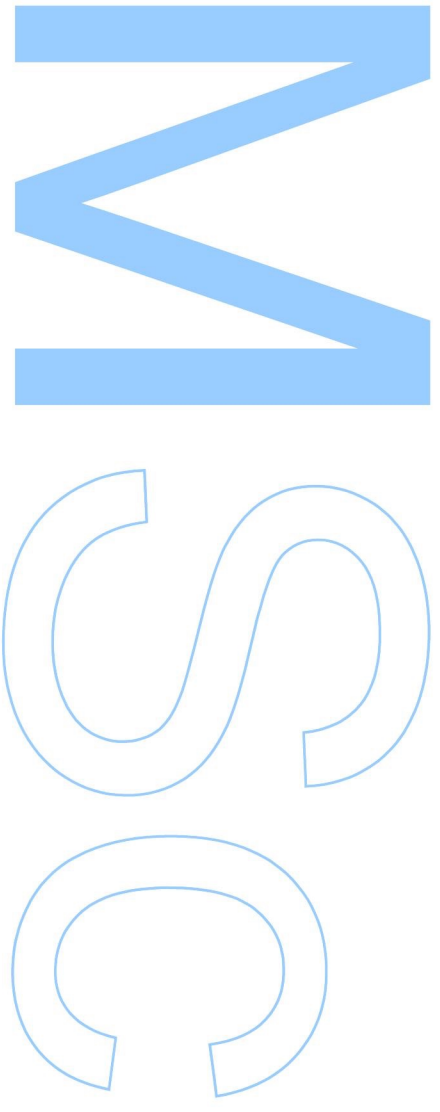
Non-Blocking Concurrent Imperative Programming with Session Types

Miguel Eduardo Pinto da Silva
Master's degree in Computer Science
Computer Science Department
2016

Supervisor
António Mário Florido, Associate Professor, Faculty of Science, University of Porto

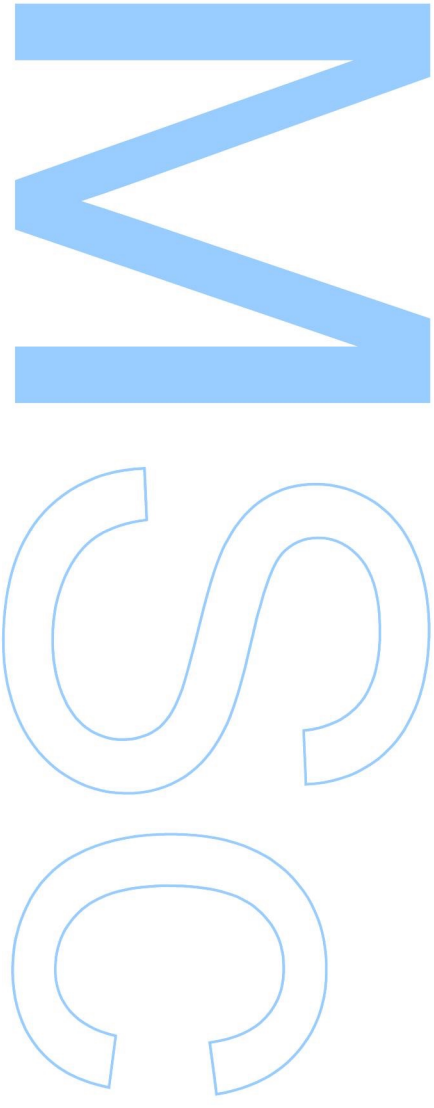
Co-supervisor
Frank Pfenning, Professor, Carnegie Mellon University



All corrections determined by the jury, and only those, were incorporated.

The President of the Jury,

Porto, _____/_____/_____



Miguel Eduardo Pinto da Silva

Non-Blocking Concurrent Imperative Programming with Session Types

Department of Computer Science
Faculty of Sciences University of Porto
June 2016

Miguel Eduardo Pinto da Silva

Non-Blocking Concurrent Imperative Programming with Session Types

Dissertation submitted to the Faculty of Sciences
University of Porto as part of the requirements for the degree of
Master in Computer Science

Supervisor: Prof. Mário Florido
Co-supervisor: Prof. Frank Pfenning

Department of Computer Science
Faculty of Sciences University of Porto
June 2016

To my parents

Acknowledgements

First, I would like to thank my advisors, Professor Mário Florido and Professor Frank Pfenning, for sharing all their knowledge and experience with me. In particular, I want to thank Professor Frank Pfenning for taking me in as an undergraduate intern in Pittsburgh, which was the beginning of this thesis. I also want to commend and thank for Professor Mário Florido's bravery for taking me as his student in a thesis he was not intimately connected with.

All my Professors share a fundamental role on my graduation, as they are the basis of everything I have learned. For this dissertation, I want to thank in particular Professor Sandra Alves, who helped me with the basics of π-calculus when I struggled the most with it, and Professor Ricardo Rocha, for the help with *pthreads* and the initial push in the right direction when I did not know where to start implementing this thesis.

To all my family, who supported me all my life. Specially, to my parents, who would not let me settle for anything less than perfection, and to my brother, who made me the geek I am.

I'm fortunate to have many good friends who made this year easier: Afonso, João Vítor, Ricardo and Tiago thank you for making my Friday and Saturday nights much more cheerful; Marvin and Rafaela, my friends from Physics, who always set a (very high!) bar for me to achieve; my friends from the Computer Science department, specially, João and Patricia.

Finally, to my dearest Catarina, who was always by my side in this year long journey, starting in Pittsburgh, and whose constant support and love were crucial for me to carry this dissertation through.

This work was partially funded by the FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program.

Miguel Eduardo Pinto da Silva
Porto, 2016

Abstract

Concurrent C0 is an imperative programming language in the C family with session-typed message-passing concurrency. The previously proposed semantics implements asynchronous (non-blocking) output; here, we extend it with non-blocking input. Our framework relies on one key idea: to postpone message reception as much as possible by interpreting receive commands as a request for a message. We also extend the semantics to include a cost, named work and span, on each operation. We implemented our ideas as a translation from a blocking intermediate language to a non-blocking language. Using the cost semantics and assuming correctness, we prove some important properties of the non-blocking model:

- The span in the non-blocking semantics is less or equal than the span in the blocking semantics.
- The work is unchanged across both semantics.

Finally, we evaluate our techniques with several benchmark programs and show the results obtained.

Keywords: Session Types, Concurrency, Asynchronous Communication, Message Passing, Work, Span.

Resumo

Concurrent C0 é uma linguagem de programação imperativa na família do C, com concurrência através da passagem de mensagens, usando *session types* como sistema de tipos. A semântica previamente proposta implementa *output* assíncrono não-bloqueante; neste trabalho, extendemos a semântica com *input* não-bloqueante. A nossa *framework* assenta numa ideia chave: adiar a recepção da mensagem o mais possível, interpretando uma acção de receber como um pedido de uma mensagem. Também extendemos a semântica para atribuir um custo, a que chamamos *work* e profundidade, a cada operação. Implementamos as nossas ideias como uma tradução de uma linguagem intermédia bloqueante para uma linguagem não-bloqueante. Usando a semântica de custo e assumindo correção, demonstramos algumas propriedades importantes, provenientes do modelo não-bloqueante:

- A profundidade calculada usando a semântica não-bloqueante é sempre menor ou igual à profundidade calculada usando a semântica bloqueante.
- O *work* é igual em ambas as semânticas.

Finalmente, avaliamos as nossas técnicas com um conjunto de programas referência e mostramos os resultados obtidos.

Palavras-chave: *Session Types*, Concurrência, Comunicação Assíncrona, Passagem de Mensagens, *Work*, Profundidade.

Contents

List of Tables

List of Figures

List of Code

Chapter 1

Introduction

This dissertation presents a non-blocking model for receiving messages, implemented as an extension to an existing imperative programming language, which improves the performance of the previous model of reception in some examples. Our model is based on the premise that we want to halt execution to wait for a message only when the data contained in that message is necessary to continue computation.

Our work is based on *session types* [16, 17], a typing system for π-calculus [24, 20], that defines communication between processes over communication channels.

In recent work, session types have been linked with linear logic via a Curry-Howard interpretation of linear propositions as types, proofs as processes, and cut reduction as communication. Variations apply for both intuitionistic [5, 6] and classical [29] linear logic.

The intuitionistic variant culminated in SILL, a functional language with session-typed concurrency [27]. The adaptation of SILL to imperative programming gave rise to Concurrent C0 [30], a session-based extension to an imperative language. This dissertation describes our modifications to Concurrent C0 to accomplish our non-blocking model.

Concurrent C0's model for message reception, which we refer to as blocking, relies on fetching messages from a queue, a usual construct in asynchronous communication using linear channels [14, 22]. Upon performing a receiving action, if the message is not in the queue, the execution of the program blocks, until it can retrieve the message

from the queue. On the other hand, our non-blocking model, by interpreting receives as requests, will only try to fetch the message when a synchronization is undergoing, which is only performed when the execution requires that message to proceed.

As an illustrative example of the difference between these two models, consider the following simple example: `x = receive(c); P; y = x+1`. We are trying to receive a value, `x`, from channel `c`, then execute a sequence of instructions, `P`, which do not use the variable `x`, and, finally, execute an operation using that variable. Under the blocking semantics, we need to stop execution until the value is ready, in spite of not needing it immediately. Under the non-blocking model, this value would only be synchronized before the instruction `y = x+1`.

We also present a compilation function from the original version of Concurrent C0 to our non-blocking one and introduce a cost semantics that computes the span and work [9], providing an abstract analytical measure of latent parallelism in the computation. Assuming correctness, we then prove that the span always improves or remains the same under the non-blocking semantics. We also prove that work, on the other hand, is constant in both semantics.

In related work, Guenot [15] has given a pure and elegant computational interpretation of classical linear logic with non-blocking input based on the solos calculus [19]. The primary notion of a process as a thread of control that pervades our work is no longer visible there and it does not immediately suggest an implementation strategy. Our work generalizes the functional notions of *futures* and *promises* [2, 13] by supporting more complex bidirectional communication [26].

The main results presented in this dissertation were previously reported in [25].

1.1 Dissertation Outline

This dissertation is structured as follows:

Chapter 2 - Background - presents the fundamental concepts that are the basis of this thesis.

Chapter 3 - Imperative Programming with Session Types - introduces the main concepts to programming with Concurrent C0, including compilation and target language, and proposes a cost semantics for the language.

Chapter 4 - Non-Blocking Receive - describes an extension to CC0, by changing how input is handled by each process. Introduces an intermediate language, new operational and cost semantics, a translation from CC0's target language to this intermediate language and some theoretical results.

Chapter 5 - Implementation and Experimental Evaluation - details how the concepts from Chapters 3 and 4 are implemented and presents some experimental comparison between the two receiving models.

Chapter 6 - Conclusion - concludes the dissertation and presents some hints for further work.

Chapter 2

Background

In this chapter, we present the theoretical background of this thesis. We give a brief overview of session types and link them with linear logic through a Curry-Howard correspondence. We also describe the operational semantics of the programming languages present in this thesis. Finally, we discuss some performance metrics for parallel algorithms.

2.1 Session Types

Message-passing concurrency is a model for concurrency where the concurrent entities communicate through message exchange, which may occur synchronously or asynchronously. *Process calculi* [7, 24] is the most prominent technique to reason about this model. It is a family of formal languages that use the abstract concept of channels to model communication between concurrent systems, named *processes*. The arguably standard process calculus is π-calculus [24, 20], whose main feature are channels that can be generated dynamically and passed in communication. Many type systems were developed over the years for this language, but *session types* are possibly the most important for dealing with one-to-one communication.

A session describes the collective conduct of the components of a concurrent system. Binary sessions focus on the interactions between two of these components, with an inherent concept of duality: when a component sends, the other receives. A *session*

A, B, C	$::=$	$\mathbf{1}$	terminate
	$\mid$	$A \otimes B$	send channel of type A and continue as B
	$\mid$	$A \oplus B$	provide either A or B
	$\mid$	$\tau \wedge B$	send value of type τ and continue as B
	$\mid$	$A \multimap B$	receive channel of type A and continue as B
	$\mid$	$A \,\&\, B$	offer choice between A or B
	$\mid$	$\tau \supset B$	receive value of type τ and continue as B
	$\mid$	$!A$	provide replicable session of type A

Figure 2.1: Session Types syntax

type [16, 17] defines the communication between processes using this notion. Session types enforce conformance to a communication protocol, organizing a session to occur over a communication channel.

Session types commonly capture input and output, choice and selection, replication and recursive behaviour. They also provide *session delegation*, assigning a session to another process through communication, and additional guarantees of system behaviour, such as *deadlock freedom* and *liveness* [12]. Figure 2.1 presents a possible syntax, matching the syntax of propositions in intuitionistic linear logic, to express some different session types.

Throughout this thesis, we use as a recurring example queues and binary search trees, to illustrate the discussed concepts. The latter also uses another data structure, linked lists, and both focus solely on integers. For now, we show only the session type definition of these two data structures (Figure 2.2). Note that these types are recursive, although we did not explicitly mention recursive session types in Figure 2.1.

$$
\begin{aligned}
&\mathsf{queue} = \&\{\mathsf{enq} : \mathsf{int} \supset \mathsf{queue}, \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{int} \wedge \mathsf{queue}\}\} \\
&\mathsf{list} = \oplus\{\mathsf{cons} : \mathsf{int} \wedge \mathsf{list}, \mathsf{nil} : \mathbf{1}\} \\
&\mathsf{tree} \;=\; \&\{\mathsf{insert} : \mathsf{key} \supset \mathsf{tree}, \\
&\qquad\quad \mathsf{find} : \mathsf{key} \supset \mathsf{bool} \wedge \mathsf{tree}, \\
&\qquad\quad \mathsf{reduce} : \mathsf{int} \supset A \multimap \mathsf{int} \wedge \mathbf{1}, \\
&\qquad\quad \mathsf{tolist} : \mathsf{list} \otimes \mathbf{1}\}
\end{aligned}
$$

Figure 2.2: Queue, list and tree interface definition.

2.2 Curry-Howard Isomorphism

The relationship between typed λ-calculus and intuitionistic logic or, more generally, type systems for models of computation and formal proof calculi, is commonly known as the Curry-Howard Isomorphism. Haskell Curry [10] connected Hilbert-style deduction systems for implication with the types of the combinators of combinatory logic. William Howard [18] noted that the proof system referred to as natural deduction in its intuitionistic version can be interpreted as a typed branch of λ-calculus.

In recent work [5, 6], the Curry-Howard isomorphism has been extended to link typed π-calculus, through session types, with intuitionistic linear logic. It uses linear logic propositions as session types, proofs as concurrent programs, and cut elimination as computation. A variant also applies for classical [29] linear logic.

This correspondence uses *linear channels* with exactly two endpoints, dubbed *provider* and *client*. We omit the discussion of *shared channels* which are necessary to model replication ($!A$ in linear logic and $!P$ in π-calculus). A *process* may be the client of various channels, but will only provide along a sole channel. As a consequence of this, and bearing in mind that a channel is provided by a single process, a channel may be seen as a unique process identifier. A basic typing judgement is of the following form:

$$\$c_1{:}A_1, \ldots, \$c_n{:}A_n \vdash P :: (\$c : A)$$

The session types A and A_i dictate the communication behaviour along channels $\$c$[1] and $\$c_i$. P is a process providing along $\$c$ and using $\$c_1, \ldots, \$c_n$.

Under the Curry-Howard correspondence, we can assign process expressions to the session types mentioned in Figure 2.1, using syntactic constructs for processes, instead of π-calculus, to emphasize the interpretation of proofs as programs. Figure 2.3 shows this relationship, with the addition of `cut` and `id` expressions, used to *spawn* a new process and *forward* between a client and the process that uses it.

This extension to the Curry-Howard isomorphism was concretized in SILL [27], a functional programming language extended with session-typed concurrency. This language includes recursive types and uses a *contextual monad* to create higher-

[1] We refer to a channel variable with a \$ before, such as $\$c$, to differentiate from a value, x.

$$
\begin{array}{rcll}
P, Q, R & ::= & \$c \leftarrow P_{\$c} \; ; \; Q_{\$c} & \mathsf{cut} \quad \text{(spawn)} \\
& | & \$c \leftarrow \$d & \mathsf{id} \quad \text{(forward)} \\
& | & \mathsf{close}\ \$c \mid \mathsf{wait}\ \$c \; ; \; Q & \mathbf{1} \\
& | & \mathsf{send}\ \$c\ (\$d \leftarrow P_{\$d}) \; ; \; Q \mid \$e \leftarrow \mathsf{recv}\ \$c \; ; \; R_{\$e} & A \otimes B, A \multimap B \\
& | & \mathsf{send}\ \$c\ \$d \; ; \; Q & \text{derived form } A \otimes B,\ A \multimap B \\
& | & \mathsf{send}\ \$c\ M \; ; \; P \mid x \leftarrow \mathsf{recv}\ \$c \; ; \; Q_x & A \wedge B, A \supset B \\
& | & \$c.lab \; ; \; P \mid \mathsf{case}\ \$c\ \{lab_i \rightarrow Q_i\}_i & \&\{lab_i : A_i\}_i, \oplus\{lab_i : A_i\}_i
\end{array}
$$

Figure 2.3: Linear process expressions

order processes. Using a similar theoretical basis as SILL, Concurrent Linear Object-Orientation [3] is an object-oriented language that types objects and channels with session types and Concurrent C0 (CC0) [30] is a type-safe C-like language with contracts, implementing session-typed communication over channels.

Inspired by linear functional languages using session types proposed by Gay and Vasconcelos [14], Wadler's GV [28] is a session-typed functional language, using simply typed, linear λ-calculus extended with session-typed communication. It differs from SILL in the point that the language itself is linear, whereas SILL is based on a traditional λ-calculus augmented with a linear contextual monad.

2.2.1 Asynchronous Communication

Introduced by DeYoung et al. [11], and later refined by Pfenning and Griffith [22], the previous concepts were extended to include asynchronous communication, linking it with polarized logic.

Asynchronous communication requires that each linear channel has a message queue [14], associating it with the proof system via continuation channels [11]. Session typing guarantees that there is no send/receive conflict: when a process executes a send followed by a receive, it will not receive its own message. This requires that the message queue controls its direction, which can be achieved by setting a flag when enqueueing a message. We write q when the direction does not matter, $\overleftarrow{q}$ when the direction is from a provider to its client and $\overrightarrow{q}$ when going from a client to the provider.

