the process environment, and if it reaches zero, the environment is cleaned up and the process terminates. For processes which have only one exit point, this is wasteful - there is no need to maintain an exit-point counter, as it is guaranteed to reach zero when an exit point instruction is executed.

To take advantage of this, the C code generation phase omits any exit point handling code when the IR process has only one exit point, avoiding unnecessary memory accesses and branches.

Tail calls, i.e., recursive process calls executed when there is a single exit point left, are optimized to reuse the current process environment instead of allocating a new one, avoiding the overhead of memory allocation and deallocation, and greatly improving performance. This requires branching on the exit-point count, to decide on whether a tail call is possible. If the optimization above is applied, the branch becomes unnecessary for most processes.

4.3 Simplifying the Generated IR Programs

The previously mentioned transformations introduce many optimization opportunities in the generated IR code itself. For example, below is a result of compiling a send of positive value type, as described in Section 4.1.3. A new continuation c_2 is introduced only for writing to c_1. In this case, it could be replaced entirely by direct writes to c_1.

```
    initContinuation(c2, send_rhs_2, c1)
    writeExpression(c2, 1)
    finish(c2, exit point)
send_rhs_2:
    ... // send rhs
```

To simplify the generated IR code, a control flow graph for each IR process is built through symbolic execution. For each instruction, the known state of the process environment is updated, storing information such as the known state of each continuation, or the known values of slots in the local data sections. For each visited IR block, a node is added to the graph, storing the nodes into which the node may jump to, as well as nodes which become detached from the main control flow, such as blocks referenced by a `pushTask` instruction. Eventually, all reachable nodes are visited.

The first applied optimization removes jumps to known locations, such as `continue` instructions of known continuations, or branches with known outcomes. After removing these instructions, the source and target IR blocks are merged. The example above becomes:

```
    initContinuation(c2, -, c1) // no target location
    writeExpression(c2, 1)
    ... // send rhs
```

Next, remote data locations are replaced by their known locations. For example, the writing location of c_2 was just initialized to be c_1. Thus, the data location c_2 can be replaced by c_1. This leaves us with a continuation c_2 which is initialized but never used, and thus, can be removed.

```
    initContinuation(c2, -, c1) // c2 is unused, can be removed
    writeExpression(c1, 1) // write directly to c1
    ... // send rhs
```

Another useful result from the symbolic execution is knowing which exit point instructions are guaranteed to not be the last instruction executed by the process. For example, in the following code, the `finish` instruction will never be the last executed instruction, as there is an outgoing flow from the `entry` block to `rhs` block, introduced by `pushTask`. In cases like these, the exit point markings can be removed from the instructions, and the exit-point counter of the process decremented. This makes most processes have a single exit point, which can be optimized as described in Section 4.2.

```
entry:
    pushTask(rhs)
    finish(c1, exit point) // becomes just finish(c1)
rhs:
    popTask(exit point)
```

Finally, many deferred drops can be turned into certain drops. If a given `deferDrop` instruction is guaranteed to be run, the instruction can be removed, and the drop bit of the process environment marked as always set. This saves memory and avoids unnecessary branching on environment cleanup.

5 Implementation and Evaluation

Contents

The original goal of this project was to design and implement an efficient compilation scheme for at least part of the CLASS language. This goal was achieved and surpassed, as the implemented compiler supports the full language, including polymorphism, recursion, exponentials, and shared state through mutable reference cells. On top of this, multiple optimizations were designed and implemented, resulting in significant performance improvements, and unexpectedly surpassing the performance of equivalent Haskell implementations in some benchmarks.

The compiler was fully implemented in Java, within the existing CLASS and SAM interpreters code base [29]. The implementation follows the architecture described in Chapter 3, and includes all the optimizations described in Chapter 4. The implementation has been extensively tested through a bench of more than 180 incremental unit tests written in the CLASS language, covering all language features. Furthermore, the performance of the generated code has been evaluated through benchmarks program and the results compared with different compiler configurations and alternatives.

5.1 Implementation

The compiler is contained within a new `compiler` namespace within the existing CLASS and SAM interpreters Java code base. The main class of the compiler is `Compiler`, which orchestrates the different compilation phases. This compiler program itself is invoked through a bash script, which also runs `gcc` to generate the final executable from the generated C code.

The compiler starts by parsing flags and options from the command line. Each of the implemented optimizations is toggled by a flag, making it possible to easily benchmark the impact of each optimization. It is possible to configure the target architecture (e.g., 32-bits or 64-bits), to enable trace logs on generated programs, and to enable the printing of the IR program at different stages, among others.

The input CLASS source code is parsed and type-checked using the original code, generating an annotated Abstract Syntax Tree (AST). The AST is visited by the `IRGenerator` visitor, producing an IR program, as described in Section 3.2. Enabled optimizations are then applied, and finally, the optimized IR program is translated to C code through the `CGenerator` visitor, as described in Section 3.3.

The IR is modeled as a set of classes representing the different instructions, expressions, and slots, following the model described in Section 3.1. Each instruction class provides methods to accept visitors, as well methods to easily transform the instructions during the optimization phases. While `IRExpression` objects occur as fields of `IRInstruction` objects, the latter are stored within `IRBlock` objects. An `IRProcess` object contains a description of the process header, and a set of labeled `IRBlock` objects, one of them being the entry block. Finally, an `IRProgram` object contains a set of `IRProcess` objects.

The C code generation phase is implemented through the `CGenerator` visitor, which visits the IR program and generates C code. Multiple utility classes are used to represent C code constructs, with the goal of making both the compiler code and the generated C code more readable.

5.2 Testing the Compiler

The project repository includes a comprehensive suite of more than 180 unit tests written in the CLASS language, which cover all features of the language, including edge cases relevant to the implemented optimizations. Each test consists of a CLASS source file, and optionally, a number of input and expected output files. Tests are categorized into folders based on the language feature they cover, such as exponentials, polymorphism, or process calls.

