Parallel Combinators for the Encore Programming Language

Daniel Sean McCain
Abstract

Parallel Combinators for the Encore Programming Language

Daniel Sean McCain

With the advent of the many-core architecture era, it will become increasingly important for programmers to utilize all of the computational power provided by the hardware in order to improve the performance of their programs. Traditionally, programmers had to rely on low-level, and possibly error-prone, constructs to ensure that parallel computations would be as efficient as possible. Since the parallel programming paradigm is still a maturing discipline, researchers have the opportunity to explore innovative solutions to build tools and languages that can easily exploit the computational cores in many-core architectures.

Encore is an object-oriented programming language oriented to many-core computing and developed as part of the EU FP7 UpScale project. The inclusion of parallel combinators, a powerful high-level abstraction that provides implicit parallelism, into Encore would further help programmers parallelize their computations while minimizing errors. This thesis presents the theoretical framework that was built to provide Encore with parallel combinators, and includes the formalization of the core language and the implicit parallel tasks, as well as a proof of the soundness of this language extension and multiple suggestions to extend the core language.

The work presented in this document shows that parallel combinators can be expressed in a simple, yet powerful, core language. Although this work is theoretical in nature, it is a useful starting point for the implementation of parallel combinators not only in Encore, but also in any language that has features similar to this language.
## Contents

1 Introduction ......................................................... 1
   1.1 Using Abstraction to Increase Parallel Performance ............ 1
      1.1.1 MapReduce ................................................. 2
      1.1.2 Dataflow-Based Models .................................. 3
      1.1.3 Parallelism and Common Data Structures ................. 3
      1.1.4 Orc ..................................................... 4
   1.2 Futures ....................................................... 5
   1.3 Encore ........................................................ 5
   1.4 Combinators ................................................... 6
   1.5 Thesis Contribution ........................................... 7

2 Extending Encore with Parallel Combinators ......................... 8
   2.1 Grammar ....................................................... 8
   2.2 Type System .................................................. 9
   2.3 Operational Semantics ....................................... 10
      2.3.1 Monoids in Encore ....................................... 10
      2.3.2 Configurations .......................................... 11
      2.3.3 Transition System ....................................... 12
   2.4 Parallelism .................................................... 13
   2.5 Type Soundness ............................................... 13
   2.6 Deadlock-freedom ............................................. 27

3 Additional Constructs ............................................... 34
   3.1 Core Language Extensions .................................... 34
      3.1.1 Let Bindings .............................................. 34
      3.1.2 Wildcards ............................................... 34
      3.1.3 Sequencing .............................................. 35
   3.2 Combining Combinators ....................................... 35
      3.2.1 Prune Combinator ....................................... 35
      3.2.2 Bind Combinator ......................................... 36
      3.2.3 Imitating Maybe Types .................................. 36
   3.3 Syntactic Sugar for Parallel Combinators ...................... 36
   3.4 Exceptions .................................................... 37
4 Conclusions and Future Work

4.1 Formalizing Exceptions ................................................. 39
4.2 Avoiding Intermediate Parallel Structures ......................... 39
4.3 Dealing with Unneeded Tasks and Futures ......................... 40
4.4 (Efficiently) Implementing Parallel Combinators .................. 40
4.5 Creating Benchmarks for the Parallel Combinators ............... 40
4.6 Exploring More Combinators ..................................... 41
4.7 Integrating the Semantics with Active Objects .................... 41

Bibliography .......................... 42
Chapter 1

Introduction

In the past, many performance optimizations were focused on the CPU, such as increasing clock speed or cache size. Many times, a CPU upgrade would cause previously written programs' performance to improve. As the speed of current CPUs stagnate, the focus of improving processing power has shifted to increasing computational hardware, such as incrementing the amount of processor cores. This architecture change is causing parallelism to become an important factor in high-performance software [1].

Many-core processors, which contain several tens, hundreds, or even thousands of processor cores [2], are starting to gain importance both because of their increasing computation power and their energy efficiency. In addition to these new processors, cloud computing allows us to have access to huge amounts of processing power by giving users access to multiple (virtual) computers. With computational hardware, including that in mainstream cellphones and computers, becoming more parallel and distributed, programmers must learn to take advantage of these capabilities if they wish to fully exploit these systems.