The direction of communication is shaped by the *polarity* of each connective. Simply put, from the provider's perspective, a *positive type* is a sending action and a *negative type* coincides with a receiving action. As such, we can split the propositions in negative or positive, with a *shift* operator linking them. Figure 2.4 depicts the syntax for this polarized logic.

Positive propositions	A^+, B^+	::=	$\mathbf{1}$	terminate
		|	$A^+ \otimes B^+$	send channel of type A^+
		|	$A^+ \oplus B^+$	provide either A^+ or B^+
		|	$\tau \wedge B^+$	send value of type τ
		|	$\downarrow A^-$	send shift, then receive
Negative propositions	A^-, B^-	::=	$A^+ \multimap B^-$	receive channel of type A^+
		|	$A^- \,\&\, B^-$	offer choice between A^- or B^-
		|	$\tau \supset B^-$	receive value of type τ
		|	$\uparrow A^+$	receive shift, then send

Figure 2.4: Polarized Session Types

2.3 Substructural Operational Semantics

This documents presents operational semantics in the form of a Substructural Operational Semantics (SSOS) [23], based on linear logic programming. It consists of a compositional specification of the operational semantics by defining a predicate on the expressions of the language, using rules analogous to those of multiset rewriting [8]. These rules use linear implication ($\multimap$) to express a transition, where the pattern on the left represents a state that is consumed and converted to the one on the right. The patterns may be composed by multiple predicates, clustered together by the multiplicative conjunction ($\otimes$). Existential quantification is used to bound fresh variables.

As an example, let us consider the SSOS for the untyped λ-calculus. Expressions are represented by the syntax x, $\lambda x.e\ x$ and $\mathsf{app}\ e_1\ e_2$. We also have *frames*, which are either $\mathsf{app}_1\ e_2$, that contain the argument part of the application while the function

part is being evaluated, $\mathsf{app}_2\ v_1$, which contains the evaluated function and waits for the argument to be evaluated, or call, a frame without arguments that is not operationally meaningful, but used to mark a point where a function returns. Each state is represented by a context carrying linear propositions: $\mathsf{comp}(f)$, where f is a frame, followed by either a $\mathsf{eval}(e)$, representing an expression to be evaluated, $\mathsf{return}(v)$, which represents a value being returned, and $\mathsf{bind}\ X\ Y$, that can either be used to associate the parameter X with argument Y or to look up the associated value of a parameter in the course of an evaluation. These linear propositions can be made *persistent* by prefixing the proposition with a !. Figure 2.5 presents the rules for this case. More comprehensive examples are presented by Pfenning and Simmons [23], in their original formulation.

$$
\begin{array}{lcl}
\mathsf{eval}(X\ D) \otimes \mathsf{!bind}\ X\ V & \multimap & \{\mathsf{return}(V\ D)\} \\
\mathsf{eval}((\lambda x.e\ x)\ D) & \multimap & \{\mathsf{return}((\lambda x.e\ x)\ D)\} \\
\mathsf{eval}((\mathsf{app}\ e_1\ e_2)\ D) & \multimap & \{\exists d_1.\mathsf{comp}((\mathsf{app}_1\ e_2)\ D\ d_1) \otimes \mathsf{eval}(e_1\ d_1)\} \\
\mathsf{comp}((\mathsf{app}_1\ e_2)\ D\ D_1) \otimes \mathsf{return}(v_1\ D_1) & \multimap & \{\exists d_2.\mathsf{comp}((\mathsf{app}_2\ v_1)\ D\ d_2) \otimes \mathsf{eval}(e_2\ d_2)\} \\
\mathsf{comp}((\mathsf{app}_2\ (\lambda x.e\ x))\ D\ D_2 \otimes \mathsf{return}(v_2\ D_2) & \multimap & \{\exists y.\exists d_0.\ \mathsf{comp}(\mathsf{call}\ D\ d_0) \otimes \\
 & & \mathsf{eval}((e\ y)\ d_0) \otimes \mathsf{!bind}\ y\ v_2\} \\
\mathsf{comp}(\mathsf{call}\ D\ D_0) \otimes \mathsf{return}(V_0\ D_0) & \multimap & \{\mathsf{return}(V_0\ D)\}
\end{array}
$$

Figure 2.5: SSOS rules for the untyped λ-calculus.

We use $\Omega \to \Omega'$ to refer to a single rewrite, from state Ω to state Ω'. We write $\to^n$ to denote a sequence of n steps and $\to^+$ as the transitive closure of $\to$.

2.4 Performance Measures in Parallel Algorithms

Throughout this document we analyze the time complexity of implemented algorithms using the work-span model [9]. As the name suggests, this model uses the concepts of *work* and *span*, also dubbed *depth* [4] in some literature, to characterize the time complexity of an algorithm as a function of the number of processes used to run it. In simple terms, work measures the time used to run the computation on a single

processor, whereas span is defined as the time required to run the program on a theoretical machine with an unbounded number of processors.

In a more detailed explanation of this model, we can view a *multithreaded computation* as a directed acyclic graph, called a *computational dag*. The vertices of the *dag* represent instructions and the edges encode dependencies between instructions. Chains of one or more instructions without parallel control are grouped in *strands*. Parallel control is represented in the structure of the graph, for example, a strand with multiple successors indicates that new threads have been spawned.

Taking this definition of computation, we can describe work as the sum of the time taken by each strand and span as the longest time to execute the strands along any path in the graph. In other words, span can be seen as the *critical path* in graph.

During the experimental evaluation of the algorithms, we also use the *speedup* measure. Although usually defined as the ratio between the running time using 1 processor and P processors, we use it as a comparison between the two implemented models, blocking and non-blocking.

Chapter 3

Imperative Programming with Sessions Types

3.1 Introduction

This chapter introduces the main concepts of programming with Concurrent C0 (CC0). For a more in-depth description of CC0, we refer the reader to the original paper [30].

C0 [21] is an imperative programming language, closely resembling C, with the goal of having fully specified semantics that avoid C's undefined behaviour [1]. It is used in Carnegie Mellon University to learn the basics of imperative algorithms and data structures and compiler design. Its main features are: dynamically checked contracts (for example, *@requires*, *@ensures*, *@asserts* and *@loop_invariant*); complete memory distinction by separating pointers from arrays; and a garbage collector, which eliminates the need to free memory. C0 is compiled to C, generating human-readable code that is sent to a C compiler.

CC0 is a session-typed concurrent extension of C0. Its typing system is derived from the Curry-Howard interpretation of linear propositions as types, proofs as processes, and cut reduction as computation, which we mentioned in Section 2.2.

3.2 Concurrent C0

A program in CC0 is a collection of processes exchanging messages through channels. These processes are *spawned* by functions that return channels. The process that calls these spawning functions is called the client. The new process at the other end of the channel is called the provider, who is said to *offer a session* over the channel. A process is the provider of only one channel, but may be the client of multiple ones.

CC0's channels have a linear semantics: there is exactly one reference to the channel besides the provider's. This captures the behavior described in the previous paragraph: the process we called the client has the unique reference to the provided channel.

Messages are sent asynchronously: processes advance in parallel without waiting for acknowledgement of the sent message being received. Programmers must specify the protocol of message exchange using session types and the linear type system enforces concordance with this protocol [22].

3.2.1 Type Definition

Session type declaration in CC0 is done using the *choice* keyword, in a syntax inspired by structs in C. The actual session is enclosed by `<...>` and we distinguish input from output by using a `?` for input and a `!` for output. Internal and external choices are also discriminated by prefixing the keyword *choice* with `?` for external choice and `!` for internal. An empty `< >` indicates the end of a session.

Programs 3.1 and 3.2 illustrate how session types are defined in CC0.

In the example of the queue, the choice named *queue* is a choice between labels `Enq`, `Deq`, `IsEmpty` and `Dealloc`. It is only used after an output prefix (`?`), which means it always represents an external choice, made by the client and sent to the provider of the queue interface. In contrast, the choice *queue_elem* is an internal choice (always prefixed by `!`), which means the provider has to send label `None` (no element in the queue) or `Some` (some element in the queue).

The example of the binary search tree is similar, the choice *tree* is an external choice, expecting to receive the labels `Insert`, `Find`, `Reduce` and `ToList`. Receiving one of

```
choice queue {
  <?int; ?choice queue>  Enq;
  <!choice queue_elem>   Deq;
  <!bool; ?choice queue> IsEmpty;
  < >                    Dealloc;
};

choice queue_elem {
  <?choice queue>        None;
  <!int; ?choice queue> Some;
};
typedef <?choice queue> queue;
```

Code 3.1: Protocol definition of a queue.

```
choice list {
  <!int; !choice list>  Cons;
  < >                   Nil;
};
typedef <!choice list> list;

typedef int key;
typedef int reduce_fn(int x, key k, int y);
choice tree {
  <?key; ?choice tree>          Insert;
  <?key; !bool; ?choice tree>   Find;
  <?int; ?reduce_fn*; !int; >   Reduce;
  <!list; >                     ToList;
};
typedef <?choice tree> tree;
```

Code 3.2: Protocol definition of a binary search tree.

these labels induces a sequence of actions encoded by the session: an `Insert` is followed by receiving a key, which is simply an integer, and continuing as a *tree*; a `Find` is trailed by receiving a key, sending a boolean value and continuing as a *tree*; a `Reduce` is pursued by receiving an integer and a pointer, sending another integer and terminating; finally, receiving the label `ToList` leads to sending a *list* and terminating. On the other hand, the choice *list* is an internal choice, either sending the label `Cons`, if there is an element in the list, or `Nil`, otherwise.

The protocols exchange messages in two directions, from the client process to the provider process and vice-versa, resulting in each direction being conveyed independently through an external (*queue* and *tree*) and internal choice (*queue_ element* and *list*).

The session type is given from the provider's perspective, the client adhere's to the dual of the type: if the provider executes a receiving action, the client executes a sending one. The compiler determines statically if the operations indicated in the protocol are fulfilled in the correct order with the fitting type akin to the channel's session type.

3.2.2 Protocol Implementation

CC0 offers two distinct categories of functions: ones that return a basic type (*int*, etc.) and others that return a channel typed by a session type. The latter implements the interfaces, which could be any session type, as explained in the previous section. Programs 3.3 and A.1 show the functions that implement the types defined in Programs 3.1 and 3.2. To improve readability, we only present the code for binary search trees in Appendix A.

The function *empty* implements the behaviour of the process that represents the end of the queue, which does not hold any element. The instruction `switch ($q)` implements the behaviour of an external choice, making `$q` wait on the reception of a label. Each label has its own `case`, implementing the behaviour expected of an empty queue:

- When receiving the label `Enq`, which adds a new element to the queue, a new empty queue process is spawned, through the instruction `queue $e = empty()`. The new element to the queue is received using the instruction `int y = recv($q)`. The former empty queue now continues as a process holding one element, through the instruction `$q = elem(y, $e)`.
- If a dequeue is requested (label `Deq`), it sends the label `None` (using the instruction `$q.None`), stating that the queue is empty.
- When queried if it is empty (label `IsEmpty`), the process returns the value **true** (through the instruction `send($q, true)`).
- Upon receiving the label `Dealloc`, which requests the destruction of the queue, it simply closes the channel (`close($q)`).

The function that implements a process holding an element, $elem(x, \$r)$, has similar behaviour so we do not go into so much detail. We just point out how dequeueing

```
queue $q elem (int x, queue $r) {
  switch ($q) {
    case Enq:
      int y = recv($q);
      $r.Enq; send($r, y);
      $q = elem(x, $r);
    case Deq:
      $q.Some; send($q, x);
      $q = $r;     // forward request
    case IsEmpty:
      send($q, false);
      $q = elem(x, $r);
    case Dealloc:
      $r.Dealloc; wait($r);
      close($q);
 }
}

queue $q empty () {
  switch ($q) {
    case Enq:
      int y = recv($q);
      queue $e = empty();
      $q = elem(y, $e);
    case Deq:
      $q.None;
      $q = empty();
    case IsEmpty:
      send($q, true);
      $q = empty();
    case Dealloc:
      close($q);
  }
}
```

Code 3.3: Implementation of a queue with constant time enqueue and dequeue operations from the client's perspective.

an element works in this implementation: after receiving the label `Deq`, the process sends the label `Some`, stating that there is at least one element in the queue, followed by that element (instruction `send($q, x)`). It then executes a *forward* (`$q = $r`), terminating the process.

Linearity ensures that a parent process cannot terminate while it has running children, all references must be completely consumed. CC0 introduces the operation called *forwarding* [30], which allows a process to terminate before its children. In cases where a given node has exactly one child and it is offering a channel with the same session type as its child, if this node has no more work to do, it can be terminated and contracted in a way that the parent node and the child node can communicate between themselves without the node in the middle.

CC0 has the usual features of an imperative programming language, such as conditionals, loops, assignments and functions, extended by communication primitives. Figure 3.1 presents the core communication syntax.

P, Q	$::=$	$\$c = spawn(P)\ ;\ Q$	spawn
	$\mid$	$\$c = \$d$	forward
	$\mid$	$\mathsf{close}(\$c)$	send end and terminate
	$\mid$	$\mathsf{wait}(\$c)\ ;\ Q$	receive end
	$\mid$	$\mathsf{send}(\$c,\ e)\ ;\ Q$	send data (including channels)
	$\mid$	$x = \mathsf{recv}(\$c)\ ;\ Q$	receive data (including channels)
	$\mid$	$\$c.lab\ ;\ Q$	send label
	$\mid$	$\mathsf{switch}(\$c)\ \{lab_i \to P_i\}_i$	receive label

Figure 3.1: Communication syntax of CC0

3.2.3 Target Language

CC0 is compiled to a target language and linked with a runtime system, both written in C, responsible for implementing communication. The compiler checks if messages are being exchanged in the correct order, in agreement with the session type, and enforces linear use of channels.

Recall from Section 2.2.1, the session typing from CC0 uses polarised logic to maintain the direction of the communication. Positive polarity indicates that information is streaming from the provider and negative to the provider. A *shift* is used to swap polarities. The runtime system explicitly tracks the polarity of each channel and the compiler infers and inserts the minimal amount of shifts into the target language. This inference allows programmers to use CC0 without knowing about shifts at all.

The target language adds two new instructions to the communication syntax of CC0, used to change the direction of communication. Figure 3.2 delineates the new instructions.

$$
\begin{array}{rcll}
P, Q & ::= & \ldots & \\
& | & \mathsf{send}(\$c,\ \mathit{shift})\ ;\ Q & \text{send a shift} \\
& | & \mathit{shift} = \mathsf{recv}(\$c)\ ;\ Q & \text{receive a shift}
\end{array}
$$

Figure 3.2: Additional communication syntax for CC0's target language

3.3 Operational Semantics

The operational semantics for CC0 is expressed using a Substructural Operational Semantics (SSOS), first presented by Pfenning and Griffith [22] for an asynchronous version of SILL [27], a functional language. We here repurpose it for an imperative language, which requires us to extend it with variable state, using memory cells, local to each process. Refer to Section 2.2.1 for a very brief explanation of the theory behind the queue concept in the configuration, as well as the original sources for this theory.

Configurations (Figure 3.3) describe executing processes, message queues connecting processes (one for each channel), and local storage cells. In this definition, $\mathsf{proc}(\$c, P)$

$$
\begin{array}{rrcl}
\text{Configurations} & \Omega & ::= & \cdot \\
& & | & \mathsf{queue}(\$c, q, \$d), \Omega \\
& & | & \mathsf{proc}(\$c, P), \Omega \\
& & | & \mathsf{cell}(\$c, x, v), \Omega
\end{array}
$$

Figure 3.3: Configuration definition, using multiset notation.

is the state of a process executing program P, offering along channel $\$c$. The message queue is represented by $\mathsf{queue}(\$c, q, \$d)$, which connects processes offering along $\$d$ with a client using $\$c$. The memory cell $\mathsf{cell}(\$c, x, v)$ holds the state of variable x with value v, in the process offering along channel $\$c$. The semantic rules we present in Table 3.1 are only for communication, thus will not use this memory cell predicate, only create it. The memory cell is used when evaluating an expression, details of which we omit here.

Queues always have a defined direction, depending whether they are filled from the provider or the client, as can be seen in Figure 3.4. They need to be initiated with the correct direction and the operational semantics maintains its correctness afterwards. We write m as a generalization for a message like a data value, label or channel.

$$\begin{array}{lrcl} \text{Queue filled by provider} & \overleftarrow{q} & ::= & \overleftarrow{\cdot} \mid \overleftarrow{m \cdot q} \mid \overleftarrow{\mathsf{end}} \mid \overleftarrow{\mathsf{shift}} \\ \text{Queue filled by client} & \overrightarrow{q} & ::= & \overrightarrow{\mathsf{shift}} \mid \overrightarrow{q \cdot m} \mid \overrightarrow{\cdot} \end{array}$$

Figure 3.4: Queue definition, filled by provider or client.

Table 3.1 shows the operational semantics of CC0, adapted from [22].

$$\begin{array}{lcl}
\mathsf{data_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{send}(\$d, v)\ ;\ P) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot v}, \$d) \otimes \mathsf{proc}(\$d, P)\} \\
\mathsf{data_r} & : & \mathsf{proc}(\$e, x = \mathsf{recv}(\$c)\ ;\ Q) \otimes \mathsf{queue}(\$c, \overleftarrow{v \cdot q}, \$d) \\
& & \quad \multimap \{\exists x.\, \mathsf{proc}(\$e, Q) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{cell}(\$e, x, v)\} \\
\mathsf{shift_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{send}(\$d, \mathsf{shift})\ ;\ P) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot \mathsf{shift}}, \$d) \otimes \mathsf{proc}(\$d, P)\} \\
\mathsf{shift_r} & : & \mathsf{proc}(\$e, \mathsf{shift} = \mathsf{recv}(\$c)\ ;\ Q) \otimes \mathsf{queue}(\$c, \overleftarrow{\mathsf{shift} \cdot q}, \$d) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d)\} \\
\mathsf{label_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \$d.lab\ ;\ P) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot lab}, \$d) \otimes \mathsf{proc}(\$d, P)\} \\
\mathsf{label_r} & : & \mathsf{proc}(\$e, \mathsf{switch}(\$c)\{lab_i \to P_i\}, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{lab_j \cdot q}, \$d) \\
& & \quad \multimap \{\mathsf{proc}(\$e, P_j) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d)\} \\
\mathsf{close} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{close}(\$d)) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot \mathsf{end}}, _)\} \\
\mathsf{wait} & : & \mathsf{proc}(\$e, \mathsf{wait}(\$c)\ ;\ Q) \otimes \mathsf{queue}(\$c, \overleftarrow{\mathsf{end}}, _) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q)\} \\
\mathsf{fwd_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \$d = \$e) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot \mathsf{fwd}}, \$e)\} \\
\mathsf{fwd_r} & : & \mathsf{proc}(\$d, P(\$c)) \otimes \mathsf{queue}(\$c, \overleftarrow{\mathsf{fwd}}, \$e) \\
& & \quad \multimap \{\mathsf{proc}(\$d, P(\$e))\}
\end{array}$$

$$\begin{array}{lll} \mathsf{spawn} & : & \mathsf{proc}(\$c, \$d = P(args)\ ;\ Q) \\ & & \quad \multimap \{\exists \$d.\ \mathsf{proc}(\$c, Q) \otimes \mathsf{queue}(\$c, \overleftarrow{\cdot}, \$d) \otimes \mathsf{proc}(\$d, P(args))\} \end{array}$$

Table 3.1: Operational semantics of Concurrent C0.