Multiple complex CLASS programs were compiled, tested and integrated into the test suite in order to further validate the correctness of the generated code when many features are combined. These programs include a functional implementation of the Game of Life cellular automaton [44], a prime sieve, a linked list implementation, a simulation of the dining philosophers problem, and other programs making extensive use of both shared state, polymorphism, recursion and exponentials.

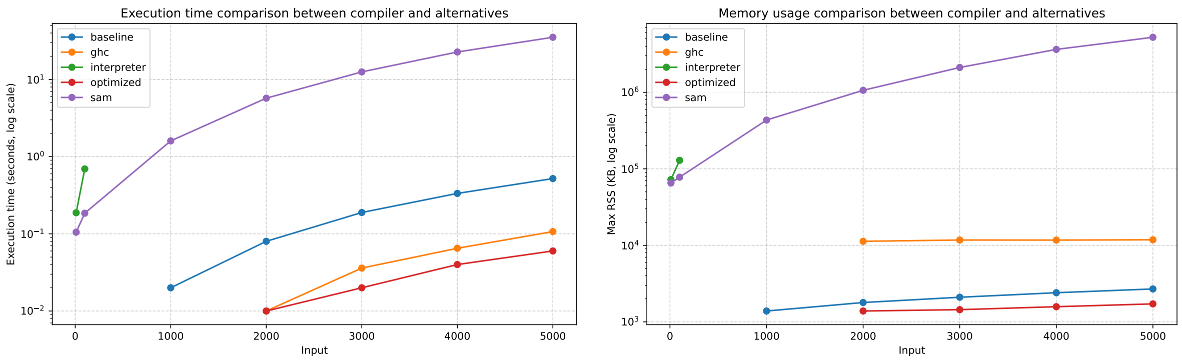


Figure 5.1: Prime sieve benchmarks averaged over 10 runs, parametrized over prime count. Compares the original interpreter, the SAM, the baseline compiler, the compiler with all optimizations enabled, and GHC.

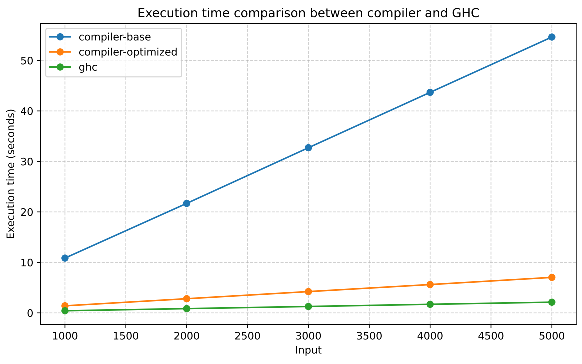
In order to easily find memory errors, a compiler flag `--profiling` can be enabled to compile the programs with allocation counters for each type of structure (e.g., processes, tasks, exponentials) and report any memory leaks when the program finishes execution. Additionally, the compilation script supports passing flags to `gcc` to enable the address and thread sanitizers, helping find and troubleshoot memory access errors and data races.

Tests are executed through a bash script `test_compiler.sh`, which runs each test in the suite, with an optional filter to select specific tests. The script compiles each test using the available debugging flags. If successfully compiled, and if input files are provided, the generated executable is run for each input file and the output is compared with the expected output file (if provided). If there are no input files, the executable is simply run to check for runtime errors. The script reports the results of each test, including compilation errors, runtime errors, memory leaks, and output mismatches. Compiler flags can also be passed to the script to test different configurations. To speed up the testing process, tests are distributed across multiple jobs.

5.3 Performance Evaluation

To measure the performance of the compiler, three benchmarking CLASS programs were developed along with equivalent Haskell implementations. Different compiler configurations were benchmarked, including the interpreted execution of the same program using both the CLASS and SAM interpreters, and the equivalent Haskell implementations, compiled with GHC `-O2`. All presented results are averaged over 10 runs, and were obtained on a machine with an Intel Core i9-12900K CPU and 32GB of RAM, running NixOS 25.05. The full benchmark results are presented in Appendix A.

The first benchmark program computes prime sieve numbers using the Turner's Sieve algorithm, generating a list of prime numbers up to a given count, taken as input. This benchmark was chosen



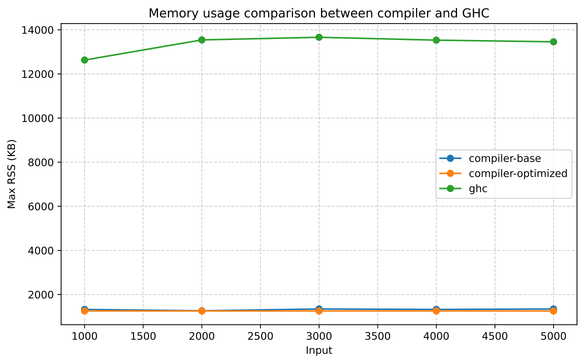


Figure 5.2: Game of Life benchmarks averaged over 10 runs, parametrized over generation count. Compares the baseline compiler, the compiler with all optimizations enabled, and GHC. Both the original interpreter and the SAM were unable to run this benchmark.

because it is a simple yet computationally intensive program. On Figure 5.1, the execution time and the maximum Resident Set Size (RSS), representing the peak memory usage, are compared between the original CLASS interpreter, the SAM interpreter, the unoptimized (baseline) compiler, the fully optimized compiler, and the equivalent Haskell implementation. The original interpreter cannot handle more than 150 primes due to operating system limits on the number of threads. The results show that the fully optimized compiler achieves a speed up of more than 100x compared to the SAM interpreter, a speedup of more than 5x compared to the unoptimized compiler, and also that it outperforms the Haskell implementation, while using significantly less memory.

The second benchmark program implements a functional version of the Game of Life cellular automaton [44]. All runs were given the same 17x17 initial configuration containing the *Kok's galaxy*[1] pattern, and were parametrized by the number of generations to compute. In this benchmark, shown in Figure 5.2, the optimized compiler performs slower than the Haskell implementation, by a factor of approximately 2.5x. The memory usage of the generated program is, however, significantly lower than the Haskell implementation. We suspect that with further optimizations, such as allocating data in registers and reusing sessions and data sections, the Haskell implementation can be matched or even surpassed.