Developing parallel software is no easy task. Traditionally, the programmer had to be aware of the hardware details of the platform that would run their program to improve performance as much as possible. This would force them to manage both program correctness and communication issues, which may lead to subtle bugs which are hard to find and may only rarely appear. While sequential programming is mature and well-studied, the same cannot be said for the parallel programming paradigm. This gives researchers ample opportunities to explore innovative solutions for building tools and languages for many-core systems.

This chapter serves as an introduction to various central concepts and related work, which help motivate the work provided by this thesis.

1.1 Using Abstraction to Increase Parallel Performance

The term parallelism may not often be well understood, as it is not always accurately described. In part, this is because it can be used interchangeably with the term concurrency in some contexts. In order to differentiate these two words, I will quote the
excellent definitions given by Simon Marlow [3]:

A parallel program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more quickly. Different parts of the computation are delegated to different processors that execute at the same time (in parallel), so that results may be delivered earlier than if the computation had been performed sequentially.

In contrast, concurrency is a program-structuring technique in which there are multiple threads of control. Notionally the threads of control execute “at the same time”; that is, the user sees their effects interleaved. Whether they actually execute at the same time or not is an implementation detail: a concurrent program can execute on a single processor through interleaved execution, or on multiple physical processors.

While parallel programming is concerned only with efficiency, concurrent programming is concerned with structuring a program that needs to interact with multiple independent external agents (for example the user, a database server, and some external clients). Concurrency allows such programs to be modular; the thread that interacts with the user is distinct from the thread that talks to the database. In the absence of concurrency, such programs have to be written with event loops and callbacks – indeed, event loops and callbacks are often used even when concurrency is available, because in many languages concurrency is either too expensive, or too difficult, to use.

It is also important to differentiate between data parallelism and task parallelism. The former refers to the distribution of data across several nodes, normally to perform the same computation on the data. The latter focuses on the distribution of tasks across several nodes, each of which may be performing different computations. These tasks may be performed by threads, for example. In order to achieve the best performance, it may be necessary to have a mix of both of these forms of parallelization.

Nvidia CUDA [4] and MPI [5] are two well known parallel-computing platforms. In order to achieve efficient computations, these platforms use low-level constructs to send, coordinate and synchronize data. Our interest is in maintaining good run times by increasing program parallelism while decreasing the use of low-level constructs, which may force the developer to understand the hardware details of a specific architecture. The use of high abstraction levels would help provide semantic guarantees, as well as simplify the understanding, debugging, and testing of a parallel program.

1.1.1 MapReduce

Google’s MapReduce [6] is a well known programming model that uses a simple interface to automatically parallelize and distribute computations to various clusters. MapReduce creates an abstraction that gives its users the power to concentrate on their computation without having to manage many of the low-level tasks. It was inspired by the map and
reduce primitives from functional programming language. Users mostly only have to concentrate on writing both of these functions to express the computation. This takes away the concern about distributing and coordinating the parallel work, as well as dealing with machine, network or data failures.

The abstractions provided by MapReduce minimizes the complexity of writing parallel programs, yet it may become difficult to manage the combination of multiple MapReduce operations. The FlumeJava [7] Java library tries to solve this by deferring the evaluation of the computation with the use of classes that represent immutable parallel collections. It constructs a dataflow graph of the execution plan, optimizes it and executes it using the underlying parallel primitives, which include MapReduce, when the results are needed. The user needs no knowledge of the details of the parallel collections nor the operations’ implementation strategies to achieve efficient computations, as FlumeJava abstracts these details away.

1.1.2 Dataflow-Based Models
An alternative to increasing parallelism is through the use of dataflow models, which allow programmers to build parallel computations by only specifying the data and control dependencies. These models tend to be more explicit about data dependencies, and can be seen as graphs, where the nodes represent data and the edges represent the computations that occur between nodes. Concurrent Collections (CnC) [8] combines this model with the use of a specification language to facilitate the implementation of parallel programs. CnC provides a separation between domain experts, which might be non-professional programmers, and tuning experts, programmers that worry about efficiently implementing parallel programs. The domain expert must provide the implementation of a computation in a separate programming language, and the tuning expert maps the CnC specification to a specific target architecture. Thus, CnC serves to help with the communication between domain and tuning experts, and has been implemented in a variety of programming languages, including C++, .NET, Java and Haskell.