We can group rules under several categories. The sending ones, data_s, shift_s, label_s, append a message, which can be some data, a label or a shift, to the end of the message queue. We show the rules of a process sending through the channel that it is providing, so we require the queue to be pointing to the client.

Two other rules also append a message to the queue, they are close and fwd_s. These two are terminating rules, when a process executes one of them, it will be concluded. Functionally, they are very similar, both decrease the number of processes in the configuration, but the close rule changes the channel on the queue predicate to the empty channel _, whereas fwd_s replaces it with another, already existing, one.

The receiving rules, data_r, shift_r, label_r, wait and fwd_r, all fetch a message from the queue, blocking the execution until the message is available, but the consequent behavior is unique to each function:

- data_r binds a new variable to receive the value contained in the message, creating a memory cell to hold it.
- shift_r changes the direction of the queue.
- label_r branches to the correct case, depending on the label received.
- wait closes the queue of the terminating process.
- fwd_r not only closes the queue of the terminating process, but also updates the references it had of the terminating process to point to its new provider.

We presented the rules from the perspective of a process receiving a message from one of its providers, so all the rules require a queue pointing to the client.

The last rule, spawn, binds a new process and creates a queue connecting the new process with its client.

3.3.1 Cost Semantics

We now introduce a cost model for the operational semantics of CC0, using the span-work performance metric. We assign each operation a weight, but only for communication, ignoring internal computation. Although this may not lead to an entirely realistic measure for the complexity of an algorithm, it still makes for a useful abstract one. Moreover, in many of our examples communication costs dominate performance.

In the cost semantics we maintain a span s for each executing process which represents the earliest global time (counting only communication steps) at which the process could have reached its current state. Because messages can only be received after they have been sent, each message is tagged with the time at which it is sent, and the recipient takes the maximum between its own span and the span carried by the message. Except for operations using *shifts* or *forwards*, which are not explicit communications by the programmer, each call to a communication function increases the span by one unit.

The work w is determined individually by each process. As with span, all operations except the ones using *shifts* or *forwards* increase work by one unit. Although each message also carries the work of the sending process, this work is ignored unless the message is an end or a forward (fwd). In these two cases, the receiving process adds the work carried by the message to its own, to propagate the work of the sending process, which is being terminated.

To accomplish these changes, it is necessary to make a slight modification to the definitions of configuration and how the queues are filled, which are presented in Figures 3.5 and 3.6. The former needs to take into account that processes must track their span and work and the latter needs to encompass span and work in the messages.

$$
\begin{array}{rrcl}
\text{Configurations} & \Omega & ::= & \cdot \\
& & \mid & \mathsf{queue}(\$c, q, \$d), \Omega \\
& & \mid & \mathsf{proc}(\$c, P, s, w), \Omega \\
& & \mid & \mathsf{cell}(\$c, x, v), \Omega
\end{array}
$$

Figure 3.5: Configuration definition, using multiset notation, in the cost semantics.

$$\begin{array}{lrcl} \text{Queue filled by provider} & \overleftarrow{q} & ::= & \overleftarrow{\cdot} \mid \overleftarrow{(m,s,w)\cdot q} \mid \overleftarrow{(\mathsf{end},s,w)} \mid \overleftarrow{(\mathsf{shift},s,w)} \\ \text{Queue filled by client} & \overrightarrow{q} & ::= & \overrightarrow{(\mathsf{shift},s,w)} \mid \overrightarrow{q\cdot(m,s,w)} \mid \overrightarrow{\cdot} \end{array}$$

Figure 3.6: Queue definition, filled by provider or client, in the cost semantics.

Table 3.1 instruments the operational semantics from Table 3.1, to include the costs discussed above.

$$\begin{array}{lcl}
\mathsf{data_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{send}(\$d, v)\ ;\ P, s, w) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q\cdot(v, s+1, w+1)}, \$d) \otimes \mathsf{proc}(\$d, P, s+1, w+1)\} \\
\mathsf{data_r} & : & \mathsf{proc}(\$e, x = \mathsf{recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1)\cdot q}, \$d) \\
& & \quad \multimap \{\exists x.\, \mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1)+1, w+1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{cell}(\$e, x, v)\} \\
\mathsf{shift_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{send}(\$d, \mathsf{shift})\ ;\ P, s, w) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q\cdot(\mathsf{shift}, s, w)}, \$d) \otimes \mathsf{proc}(\$d, P, s, w)\} \\
\mathsf{shift_r} & : & \mathsf{proc}(\$e, \mathsf{shift} = \mathsf{recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{shift}, s_1, w_1)\cdot q}, \$d) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d)\} \\
\mathsf{label_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \$d.lab\ ;\ P, s, w) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q\cdot(lab, s+1, w+1)}, \$d) \otimes \mathsf{proc}(\$d, P, s+1, w+1)\} \\
\mathsf{label_r} & : & \mathsf{proc}(\$e, \mathsf{switch}(\$c)\{lab_i \to P_i\}, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(lab_j, s_1, w_1)\cdot q}, \$d) \\
& & \quad \multimap \{\mathsf{proc}(\$e, P_j, \mathsf{max}(s, s_1)+1, w+1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d)\} \\
\mathsf{close} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{close}(\$d), s, w) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q\cdot(\mathsf{end}, s+1, w+1)}, _)\} \\
\mathsf{wait} & : & \mathsf{proc}(\$e, \mathsf{wait}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{end}, s_1, w_1)}, _) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1)+1, w+w_1+1)\} \\
\mathsf{fwd_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \$d = \$e, s, w) \\
& & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q\cdot(\mathsf{fwd}, s, w)}, \$e)\} \\
\mathsf{fwd_r} & : & \mathsf{proc}(\$d, P(\$c), s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{fwd}, s_1, w_1)}, \$e) \\
& & \quad \multimap \{\mathsf{proc}(\$d, P(\$e), \mathsf{max}(s, s_1), w+w_1)\} \\
\mathsf{spawn} & : & \mathsf{proc}(\$c, \$d = P(args)\ ;\ Q, s, w) \\
& & \quad \multimap \{\exists \$d.\, \mathsf{proc}(\$c, Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{\cdot}, \$d) \otimes \mathsf{proc}(\$d, P(args), s, 0)\}
\end{array}$$

Table 3.2: Cost Semantics for Concurrent C0.

Sending rules, except shift_s, increase span and work the process by one unit and create a message with span and work equal to their new values.

A process executing a receiving function must synchronize its span, so it takes the maximum of its current span and the one carried by the received message. In addition

to this, wait and fwd_r, also add the work carried by the received message to the work of the process.

When a process is being spawned, it starts with 0 work and span equal to the process that spawned it. This rule does not count as communication, so the span and work of the original process remain unaltered.

Chapter 4

Non-blocking Receive

4.1 Introduction

In this chapter, we present the main contribution of this thesis, a new model for message reception whose goal is to block the execution of the process to wait for a message only when the data contained in this message is necessary to continue the execution. We propose a change to the cost semantics, a modified target language for CC0, as well as a translation from the original target language to the non-blocking one, and some theoretical results regarding this model.

4.2 Non-blocking Receive

Receiving a message in CC0 blocks the execution of the program, a behavior matching the operational semantics. A receiving function (`recv` or `wait`) only succeeds when it is possible to retrieve the corresponding message from the queue of the associated channel.

This model for message reception may not be the optimal choice for some algorithms, where an arbitrary imposed order on messages received on two different channels might prevent other computation to proceed. An extreme case is when a received value is not actually ever needed.

Our alternative follows two principles. One, the difference should be invisible to the programmer who should not need to know exactly when a message is received. Second, the implementation still needs to adhere to the protocol defined by the session type, which forces the order of sends and receives.

Our model involves postponing reception as much as possible, by interpreting receives as a request for a message. The request is saved on the channel until a synchronization is necessary. A synchronization is triggered when some data contained on any of the requests is required to continue execution. For example, if a process requested a shift from a channel, a synchronization is required to correct the polarity of the channel, that is, drain the message queue in order to change the direction of communication using the same queue.

The requests are handled in the order they were made, which guarantees that the session type is still being respected. Furthermore, all these changes only occur in an intermediate language, allowing CC0 to keep the same source-level syntax. Figure 4.1 presents the new constructs used to introduce this non-blocking model into the target language.

$$
\begin{array}{lcll}
P, Q & ::= & \ldots & \\
& | & \mathsf{async\_wait}(\$c);\ Q & \text{request an } \mathsf{end} \\
& | & x = \mathsf{async\_recv}(\$c);\ Q & \text{request data} \\
& | & \mathit{shift} = \mathsf{async\_recv}(\$c);\ Q & \text{request a } \mathsf{shift} \\
& | & \mathsf{sync}(\$c, x);\ Q & \text{synchronize variable} \\
& | & \mathsf{sync}(\$c, \mathsf{shift});\ Q & \text{synchronize } \mathsf{shift} \\
& | & \mathsf{sync}(\$c, \mathsf{end});\ Q & \text{synchronize } \mathsf{end}
\end{array}
$$

Figure 4.1: Non-blocking syntax for CC0's target language

Programs 4.1 and A.2 present the non-blocking versions of a queue and a binary search tree (in appendix A), respectively. They are presented in a simplified version target language, where shifts are included but without the heavy syntax introduced by the compilation to C. More details on this difference can be found in Chapter 5, where we present the implementation of this system, using the actual low level syntax of the target language.

```
queue $q elem (int x, queue $r) {
  switch ($q) {
    case Enq:
      int y = async_recv($q);
      $r.Enq;
      sync($q, y); send($r, y);
      $q = elem(x, $r);
    case Deq:
      shift = async_recv($q);
      sync($q, shift);
      $q.Some; send($q, x); send($q, shift);
      $q = $r;     // forward request
    case IsEmpty:
      shift = async_recv($q);
      sync($q, shift);
      send($q, false); send($q, shift);
      $q = elem(x, $r);
    case Dealloc:
      shift = async_recv($q);
      $r.Dealloc; send($r, shift);
      async_wait($r);
      sync($r, end); sync($q, shift);
      close($q);
  }
}

queue $q empty () {
  switch ($q) {
    case Enq:
      int y = async_recv($q);
      queue $e = empty();
      sync($q, y);
      $q = elem(y, $e);
    case Deq:
      shift = async_recv($q);
      sync($q, shift);
      $q.None; send($q, shift);
      $q = empty();
    case IsEmpty:
      shift = async_recv($q);
      sync($q, shift);
      send($q, true); send($q, shift);
      $q = empty();
    case Dealloc:
      shift = async_recv($q);
      sync($q, shift);
      close($q);
  }
}
```

Code 4.1: Non-blocking implementation of a queue, in CC0's simplified target language.

4.3 Cost Semantics

The new non-blocking model introduces changes in both operational semantics and cost semantics. We present only the latter, which includes the changes to the former.

Recall from Section 3.3.1, our definition of configuration has three predicates: $\mathsf{proc}(\$c, P, s, w)$ represents a process offering along channel $\$c$, executing program P, with span s and work w; queue represents the message queue, connecting two processes; and $\mathsf{cell}(\$c, x, v)$ holds the state of variable x with value v, in the process offering along channel $\$c$.

This definition of configuration needs to be updated, modifying the queue predicate to include the queue of requests. The new configuration and the definition of queue of requests are shown in Figure 4.2.

$$
\begin{array}{rrcl}
\text{Configurations} & \Omega & ::= & \cdot \\
& & \mid & \mathsf{queue}(\$c, q, \$d, r), \Omega \\
& & \mid & \mathsf{proc}(\$c, P, s, w), \Omega \\
& & \mid & \mathsf{cell}(\$c, x, v), \Omega \\
\text{Request queue} & r & ::= & \cdot \mid x \cdot r \mid \mathsf{end} \mid \mathsf{shift}
\end{array}
$$

Figure 4.2: Definitions of configuration in a non-blocking environment and the request queue.

We define a receive request to increase both span and work by one unit. Upon synchronising any request, the span must also be synchronized with the value carried by the message. As before, during the synchronization, any end or fwd message requires the addition of the work contained in the message to the work of the receiving process. Table 4.1 presents the rules for the new instructions added with the non-blocking receive model. The rules for the other instructions remain unchanged.

$$
\begin{array}{lll}
\mathsf{label\_r} & : & \mathsf{proc}(\$e, \mathsf{switch}(\$c)\{lab_i \to P_i\}, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(lab_j, s_1, w_1) \cdot q}, \$d, \cdot) \\
& & \quad \multimap \{\mathsf{proc}(\$e, P_j, \mathsf{max}(s, s_1) + 1, w + 1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, \cdot)\} \\
\mathsf{data\_async\_r} & : & \mathsf{proc}(\$e, x = \mathsf{async\_recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \\
& & \quad \multimap \{\exists x.\, \mathsf{proc}(\$e, Q, s + 1, w + 1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r \cdot x)\} \\
\mathsf{shift\_async\_r} & : & \mathsf{proc}(\$e, \mathsf{shift} = \mathsf{async\_recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r \cdot \mathsf{shift})\} \\
\mathsf{wait\_async} & : & \mathsf{proc}(\$e, \mathsf{async\_wait}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q, s + 1, w + 1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r \cdot \mathsf{end})\}
\end{array}
$$

$$\begin{aligned}
\mathsf{sync_data\ 1} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ x)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d, y \cdot r) \\
&\quad \multimap \{\mathsf{proc}(\$e, \mathsf{sync}(\$c,\ x)\ ;\ Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \otimes \mathsf{cell}(\$e, y, v)\} \\
\mathsf{sync_data\ 2} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ x)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d, x \cdot r) \\
&\quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \otimes \mathsf{cell}(\$e, x, v))\} \\
\mathsf{sync_shift\ 1} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{shift})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d, y \cdot r) \\
&\quad \multimap \{\mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{shift})\ ;\ Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \otimes \mathsf{cell}(\$e, y, v)\} \\
\mathsf{sync_shift\ 2} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{shift})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{shift}, s_1, w_1)}, \$d, \mathsf{shift}) \\
&\quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overrightarrow{\cdot}, \$d, \cdot)\} \\
\mathsf{sync_wait\ 1} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d, y \cdot r) \\
&\quad \multimap \{\mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, \mathsf{max}(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \otimes \mathsf{cell}(\$e, y, v)\} \\
\mathsf{sync_wait\ 2} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{end}, s_1, w_1)}, \$d, \mathsf{end}) \\
&\quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1), w + w_1)\} \\
\mathsf{sync_fwd} &: \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ X)\ ;\ Q(\$c), s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{fwd}, s_1, w_1)}, \$d, r) \otimes \mathsf{queue}(\$d, \overleftarrow{q}, \$f, \cdot) \\
&\quad \multimap \{\mathsf{proc}(\$e, \mathsf{sync}(\$d,\ X)\ ;\ Q(\$d), \mathsf{max}(s, s_1), w + w_1) \otimes \mathsf{queue}(\$d, \overleftarrow{q}, \$f, r)\}
\end{aligned}$$

Table 4.1: Operational and cost semantics rules for the non-blocking receive model.

The non-blocking receiving rules, data_async_r, shift_async_r and wait_async, simply add a new request to the request queue, independently of what is currently in the message queue. The rule data_async_r, which is receiving data on a variable, also needs to bind this variable, through the existential quantifier $\exists$.

From a cost point of view, these rules only induce an increase of one unit in both span and work, except shift_async_r who deals with a shift so does not change these metrics, as discussed previously.

The synchronization rules possess two cases each. Since the requests are stored in the queue, and to preserve the order of messages, if we are synchronizing a variable but there are other requests before that variable, these need to be handled first. A synchronization is only successful when the message is present in the message queue, and the type of this message must match the type of the request. This means that, if, for example, we are synchronizing a variable, we cannot get a message with a shift, end or a label. Synchronizing a shift or end empties the request queue completely, due to the nature of these operation: if we synchronize a shift, then we will start sending; if we synchronize a end, then we will close that client channel. This matches the behavior of the rules shift_r and wait from Table 3.1.

Looking at synchronization from the perspective of the cost semantics, synchronizing always requires us to also synchronize the span of the process with the one from message. If a process is synchronizing a end request, the work carried by the message is added to the process' work, otherwise the work of the process is unchanged.

When synchronizing, it is possible that a fwd message is received from the queue. When this happens, the queue of requests is moved to the queue predicate of the new client channel. The cost of this operation is equal, in both work and span, to the cost of synchronizing a end.

We also repeated the rule to receive a label, which, operationally, does not suffer any change, but we want to emphasize that this rule requires an empty queue of requests to be successful. Receiving a label induces a change in the instructions to be executed, so we opted to keep it as a blocking function. Alternatively, we could have changed it to a non-blocking functions, but it would need to always follow this sequence of instructions: `label = async_recv($c); sync($c, label)`. As a consequence of receiving a label using a blocking function, to preserve the order of messages received, we need to completely synchronize the channel before receiving a label.

4.4 Translation

Previously in this chapter, we have mentioned that, to accomplish our non-blocking model, we introduced a new set of instructions. To establish the connection between CC0's target language and our modified one, we developed a translation from the former to the latter. We represent this translation using the $[\![]\!]$ notation. The translation is applied independently to each function in the whole program.

Our translation uses an auxiliary table of requests, σ, to determine where to include the synchronization functions. This table is local to each process, it is a list of pairs, $[(X, \$d)]$, where $\$d$ is the channel to which the request was done and X is either a variable, shift or end. The way this table is built is clarified by the rules in Table 4.2, but consider the example of the non-blocking reception of a value, through the instruction `x = async_recv($d)`. This instruction would add the pair $(x, \$d)$ to the end of the table of requests, which conceptually works as a queue, similar to the request queue of the operational semantics.