A third benchmark, covering the shared state features of CLASS, implements a queue shared between a producer and a consumer. The producer pushes a number of items into the queue, while the consumer pops them concurrently. The Haskell implementation uses `MVar` to implement the shared state, as it bears a strong resemblance to the cells of CLASS. The results, shown in Appendix A, indicate that the optimized compiler performs only slightly better than the baseline compiler, and both are significantly faster than the SAM, but slower than the Haskell implementation.

[1] `https://conwaylife.com/wiki/Kok%27s_galaxy`, accessed 2025-10-04

6 Conclusion

Contents

This chapter concludes the thesis. On the first section, a summary of the main points discussed in the previous chapters is presented, along with the contributions of this work. The second section outlines potential areas for future work and improvements.

6.1 Conclusions

In this thesis, we successfully designed and implemented a novel type-directed compilation scheme for the linear session-typed language CLASS [1–3], generating efficient low-level C code. To the best of our knowledge this is the first work demonstrating the efficient compilation of linear session basic languages to machine code. The primary goal — to preserve the semantics and safety guarantees of CLASS while achieving competitive performance — was achieved. The implemented compiler produces code that is orders-of-magnitude faster than the interpreted execution of CLASS programs, and is competitive with equivalent Haskell programs compiled with GHC.

This work introduces several key contributions to the field of session-typed programming languages and their compilation. The first Intermediate Representation (IR) specifically designed for efficiently translating linear/session-typed languages to low level code was developed. This IR was heavily refined and iterated upon during its development, with various abstraction levels and approaches to memory management and session communication being explored. The final design was chosen due to its performance potential and its ability to easily encode the SAM execution model, while avoiding garbage collection and complex runtime systems.

During execution, data is stored within heap-allocated environments, stack-frame like structures associated to process calls. Due to the coroutine-like nature of session-based programs, environments are not organized in a traditional stack. Instead of keeping a single return address, each environment may reference multiple other environments through sessions, forming a graph-like structure. Environments are destroyed when the program counter leaves them for the last time, using a reference counting scheme.

Communication over sessions is compiled into direct memory writes and reads into and from these frames, with sizes and offsets determined at compile time, excluding polymorphic code sections. When a sequence of writes on a given session finishes, control is passed back to the continuation stored in the active frame, following the execution model of the SAM.

Although the execution model of the generated code is sequential, the compilation scheme also supports concurrency through the shared state constructs of CLASS. Shared state cells are compiled into heap-allocated reference-counted structures, protected by mutexes, allowing for safe concurrent access. These cells can then be shared freely between multiple threads.

Various optimizations were explored and implemented, including common transformations such as inlining, monomorphization of polymorphic code, and tail call elimination. The most noteworthy optimizations explored make use of the information available in session types to reduce the number of memory copies, jumps and runtime checks. For example, a send of a session holding some value through another session can be transformed into a send of the value itself, avoiding multiple operations. Additionally, exponentials were optimized by storing their values directly if possible, cloning them as needed. This change avoids the overhead of recomputing the values of exponentials every time they are used.

An implementation of the developed scheme was created, covering all of the features of CLASS, including polymorphism, recursion, exponentials and shared state. The implementation, available on GitHub[1], was built in Java, on top of the existing SAM codebase. Most of the discussed optimizations are configurable through compiler flags. A suite of tests was created to validate the correctness of the implementation, and the performance of the generated code was evaluated through a series of benchmarks, across different configurations. The compiler was shown to generate programs competitive

[1]https://github.com/RiscadoA/class

with equivalent Haskell GHC compiled programs, being able to surpass them in some cases.

6.2 Future Work

The developed solution, while already showing promising results, still has several avenues for future improvements and optimizations. One such avenue is compiling to a lower level target language, such as LLVM IR, or assembly language. For example, this could allow for storing frequently accessed data entirely in CPU registers, and committing them to memory only when absolutely necessary.

The IR could also be optimized further, by removing, for example, redundant moves and copies of data, which are frequently generated by the current compilation scheme. Both continuations and data sections could be reused if possible, reducing the memory footprint of the generated programs, and improving cache locality.

It would also be interesting to construct a formal proof of correctness for the compilation scheme, demonstrating that the generated IR code preserves the semantics and safety of CLASS. As the compiled code strongly resembles the SAM execution model, the proof could be based on its existing proof of correctness.

Concurrent cuts have not been addressed in this work, as the focus was on generating efficient sequential code. The type system of the CLASS language could be modified to distinguish between sequential and concurrent sessions, allowing for the generation of efficient code for both sessions types.

Finally, a standard library for CLASS could also be developed, providing native implementations of common data structures such as arrays and maps, as currently these are emulated using recursive session types. This would greatly improve the usability and performance of CLASS programs. This could be achieved by, for example, implementing a Foreign Function Interface (FFI) to C libraries.

Bibliography

[1] P. Rocha and L. Caires, “Propositions-as-types and shared state,” *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. [Online]. Available: https://doi.org/10.1145/3473584

[2] ——, “Safe session-based concurrency with shared linear state,” in *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 421–450. [Online]. Available: https://doi.org/10.1007/978-3-031-30044-8_16

[3] P. Rocha, “Class: A logical foundation for typeful programming with shared state,” Ph.D. dissertation, NOVA University Lisbon, July 2022, supervised by Luís Caires.

[4] L. CAIRES, F. PFENNING, and B. TONINHO, “Linear logic propositions as session types,” *Mathematical Structures in Computer Science*, vol. 26, no. 3, pp. 367–423, 2016.

[5] P. Wadler, “Propositions as sessions,” *SIGPLAN Not.*, vol. 47, no. 9, p. 273–286, Sep. 2012. [Online]. Available: https://doi.org/10.1145/2398856.2364568

[6] ——, “Linear types can change the world!” in *Programming Concepts and Methods*, 1990. [Online]. Available: https://api.semanticscholar.org/CorpusID:58535510

[7] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *Programming Languages and Systems*, C. Hankin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138.