Another option could be using FlowPools [9], a deterministic dataflow collections abstractions for Scala. FlowPools can be seen as concurrent data structures which, through the use of a functional interface, easily compose to create more complex computations. These structures can also be generated dynamically at run time. Users are given the possibility to think about their computations as sequential programs, whereas the task of performing and optimizing the parallelization is given to the framework, which is done by using efficient non-blocking data structures to build complex parallel dataflow programs.

1.1.3 Parallelism and Common Data Structures
A problem that may appear with the previously mentioned solutions is that they excel in flat data parallelism, but not in the use of sparse data. Data Parallel Haskell (DPH) [10], influenced by the NESL programming language [11], extends the Glasgow Haskell Compiler (GHC) with the capabilities to perform efficient computations on ir-
regular data. A new data type is provided to the user, the parallel array, as well as multiple functions that operate on this data type. The provided functions are named after their sequential counterparts, differing not only in their implementation but also in their name (the parallel versions have a suffixed ‘P’ in their function name). These familiar function names, in conjunction to the provided syntactic sugar, provides users of DPH with the means to easily write parallel programs that use sparse data. A solution similar to this was proposed by Bergstrom et al. [12], where an implementation of Parallel ML was extended to include data-only flattening, which tries to take advantage of modern Multiple-Instruction-Multiple-Data architectures.

Aleksandar Prokopec et al. extended the Scala programming language collection library with a wide variety of parallel structures and operators [13], in the hope that implicit parallelism will facilitate writing reliable parallel code. Similar to DPH, this framework allows users to easily switch from sequential collections to their parallel counterparts. This is done by splitting a collection of data into subcollections which are sent to different processors, applying some operation on these subcollections, and recombining the data into a final collection. Adaptive scheduling, done by work stealing, is applied to utilize all available computation cores as efficiently as possible. Each processor maintains a task queue, and when a processor has no more elements in its queue, it can steal the oldest tasks in other queues. The function names do not differ from their sequential counterparts, nor does it specialize in sparse data, as opposed to DPH.

1.1.4 Orc

The Orc programming language [14] was the primary source of inspiration for the parallel combinators in this thesis. It expresses the control flow in concurrent programs with high-level primitive constructs, called combinators. The power of Orc comes from its implicitly distributed combinators, which are used to compose expressions. Expressions in Orc call external services, called sites, which may publish any number of results. If a site explicitly reports that they will not respond, it is considered halted. The following are the four combinators found in Orc, using \( F \) and \( G \) as expressions that contain site calls:

- The parallel combinator, \( F | G \), executes two independent expressions in parallel and publishes the results of both.
- The sequential combinator, \( F > x > G \), executes the first expression, \( F \), where each of its published values are bound to the variable \( x \) and given to a second expression, \( G \), which will run in parallel to, and independently from, the first one.
- The pruning combinator, \( F < x < G \), executes two expressions, \( F \) and \( G \), in parallel. The first expression will suspend if it depends on the value of the variable \( x \). Once the second expression publishes a value, its execution is terminated, \( x \) is assigned that value, and the first expression proceeds with its execution. This is Orc’s only mechanism to terminate parts of a computation.
- The otherwise combinator, \( F ; G \), only executes the second expression if the first one halts and has published no values.

Although Orc is a minimal language that was designed only to test and experiment
with its combinators [15], its approach to parallelism and expressive power has served as source of inspiration to library implementors of other programming languages [16].

1.2 Futures

Futures, first introduced by Baker and Hewitt [17], are computations that are evaluated asynchronously from the current computation flow. They provide programmers with the capability to start a computation in advance, usually in parallel, and request its value later.