Our translation function, $[\![]\!]$, takes as an argument a pair, $(\mathsf{Instruction}, \sigma)$, and returns a new sequence of instructions and a new table of requests. Instruction represents not only communication syntax but also flow control directives and an artificial no operation instruction, which is simply skipped by the runtime environment. The rules for the translation are included in Table 4.2. The definitions of all the auxiliary functions are presented in Appendix B.

We use the operator $\cup$, borrowed from set notation, to represent union of two lists of requests. The list on the right side of the operator is appended to the list on left and repeated tuples are removed. We also use the set difference operator, $\setminus$, to remove the pairs of the list of requests on the right from the list on left.

receive-shift: $[\![(\mathsf{shift} = \mathsf{recv}(\$d),\ \sigma)]\!] = (\mathsf{shift} = \mathsf{async\_recv}(\$d),\ \sigma \cup [(\mathsf{shift}, \$d)])$

receive-value: $[\![(x = \mathsf{recv}(\$d),\ \sigma)]\!] = (x = \mathsf{async\_recv}(\$d),\ \sigma \cup [(x, \$d)])$

wait: $[\![(\mathsf{wait}(\$d),\ \sigma)]\!] = (\mathsf{async\_wait}(\$d),\ \sigma \cup [(\mathsf{end}, \$d)])$

receive-label: $[\![(\mathsf{switch}(\$d)\{lab_i \rightarrow P_i\},\ \sigma)]\!] = (sync\_instructions\ ;\ \mathsf{switch}(\$d)\{lab_i \rightarrow P'_i\},\ \bigcup_{i \in I} \sigma_i)$

$$\text{where } l = \mathsf{check\_shift}\ \sigma\ \$d$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma$$
$$(P'_i, \sigma_i) = [\![(P_i,\ \sigma')]\!]$$

spawn: $[\![(\$d = f(\mathsf{args}),\ \sigma)]\!] = (sync\_instructions\ ;\ \$d = f(\mathsf{args}),\ \sigma')$

$$\text{where } l_i = \mathsf{check\_arg}\ \sigma\ \mathsf{arg_i}$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ (\bigcup_{i \in I} l_i)\ \sigma$$

close: $[\![(\mathsf{close}(\$d),\ \sigma)]\!] = (sync\_instructions\ ;\ \mathsf{close}(\$d),\ [])$

$$\text{where } l = \mathsf{sync\_all}\ \sigma$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma$$

send-exp: $[\![(\mathsf{send}(\$d, e),\ \sigma)]\!] = (sync\_instructions\ ;\ \mathsf{send}(\$d, e),\ \sigma')$

$$\text{where } l_1 = \mathsf{check\_shift}\ \sigma\ \$d$$
$$l_2 = \mathsf{check\_exp}\ \sigma\ e$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ (l_1 \cup l_2)\ \sigma$$

send-shift: $[\![(\mathsf{send}(\$d, \mathsf{shift}),\ \sigma)]\!] = (sync\_instructions\ ;\ \mathsf{send}(\$d, \mathsf{shift}),\ \sigma')$

$$\text{where } l = \mathsf{check\_shift}\ \sigma\ \$d$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma$$

send-label: $[\![(\$d.lab,\ \sigma)]\!] = (sync\_instructions\ ;\ \$d.lab,\ \sigma')$

$$\text{where } l = \mathsf{check\_shift}\ \sigma\ \$d$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma$$

forward: $[\![(\$d = \$e,\ \sigma)]\!] = (sync\_instructions\ ;\ \$d = \$e,\ [])$

$$\text{where } l = \mathsf{sync\_all}\ \sigma$$
$$(sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma$$

$$
\begin{array}{lll}
\text{sequence:} & [\![(P\ ;\ Q,\ \sigma)]\!] = (P'\ ;\ Q',\ \sigma'') & \\
& \quad \text{where}\ (P', \sigma') = [\![(P, \sigma)]\!] & \\
& \qquad\ \ (Q', \sigma'') = [\![(Q, \sigma')]\!] & \\
\text{assignment:} & [\![(x = e,\ \sigma)]\!] = (sync\_instructions\ ;\ x = e,\ \sigma) & \\
& \quad \text{where}\ l_1 = \mathsf{check\_exp}\ \sigma\ x & \\
& \qquad\ \ l_2 = \mathsf{check\_exp}\ \sigma\ e & \\
& \qquad\ \ (sync\_instructions, \sigma') = \mathsf{generate\_sync}\ (l_1 \cup l_2)\ \sigma & \\
\text{if:} & [\![(\text{if}\ (b)\ \text{then}\ P\ \text{else}\ Q,\ \sigma)]\!] = (sync\_instructions\ ;\ \text{if}\ (b)\ \text{then}\ P\ \text{else}\ Q,\ \sigma_1 \cup \sigma_2) & \\
& \quad \text{where}\ l = \mathsf{check\_exp}\ \sigma\ b & \\
& \qquad\ \ (sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma & \\
& \qquad\ \ (P', \sigma_1) = [\![(P, \sigma')]\!] & \\
& \qquad\ \ (Q', \sigma_2) = [\![(Q, \sigma')]\!] & \\
\text{while :} & [\![(\text{while}\ (b)\ \text{do}\ P,\ \sigma)]\!] = (sync\_instructions\ ;\ \text{while}\ (b)\ \text{do}\ P',\ \sigma_2) & \\
& \quad \text{where}\ l = \mathsf{check\_exp}\ \sigma\ b & \\
& \qquad\ \ (sync\_instructions, \sigma') = \mathsf{generate\_sync}\ l\ \sigma & \\
& \qquad\ \ \sigma_1 = \mathsf{loop\_carried\_req}\ P\ \sigma' & \\
& \qquad\ \ (P', \sigma_2) = [\![(P, \sigma' \cup \sigma_1)]\!] & \\
\text{func:} & [\![(f(args),\ \sigma)]\!] = (sync\_instructions\ ;\ f(args),\ \sigma') & \\
& \quad \text{where}\ l_i = \mathsf{check\_arg}\ \sigma\ arg_i & \\
& \qquad\ \ l' = \mathsf{check\_waits}\ \sigma & \\
& \qquad\ \ (sync\_instructions, \sigma') = \mathsf{generate\_sync}\ (\bigcup_{i \in I} l_i) \cup l'\ \sigma & \\
& \qquad\ \ \text{if}\ \sigma' \neq \{\} & \\
& \qquad\ \ \text{then}\ \mathsf{recurse}\ f\ \sigma' &
\end{array}
$$

Table 4.2: Translation scheme from blocking to non-blocking versions.

To translate a receive-value, a receive-shift or a wait, there is no synchronization needed, the first is a consequence of defining the receive rule to bind a new variable. The functions async_recv and async_wait produce a request so we have to include a pair in the table of requests, indicating both the variable or message type and the channel tied to the request.

In contrast to the other receives, receiving a label requires us to check first if the channel is ready (through check_shift), to ensure that messages arrive in the order specified by the session type. It recursively calls the translation function to each `case` and returns a request table that is the union of the tables returned by each `case`, which is relevant for recursive session types that may carry requests from one iteration to the next.

The translations for close and forward are equal. To close a channel we need to synchronize it and all its clients, so sync_all generates a list with a number of pairs equal to the number of requests in σ. The auxiliary function generate_sync takes this list and returns the minimum sequence of instructions needed to synchronize the channel. If there is no need to perform any synchronization, generate_sync produces the no operation instruction we mentioned as part of the definition of Instruction.

The translation of a spawn instruction simply needs to check if any argument needs to be synchronized, by applying the auxiliary function check_arg to each argument. The sending functions are all similar, they need to check if the channel is in the correct direction and, in the case of send-expression, the variables in the expression are all synchronized.

There are two cases for the translation of an assignment, we presented only the most general one. An assignment can either be bounding or not, the rule presented in Table 4.2 is relative to the non-bounding case. Since x may not be a new variable, it might have a request pending on its value, thus we need to check it for requests, in addition to the expression.

On flow control instructions, the translation is applied recursively. If it is a while loop, we need to be mindful of loop carried requests, which requires us to go over the body of the loop twice. The function loop_carried_req determines the table of requests after the execution of P, which is then passed as argument for the recursive translation of P.

Recall that CC0 possesses two types of functions: one that returns a basic data type, such as `int`, and others that return a channel. The translation of the former is a special case of the assignment, the latter can also be divided in two cases. The first is spawning a new process, which we already mentioned, and the second is changing the context of the current process by entering a new function that returns a channel of the same type, corresponding to the rule *func* in Table 4.2.

Changing the context of the function, requires that all the arguments are synchronized, because references to variables declared locally are lost when swapping to a new procedure. This is also the case for client channels that are waiting to be terminated. This is handled by the auxiliary function check_waits, that sifts through the requests to find a wait request. Any open client channels are passed on to the new function, including ones with pending requests, which do not need to be synchronized beforehand.

If there is any pending request, the translation is applied recursively to this new function, with the non-empty table of requests given as argument, through the auxiliary function recurse. This auxiliary function calls another translation function with rules very similar to the ones presented in this section but with the difference that it only consumes requests and does not generate them, stopping when the table of requests is empty. The recursive call is applied to the already-translated version of the other function.

4.5 Impact of Non-Blocking Receive

In this section, we discuss some properties about our non-blocking model. We argue that the span of any process in a blocking configuration is greater or equal than the correspondent process in the non-blocking configuration. We also claim that the total work of a blocking configuration is equal to the work of the non-blocking configuration. Both these results are included in Theorem 1.

Throughout this section, we will assume that all computations are terminating, satisfying both global and local progress.

As a general rule, when we want to note explicitly that we are referring to a process in a non-blocking context, we use the translation notation, $[\![]\!]$. So, for example, when writing non-blocking operational semantics' transitions, we use $\mathsf{proc}([\![\$c]\!], [\![P]\!], s, w)$ to contrast with the blocking operational semantics, which would be written as $\mathsf{proc}(\$\mathrm{c}, P, s, w)$.

We define the operator $\in$ for processes and configurations as: $\$c \in \Omega \implies \exists \mathsf{proc}(\$d, P, s, w) \in \Omega.\ \$c = \$d$.

Definition 1 *We define a predicate,* $\mathsf{messages}(\Omega)$*, which, given a configuration, returns a list of all messages inside a queue of messages, in each* queue *predicate in the configuration* Ω*.*

Definition 2 *Given a configuration* Ω*, we define* $\mathsf{work}(\Omega)$ *as the sum of the work of all processes in the configuration and the sum of the work carried by each message whose data is* end *or* fwd*:*

$$\mathsf{work}(\Omega) = \sum_{\mathsf{proc}(\$c, P, w_{\$c}) \in \Omega} w_{\$c} \; + \sum_{m \in \mathsf{messages}(\Omega)} \begin{cases} \mathsf{work}(m) & \textit{if } \mathsf{data}(m) = \mathsf{end} \vee \mathsf{data}(m) = \mathsf{fwd} \\ 0 & \textit{otherwise} \end{cases}$$

Definition 3 *Given a blocking configuration, Ω, and its non-blocking correspondent, $[\![\Omega]\!]$, we define:* $\mathsf{span}([\![\Omega]\!]) \leq \mathsf{span}(\Omega)$ *if and only if* $\forall \mathsf{proc}(\$c, P, s_{\$c}, w_{\$c}) \in \Omega.\ \exists \mathsf{proc}([\![\$c]\!], [\![P]\!], s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \in [\![\Omega]\!].\ s_{[\![\$c]\!]} \leq s_{\$c}$.

We define an inverse function of the translation, called pgr_readback, that compiles a non-blocking program to a blocking one. The rules of this inverse compilation are presented in Table 4.3. They are, mostly, the identity function, with the exception of requests, which are compiled to the blocking counterparts, and synchronizations, who are replaced by a new *skip* rule.

We extend the blocking semantics with an artificial skip rule, used only on the following proofs. Executing a skip changes neither the span nor the work of a process, and the process does not interact with any message queues.

receive-shift:	$\mathsf{pgr\_readback}(\mathsf{shift} = \mathsf{async_recv}(\$d)) = \mathsf{shift} = \mathsf{recv}(\$d)$
receive-value:	$\mathsf{pgr\_readback}(x = \mathsf{async\_recv}(\$d)) = x = \mathsf{recv}(\$d)$
wait:	$\mathsf{pgr\_readback}(\mathsf{async\_wait}(\$d)) = \mathsf{wait}(\$d)$
receive-label:	$\mathsf{pgr\_readback}(\mathsf{switch}(\$d)\{lab_i \rightarrow P_i) = \mathsf{switch}(\$d)\{lab_i \rightarrow P'_i\}$ where $P'_i = \mathsf{pgr\_readback}(P_i)$
spawn:	$\mathsf{pgr\_readback}(\$d = f(\mathsf{args})) = \$d = f(\mathsf{args})$
close:	$\mathsf{pgr\_readback}(\mathsf{close}(\$d)) = \mathsf{close}(\$d)$
send-expression:	$\mathsf{pgr\_readback}(\mathsf{send}(\$d, e)) = \mathsf{send}(\$d, e)$
send-shift:	$\mathsf{pgr\_readback}(\mathsf{send}(\$d, \mathsf{shift})) = \mathsf{send}(\$d, \mathsf{shift})$
send-label:	$\mathsf{pgr\_readback}(\$d.lab) = \$d.lab$
forward:	$\mathsf{pgr\_readback}(\$d = \$e) = \$d = \$e$
sequence:	$\mathsf{pgr\_readback}(P\ ;\ Q) = P'\ ;\ Q'$ where $P' = \mathsf{pgr\_readback}(P)$, $Q' = \mathsf{pgr\_readback}(Q)$
assignment:	$\mathsf{pgr\_readback}(x = e) = x = e$
if:	$\mathsf{pgr\_readback}(\text{if } (b) \text{ then } P \text{ else } Q) = \text{if } (b) \text{ then } P' \text{ else } Q'$ where $P' = \mathsf{pgr\_readback}(P)$, $Q' = \mathsf{pgr\_readback}(Q)$
while :	$\mathsf{pgr\_readback}(\text{while } (b) \text{ do } P) = \text{while } (b) \text{ do } P'$ where $P' = \mathsf{pgr\_readback}(P)$
func:	$\mathsf{pgr\_readback}(f(args)) = f(args)$
sync:	$\mathsf{pgr\_readback}(\mathsf{sync}(\$c, X)) = \mathsf{skip}$

Table 4.3: Readback scheme from non-blocking to blocking programs.

Conjecture 1 ***(Correctness)*** *If* $[\![\Omega]\!] \to C$*, using the non-blocking operational semantics, then* $\exists \Omega'.\ \Omega \to \Omega'$*, using the blocking operational semantics, and* $C = [\![\Omega']\!]$*.*

We do not have a rigorous proof for Conjecture 1. Here we give an hint based on what we tried to do of how such a proof could be done.

Essentially we would take the blocking semantics as valid and show that any non-blocking computation can be simulated by a blocking one. To do this, we would show that for any given complete computation in the non-blocking semantics we can assemble a corresponding blocking computation. This requires us to define some operation that allows us to look ahead to obtain a value something was eventually synchronized to. The key induction would be over the complete non-blocking computation, relating states. This would enable us to prove by induction that each process in the non-blocking semantics has a corresponding state in the blocking semantics. The same had to be done for queues.

Note that this correctness includes full-value correction, every synchronized variable in a memory cell has the same value in both semantics.

In the next theorem, we assume correctness of our non-blocking semantics and prove that work is the same across configurations and span decreases or remains the same when going from blocking to non-blocking configurations.

Theorem 1 *Assuming that Conjecture 1 holds,* $[\![\Omega_1]\!] \to [\![\Omega_1']\!]$*, using the non-blocking cost semantics,* $\exists \Omega_2'.\ \Omega_2 \to \Omega_2'$*, using the blocking cost semantics, and* $\Omega_2' = \mathsf{readback}([\![\Omega_1']\!])$*.*

1. *If* $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$*, then* $\mathsf{work}([\![\Omega_1']\!]) = \mathsf{work}(\Omega_2')$*.*
2. *If* $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$*, then* $\mathsf{span}([\![\Omega_1']\!]) \leq \mathsf{span}(\Omega_2')$*.*

Proof: By induction on the operational semantics.

- **Case:** Send-expression:

$$\begin{aligned}&\mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], \mathsf{send}([\![\$c]\!], E)\ ;\ Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]})\\ &\multimap \{\mathsf{queue}([\![\$d]\!], \overleftarrow{q \cdot [\![\mathrm{m}]\!]}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], Q, s_{[\![\$c]\!]}, w'_{[\![\$c]\!]})\}\end{aligned}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{queue}(\$d, \overleftarrow{q'}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{send}(\$c, E)\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \\ \multimap \{\mathsf{queue}(\$d, \overleftarrow{q' \cdot m}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c})\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + 1$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + 1$. $w'_{\$c} = w_{\$c} + 1$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + 1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]}+1$, $s'_{\$c} = s_{\$c}+1$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

Note that $\mathsf{span}([\![\mathrm{m}]\!]) \leq \mathsf{span}(m)$, as a consequence of $s'_{[\![\$c]\!]} \leq s'_{\$c}$.

- **Case:** Send-shift:

$$\mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], \mathsf{send}([\![\$c]\!], \mathsf{shift})\ ;\ Q, s_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \\ \multimap \{\mathsf{queue}([\![\$d]\!], \overleftarrow{q \cdot [\![\mathrm{m}]\!]}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]})\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{queue}(\$d, \overleftarrow{q'}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{send}(\$c, \mathsf{shift})\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \\ \multimap \{\mathsf{queue}(\$d, \overleftarrow{q' \cdot m}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c})\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]}$, $s'_{\$c} = s_{\$c}$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

Note that $\mathsf{span}([\![\mathrm{m}]\!]) \leq \mathsf{span}(m)$, as a consequence of $s'_{[\![\$c]\!]} \leq s'_{\$c}$.

- **Case:** Send-label:

$$\mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], [\![\$c]\!].lab\ ;\ Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \\ \multimap \{\mathsf{queue}([\![\$d]\!], \overleftarrow{q \cdot [\![\mathrm{m}]\!]}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]})\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{queue}(\$d, \overleftarrow{q'}, \$c) \otimes \mathsf{proc}(\$c, \$c.lab\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \\ \multimap \{\mathsf{queue}(\$d, \overleftarrow{q' \cdot m}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c})\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + 1$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + 1$. $w'_{\$c} = w_{\$c} + 1$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + 1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]}+1$, $s'_{\$c} = s_{\$c}+1$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

Note that $\mathsf{span}([\![\mathrm{m}]\!]) \leq \mathsf{span}(m)$, as a consequence of $s'_{[\![\$c]\!]} \leq s'_{\$c}$.