[8] D. Mostrous and V. T. Vasconcelos, “Affine Sessions,” in *Proc. of COORDINATION 2014*, ser. LNCS, vol. 8459. Springer, 2014, pp. 115–130.

[9] N. D. Matsakis and F. S. Klock, “The rust language,” in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[10] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Russi, D. Sezer, T. Zakian, and R. Zhou, "Move: A Language with Programmable Resources," *Libra Assoc*, 2019.

[11] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. W. Barrett, and D. L. Dill, "The move prover," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 137–150. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_7

[12] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear haskell: practical linearity in a higher-order polymorphic language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158093

[13] A. Das and F. Pfenning, "Rast: A language for resource-aware session types," *Log. Methods Comput. Sci.*, vol. 18, no. 1, 2022.

[14] R. Chen, S. Balzer, and B. Toninho, "Ferrite: A Judgmental Embedding of Session Types in Rust," in *36th European Conference on Object-Oriented Programming, ECOOP 2022*, ser. LIPIcs, K. Ali and J. Vitek, Eds., vol. 222, 2022, pp. 22:1–22:28.

[15] M. Willsey, R. Prabhu, and F. Pfenning, "Design and implementation of concurrent C0," in *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016*, ser. EPTCS, I. Cervesato and M. Fernández, Eds., vol. 238, 2016, pp. 73–82.

[16] 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*, ser. EPTCS, F. Martins and D. Orchard, Eds., vol. 291, 2019, pp. 12–23.

[17] J. Jacobs and S. Balzer, "Higher-order leak and deadlock free locks," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 1027–1057, 2023.

[18] R. Milner, *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag, 1980. [Online]. Available: https://doi.org/10.1007/3-540-10235-3

[19] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540192900084

[20] ——, "A calculus of mobile processes, ii," *Information and Computation*, vol. 100, no. 1, pp. 41–77, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540192900095

[21] J.-Y. Girard, “Linear logic,” *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397587900454

[22] L. Caires and F. Pfenning, “Session types as intuitionistic linear propositions,” in *CONCUR 2010 - Concurrency Theory*, P. Gastin and F. Laroussinie, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 222–236.

[23] P. Rocha and L. Caires, “Propositions-as-types and shared state (artifact),” *Proc. ACM Program. Lang.*, May 2021.

[24] ——, “Safe session-based concurrency with shared linear state (artifact),” January 2023.

[25] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, “Session types for rust,” in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, ser. WGP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–22. [Online]. Available: https://doi.org/10.1145/2808098.2808100

[26] L. Caires and B. Toninho, “The session abstract machine,” in *Programming Languages and Systems: 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 206–235. [Online]. Available: https://doi.org/10.1007/978-3-031-57262-3_9

[27] P. J. Landin, “The Mechanical Evaluation of Expressions,” *The Computer Journal, Volume 6, Issue 4, January 1964*, vol. 6, no. 4, p. 308–320, 1964.

[28] Y. Lafont, “The linear abstract machine,” *Theoretical Computer Science*, vol. 59, no. 1, pp. 157–180, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397588901004

[29] L. Caires and B. Toninho, “The Session Abstract Machine (Artifact),” 2024.

[30] M. Hofmann, “A type system for bounded space and functional in-place update,” *Nordic J. of Computing*, vol. 7, no. 4, p. 258–289, Dec. 2000.

[31] S. Abramsky, “Computational interpretations of linear logic,” *Theoretical Computer Science*, vol. 111, no. 1, pp. 3–57, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759390181R

[32] D. N. Turner and P. Wadler, “Operational interpretations of linear logic,” *Theoretical Computer Science*, vol. 227, no. 1, pp. 231–248, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397599000547

[33] A. Barber and G. Plotkin, *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1996.

[34] K. Honda, "Types for dyadic interaction," in *CONCUR'93*, E. Best, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–523.

[35] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro, "Foundations of session types and behavioural contracts," *ACM Comput. Surv.*, vol. 49, no. 1, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2873052

[36] N. Yoshida, R. Hu, R. Neykova, and N. Ng, "The scribble protocol language," in *Trustworthy Global Computing*, M. Abadi and A. Lluch Lafuente, Eds. Cham: Springer International Publishing, 2014, pp. 22–41.

[37] A. Scalas, N. Yoshida, and E. Benussi, "Effpi: verified message-passing programs in dotty," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, ser. Scala '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 27–31. [Online]. Available: https://doi.org/10.1145/3337932.3338812

[38] R. Pucella and J. A. Tov, "Haskell session types with (almost) no class," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, ser. Haskell '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 25–36. [Online]. Available: https://doi.org/10.1145/1411286.1411290

[39] L. Caires, F. Pfenning, and B. Toninho, "Towards concurrent type theory," in *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, ser. TLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–12. [Online]. Available: https://doi.org/10.1145/2103786.2103788

[40] M. Giunti and V. T. Vasconcelos, "Linearity, session types and the pi calculus," *Mathematical Structures in Computer Science*, vol. 26, no. 2, pp. 206–237, 2016.

[41] A. Ahmed, M. Fluet, and G. Morrisett, "Lˆ 3: a linear language with locations," *Fundamenta Informaticae*, vol. 77, no. 4, pp. 397–449, 2007.

[42] W. Kokke and O. Dardha, "Deadlock-free session types in linear haskell," in *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–13. [Online]. Available: https://doi.org/10.1145/3471874.3472979

[43] N. Lagaillardie, R. Neykova, and N. Yoshida, “Stay safe under panic: Affine rust programming with multiparty session types,” 2022. [Online]. Available: https://arxiv.org/abs/2204.13464

[44] M. Gardner, “The fantastic combinations of John Conway’s new solitaire game “life” by Martin Gardner,” *Scientific American*, vol. 223, pp. 120–123, 1970.