Future variables are variables that will eventually contain the result of the future. These variables can be seen as containers that will at some point contain the value of a future computation. Once this computation writes the result to the future, it is said to be fulfilled. Depending on the semantics of the language that implements them, one might be able to pass these variables around, wait for their result, or use operators on them to create other futures.

Work done by a computation involving futures should require no more work than a sequential version of the same computation. Thus, they are an attractive addition to a language that can run computations in parallel.

Futures were first featured in the MultiLisp programming language [18], which used the future keyword to explicitly annotate expressions that could be evaluated in parallel with the rest of the program. These expressions would immediately return a future, which could be used in other expressions. If the value of the future is needed in an operation, the computation would be suspended until the value of the future is determined. These futures are considered implicit, as the programmer does not have to worry whether a variable is a future or not. If futures are explicit, it is necessary to state when to retrieve the value of a future variable. A few examples of modern languages that support futures are C++ (starting from C++11 [19]), Racket [20], and Java [21].

Speculations are similar to futures in that they represent values which are retrieved at a later time. They are, unlike futures, delayed and not always evaluated. This makes speculations useful in lazy languages, as their values may not always be needed.

The value of futures, as opposed to speculations, are generally needed. In this thesis, we consider that this may not be the case.

1.3 Encore

Encore [22] is a programming language that is being developed as part of the EU FP7 UpScale project1. Encore is a general purpose parallel object-oriented programming language that supports scalable performance. This is done by a combination of actor-based concurrency, asynchronous communication and guaranteed race freedom. While

---

1The project is partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations
the language contains many features, this section will focus on a few core language elements.

Parallelism in Encore is currently achieved by using active objects, which are objects that run in parallel. They have encapsulated fields and methods, which are accessed as other object-oriented languages. Active objects always immediately return a future as a result to their method calls, which will eventually contain the result of the executed method. However, there is no guarantee that the method is run immediately. Even though active objects are run in parallel, they each have a single thread of execution. This means that all method calls are internally serialized in an active object, and its methods are run in a sequential order.

Futures are first-class elements of Encore, which can be passed around as variables, blocked on in multiple ways, and chained to functions, similar to how callbacks work in other languages, such as Javascript. They are, however, computationally expensive to block on, as they block the active object’s thread of control.

Encore also has passive objects that return results synchronously, i.e. the calling objects block until the passive object returns its result. These correspond to the “regular” classes in other languages. These object are used to pass data between active classes and help represent state.

The language also includes anonymous functions, which can be assigned to variables, used as arguments and returned from methods. With the use of polymorphism and type inference, high-level functions can be easily written and used in Encore.

As Encore is still a young language (it has been worked on for about two years, at the time of the writing of this thesis), its syntax and some of its semantics may change. Many of the core features of the language have already been chosen and may be represented as a core language [22]. This thesis uses and extends some of these core elements, while using little of the current Encore syntax.

1.4 Combinators

A combinator is, as described by John Hughes, “a function which builds program fragments from program fragments” [23]. In essence, combinators provide the means to build powerful functions by using and combining other, smaller, functions. This gives developers the possibility to program tools without the need of specifying how the program works in great detail. This is done by focusing on high-level operations such as higher-order functions and previously defined combinators, features that are commonly found in functional programming languages.

A few examples of common combinators are the map function (which applies a function to a collection of elements), the filter function (using a predicate function on a collection of elements, it returns the collection of elements that satisfy the predicate), and the fold function (which reduces a collection of objects into a single object by using a supplied function). The use and composition of combinators have given rise to powerful libraries, such as QuickCheck [24], or parser combinators [25-28] such as Parsec [29]. Both of these examples, while having been originally been written in Haskell, have been
ported to, and inspired the creation of tools in, multiple other languages.

1.5 Thesis Contribution

Given the advantages of high-level parallel programming abstractions and combinators, such as those provided by the Orc programming language, it would seem beneficial to combine the expressive power of these features with futures. Including such combinators in Encore would give the language a simple, yet powerful tool which could be used to elegantly solve problems with implicit parallelization.

Implementing such an extension in a programming language may seem like a fairly simple task, yet there are multiple subtleties which must be taken into account in order to avoid issues during run time. Thus, it is useful to have a theoretical framework to base the implementation of such an extension on, as it gives the possibility to prove important properties, such as deadlock-freedom.