- **Case:** Close:

$$\begin{aligned}&\mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], \mathsf{close}([\![\$c]\!]), s_{[\![\$c]\!]}, w_{[\![\$c]\!]})\\ &\quad \multimap \{\mathsf{queue}([\![\$d]\!], \overleftarrow{q \cdot [\![\mathrm{m}]\!]}, \_, r)\}\end{aligned}$$

is the transition in the non-blocking operational semantics,

$$\begin{aligned}&\mathsf{queue}(\$d, \overleftarrow{q'}, \$c) \otimes \mathsf{proc}(\$c, \mathsf{close}(\$c), s_{\$c}, w_{\$c})\\ &\quad \multimap \{\mathsf{queue}(\$d, \overleftarrow{q' \cdot m}, \_)\}\end{aligned}$$

is the transition in the blocking operational semantics.

1. $\mathsf{work}([\![m]\!]) = w_{[\![\$c]\!]} + 1$ and $\mathsf{data}([\![m]\!]) = \mathsf{end}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + \mathsf{data}([\![m]\!]) - w_{[\![\$c]\!]} = \mathsf{work}([\![\Omega_1]\!]) + 1$. $\mathsf{work}(m) = w_{\$c}+1$ and $\mathsf{data}(m) = \mathsf{end}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + \mathsf{work}(m) - w_{\$c} = \mathsf{work}(\Omega_2)+1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so, by removing a process, it still follows that $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

Note that $\mathsf{span}([\![\mathrm{m}]\!]) = s_{[\![\$c]\!]} + 1$ and $\mathsf{span}(m) = s_{\$c} + 1$. By theorem hypothesis, $s_{[\![\$c]\!]} \leq s_{\$c}$, so it follows that $\mathsf{span}([\![\mathrm{m}]\!]) \leq \mathsf{span}(m)$.

- **Case:** Send-Forward:

$$\begin{aligned}&\mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$c]\!], r) \otimes \mathsf{proc}([\![\$c]\!], [\![\$c]\!] = [\![\$e]\!], s_{[\![\$c]\!]}, w_{[\![\$c]\!]})\\ &\quad \multimap \{\mathsf{queue}([\![\$d]\!], \overleftarrow{q \cdot [\![\mathrm{m}]\!]}, [\![\$e]\!], r)\}\end{aligned}$$

is the transition in the non-blocking operational semantics,

$$\begin{aligned}&\mathsf{queue}(\$d, \overleftarrow{q'}, \$c) \otimes \mathsf{proc}(\$c, \$c = \$e, s_{\$c}, w_{\$c})\\ &\quad \multimap \{\mathsf{queue}(\$d, \overleftarrow{q' \cdot m}, \$e)\}\end{aligned}$$

is the transition in the blocking operational semantics.

1. $\mathsf{work}(\llbracket m \rrbracket) = w_{\llbracket \$c \rrbracket}$ and $\mathsf{data}(\llbracket m \rrbracket) = \mathsf{fwd}$, so $\mathsf{work}(\llbracket \Omega_1' \rrbracket) = \mathsf{work}(\llbracket \Omega_1 \rrbracket) + \mathsf{data}(\llbracket m \rrbracket) - w_{\llbracket \$c \rrbracket} = \mathsf{work}(\llbracket \Omega_1 \rrbracket)$. $\mathsf{work}(m) = w_{\$c}$ and $\mathsf{data}(m) = \mathsf{end}$, so $\mathsf{work}(\Omega_2') = \mathsf{work}(\Omega_2) + \mathsf{work}(m) - w_{\$c} = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}(\llbracket \Omega_1 \rrbracket) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}(\llbracket \Omega_1' \rrbracket) = \mathsf{work}(\Omega_2')$.
2. By theorem hypothesis, $\mathsf{span}(\llbracket \Omega_1 \rrbracket) \leq \mathsf{span}(\Omega_2)$, so, by removing a process, it still follows that $\mathsf{span}(\llbracket \Omega_1' \rrbracket) \leq \mathsf{span}(\Omega_2')$.

Note that $\mathsf{span}(\llbracket \mathrm{m} \rrbracket) = s_{\llbracket \$c \rrbracket}$ and $\mathsf{span}(m) = s_{\$c}$. By theorem hypothesis, $s_{\llbracket \$c \rrbracket} \leq s_{\$c}$, so it follows that $\mathsf{span}(\llbracket \mathrm{m} \rrbracket) \leq \mathsf{span}(m)$.

- **Case:** Spawn:

$$\mathsf{proc}(\llbracket \$c \rrbracket, \llbracket \$d \rrbracket = P(args)\ ;\ Q, s_{\llbracket \$c \rrbracket}, w_{\llbracket \$c \rrbracket}) \multimap \{\mathsf{proc}(\llbracket \$c \rrbracket, Q, s_{\llbracket \$c \rrbracket}, w'_{\llbracket \$c \rrbracket}) \otimes \mathsf{queue}(\llbracket \$d \rrbracket, \cdot, \llbracket \$e \rrbracket, \cdot) \otimes \mathsf{proc}(\llbracket \$e \rrbracket, P(args), s_{\llbracket \$e \rrbracket}, w_{\llbracket \$e \rrbracket})\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \$d = P(args)\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, \cdot, \$e) \otimes \mathsf{proc}(\$e, \mathsf{pgr\_readback}(P), s_{\$e}, w_{\$e})\}$$

is the transition in the blocking operational semantics.

1. $w'_{\llbracket \$c \rrbracket} = w_{\llbracket \$c \rrbracket}$ and $w_{\llbracket \$e \rrbracket} = 0$, so $\mathsf{work}(\llbracket \Omega_1' \rrbracket) = \mathsf{work}(\llbracket \Omega_1 \rrbracket)$. $w'_{\$c} = w_{\$c}$ and $w_{\$e} = 0$, so $\mathsf{work}(\Omega_2') = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}(\llbracket \Omega_1 \rrbracket) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}(\llbracket \Omega_1' \rrbracket) = \mathsf{work}(\Omega_2')$.
2. $s'_{\llbracket \$c \rrbracket} = s_{\llbracket \$c \rrbracket} = s_{\llbracket \$e \rrbracket}$, $s'_{\$c} = s_{\$c} = s_{\$e}$. By theorem hypothesis, $\mathsf{span}(\llbracket \Omega_1 \rrbracket) \leq \mathsf{span}(\Omega_2)$, so $s_{\llbracket \$c \rrbracket} \leq s_{\$c}$. It follows that $s'_{\llbracket \$c \rrbracket} \leq s'_{\$c}$, $s_{\llbracket \$e \rrbracket} \leq s_{\$e}$ and $\mathsf{span}(\llbracket \Omega_1' \rrbracket) \leq \mathsf{span}(\Omega_2')$.

- **Case:** Receive-label:

$$\mathsf{proc}(\llbracket \$c \rrbracket, \mathsf{switch}(\llbracket \$d \rrbracket)\{lab_i \to P_i\}, s_{\llbracket \$c \rrbracket}, w_{\llbracket \$c \rrbracket}) \otimes \mathsf{queue}(\llbracket \$d \rrbracket, \overleftarrow{\llbracket m \rrbracket \cdot q}, \llbracket \$e \rrbracket, \cdot) \multimap \{\mathsf{proc}(\llbracket \$c \rrbracket, P_j, s'_{\llbracket \$c \rrbracket}, w'_{\llbracket \$c \rrbracket}) \otimes \mathsf{queue}(\llbracket \$d \rrbracket, \overleftarrow{q}, \llbracket \$e \rrbracket, \cdot)\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{switch}(\$d)\{lab_i \to \mathsf{pgr\_readback}(P_i)\}, s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{m \cdot q'}, \$e) \multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(P_j), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{q'}, \$e)\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + 1$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + 1$. $w'_{\$c} = w_{\$c} + 1$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + 1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = \mathsf{max}(\mathsf{span}([\![m]\!]), s_{[\![\$c]\!]}) + 1$, $s'_{\$c} = \mathsf{max}(\mathsf{span}(m), s_{\$c}) + 1$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$ and $\mathsf{span}([\![m]\!]) \leq \mathsf{span}(m)$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Receive-value:

$$\mathsf{proc}([\![\$c]\!], x = \mathsf{async\_recv}([\![\$d]\!]) \,;\, Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r)$$
$$\multimap \{\exists x.\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r \cdot x)\}$$

  is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, x = \mathsf{recv}(\$d) \,;\, \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{m \cdot q'}, \$e)$$
$$\multimap \{\exists x.\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{q'}, \$e) \otimes \mathsf{cell}(\$c, x, \mathsf{data}(m))\}$$

  is the transition in the blocking operational semantics.

  1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + 1$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + 1$. $w'_{\$c} = w_{\$c} + 1$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + 1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
  2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]} + 1$, $s'_{\$c} = \mathsf{max}(\mathsf{span}(m), s_{\$c}) + 1$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Receive-shift:

$$\mathsf{proc}([\![\$c]\!], \mathsf{shift} = \mathsf{async\_recv}([\![\$d]\!]) \,;\, Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r)$$
$$\multimap \{\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r \cdot \mathsf{shift})\}$$

  is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{shift} = \mathsf{recv}(\$d) \,;\, \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{m}, \$e)$$
$$\multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, \overrightarrow{\cdot}, \$e)\}$$

  is the transition in the blocking operational semantics.

  1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.

2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]}$, $s'_{\$c} = \mathsf{max}(\mathsf{span}(m), s_{\$c})$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Wait:

$$\mathsf{proc}([\![\$c]\!], \mathsf{async\_wait}([\![\$d]\!]) \,;\, Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r)$$
$$\multimap \{\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r \cdot \mathsf{end})\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{wait}(\$d) \,;\, \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{m}, \_)$$
$$\multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c})\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + 1$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + 1$. $w'_{\$c} = w_{\$c} + 1 + \mathsf{work}(m)$ and $\mathsf{data}(m) = \mathsf{end}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + 1 + \mathsf{work}(m) - \mathsf{work}(m) = \mathsf{work}(\Omega_2) + 1$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = s_{[\![\$c]\!]} + 1$, $s'_{\$c} = \mathsf{max}(\mathsf{span}(m), s_{\$c}) + 1$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Receive-Forward:

$$\mathsf{proc}([\![\$c]\!], \mathsf{sync}([\![\$d]\!], X) \,;\, Q([\![\$d]\!]), s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{[\![m]\!]}, [\![\$e]\!], r) \otimes \mathsf{queue}([\![\$e]\!], \overleftarrow{q'}, [\![\$f]\!], \cdot)$$
$$\multimap \{\mathsf{proc}([\![\$c]\!], \mathsf{sync}([\![\$e]\!], X) \,;\, Q([\![\$e]\!]), s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$e]\!], \overleftarrow{q'}, [\![\$f]\!], r)\}$$

is the transition in the non-blocking operational semantics, where $\mathsf{data}([\![m]\!]) = \mathsf{fwd}$,

$$\mathsf{proc}(\$c, \mathsf{skip} \,;\, \mathsf{pgr\_readback}(Q(\$d)), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, \overleftarrow{m}, \$e)$$
$$\multimap \{\mathsf{proc}(\$c, \mathsf{skip} \,;\, \mathsf{pgr\_readback}(Q(\$e)), s'_{\$c}, w'_{\$c})\}$$

is the transition in the blocking operational semantics, where $\mathsf{data}(m) = \mathsf{fwd}$.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + \mathsf{work}([\![m]\!])$ and $\mathsf{data}([\![m]\!]) = \mathsf{fwd}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + \mathsf{work}([\![m]\!]) - \mathsf{work}([\![m]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c} + \mathsf{work}(m)$ and $\mathsf{data}(m) = \mathsf{fwd}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2) + \mathsf{work}(m) - \mathsf{work}(m) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.

2. $s'_{[\![\$c]\!]} = \mathsf{max}(\mathsf{span}([\![m]\!]), s_{[\![\$c]\!]})$, $s'_{\$c} = \mathsf{max}(\mathsf{span}(m), s_{\$c})$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$ and $\mathsf{span}([\![m]\!]) \leq \mathsf{span}(m)$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Synchronization-value:

$$\mathsf{proc}([\![\$c]\!], \mathsf{sync}([\![\$d]\!], x)\ ;\ Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{[\![m]\!] \cdot q}, [\![\$e]\!], x \cdot r)$$
$$\multimap \{\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{q}, [\![\$e]\!], r) \otimes \mathsf{cell}([\![\$c]\!], x, \mathsf{data}([\![m]\!]))\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{skip}\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e)$$
$$\multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e)\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
2. $s'_{[\![\$c]\!]} = \mathsf{max}(\mathsf{span}([\![m]\!]), s_{[\![\$c]\!]})$, $s'_{\$c} = s_{\$c}$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$, $\mathsf{span}([\![m]\!]) \leq \mathsf{span}(m)$, m who was previously received by a blocking function, and $s_{\$c} \geq \mathsf{span}(m)$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Synchronization-shift:

$$\mathsf{proc}([\![\$c]\!], \mathsf{sync}([\![\$d]\!], \mathsf{shift})\ ;\ Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{[\![m]\!]}, [\![\$e]\!], \mathsf{shift})$$
$$\multimap \{\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overrightarrow{\cdot}, [\![\$e]\!], \cdot)\}$$

is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{skip}\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e)$$
$$\multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e)\}$$

is the transition in the blocking operational semantics.

1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.

2. $s'_{[\![\$c]\!]} = \mathsf{max}(\mathsf{span}([\![m]\!]), s_{[\![\$c]\!]})$, $s'_{\$c} = s_{\$c}$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$, $\mathsf{span}([\![m]\!]) \leq \mathsf{span}(m)$, m who was previously received by a blocking function, and $s_{\$c} \geq \mathsf{span}(m)$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- **Case:** Synchronization-wait:

$$\mathsf{proc}([\![\$c]\!], \mathsf{sync}([\![\$d]\!], \mathsf{end})\ ;\ Q, s_{[\![\$c]\!]}, w_{[\![\$c]\!]}) \otimes \mathsf{queue}([\![\$d]\!], \overleftarrow{[\![m]\!]}, [\![\$e]\!], \mathsf{end}) \multimap \{\mathsf{proc}([\![\$c]\!], Q, s'_{[\![\$c]\!]}, w'_{[\![\$c]\!]})\}$$

  is the transition in the non-blocking operational semantics,

$$\mathsf{proc}(\$c, \mathsf{skip}\ ;\ \mathsf{pgr\_readback}(Q), s_{\$c}, w_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e) \multimap \{\mathsf{proc}(\$c, \mathsf{pgr\_readback}(Q), s'_{\$c}, w'_{\$c}) \otimes \mathsf{queue}(\$d, q', \$e)\}$$

  is the transition in the blocking operational semantics.

  1. $w'_{[\![\$c]\!]} = w_{[\![\$c]\!]} + \mathsf{work}([\![m]\!])$ and $\mathsf{data}([\![m]\!]) = \mathsf{end}$, so $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}([\![\Omega_1]\!]) + \mathsf{work}([\![m]\!]) - \mathsf{work}([\![m]\!]) = \mathsf{work}([\![\Omega_1]\!])$. $w'_{\$c} = w_{\$c}$, so $\mathsf{work}(\Omega'_2) = \mathsf{work}(\Omega_2)$. By theorem hypothesis, $\mathsf{work}([\![\Omega_1]\!]) = \mathsf{work}(\Omega_2)$, so it follows that $\mathsf{work}([\![\Omega'_1]\!]) = \mathsf{work}(\Omega'_2)$.
  2. $s'_{[\![\$c]\!]} = \mathsf{max}(\mathsf{span}([\![m]\!]), s_{[\![\$c]\!]})$, $s'_{\$c} = s_{\$c}$. By theorem hypothesis, $\mathsf{span}([\![\Omega_1]\!]) \leq \mathsf{span}(\Omega_2)$, so $s_{[\![\$c]\!]} \leq s_{\$c}$, $\mathsf{span}([\![m]\!]) \leq \mathsf{span}(m)$, m who was previously received by a blocking function, and $s_{\$c} \geq \mathsf{span}(m)$. It follows that $s'_{[\![\$c]\!]} \leq s'_{\$c}$ and $\mathsf{span}([\![\Omega'_1]\!]) \leq \mathsf{span}(\Omega'_2)$.

- There are three missing cases, assignments, conditionals and loops, whose operational rules we did not detail in this thesis. By assuming correctness, the evaluation of expressions will always yield the same value, so these rules will not change work nor span.

Chapter 5

Implementation and Experimental Evaluation

In this chapter, we discuss the low-level implementation of the concepts we presented in the previous chapters. Specifically, we delve into the real target language of CC0 and the runtime system that enables the session typed channels, as well as the changes to this runtime system to allow non-blocking reception. CC0's target language is C and the runtime system is also written in C.

Finally, we present some experimental results with the blocking and non-blocking semantics that were the basis for the ideas in Section 4.5.

5.1 Implementation

We divide the implementation section in two parts, one for the existing blocking runtime and the other for our non-blocking one. In the first, we give an overview of how the existing blocking version of CC0 is implemented, presenting code of the most crucial functions and the ones we had to change to implement the non-blocking runtime. We also show how span and work are calculated in the runtime environment. The second part focuses on describing the modifications to the functions and structures on the first part to accomplish the non-blocking model.

5.1.1 Blocking Runtime

The first functional runtime environment for CC0 was called *concur2* [30]. This runtime has been optimized in more recent versions, but our work predates these newer, more efficient, implementations. The runtime uses the *pthread* library to enable parallelism in multi-core machines and each process in CC0 is mapped 1-to-1 to a *pthread* thread.

Each process has a channel associated to it, which is the channel they provide, their unique identifier. This channel is represented by a `struct`, containing information about the message queue's direction, the actual message queue and a pthread mutex and condition variable. These two are used to obtain mutual exclusivity on the queue: only one of the processes at the two end of the channel may write to the queue at a given time. Code 5.1 shows the declaration of this `struct`.

```
struct channel {
    channel_dir_e queue_dir;
    channel_dir_e provider_dir;
    channel_dir_e client_dir;
    queue_t* msgs;
    pthread_mutex_t m;
    pthread_cond_t c;
};
```

Code 5.1: Channel structure.

The direction of the queue is simply an `enum` structure in C, with the options `TO_PROVIDER` and `TO_CLIENT`.

Spawning a process is done in the target language, each function that returns a channel has an associated spawning procedure that returns the reference of the channel to the client. Code 5.2 shows how a process running the *empty* function that implements the end of a queue is spawned.