A
Benchmark Results

This appendix contains the full benchmark results that were summarized in Chapter 5. The source code for the benchmarks is too large to include in this document, but is available in the GitHub repository[1] containing the implementation of the compiler and the runtime system. The programs used for the benchmarks were the prime sieve, at `examples/sam/sam-sieve-sum-fast-stop-intp.clls`, the Game of Life, at `tests/complex/game_of_life_2.clls`, and finally, the shared queue, which can be found at `examples/LFCS/xqueue-n.clls`.

Each of the benchmarks was run 10 times for each data point, and the average execution time and memory usage were recorded. The execution time and memory usage were measured using the GNU `time -v` command, which provides detailed information about the resources used by a program. The maximum Resident Set Size (RSS) was used as the measure of memory usage. All results were obtained on a machine with an Intel Core i9-12900K CPU and 32GB of RAM, running NixOS 25.05.

To measure the impact of each implemented optimization, multiple compiler configurations were tested. The `base` configuration has all optimizations disabled, except for the exponential value optimization, which is not configurable. The `optimized` configuration has all optimizations enabled. The rest of the configurations are split into two groups: the `<set>` configurations which have only the specified optimizations enabled, and the `no-<set>` configurations which have all optimizations enabled except for

[1] https://github.com/RiscadoA/class

the specified one. The benchmarked optimization sets are described below:

- `inlining`: Inlining and monomorphization of process calls.
- `values`: Enables the optimizations which change the representation of sends of value types and affine types, as described in Sections 4.1.3 and 4.1.5.
- `endpoints`: Enables tail calls, uses symbolic execution to find and eliminate redundant exit points, and avoids branching on processes with a single exit point, as described in Sections 4.2 and 4.3.
- `control-flow`: Uses symbolic execution to find and eliminate known jumps, branches and always set drop bits, as described in Section 4.3.
- `locations`: Replaces data locations referring to known continuations with direct data locations, as described in Section 4.3.

The Haskell implementation of the benchmarks was compiled using GHC 9.8.4 with the `-O2` optimization flag.

A.1 Prime Sieve Results

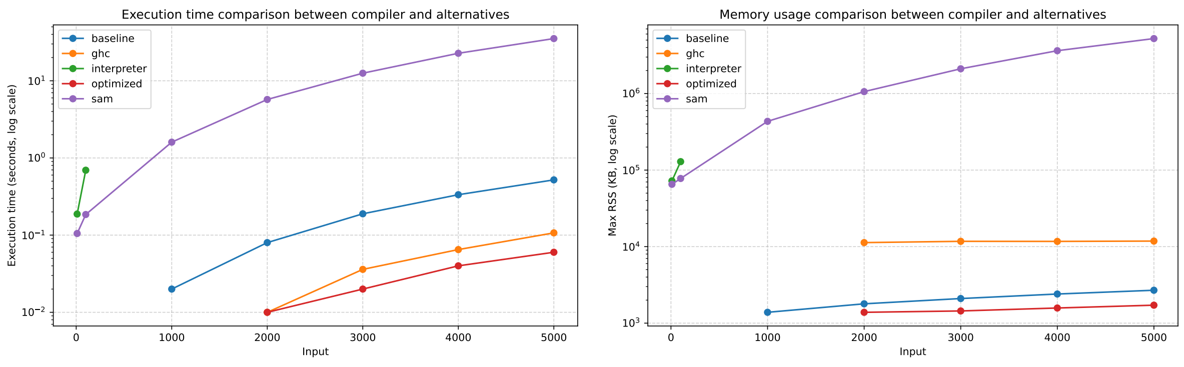


Figure A.1: Same as Figure 5.1 (repeated for reference). Compares the prime sieve benchmarks across the original interpreter, the SAM, the baseline compiler, the compiler with all optimizations enabled, and GHC.

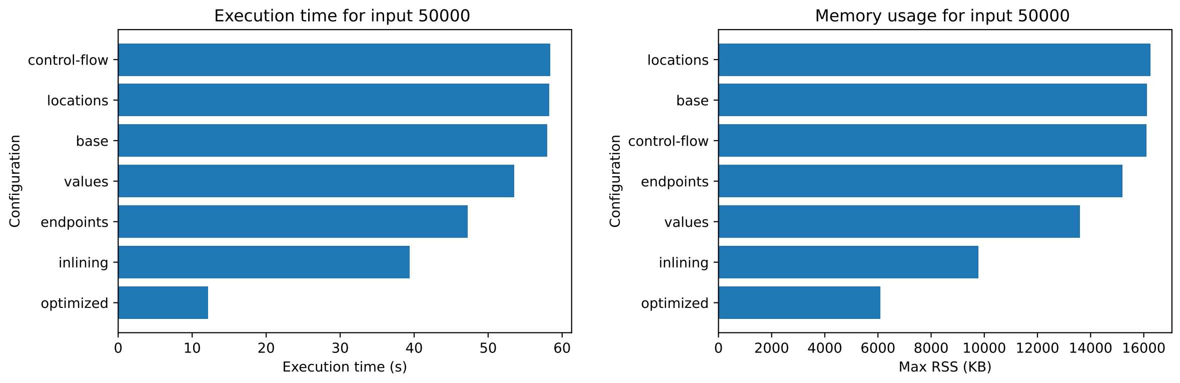


Figure A.2: Prime sieve benchmarks averaged over 10 runs, with a fixed prime count of 50000. Compares the baseline compiler, the fully optimized compiler, and multiple compiler configurations with only specific optimizations enabled.