This thesis presents the theoretical framework that was built to provide Encore with such an extension. This includes the formalization of the core language and the implicit parallel tasks, as well as a proof of the soundness of this language extension and multiple suggestions to extend the core language.

Although this thesis focuses on Encore, its content is applicable to any programming language that uses futures and wishes to combine their use with parallel combinators.
Chapter 2

Extending Encore with Parallel Combinators

Formalizing the semantics of the parallel combinator extension before adding it to Encore helps not only to prove properties about this new extension, but also to accurately describe how its different components should work and interact once implemented.

The formalization of Encore has already been expressed in the $\mu$Encore calculus [22], yet it is unnecessarily complex for expressing parallel combinators in the language. Although Encore is an object-oriented programming language, a paradigm for which there are already multiple simple formalizations [30, 31], there was no need to use any object-orientedness to express the combinators, due to their functional nature. The semantics of the combinators from the Orc programming language [32, 33] were not expressive enough to use in this thesis, as they do not clearly show where published values come from. Other formalizations for languages with futures already exist [34–36], yet they either have to be changed or can be simplified in order to obtain an elegant calculus that is closer to expressing the traits found in this extension and its implementation in Encore.

Thus, a small calculus was created in order to express the core language of the subset of Encore necessary to include the parallel combinator extension. This chapter introduces this core language, as well as its properties and their proofs.

2.1 Grammar

The following syntax represents a small subset of the Encore programming language, including the new syntax for the parallel combinators:

\[
e ::= \text{expressions} \\
| \quad v \quad \text{values} \\
| e \ e \ \text{function application} \\
| \text{async} \ e \ \text{asynchronous expression} \\
| \text{get} \ e \ \text{get expression}
\]
It is not necessary to include the full syntax of Encore to express the extension, as the expressions already included in Encore that are of particular interest for the parallel combinators are the asynchronous expression, the get expression and future chaining. Asynchronous expressions are those that should be ran in parallel, and return futures as a result. Get expressions block on futures, waiting for their results to be available. Future chains run an expression which uses the value of a future, as soon as it is fulfilled.

Containers are not an integral part of Encore, yet are an important structure that are used in the parallel combinator extension, as they represent an abstraction for data structures in Encore. Examples of containers are lists, sets and multisets.

The parallel combinator extension is composed of the parallel singleton, a data structure that contains parallel expressions, as well as the different functions that operate on these expressions. The elements that form part of this new language extension are all of the expressions from and below the parallel singleton in the expression grammar, and all of the values from and below the empty parallel structure in the value grammar.

### 2.2 Type System

Type systems provide guarantees regarding the prevention of execution errors during a program’s run time. The language used for the subset of Encore with parallel expres-
sions is sufficiently expressed with a first-order type system. While these systems lack
the power of type parametrization and type abstraction, they provide the higher order
functions needed in this language extension, thus significantly reducing the complexity
of type expressions in the parallel combinators.

The set of included types is generated by the following grammar:
\[
\tau ::= K \mid \text{Fut} \tau \mid \text{Ct} \tau \mid \text{Par} \tau \mid \tau \rightarrow \tau
\]

\(K\) represents basic types (such as integers or booleans). \(\text{Fut} \tau\) represents the type of
future variables of type \(\tau\). \(\text{Ct} \tau\) represents the type of containers, which are a structures
that are composed of elements of type \(\tau\). \(\text{Par} \tau\) represents the type of parallel structures
with elements of type \(\tau\). \(\tau \rightarrow \tau\) represents the function type, and is right-associative
(i.e. the expression \(\tau_1 \rightarrow \tau_2 \rightarrow \tau_3\) stands for \(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)\)).