This function takes an argument `n_clock *c`, which is a structure used to track the span, work and number of processes spawned of a process, in this case, the client's. We refer to this structure as *clock* in the rest of this chapter. The new process initialises the last two metrics start as 0, but span inherits the client's span. This is done in lines 6 to 10 of Code 5.2. The function `new_channel_negative` is used to create a channel with a queue with negative polarity, but it simply calls the function `new_channel`

```
queue empty_spawn(n_clock *c) {
  channel_t* cid = new_channel_negative(-1);
  queue_p $q = provider_handle(cid);
  empty_args_t *empty_args = (empty_args_t*) malloc(sizeof(struct empty_args));
  empty_args->$q = $q;
  n_clock *nc = (n_clock*)malloc(sizeof(n_clock));
  nc->span = c->span;
  nc->work = 0;
  nc->nprocs_spawned = 0;
  empty_args->c = nc;
  spawn_process(&empty_unpack, (void *)empty_args);
  c->nprocs_spawned = (c->nprocs_spawned)+1;
  return client_handle(cid);
}
```

Code 5.2: Spawning a process running the *empty* function.

(Code 5.3) with the correct capacity. This capacity is determined by the compiler, which calculates the *type width* of the session type of the channel.

Type width is a property of certain session types that allow us to infer how many values may be buffered at any given time. Willsey et al. [30] offers a complete explanation on how type width is calculated in CC0. To summarize, a session type may be seen as a colored directed graph, with receiving nodes colored differently than sending ones. A walk in this graph is a possible sequence of sent or received types and the width of the type is the longest walk in the graph. If the graph possesses a cycle, then the width is infinite, represented by `-1`. If this succeeds, the message queue is started with a minimum size and doubles its size every time the capacity is reached.

```
static inline channel_t* new_channel(channel_dir_e dir, int capacity) {
    channel_t* chan = malloc(sizeof(channel_t));
    assert(chan);
    chan->msgs = new_queue(capacity);
    assert(chan->msgs);
    pthread_mutex_init(&chan->m, NULL);
    pthread_cond_init(&chan->c, NULL);
    return chan;
}
```

Code 5.3: Creating a new channel in *concur2*.

Messages in CC0 are also a C `struct` (Code 5.4), containing the type of the message, the clock and the value carried by the message, given by a `union` of several possible basic types.

```
typedef struct {
  msg_type_e type;
  n_clock *c;
    union {
        int label;
        int n;
        bool b;
        void* p;
        channel_t* forward;
        client_handle_t* handle;
    };
} channel_msg_t;
```

Code 5.4: Structure of a message.

The procedure of swapping messages happens in four stages: the message is sent, packed, unpacked and, finally, received. The runtime *concur2* provides a set of functions that discriminate both the channel to which the message is being sent or received (if it is the provider or a client) and the type of the message. These functions are similar to each other, so we only present the example of an integer, implemented by functions *client_send_int*, *provider_send_int*, *client_recv_int* and *provider_recv_int*. Code 5.5 clears the similarity between provider and client functions, so, in the rest of this chapter, we only show the examples for provider functions.

The functions from Code 5.5 simply call more generic functions with the correct message type and update the clock. The actual sending and receiving are done by the function *provider_send_msg* and *provider_recv_msg*. Sending a message requires packing and enqueueing it, the implementation of which is orthogonal to our work. Receiving a message is presented in Code 5.6, where it is also shown an extract of the function used to unpack the message, demonstrating how span and work are synchronized.

Referring to Code 5.6, lines 7 to 9 are responsible for blocking the program if the message we want to receive was not queued yet. The function *queue_dequeue*, in line 11, fetches the message from the queue. The function *unpack_msg* verifies if the expected type is equal to the type of the message and changes the return location (`ret_loc`) to point to the value carried by the message.

Returning to our example of the queue, Program 5.7 presents the implementation of this data structure in the low-level blocking target language of CC0. We omit details

```
int client_recv_int(client_handle_t* client, n_clock *c){
    int n;
    client_recv_msg(client, INT, (void**) &n, c);
    c->work = (c->work)+1;
    c->span = (c->span)+1;
    return n;
}
int client_send_int(int n, client_handle_t* client, n_clock *c){
  c->work = (c->work)+1;
  c->span = (c->span)+1;
  return client_send_msg(client, INT, (void*) (intptr_t) n, c);
}
int provider_recv_int(provider_handle_t* provider, n_clock *c){
  int n;
  provider_recv_msg(provider, INT, (void**) &n, c);
  c->work = (c->work)+1;
  c->span = (c->span)+1;
  return n;
}
int provider_send_int(int n, provider_handle_t* provider, n_clock *c){
  c->work = (c->work)+1;
  c->span = (c->span)+1;
  return provider_send_msg(provider, INT, (void*) (intptr_t) n, c);
}
```

Code 5.5: Send and receive functions of an integer, for provider and client channels.

such as the spawn functions, which are all similar to Code 5.2, and pthread specific syntax needed to pass arguments to the functions that each process executes.

5.1.2 Non-blocking Runtime

The main object of our non-blocking model is the request. It is represented by a `struct`, containing the type of the message requested, the return location and a boolean value indicating if the request has already been processed. This structure is realized in Code 5.8.

The queue of requests discussed in Chapter 4 is stored in the channel structure. Each channel possesses a queue filled by the provider and another filled by the client, as well as variables to control the capacity and the next request to be handled. The queue of requests is not polarized like the queue of messages, so it requires this distinction between client and provider. The reason for this is the case when a process finishes sending and starts requesting right after. The process on the other end of the channel

```
1  static int provider_recv_msg(provider_handle_t* handle,
2                               msg_type_e expected_type, void** ret_loc, n_clock *c) {
3      channel_t* chan = handle->chan;
4      pthread_mutex_lock(&chan->m);
5
6      assert(chan->provider_dir == TO_PROVIDER);
7      while (chan->queue_dir != TO_PROVIDER || queue_size(chan->msgs) == 0) {
8        pthread_cond_wait(&chan->c, &chan->m);
9      }
10     channel_msg_t* msg;
11     assert(queue_dequeue(chan->msgs, (void**)&msg) == 0);
12     channel_t* forward = unpack_msg(msg, expected_type, ret_loc, c);
13     if (forward) {
14         assert(queue_size(chan->msgs) == 0);
15         pthread_mutex_unlock(&chan->m);
16         free_channel(chan);
17         handle->chan = forward;
18         return provider_recv_msg(handle, expected_type, ret_loc, c);
19     }
20     if (expected_type == SHIFT) {
21         assert(queue_size(chan->msgs) == 0);
22         chan->provider_dir = TO_CLIENT;
23     }
24     pthread_mutex_unlock(&chan->m);
25     return SUCCESS;
26 }
27
28 channel_t* unpack_msg(channel_msg_t* msg, msg_type_e expected_type,
29                                 void **ret_loc, n_clock *c) {
30     assert(msg != NULL);
31     n_clock *c_recv = msg->c;
32     c->span = (c->span) > (c_recv->span)? (c->span) : (c_recv->span);
33     if (msg->type == FORWARD || msg->type == DONE) {
34       c->work = (c->work) + (c_recv->work);
35       c->nprocs_spawned = (c->nprocs_spawned) + (c_recv->nprocs_spawned);
36     }
37     // ....
38 }
```

Code 5.6: Receiving a message in *concur2*.

```
void empty(queue_p $q, n_clock *c) {
  switch (provider_recv_label($q, c)) {
  case Enq: {
    int y = provider_recv_int($q, c);
    queue $e = empty_spawn(c);
    elem($q, y, $e, c);
  }
  case Deq: {
    provider_recv_shift($q,c);
    provider_send_label(None, $q, c);
    provider_send_shift($q, c);
    empty($q, c);
  }
  case IsEmpty: {
    provider_recv_shift($q, c);
    provider_send_int(true, $q, c);
    provider_send_shift($q, c);
    empty($q, c);
  }
  case Dealloc: {
    provider_recv_shift($q, c);
    provider_send_done($q, c);
  }
  }
}

void elem(queue_p $q, int x, queue $r, n_clock *c) {
  switch (provider_recv_label($q, c)) {
  case Enq: {
    int y = provider_recv_int($q, c);
    client_send_label(Enq, $r, c);
    client_send_int(y, $r, c);
    elem($q, x, $r, c);
  }
  case Deq: {
    provider_recv_shift($q, c);
    provider_send_label(Some, $q, c);
    provider_send_int(x, $q, c);
    provider_send_shift($q, c);
    forward($q, $r, c);
  }
  case IsEmpty: {
    provider_recv_shift($q, c);
    provider_send_int(false, $q, c);
    provider_send_shift($q, c);
    elem($q, x, $r, c);
  }
  case Dealloc: {
    provider_recv_shift($q, c);
    client_send_label(Dealloc, $r, c);
    client_send_shift($r, c);
    client_recv_done($r, c);
    provider_send_done($q, c);
  }
  }
}
```

Code 5.7: Implementation of a queue, in the low-level blocking target language of CC0.

```
typedef struct request {
    msg_type_e expected_type;
    void **ret_loc;
    int treated;
} request;
```

Code 5.8: Request structure.

may not have finished synchronizing when the first process started requesting. Without the separation, the first process would overwrite the the requests from the other process.

Code 5.9 shows the added information to the channel `struct`.

```
1  struct channel {
2      // ...
3      request **pending_reqs_provider;
4      int nreqs_provider;
5      int first_req_provider;
6      int reqs_cap_provider;
7      request **pending_reqs_client;
8      int nreqs_client;
9      int first_req_client;
10     int reqs_cap_client;
11 };
```

Code 5.9: Queues of requests declaration in the channel structure.

The variables pertaining to the request queue are initialized in the `new_channel` function. This initialization only requires allocating memory for the queues, setting the first requests and number of requests to zero and the capacity, which also uses the type width to be determined. These queues behave like the message queues if the type width is infinite: their capacity is set at an initial value and doubled each time the queue is filled up.

Sending functions behave exactly the same way in the non-blocking model, however, receiving ones follow a new arrangement. The functions `provider_recv_int`, and similar, are now `async_provider_recv_int`. If it is a function that sets the value of a variable, this family of `async` functions takes the pointer to this variable as an argument, or a `NULL` pointer otherwise.

The receiving functions call a more general `async_provider_recv_msg`, who builds a request, updates the queue of requests, increasing its size if needed. As prescribed by

our cost semantics, this procedure of building a request increases the span and work by one unit, unless we it is a request for a shift. Code 5.10 illustrates how the request for an integer value is processed, in the aforementioned functions.

```
void async_provider_recv_int(provider_handle_t* provider, int *n, n_clock *c){
    async_provider_recv_msg(provider, INT, (void **) n);
    c->work++;
    c->span++;
}
void async_provider_recv_msg(provider_handle_t *provider,
                             msg_type_e expected_type, void **ret_loc) {
    channel_t* chan = provider->chan;
    pthread_mutex_lock(&chan->m);
    request *r = (request *)malloc(sizeof(request));
    r->expected_type = expected_type;
    r->ret_loc = ret_loc;
    r->treated = 0;
    chan->pending_reqs_client[chan->nreqs_client] = r;
    chan->nreqs_client++;
    if (chan->nreqs_client == chan->reqs_cap_client) {
        int new_cap = chan->reqs_cap_client * 2;
        request** new_buf = (request**)malloc(sizeof(request*)*new_cap);
        int i;
        for (i = chan->first_req_client; i<chan->nreqs_client; i++)
            new_buf[i] = chan->pending_reqs_client[i];
        chan->reqs_cap_client = new_cap;
        chan->pending_reqs_client = new_buf;
    }
    pthread_mutex_unlock(&chan->m);
}
```

Code 5.10: Functions to create a request, exemplified by a request for an integer value.

The next building block of our model are the synchronization functions, split in `provider_sync` which is called by a process and an auxiliary function, whose behavior is similar to the original `provider_recv_msg`. The synchronization function receives a pointer as argument, which determines until which point the channel is synchronized. A `NULL` pointer synchronizes the channel completely, whereas a pointer to a variable stops the synchronization loop when that same pointer is seen upon processing a request.

The auxiliary functions handle fetching messages from the queue. If a message is not available, the execution blocks, which is inevitable because this synchronization function is only called when the program needs that message to proceed. Code 5.11 shows how these two functions are implemented.

```
int provider_sync_aux(provider_handle_t *handle, int reqnumber,
```

```
                    void** elem, n_clock *c) {
    channel_t *chan = handle -> chan;
    pthread_mutex_lock(&chan->m);
    request *r = chan->pending_reqs_client[reqnumber];
    if (r->treated) {
        pthread_mutex_unlock(&chan->m);
        return -1;
    }
    assert(chan->provider_dir == TO_PROVIDER);
    while (chan->queue_dir != TO_PROVIDER || queue_size(chan->msgs) == 0) {
        pthread_cond_wait(&chan->c, &chan->m);
    }
    channel_msg_t* msg;
    assert(queue_dequeue(chan->msgs, (void**)&msg) == 0);
    channel_t *forward = unpack_msg(msg, r->expected_type, r->ret_loc, c);
    if (forward) {
        assert(queue_size(chan->msgs) == 0);
        forward->pending_reqs_client = chan->pending_reqs_client;
        forward->nreqs_client = chan->nreqs_client;
        forward->first_req_client = chan->first_req_client;
        forward->pending_reqs_provider = chan->pending_reqs_provider;
        forward->nreqs_provider = chan->nreqs_provider;
        forward->first_req_provider = chan->first_req_provider;
        pthread_mutex_unlock(&chan->m);
        free_channel(chan);
        handle->chan = forward;
        return provider_sync_aux(handle, reqnumber, elem, c);
    }
    if (r->expected_type == SHIFT) {
        assert(queue_size(chan->msgs) == 0);
        chan->provider_dir = TO_CLIENT;
    }
    r->treated = 1;
    pthread_mutex_unlock(&chan->m);
    return r->ret_loc == elem;
}
void provider_sync(provider_handle_t *handle, void** elem, n_clock *c) {
    channel_t *chan = handle -> chan;
    int i = chan->first_req_client;
    int res;
    while (i<chan->nreqs_client && (res=provider_sync_aux(handle,i,elem,c)) == 0)
        i++;
    if (res == -1)   return;
    chan = handle -> chan;
    pthread_mutex_lock(&chan->m);
    if (elem == NULL) {
        chan->first_req_client = 0;
        chan->nreqs_client = 0;
    }
    else {
        if (i == chan->nreqs_client)  i--;
        chan->first_req_client = i+1;
```

```
    }
    pthread_mutex_unlock(&chan->m);
}
```

Code 5.11: Functions to synchronize a channel.

5.2 Working Example

In this section, we explain in detail the non-blocking implementation of a queue, presented in Program 5.12. This implementation is in CC0's target language and follows the same structure of the version presented in Program 4.1, which is simplified to avoid the heavy notation of the runtime functions we showcased in this chapter.

```
1  void elem(queue_p $q, int x, queue $r, n_clock *c) {
2    switch (provider_recv_label($q, c)) {
3      case Enq: {
4        int y;
5        async_provider_recv_int($q, &y, c);
6        client_send_label(Enq, $r, c);
7        provider_sync($q,(void**)&y,c);
8        client_send_int(y, $r, c);
9        elem($q,x,$r,c);
10     }
11     case Deq: {
12       async_provider_recv_shift($q);
13       provider_sync($q,NULL,c);
14       provider_send_label(Some, $q, c);
15       provider_send_int(x, $q, c);
16       provider_send_shift($q, c);
17       forward($q, $r, c);
18     }
19     case IsEmpty: {
20       async_provider_recv_shift($q);
21       provider_sync($q,NULL,c);
22       provider_send_bool(false, $q, c);
23       provider_send_shift($q, c);
24       elem($q,x,$r,c);
25     }
26     case Dealloc: {
27       async_provider_recv_shift($q);
28       client_send_label(Dealloc, $r, c);
29       client_send_shift($r, c);
30       async_client_recv_done($r, c);
31       client_sync($r,NULL,c);
32       provider_sync($q,NULL,c);
33       provider_send_done($q, c);
34     }
```

```
35      }
36  }
37  void empty(queue_p $q, n_clock *c) {
38    switch (provider_recv_label($q, c)) {
39      case Enq: {
40        int y;
41        async_provider_recv_int($q, &y, c);
42        queue $e = empty_spawn(c);
43        provider_sync($q,(void**)&y,c);
44        elem($q, y, $e, c);
45      }
46      case Deq: {
47        async_provider_recv_shift($q);
48        provider_sync($q,NULL,c);
49        provider_send_label(None, $q, c);
50        provider_send_shift($q, c);
51        empty($q,c);
52      }
53      case IsEmpty: {
54        async_provider_recv_shift($q);
55        provider_sync($q,NULL,c);
56        provider_send_bool(true, $q, c);
57        provider_send_shift($q, c);
58        empty($q,c);
59      }
60      case Dealloc: {
61        async_provider_recv_shift($q);
62        provider_sync($q,NULL,c);
63        provider_send_done($q, c);
64      }
65      }
66  }
```

Code 5.12: Non-blocking implementation of a queue, in CC0's low-level target language.

We focus on explaining the *elem* function, which is the one carrying more changes.

Line 1 shows the declaration of the function, with the arguments, `queue_p $q` representing the channel that the process is providing, `int x` holds the element of the queue, `queue $r` is a client channel of our process, depicting the rest of the queue, and, finally, `n_clock *c` is the process' clock, measuring span, work and the number of processes spawned.

Line 2 represents the instruction to receive a label. Note that there is no synchronization before, which means that all requests were synchronized when this function is called. Each different label has a different set of instructions to execute, implementing the type explained in Code 3.1.

The reception of the label `Enq` is followed by receiving an integer. We first declare the new variable (line 4) and store the request using the address of this variable (line 5). We do not need the value to be enqueued immediately so, first, we can propagate the label to our client (line 6), after which we need to synchronize (line 7), by passing the address of the variable, delegating to the runtime system the job of assigning the variable to the adequate memory position. This synchronization was required considering that, in line 8, we need to send the value we received to our client.

After receiving the label `Deq`, we start sending, so first we need to receive a shift. Internally, our request (line 12) stores a `NULL` pointer, indicating that the request is either a shift or a `end`. We need to synchronize the channel directly after, in order to execute the sending functions, passing a `NULL` pointer to the synchronization functions, implying that the channel must be completely synchronized. Lines 14-17 do not suffer any change from the blocking model and simply implement the rest of the type.