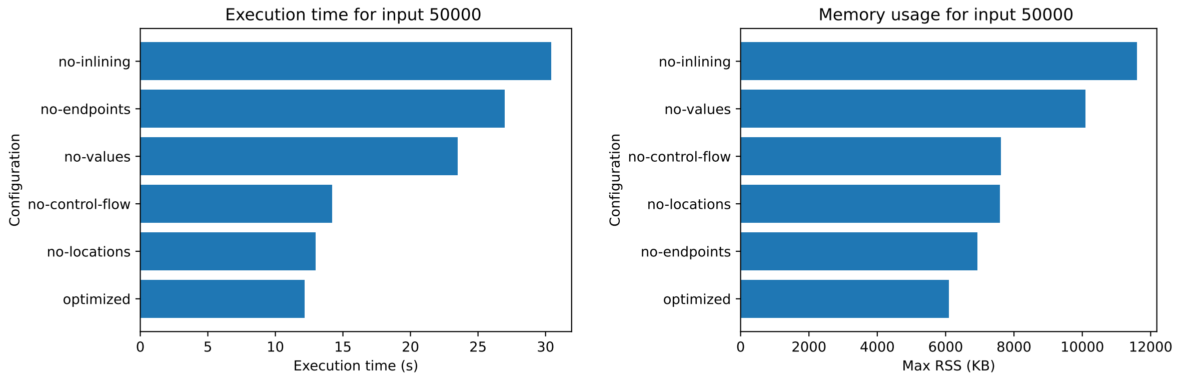


Figure A.3: Prime sieve benchmarks averaged over 10 runs, with a fixed prime count of 50000. Compares the fully optimized compiler, and multiple compiler configurations with only specific optimizations disabled.

A.2 Game of Life Results

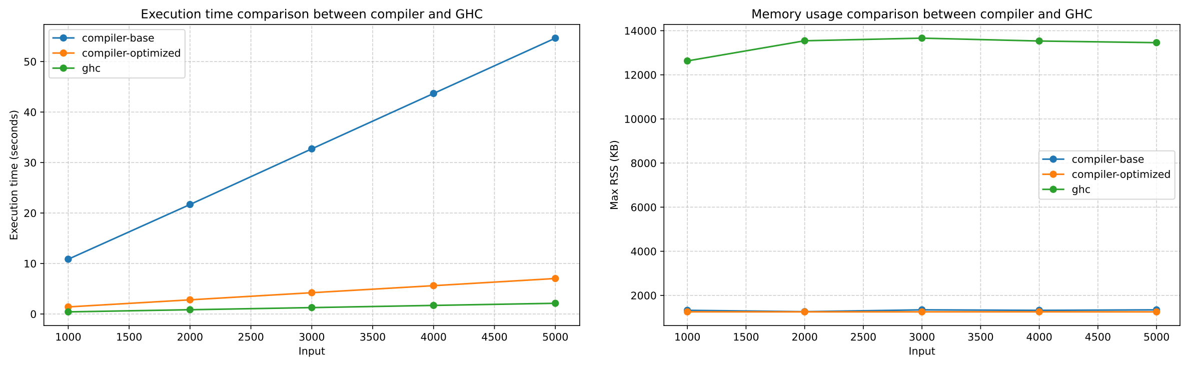


Figure A.4: Same as Figure 5.2 (repeated for reference). Compares the Game of Life benchmarks across the baseline compiler, the compiler with all optimizations enabled, and GHC.

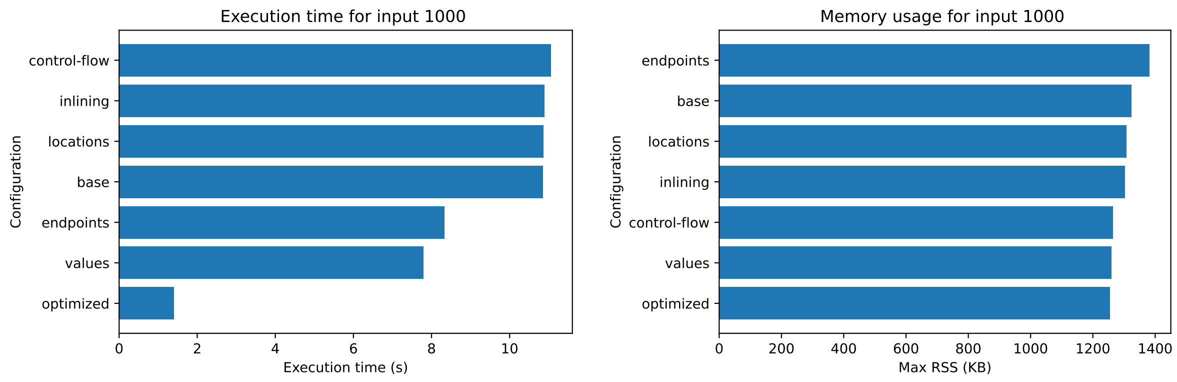


Figure A.5: Game of Life benchmarks averaged over 10 runs, with a fixed generation count of 1000. Compares the baseline compiler, the fully optimized compiler, and with only specific optimizations enabled.

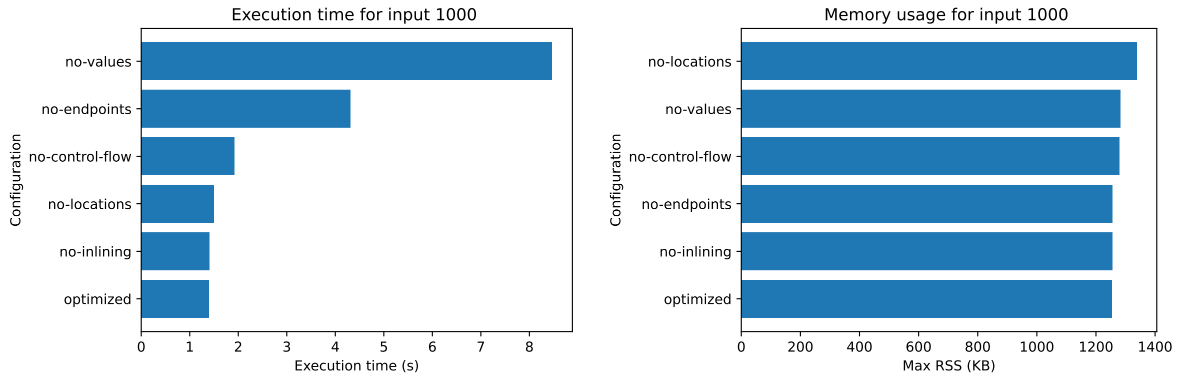


Figure A.6: Game of Life benchmarks averaged over 10 runs, with a fixed generation count of 1000. Compares the fully optimized compiler, and multiple compiler configurations with only specific optimizations disabled.