The typing context \(\Gamma\), a sequence of variables and their types, will also include future
variables and their types. If the context is empty (written as \(\emptyset\)), it will be omitted by
writing \(\Gamma \vdash e : \tau\), representing the expression \(e\) with type \(\tau\) under the empty set of
assumptions. The typing context has the following grammar:
\[
\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, f : \text{Fut} \tau
\]

Judgments are assertions about the expressions in the language. The judgments that
are used are \(\Gamma \vdash \cdot\), which states that \(\Gamma\) is a well-formed environment, and \(\Gamma \vdash e : \tau\),
which states that \(e\) is a well-formed term of type \(\tau\) in \(\Gamma\). The validity of these judgments
is defined by the type rules in figure 2.1.

2.3 Operational Semantics

Formal semantics can be used to rigorously specify the meaning of a programming lan-
guage, helping to form the basis to analyze and implement it by leaving no ambiguities
in its interpretation. Operational semantics worry about how computations take effect,
and do so by concisely describing the evaluation steps that take place. This chapter
introduces the Encore language semantics with the parallel combinators extension, as
well the use of the monoid and task abstraction.

2.3.1 Monoids in Encore

Containers and parallel combinators in this thesis are monoids, an abstraction that
significantly simplifies their syntax and representation. The use of monoids in the En-
core syntax helps to further disconnect the language specification from the language
implementation; the implementation should follow the specification, not the other way
around.

A monoid is an algebraic structure \((S, \circ, e_S)\) that has an associative binary operation,
\(\circ\), and an identity element \(e_S\), which satisfies the following properties:

**Closure:** \(\forall a, b \in S : a \circ b \in S\)
Associativity: ∀a, b, c ∈ S : (a ◦ b) ◦ c = a ◦ (b ◦ c)

Identity element: ∃e ∈ S : ∀a ∈ S : e ◦ a = a ◦ e = a

If the order of the operations on the monoid is not important, the monoid is considered a commutative monoid. These monoids also satisfy the commutative axiom:

Commutativity: ∀a, b ∈ S : a ◦ b = b ◦ a

A monoid homomorphism \( \phi \) is a mapping from one monoid \((S, \circ, e_S)\) to another \((T, *, e_T)\) such that for all elements \(a\) and \(b\) in \(S\), \(\phi(a \circ b) = \phi(a) * \phi(b)\) and \(\phi(e_S) = e_T\).

The expression \([e_1] \circ \cdots \circ [e_n]\) represents a (possibly empty) container, as described in section 2.1. The symbol \(\circ\) is the associative binary operation of the monoid, and the identity element is \([\]\).

The expression \({e_1} | \cdots | {e_n}\) represents a (possibly empty) monoidal structure of (possibly repeating) parallel expressions \(e\) and values \(v\). This expression is also a monoidal structure, using \(|\) as the associative binary operation of the monoid, and \(\{\}\) as the identity element.

2.3.2 Configurations

A configuration config is a multiset of futures, tasks and dependent tasks (chains). The union operator on configurations is denoted by a whitespace, and the empty configuration by \(\varepsilon\). The configuration grammar is the following:

\[
\text{config ::= } \varepsilon \mid (\text{fut } f) \mid (\text{fut } f \: v) \mid (\text{task } e \: f) \mid (\text{chain } f_1 \: e \: f_2) \mid \text{config config}
\]

A future variable \(f\) refers to memory location where the return value of an asynchronous execution can be retrieved.

The following objects are used in the configuration:

- \((\text{fut } f)\): the future \(f\) contains no value yet.
- \((\text{fut } f \: v)\): the future \(f\) contains the value \(v\).
- \((\text{task } e \: f)\): the value of the expression \(e\) will be stored in the future \(f\).
- \((\text{chain } f_1 \: e \: f_2)\): the value of the expression \(e\) depends on the future variable \(f_1\) and, once fulfilled, will deposit the result of \(e\) in the future \(f_2\).

Tasks are the main configurations used to produce calculations. In order for the first transitions in the system to occur, there must be a “main” task, which is expressed as \((\text{task } e_0 \: f_{\text{main}})\). The future variable \(f_{\text{main}}\) will contain the final value of the execution of the initial expression \(e_0\). Thus, the initial environment and configuration of the system is \(f_{\text{main}} \vdash (\text{task } e_0 \: f_{\text{main}}) \: (\text{fut } f_{\text{main}})\).