The case for the label `IsEmpty` is equivalent to that of the label `Deq`. The label `Dealloc` is the one that carries the most potential benefit for our system. After receiving the label, we request a shift (line 27) and, before synchronizing this shift, we propagate the label (line 28) and send a shift (line 29) to our client. We are expecting our client to execute a `close`, so we request a `end` (line 30), which we synchronize (line 31) immediately after. When all this is executed, we finally synchronize the shift (line 32) from our provider, and we `close` (line 33) the channel.

5.3 Experimental Evaluation

The benchmarking suite[1], with a small explanation of each file, can be found in Table 5.1, adapted from [30]. A more detailed explanation follows.

- `bitstring1` executes a sequence of 2000 increments and checks if the value is correct.
- `bitstring3` executes the equivalent of 500 increments, using internal choice, by spawning one new process for each increment.

[1] available at `http://www.cs.cmu.edu/~fp/misc/cc0-bench.tgz`

- `bst` implements an unbalanced binary search tree, used to sort an array of 7 elements.
- `insert-sort` sorts an array of 6 elements, using insertion sort.
- `mergesort1` sorts an array of 6 elements using Fibonacci trees.
- `mergesort3` sorts an array of 6 elements using a bottom up strategy and binary trees.
- `mergesort4` sorts an array of 6 elements using traditional merge sort.
- `odd-even-sort1` sorts an array of 6 elements using original odd-even-sort.
- `odd-even-sort4` sorts an array of 6 elements, only the `tail` element counts down to 0 and each element swaps between right and left while the network is being built.
- `odd-even-sort6` sorts an array of 6 elements emphazising message swapping.
- `parfib` calculates the eighth number of Fibonacci, using message swapping.
- `primes` calculates the first 500 primes using a sequential prime sieve.
- `queue-notail`, a queue of 3000 elements, with looping processes instead of tail recursion.
- `queue`, a queue with 3000 elements, with processes executing tail recursion.
- `reduce` executes a reduce operation on a tree with 1024 elements and each element is a leaf.
- `seg` places 1000 elements in a segmented list and calculates their sum.
- `siege-eager` is the implementation of an eager sieve of Erastothenes, calculates the 500th prime.
- `sieve-lazy` implements a lazy version of the sieve of Erastothenes, calculating the 500th prime.
- `stack` implements a stack of 2000 elements.

bitstring1	bitstrings with external choice	parfib	parallel naive Fibonacci, simulating fork/join
bitstring3	bitstrings with internal choice	primes	prime sieve (sequential)
bst	binary search trees, tree sort	queue-notail	queues without tail calls
insert-sort	insertion sort using a queue	queue	queues written naturally
mergesort1	mergesort with Fibonacci trees	reduce	reduce and scan on parallel sequences
mergesort3	mergesort with binary trees	seg	list segments
mergesort4	mergesort with binary trees, sequential merge	sieve-eager	eager prime sieve merge
odd-even-sort1	odd/even sort, v1	sieve-lazy	lazy prime sieve
odd-even-sort4	odd/even sort, v4	stack	a simple stack
odd-even-sort6	odd/even sort, v6		

Table 5.1: CC0 benchmarking suite

All benchmarks were run on a 2015 Macbook Pro, with a 2.7 GHz Intel Core I5 (2 cores) processor and 8 GB RAM. The detailed results are presented in Table 5.2. Figures 5.1 and 5.2 aggregate these results in a more friendly outlook, comparing span in the two models and the speedup when going from the blocking to the non-blocking model.

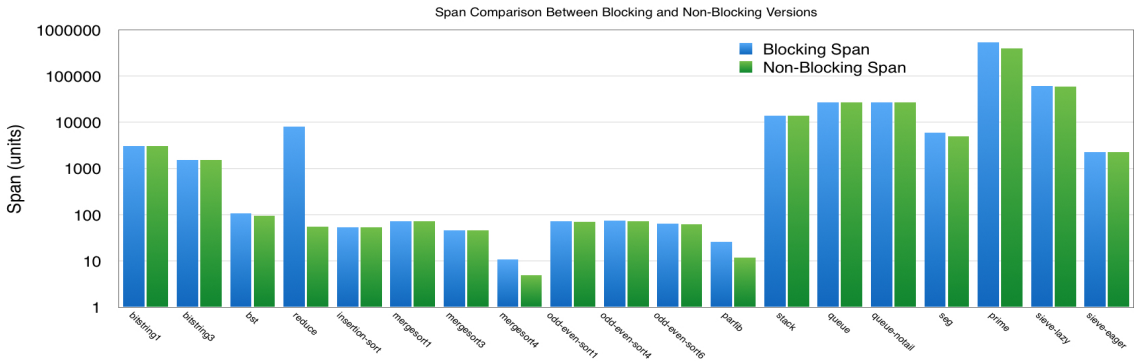


Figure 5.1: Blocking and Non-Blocking span benchmarks on a log scale.

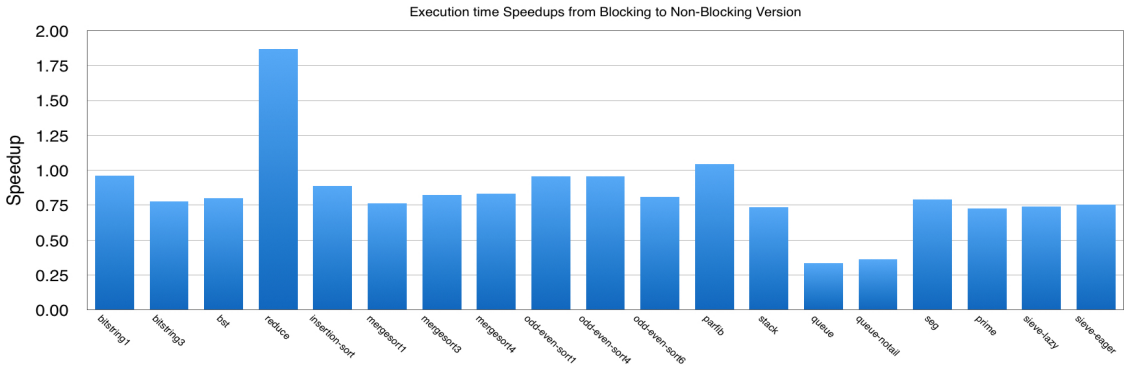


Figure 5.2: Execution time speedup from Blocking to Non-Blocking Version.

Program	Version	Span	Work	Number of Processes	Execution time (ms)	Speedup ($\frac{\text{Blocking Execution Time}}{\text{Non-Blocking Execution Time}}$)
bitstring1	Blocking	3096	8084	11	12.69	0.96
	Non Blocking	3072	8084	11	13.17	
bitstring3	Blocking	1525	4016	501	9.39	0.78
	Non Blocking	1525	4016	501	12.10	
bst	Blocking	109	259	37	0.76	0.80
	Non Blocking	97	259	37	0.95	
reduce	Blocking	8191	12282	2047	239.80	1.87
	Non Blocking	56	12282	2047	128.17	
insertion-sort	Blocking	54	96	7	0.24	0.89
	Non Blocking	54	96	7	0.27	
mergesort1	Blocking	73	215	53	0.93	0.76
	Non Blocking	73	215	53	1.22	
mergesort3	Blocking	46	155	15	0.33	0.83
	Non Blocking	46	155	15	0.40	
mergesort4	Blocking	11	30	15	0.25	0.83
	Non Blocking	5	30	15	0.30	
odd-even-sort1	Blocking	72	214	15	0.44	0.96
	Non Blocking	70	214	15	0.46	
odd-even-sort4	Blocking	76	280	15	0.46	0.96
	Non Blocking	74	280	15	0.48	
odd-even-sort6	Blocking	65	196	21	0.42	0.81
	Non Blocking	62	196	21	0.52	
parfib	Blocking	26	268	67	1.20	1.04
	Non Blocking	12	268	67	1.15	
stack	Blocking	14011	20012	2002	208.75	0.73
	Non Blocking	14011	20012	2002	284.30	
queue	Blocking	27010	18024012	3001	9021.79	0.34
	Non Blocking	27007	18024012	3001	26734.11	
queue-notail	Blocking	27010	18024012	3001	9161.20	0.36
	Non Blocking	27007	18024012	3001	25184.28	
seg	Blocking	6007	12010	3004	252.43	0.79
	Non Blocking	5005	12010	3004	319.15	
prime	Blocking	532192	532192	502	1376.44	0.73
	Non Blocking	399144	532192	502	1894.44	
sieve-lazy	Blocking	60189	909998	7137	732.68	0.74
	Non Blocking	59692	909998	7137	987.29	
sieve-eager	Blocking	2286	157738	26464	693.71	0.75
	Non Blocking	2286	157738	26464	919.53	

Table 5.2: Blocking and Non-blocking benchmarks

Most of the examples across our benchmark suite bring about a small improvement to span, where reduce, mergesort4, and parfib see the most noticeable ones. In the case of reduce, some of the difference could be recovered by fine-tuning the CC0 source program; it is interesting that our generic technique can make up for a performance bug (when considered under the blocking semantics) introduced by the programmer. This shows that, at the very least, our implementation can help identify some performance issues in the given code.

As Figure 5.2 shows, the overhead of maintaining the request queue is considerable in some examples. Only in reduce and parfib do we realize an actual performance improvement. The queue examples show the most dire decrease in performance, which can be partially explained by the overhead on increasing the capacity of the request queues. Another possible explanation is the high amount of processes created for these two examples, which, with the overhead of the heavy requests queues, may not fit in memory, requiring data to be stored in the disk.

As expected, both the number of processes spawned and the work is constant when going from the blocking to the non-blocking version.

Chapter 6

Conclusion

In this dissertation, we implemented a non-blocking model of receiving messages in Concurrent C0, a language based on a Curry-Howard correspondence between session types and linear logic.

This model led to a new semantics of the language, a non-blocking semantics, opposed to the existing blocking one. We instrumented both these semantics to include costs on each operation, using the span and work model, which provide an abstract analytical measure of parallelism in the computation.

We also defined a translation function from the original Concurrent C0 to our non-blocking extension and demonstrated some properties about our model, assuming correctness:

- Span in a non-blocking configuration is always less or equal than the span on a blocking configuration.

- Work is the same across both models.

Finally, we executed an experimental evaluation of both models on a set of representative examples. We found that it is hard to reap the practical benefit of span's improvement, as the overhead of the implementation of the data structures that support our model is too great to gain an increase in execution time.

6.1 Future Work

We would like to point out some possible future continuation to the work presented in this thesis:

- A rigorous proof of Conjecture 1 to confirm correctness of our non-blocking model was not included in this thesis, although we presented a sketch of a possible proof.
- Since recipient behavior on input is opaque to the sender, we may be able to craft an optimization, which avoids the slowdown in the common case where no improvement in the span is available. For this purpose, we would likely combine our fully dynamic technique with static dependency analysis to use non-blocking input only where promising.
- Our cost semantics could be adapted as a tool in performance debugging, which may be helpful in particular to novice programmers with little experience in concurrency.
- Our translation function has a working version, implemented in Haskell. However, it was not incorporated into the CC0 compiler, which is written in ML.
- Finally, just as there is a connection between session types and intuitionistic linear logic and asynchronous communication and polarised logic, we suspect that there might be a connection between non-blocking asynchronous communication and some branch of logic.

Bibliography

[1] Arnold, R. (2010). C0, an imperative programming language for novice computer scientists.

[2] Baker, Jr., H. C. and Hewitt, C. (1977). The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA. ACM.

[3] Balzer, S. and Pfenning, F. (2015). Objects as session-typed processes. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 13–24, New York, NY, USA. ACM.

[4] Blelloch, G. E. (1996). Programming parallel algorithms. *Commun. ACM*, 39(3):85–97.

[5] Caires, L. and Pfenning, F. (2010). Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France. Springer LNCS 6269.

[6] Caires, L., Pfenning, F., and Toninho, B. (2016). Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26:367–423.

[7] Cardelli, L. and Gordon, A. D. (2000). Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213.

[8] Cervesato, I. and Scedrov, A. (2009). Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044 – 1077. Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[9] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*, chapter 27. McGraw-Hill Higher Education, 2nd edition.

[10] Curry, H. B., Feys, R., Craig, W., and Craig, W. (1958). *Combinatory logic, vol. 1*. North-Holland Publ.

[11] DeYoung, H., Caires, L., Pfenning, F., and Toninho, B. (2012). Cut reduction in linear logic as asynchronous session-typed communication.

[12] Dezani-Ciancaglini, M., de'Liguoro, U., and Yoshida, N. (2008). *On Progress for Structured Communications*, pages 257–275. Springer Berlin Heidelberg, Berlin, Heidelberg.

[13] Friedman, D. P., David, and Wise, S. (1978). Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, pages 289–296.

[14] Gay, S. J. and Vasconcelos, V. T. (2010). Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50.

[15] Guenot, N. (2014). Session types, solos, and the computational contents of the sequent calculus. Talk at the Types Meeting.

[16] Honda, K. (1993). Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715.

[17] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems*, ESOP'98, pages 122–138. Springer LNCS 1381.

[18] Howard, W. A. (1995). The formulae-as-types notion of construction.

[19] Laneve, C. and Victor, B. (2003). Solos in concert. *Mathematical Structures in Computer Science*, 13(5):657–683.

[20] Milner, R. (1999). *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

[21] Pfenning, F. and Arnold, R. (2010). C0 language.

[22] Pfenning, F. and Griffith, D. (2015). Polarized substructural session types. In Pitts, A., editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England. Springer LNCS 9034. Invited talk.

[23] Pfenning, F. and Simmons, R. J. (2009). Substructural operational semantics as ordered logic programming. In *Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on*, pages 101–110.

[24] Sangiorgi, D. and Walker, D. (2001). *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[25] Silva, M., Florido, M., and Pfenning, F. (2016). Non-blocking concurrent imperative programming with session types. In *Proceedings of the 4th International Workshop on Linearity (LINEARITY'16)*.

[26] Toninho, B., Caires, L., and Pfenning, F. (2012). Functions as session-typed processes. In Birkedal, L., editor, *15th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS'12, pages 346–360, Tallinn, Estonia. Springer LNCS.

[27] Toninho, B., Caires, L., and Pfenning, F. (2013). Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 350–369, Berlin, Heidelberg. Springer-Verlag.

[28] Wadler, P. (2012). Propositions as sessions. In *ACM SIGPLAN Notices*, volume 47, pages 273–286. ACM.

[29] Wadler, P. (2015). Propositions as types. *Commun. ACM*, 58(12):75–84.

[30] Willsey, M., Prabhu, R., and Pfenning, F. (2016). Design and implementation of concurrent c0. In *Proceedings of the 4th International Workshop on Linearity (LINEARITY'16)*.

Appendix A

Binary Search Tree Implementations

```
list $c nil() {   // implements the empty list
  $c.Nil;
  close($c);
}

list $c cons(int n, list $d) {    // a process holding 1 element of the list
  $c.Cons;
  send($c, n);
  $c = $d;
}

list $c append(list $d, list $e) { // appends list $e to the end of list $d
  switch ($d) {
    case Nil: {
      wait($d);
      $c = $e;
    }
    case Cons: {
      $c.Cons;
      int x = recv($d);
      send($c, x);
      $c = append($d, $e);
    }
  }
}

tree $c node(int x, tree $l, tree $r) { // process implementing a node of the tree
  switch ($c) {
    case Insert: {
      int y = recv($c);
      if (y < x) {     // it is a search tree, but not balanced, order must be preserved
        $l.Insert; send($l, y);    // y is lesser than x so it goes to the left subtree
        $c = node(x, $l, $r);      // recursive call
      } else if (x == y) {
```

```
        $c = node(y, $l, $r);      // y is the same as x, so no insert is done
      } else {
        $r.Insert; send($r, y);  // y is greater than x so it goes to the right subtree
        $c = node(x, $l, $r);    // recursive call
      }
    }
    case Find: {
      int y = recv($c);
      if (y < x) {
        $l.Find; send($l, y);  // value is lesser, search in left subtree
        bool b = recv($l); send($c, b);
      } else if (x == y) {
        send($c, true);    // found the value
      } else {
        $r.Find; send($r, y);  // value is greater, search in right subtree
        bool b = recv($r); send($c, b);
      }
      $c = node(x, $l, $r); // recursive call
    }
    case Reduce: {
      int init = recv($c);
      reduce_fn* f = recv($c);  // f is the reduction function
      $l.Reduce; send($l, init); send($l, f);  // propagates label and values to the right subtree
      int y = recv($l); wait($l);  // receives result and closes the client
      $r.Reduce; send($r, init); send($r, f);  // propagates label and values to the left subtree
      int z = recv($r); wait($r); // receives result and closes the client
      send($c, (*f)(y, x, z)); // applies the reduction function and sends the result
      close($c); // closes the provider
    }
    case ToList : {
      $r.ToList; list $right = recv($r); // receives the right subtree as a list
      wait($r);
      $right = cons(x, $right);
      $l.ToList; list $left = recv($l);  // receives the left subtree as a list
      wait($l);
      list $d = append($left, $right);  // joins the whole tree as a list
      send($c, $d);
      close($c);
    }
  }
}

tree $c leaf() {  // a process implementing a leaf of the tree
  switch ($c) {
    case Insert: {
      int x = recv($c);
      tree $l = leaf();    // creates new child nodes
      tree $r = leaf();
      $c = node(x, $l, $r); // continues as a node
    }
    case Find: {
      int x = recv($c);
```

```
      send($c, false);  // did not find the element
      $c = leaf(); // recursive call
    }
    case Reduce: {
      int init = recv($c);
      reduce_fn* f = recv($c);
      send($c, init);  // has no element so does not apply the reduction
      close($c);
    }
    case ToList: {
      list $d = nil();  // has no element, so send the empty list
      send($c, $d); close($c);
    }
  }
}
```