A.3 Shared Queue Results

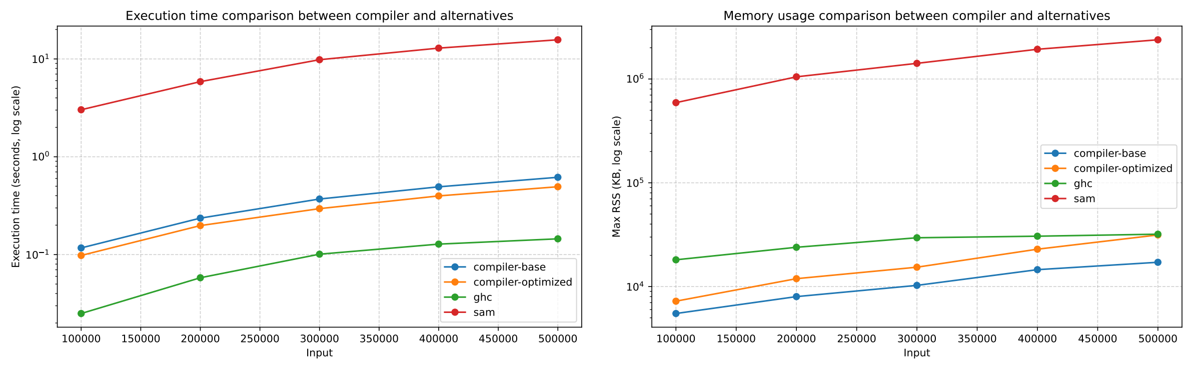


Figure A.7: Shared queue benchmarks averaged over 10 runs, parametrized over item count. Compares the SAM, the baseline compiler, the compiler with all optimizations enabled, and GHC. The original interpreter was were unable to run this benchmark.

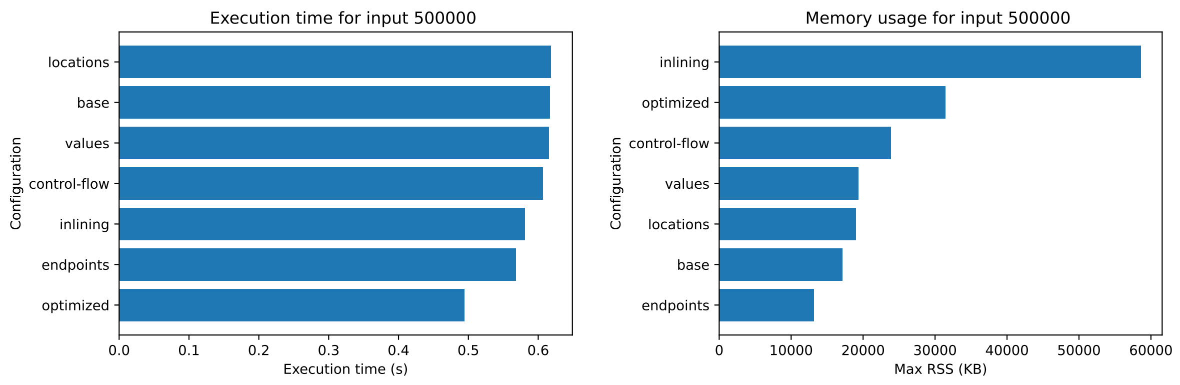


Figure A.8: Shared queue benchmarks averaged over 10 runs, with a fixed item count of 500000. Compares the baseline compiler, the fully optimized compiler, and with only specific optimizations enabled.

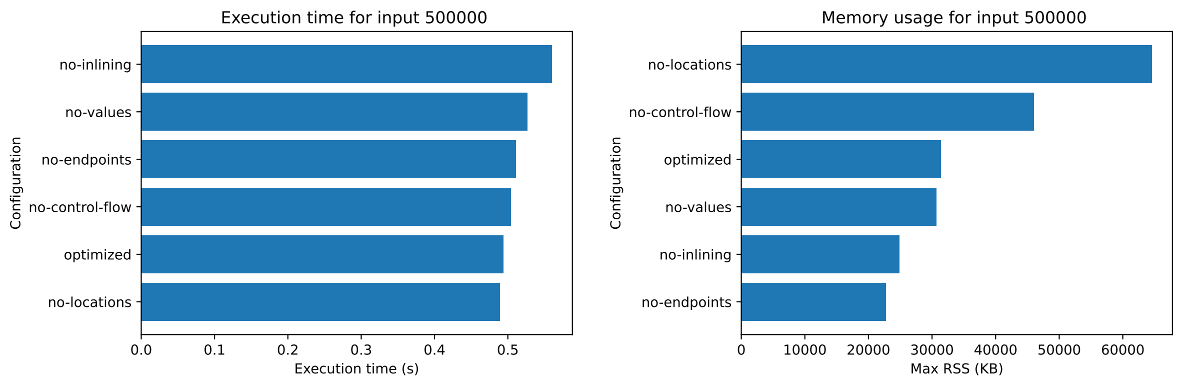


Figure A.9: Shared queue benchmarks averaged over 10 runs, with a fixed item count of 500000. Compares the fully optimized compiler, and multiple compiler configurations with only specific optimizations disabled.

B

IR Specification

This appendix specifies the syntax and the full instruction set of the Intermediate Representation (IR) developed during this work. Table B.1 lists all instructions of the IR, along with a brief description of each instruction. The full BNF syntax of IR programs is provided in the Listing B.1, containing the concrete syntax of all arguments used in the instructions.

In the instruction set specification, for convenience, the remote continuation referenced by a continuation `c` is denoted as `c.r`.

Table B.1: IR instruction set

Instruction	**Description**
`initContinuation(c, lab, loc)`	Initializes continuation `c`, with return address `lab`, writing location `loc` and remote continuation `c`.
`bindContinuation(src, c, loc)`	Binds the continuation stored at `src` to continuation `c`, setting the writing location of `c.r` to `loc`.
`continue(c, lab)`	Transfers control to the return address of continuation `c`. Sets the return address of `c.r` to `lab`.
`finish(c)`	Transfers control to the return address of continuation `c`. Sets `c.r.r` to `c.r`, unlinking `c.r` from `c`.
`forward(c1, c2)`	Copies `c1` to `c2.r` and `c2` to `c1.r`, linking `c1.r` and `c2.r`. Transfers control to the return address of `c2`.
`pushTask(lab)`	Pushes a new task with code address `lab` onto the task queue.
`popTask()`	Pops a task from the task queue and transfers control to it.
`jump(lab)`	Unconditional jump to label `lab`.
`branchTag(loc, case, ...)`	Branches on the tag stored at `loc`. Takes as arguments a list of cases for each tag `i`.
`branchExpr(expr, if, else)`	Branches on the result of a boolean expression. Takes as arguments a `if` and an `else` case.
`branchFlags(reqs, if, else)`	Similar to `branchExpr`, but branches on the whether the flag requirements `reqs` are satisfied.
`callProcess(pid, pargs)`	Calls process `pid` with type parameters, continuation arguments and data arguments, specified in `pargs`.
`callExponential(loc, c, d)`	Calls the exponential closure stored at `loc`, initializing a new continuation `c` as its argument, with writing location `d`.
`writeExponential(loc, pid, eargs)`	Writes to `loc` a new exponential closure from process `pid`, with type parameters and data arguments specified in `eargs`.
`writeExpression(loc, expr)`	Evaluates `expr` and writes the result to `loc`.
`writeScan(loc, sl)`	Reads a slot `sl` from the standard input and writes it to `loc`.
`writeTag(loc, i)`	Writes a tag `i` to location `loc`.
`writeContinuation(loc, c)`	Writes a continuation reference `c` to location `loc`.
`writeCell(loc, st)`	Writes a new cell with capacity for slot tree `st` to location `loc`.
`writeType(loc, st, tflg)`	Writes a type parameter to `loc` described by a slot tree `st` and with flag requisites `tflg`.
`moveSlots(dst, src, st)`	Moves the slot tree `st` from location `src` into `dst`.
`cloneSlots(dst, src, st)`	Clones the slot tree `st` from location `src` into `dst`.
`dropSlots(loc, st)`	Drops the slot tree `st` at location `loc`.
`lockCell(loc)`	Locks the cell at `loc`.
`unlockCell(loc)`	Unlocks the cell at `loc`.
`acquireCell(loc)`	Increases the reference count of the cell at `loc`.
`releaseCell(loc)`	Decreases the reference count of the cell at `loc`. If the reference count reaches zero, the cell is deallocated.
`launchThread(lab)`	Launches a new thread, starting execution at label `lab`.
`sleep(int)`	Sleeps for a number of milliseconds.
`print(expr)`	Evaluates `expr` and prints the result to the standard output.
`printLine(expr)`	Same as `print`, but adds a newline character after printing.
`panic(str)`	Prints the string `str` to the standard error and aborts execution.

Listing B.1: BNF Grammar of the Intermediate Representation

```
nl   ::= "\n"
pid  ::= [a-zA-Z_][a-zA-Z0-9_]*
lab  ::= [a-zA-Z_][a-zA-Z0-9_]*
int  ::= [0-9]+
str  ::= /* double-quoted string */
bool ::= "true" | "false"

/* type, continuation, data and drop bit identifiers */
t, t1, t2 ::= "t" int
c, c1, c2 ::= "c" int
d, d1, d2 ::= "d" int
e         ::= "e" int

/* slots, slot trees, slot tree combinations, offsets */
sl  ::= "exponential"     | "int" | "bool" | "string" | t
      | "cell" "[" st "]" | "tag" | "cont" | "type"
st  ::= "[]" | "[" sl (";" sl)* (";" st)? "]"
      | "tag[" st ("|" st)* "]"
      | "flags" "<" reqs ">" "[" st "|" st "]"
sc  ::= "<" st ("+" st)* ">"
ofs ::= st "~" st

/* locations */
locb           ::= c | d | "c" "(" loc ")"
loc, src, dst ::= locb ("." ofs)?

/* type flag requirements */
reqs ::= "no" | "yes" | "if" "(" flag ("," flag)* ")"
flag ::= t "." ("value" | "droppable" | "cloneable")

/* type, continuation and data arguments */
tflg ::= ("(" flag "=" reqs ")")*
targ ::= t "<-" loc | t "<-" st tflg
carg ::= c1 "<-" (loc | c2) ("+" ofs)?
darg ::= d "<-[" st "]" loc | d "=[" st "]" loc

/* process and exponential arguments */
parg  ::= targ | carg | darg
pargs ::= (parg ("," parg)*)?
earg  ::= targ | darg
eargs ::= (earg ("," earg)*)?

/* branch cases */
case, if, else ::= lab ":" int

/* program structure */
prog  ::= proc+
proc  ::= pid ":" nl entry* block+
entry ::= "exit points:" int nl
        | "types:" (t+ | "none") nl
        | "continuations:" (c+ | "none") nl
        | "data" d ":" sc ("(" c ")")? nl
        | "drop" e ":" st "at" loc "(always)"? nl
block ::= lab ":" nl (instr nl)+
expr  ::= expr "+" expr | expr "-" expr | expr "*" expr
        | expr "/" expr | expr "%" expr
        | expr "=" expr | expr ">" expr | expr "<" expr
        | expr "and" expr | expr "or" expr | "not" expr
        | "(" expr ")" | int | str | bool
        | ("move" | "clone") "(" loc "," sl ")"
instr ::= /* consult instruction set table */
```