Code A.1: Implementation of an unbalanced binary search tree, in CC0.

```
1  list $c nil() {  // no change from the blocking version
2    $c.Nil;
3    close($c);
4  }
5
6  list $c cons(int n, list $d) { // no change from the blocking version
7    $c.Cons;
8    send($c, n);
9    $c = $d;
10 }
11
12 list $c append(list $d, list $e) {
13   switch ($d) {
14     case Nil: {
15       async_wait($d);
16       client_sync($d, end);  // needs to sync immediately because of forward
17       $c = $e;
18     }
19     case Cons: {
20       $c.Cons;
21       int x = async_recv($d);
22       sync($d, x);   // needs to sync immediately to send the variable
23       send($c, x);
24       $c = append($d, $e);
25     }
26   }
27 }
28
29 tree $c node(int x, tree $l, tree $r) {
30   switch ($c) {
31     case Insert: {
32       int y = async_recv($c);
33       sync($c, y); // needs to sync immediately because of line 34
34       if (y < x) {
```

```
35          $l.Insert; send($l, y);
36          $c = node(x, $l, $r);
37        } else if (x == y) {
38          $c = node(y, $l, $r);
39        } else {
40          $r.Insert; send($r, y);
41          $c = node(x, $l, $r);
42        }
43      }
44      case Find: {
45        int y = async_recv($c);
46        shift = async_recv($c);
47        sync($c, y);  // only needs to sync the y because of line 48
48        if (y < x) {
49          client_sync($l, shift);   // needs to sync shift because unsync'ed shift in line 52
50          $l.Find; send($l, y); send($l, shift);
51          bool b = async_recv($l);
52          shift = async_recv($l);
53          sync($l, b);  // syncs b to send it
54          sync($c, shift);  // syncs shift to correct polarity
55          send($c, b); send($c, shift);
56        } else if (x == y) {
57          sync($c, shift);  // syncs shift to be able to send
58          send($c, true); send($c, shift);
59        } else { // case for the right subtree is similar to left subtree
60          client_sync($r, shift);
61          $r.Find; send($r, y); send($r, shift);
62          bool b = async_recv($r); shift = async_recv($r);
63          sync($r, b); sync($c, shift);
64          send($c, b); send($c, shift);
65        }
66        $c = node(x, $l, $r);
67      }
68      case Reduce: {
69        int init = async_recv($c);
70        reduce_fn* f = async_recv($c);
71        shift = async_recv($c);
72        sync($l, shift); $l.Reduce;  // needs to sync shift from line 52
73        sync($c, init); // syncs init to send
74        send($l, init);
75        sync($c, f); // syncs f to send
76        send($l, f);
77        int y = async_recv($l); async_wait($l);
78
79        sync($r, shift); $r.Reduce; // similar to left subtree
80        send($r, init);  send($r, f);
81        int z = async_recv($r); async_wait($r);
82
83        sync($l, y); sync($r, z);
84        sync($c, shift); send($c, (*f)(y, x, z));
85
86        sync($l, end); // syncs requests from line 77, will close after
```

```
87          sync($r, end);
88          close($c);
89        }
90        case ToList : { // syncs are similar to case Reduce
91          shift = async_recv($c);
92          sync($r, shift); $r.ToList; send($r, shift);
93          list $right = async_recv($r);
94          async_wait($r);
95          sync($r, $right);
96          $right = cons(x, $right);
97          sync($l, shift); $l.ToList; send($l, shift);
98          list $left = async_recv($l);
99          async_wait($l);
100         sync($l, $left);
101         list $d = append($left, $right);
102         send($c, $d);
103         sync($r, end); sync($l, end);
104         close($c);
105       }
106     }
107   }
108
109   tree $c leaf() {
110     switch ($c) {
111       case Insert: {
112         int x = async_recv($c);
113         tree $l = leaf();
114         tree $r = leaf();
115         sync($c, x); // syncs to continue as node
116         $c = node(x, $l, $r);
117       }
118       case Find: {
119         int x = async_recv($c);
120         shift = async_recv($c);
121         sync($c, shift); // syncs to send
122         send($c, false); send($c, shift);
123         $c = leaf();
124       }
125       case Reduce: {
126         int init = async_recv($c);
127         reduce_fn* f = async_recv($c);
128         shift = async_recv($c);
129         sync($c, shift); // syncs the 3 requests at once
130         send($c, init);
131         close($c);
132       }
133       case ToList: {
134         shift = async_recv($c);
135         list $d = nil();
136         sync($c, shift); // syncs to send
137         send($c, $d); close($c);
138       }
```

```
139     }
140 }
```

Code A.2: Non-blocking implementation of an unbalanced binary search tree, in CC0's simplified target language.

Appendix B

Translation auxiliary functions

We present three set of definitions. In Table B.1, we show the definitions of the auxiliary functions used in Table 4.2. Table B.2 showcases the rules to ascertain the loop-carried requests in a while cycle. Finally, Table B.3 handles the rules from recursive calls or calls to other functions.

The rules in Tables 4.2, B.2 and B.3 are very similar. Table B.2 differs from the other two by not generating any instruction, simply updates and returns a new table. Table B.3 differs from Table 4.2 in the receive functions, who do not add a request to the table, and adds extra rules for synchronization functions, who remove requests from the table.

The declaration of each function will appear with a type defined in Haskell style. For example, the translation function in Table 4.2 would be declared as: $[\![]\!] :: (\mathsf{Instruction}, [(\mathsf{String}, \mathsf{Channel})]) \rightarrow (\mathsf{Instruction}, [(\mathsf{String}, \mathsf{Channel})])$. To abbreviate, we define $\mathsf{Table} = [(\mathsf{String}, \mathsf{Channel})]$.

$$
\begin{aligned}
&\mathsf{check\_exp} :: \mathsf{Table} \rightarrow \mathsf{Exp} \rightarrow \mathsf{Table} \\
&\mathsf{check\_exp}\ \sigma\ n &&= [] \\
&\mathsf{check\_exp}\ \sigma\ b &&= [] \\
&\mathsf{check\_exp}\ []\ x &&= [] \\
&\mathsf{check\_exp}\ ((y, \$c) : \sigma)\ x &&= \text{if } x = y \\
&&&\quad \text{then } [(y, \$c)] \\
&&&\quad \text{else } \mathsf{check\_exp}\ \sigma\ x \\
&\mathsf{check\_exp}\ \sigma\ e_1 + e_2 &&= (\mathsf{check\_exp}\ \sigma\ e_1) \cup (\mathsf{check\_exp}\ \sigma\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{check\_shift} :: \mathsf{Table} \to \mathsf{Channel} \to \mathsf{Table}\\
&\mathsf{check\_shift}\ [\ ]\ \$d = [\ ]\\
&\mathsf{check\_shift}\ ((y, \$c) : \sigma)\ \$d = \text{if } \$c = \$d \ \wedge\ y = \mathsf{shift}\\
&\qquad\qquad \text{then } [(y, \$c)]\\
&\qquad\qquad \text{else } \mathsf{check\_shift}\ \sigma\ \$d\\
&\mathsf{check\_wait} :: \mathsf{Table} \to \mathsf{Channel} \to \mathsf{Table}\\
&\mathsf{check\_wait}\ [\ ]\ \$d = [\ ]\\
&\mathsf{check\_wait}\ ((y, \$c) : \sigma)\ \$d = \text{if } \$c = \$d \ \wedge\ y = \mathsf{end}\\
&\qquad\qquad \text{then } [(y, \$c)]\\
&\qquad\qquad \text{else } \mathsf{check\_wait}\ \sigma\ \$d\\
&\mathsf{check\_waits} :: \mathsf{Table} \to \mathsf{Table}\\
&\mathsf{check\_waits}\ [\ ] = [\ ]\\
&\mathsf{check\_waits}\ ((y, \$c) : \sigma) = \text{if } y = \mathsf{end}\\
&\qquad\qquad \text{then } [(y, \$c)] \cup (\mathsf{check\_waits}\ \sigma)\\
&\qquad\qquad \text{else } \mathsf{check\_waits}\ \sigma\\
&\mathsf{check\_arg} :: \mathsf{Table} \to \mathsf{Exp} \to \mathsf{Table}\\
&\mathsf{check\_arg}\ \sigma\ A = \mathsf{check\_exp}\ \sigma\ A\\
&\mathsf{check\_arg} :: \mathsf{Table} \to \mathsf{Channel} \to \mathsf{Table}\\
&\mathsf{check\_arg}\ \sigma\ \$d = \mathsf{check\_wait}\ \sigma\ \$d\\
&\mathsf{sync\_all} :: \mathsf{Table} \to \mathsf{Table}\\
&\mathsf{sync\_all}\ \sigma = \sigma\\
&\mathsf{aggr\_reqs} :: \mathsf{Channel} \to \mathsf{Table} \to \mathsf{Table}\\
&\mathsf{aggr\_reqs}\ \$c\ [\ ] = [\ ]\\
&\mathsf{aggr\_reqs}\ \$c\ ((x, \$d) : l) = \text{if } \$c = \$d\\
&\qquad\qquad \text{then } (\mathsf{aggr\_reqs}\ \$c\ l) \cup [(x, \$d)]\\
&\qquad\qquad \text{else } (\mathsf{aggr\_reqs}\ \$c\ l)\\
&\mathsf{gen\_instr} :: \mathsf{Table} \to \mathsf{Instruction}\\
&\mathsf{gen\_instr}\ ((x, \$c) : l) = \mathsf{sync}(\$c, x)\\
&\mathsf{gen\_instr}\ ((\mathsf{shift}, \$c) : l) = \mathsf{sync}(\$c, \mathsf{shift})\\
&\mathsf{gen\_instr}\ ((\mathsf{end}, \$c) : l) = \mathsf{sync}(\$c, \mathsf{end})\\
&\mathsf{rem\_sync} :: \mathsf{Table} \to \mathsf{Table} \to \mathsf{Table}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{rem\_sync}\ ((x,\$c):l)\ ((y,\$d):\sigma) = \text{if } \$c \neq \$d \\
&\qquad\qquad \text{then } ((y,\$d) : (\mathsf{rem\_sync}\ ((x,\$c):l)\ \sigma)) \\
&\qquad\qquad \text{else if } \quad y = x \\
&\qquad\qquad\qquad \text{then} \sigma \\
&\qquad\qquad\qquad \text{else } \mathsf{rem\_sync}\ ((x,\$c):l)\ \sigma \\
&\mathsf{update\_table} :: \mathsf{Table} \to \mathsf{Table} \to \mathsf{Table} \\
&\mathsf{update\_table}\ []\ \sigma \qquad\qquad = \sigma \\
&\mathsf{update\_table}\ ((x,\$c):l)\ \sigma = \sigma' \\
&\qquad\qquad \text{where} l_1 = \mathsf{aggr\_reqs}\ \$c\ ((x,\$c):l) \\
&\qquad\qquad\qquad \sigma_1 = \mathsf{rem\_sync}\ l_1\ \sigma \\
&\qquad\qquad\qquad \sigma' = \mathsf{update\_table}\ l \setminus l_1\ \ \sigma_1 \\
&\mathsf{generate\_sync} :: \mathsf{Table} \to \mathsf{Table} \to (\mathsf{Instruction}, \mathsf{Table}) \\
&\mathsf{generate\_sync}\ []\ \sigma \qquad\qquad = (NULL, \sigma) \\
&\mathsf{generate\_sync}\ ((x,\$c):l)\ \sigma = (sync_1\ ;\ sync_r, \sigma') \\
&\qquad\qquad \text{where} l_1 = \mathsf{aggr\_reqs}\ \$c\ ((x,\$c):l) \\
&\qquad\qquad\qquad sync_1 = \mathsf{gen\_instr}\ l_1 \\
&\qquad\qquad\qquad \sigma_1 = \mathsf{rem\_sync}\ l_1\ \sigma \\
&\qquad\qquad\qquad (sync_r, \sigma') = \mathsf{generate\_sync}\ l \setminus l_1\ \ \sigma_1
\end{aligned}
$$

Table B.1: Auxiliary functions for the translation.

In the check_exp function, we omitted the rules for every type of expression, there would be a case for each arithmetic and boolean operator.

The generate_sync function requires some explanation. It receives as input a list of requests to be synchronized at that point, as well as the table of requests. Each channel in this list of requests is handled separately. First, we aggregate all the requests using a specific channel (aggr_reqs function), putting the most recent request, which is the last to be handled, in the front of this new list, which contains requests only from one channel. We then generate a synchronization instruction (gen_instr) based on the first request of this list, note that, by synchronizing on the most recent request, all other earlier requests will also be synchronized. The table of requests is updated to remove these synchronized requests, and generate_sync is called recursively until there are no more requests to synchronize.

The update_table function behaves similarly to generate_sync but does not generate any instruction. It will be used as an auxiliary function on Table B.2.

$$
\begin{aligned}
&\mathsf{loop\_carried\_req} :: \mathsf{Instruction} \to \mathsf{Table} \to \mathsf{Table}\\
&\mathsf{loop\_carried\_req}\ (\mathsf{shift} = \mathsf{recv}(\$d))\ \sigma = \sigma \cup [(\mathsf{shift}, \$d)]\\
&\mathsf{loop\_carried\_req}\ (x = \mathsf{recv}(\$d))\ \sigma = \sigma \cup [(x, \$d)]\\
&\mathsf{loop\_carried\_req}\ (\mathsf{wait}(\$d))\ \sigma = \sigma \cup [(\mathsf{end}, \$d)]\\
&\mathsf{loop\_carried\_req}\ (\mathsf{switch}(\$d)\{lab_i \to P_i\})\ \sigma = \bigcup_{i \in I} \sigma_i\\
&\qquad \text{where } l = \mathsf{check\_shift}\ \sigma\ \$d\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ l\ \sigma\\
&\qquad\quad \sigma_i = \mathsf{loop\_carried\_req}\ P_i\ \sigma'\\
&\mathsf{loop\_carried\_req}\ (\$d = f(\mathsf{args}))\ \sigma = \sigma'\\
&\qquad \text{where } l_i = \mathsf{check\_arg}\ \sigma\ \mathsf{arg_i}\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ (\bigcup_{i \in I} l_i)\ \sigma\\
&\mathsf{loop\_carried\_req}\ (\mathsf{close}(\$d))\ \sigma = []\\
&\mathsf{loop\_carried\_req}\ (\mathsf{send}(\$d, e))\ \sigma = \sigma'\\
&\qquad \text{where } l_1 = \mathsf{check\_shift}\ \sigma\ \$d\\
&\qquad\quad l_2 = \mathsf{check\_exp}\ \sigma\ e\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ (l_1 \cup {}_2)\ \sigma\\
&\mathsf{loop\_carried\_req}\ (\mathsf{send}(\$d, \mathsf{shift}))\ \sigma = \sigma'\\
&\qquad \text{where } l = \mathsf{check\_shift}\ \sigma\ \$d\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ l\ \sigma\\
&\mathsf{loop\_carried\_req}\ (\$d.lab)\ \sigma = \sigma'\\
&\qquad \text{where } l = \mathsf{check\_shift}\ \sigma\ \$d\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ l\ \sigma\\
&\mathsf{loop\_carried\_req}\ (\$d = \$e)\ \sigma = []\\
&\mathsf{loop\_carried\_req}\ (P\ ;\ Q)\ \sigma = \sigma''\\
&\qquad \text{where } \sigma' = \mathsf{loop\_carried\_req}\ P\ \sigma\\
&\qquad\quad \sigma'' = \mathsf{loop\_carried\_req}\ Q\ \sigma'\\
&\mathsf{loop\_carried\_req}\ (x = e)\ \sigma = \sigma'\\
&\qquad \text{where } l_1 = \mathsf{check\_exp}\ \sigma\ x\\
&\qquad\quad l_2 = \mathsf{check\_exp}\ \sigma\ e\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ (l_1 \cup l_2)\ \sigma\\
&\mathsf{loop\_carried\_req}\ (\text{if } (b) \text{ then } P \text{ else } Q)\ \sigma = \sigma_1 \cup \sigma_2\\
&\qquad \text{where } l = \mathsf{check\_exp}\ \sigma\ b\\
&\qquad\quad \sigma' = \mathsf{update\_table}\ l\ \sigma\\
&\qquad\quad \sigma_1 = \mathsf{loop\_carried\_req}\ P\ \sigma'\\
&\qquad\quad \sigma_2 = \mathsf{loop\_carried\_req}\ Q\ \sigma'
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{loop\_carried\_req}\ (\text{while}\ (b)\ \text{do}\ P)\ \sigma = \sigma_1 \\
&\qquad\qquad \text{where} l = \mathsf{check\_exp}\ \sigma\ b \\
&\qquad\qquad\qquad \sigma' = \mathsf{update\_table}\ l\ \sigma \\
&\qquad\qquad\qquad \sigma_1 = \mathsf{loop\_carried\_req}\ P\ \sigma' \\
&\mathsf{loop\_carried\_req}\ (f(args))\ \sigma = []
\end{aligned}
$$

Table B.2: Definition of translation rules for while loops.

We now present the rules for the recursive call of translation to a new function. We will only present the rules that suffer any change from their original definition.

$$
\begin{aligned}
&\mathsf{recurse} :: \mathsf{Instruction} \rightarrow \mathsf{Table} \rightarrow (\mathsf{Instruction}, \mathsf{Table}) \\
&\mathsf{recurse}\ (\mathsf{shift} = \mathsf{recv}(\$d))\ \sigma = (\mathsf{shift} = \mathsf{recv}(\$d), \sigma) \\
&\mathsf{recurse}\ (x = \mathsf{recv}(\$d))\ \sigma = (x = \mathsf{recv}(\$d), \sigma) \\
&\mathsf{recurse}\ (\mathsf{wait}(\$d))\ \sigma = (\mathsf{wait}(\$d), \sigma) \\
&\mathsf{recurse}\ (\mathsf{sync}(\$d, x))\ \sigma \quad = (\mathsf{sync}(\$d, x), \sigma_1) \\
&\qquad\qquad\qquad \text{where}\ \sigma_1 = \mathsf{rem\_sync}\ [(x, \$d)]\ \sigma \\
&\mathsf{recurse}\ (\mathsf{sync}(\$d, \mathsf{shift}))\ \sigma = (\mathsf{sync}(\$d, \mathsf{shift}), \sigma_1) \\
&\qquad\qquad\qquad \text{where}\ \sigma_1 = \mathsf{rem\_sync}\ [(\mathsf{shift}, \$d)]\ \sigma \\
&\mathsf{recurse}\ (\mathsf{sync}(\$d, \mathsf{end}))\ \sigma = (\mathsf{sync}(\$d, \mathsf{end}), \sigma_1) \\
&\qquad\qquad\qquad \text{where}\ \sigma_1 = \mathsf{rem\_sync}\ [(\mathsf{end}, \$d)]\ \sigma \\
&\mathsf{recurse}\ (P\ ;\ Q)\ \sigma = (P'\ ;\ Q',\ \sigma'') \\
&\qquad\qquad \text{where} (P', \sigma') = \mathsf{recurse}\ P\ \sigma \\
&\qquad\qquad\qquad \text{if}\ \sigma' = [] \\
&\qquad\qquad\qquad \text{then}\ (Q', \sigma'') = (Q, \sigma') \\
&\qquad\qquad\qquad \text{else}\ (Q', \sigma'') = \mathsf{recurse}\ Q\ \sigma'
\end{aligned}
$$

Table B.3: Definition of translation rules for function calls.