The term complete system configuration refers to the collection of configurations that exists in each execution step starting from the initial configuration. In the complete system configuration, each subconfiguration \((\text{fut } f)\) either has a corresponding \((\text{task } e \: f)\) or a corresponding \((\text{chain } f' \: e \: f)\), and vice-versa.
Lemma 2.3.1 (Complete system configuration reduction rule property). If the configuration $\text{config}$ has the property of being a complete system configuration, this property will be preserved by all the reduction rules.

Proof. By induction on a derivation of $\text{config}$. \qed

Terminal configurations are configurations in their terminal (i.e. final) form. They are composed of future configurations with attached values.

2.3.3 Transition System

This transition system uses reduction semantics [37], a variation of structural operational semantics [38] that represents a separation between an evaluation step and the position in an expression where it has taken place. This is done by decomposing an expression into two components: the redex, which is where the evaluation takes place, and a reduction context $C$, which contains a hole, $\langle \rangle$, where the evaluated redex is placed. Once the expression is decomposed, the redex is reduced into a contractum by using the corresponding contraction rule, which is then “plugged in” the hole of the context. $C[e]$ represents the expression that is obtained by replacing the hole of $C$ with the expression $e$. Thus, if the expression $e$ in $C[e]$ reduces to $e'$, the resulting expression after this process would be $C[e']$. The evaluation and task contexts of the semantics of this language extension are shown in figure 2.4.

Structural congruence equivalences allow us to compare configurations when their order is not important. The congruence equivalences, shown in figure 2.5, were inspired by those of $\pi$-calculus [39, 40].

The equivalence rules for the parallel combinators, shown in figure 2.6, permit the system to optimize the operations that will be performed on parallel structures. PeekM expresses that a peek operation on a parallel structure can be seen as the result of performing a peek on the elements of the parallel structure, and another peek on these “subpeeks”. PeekS states that performing two peek operations on an expression can be simplified into a single peek operation. LiftFutGet shows a resemblance between a lifted future and a singleton parallel expression where a get operation is performed on the future. This last rule is of particular use to the extract combinator.

Some of the parallel combinators can be defined recursively. Those that have this specification are defined in figure 2.7. The sequence combinator applies a function to the elements of a parallel combinator. Although it is equivalent to the map function in other programming languages, this particular syntax has been chosen due to its resemblance to both the sequential combinator in Orc and to the chain operator. The extract combinator transforms a parallel structure into a container. The join combinator flattens a parallel structure.

The full list of the operational semantic rules is shown in figure 2.8. Some of these rules use the fresh $x$ predicate, which is used to assert that the variable $x$ is globally unique. The rules can be divided into three categories: Encore-related, future-handling and parallel combinators.
The Encore-related rules are those that are already found, in one form or another, in the Encore programming language. $\beta$-red represents a $\beta$-reduction, where all of the variables $x$ in the expression $e$ are substituted by the value $v$. The Async rule creates a new future computation, where the expression $e$ is evaluated in a task and a future variable $f$ is immediately returned. The Get rule retrieves the value of a future variable once it has been fulfilled. The Chain rule supplies a future variable and an expression, and immediately returns a new future variable which, once fulfilled, will contain the value of computation of the provided expression using the supplied future variable as a result.

Future-handling rules are not performed in contexts, but on configurations. The ChainVal rule starts the computation of an expression once the future that it is depending on has been fulfilled. The FutVal rule creates a future-with-value configuration once its related task has fully evaluated its expression.

Parallel combinator rules describe the combinators in this new language extension. The SeqLF rule is a special case of the sequence operator, where a chain is applied to a lifted future. The OtherV rule eliminates the second expression of an otherwise combinator once the first expression contains a value. The OtherH rule eliminates the first expression of an otherwise combinator once it has been evaluated to an empty parallel structure. PeekV non-deterministically chooses a value from a parallel structure as the value of a peek operator. PeekH supplies an empty parallel structure if its expression is already one. The Struct rule states that structurally congruent terms have the same reductions.

## 2.4 Parallelism

The previous section defines the reduction rules that may be used in a sequential version of the language. Certain operations, such as async, would have to be defined in the evaluation context in order for the sequential version to be deterministic, yet this would be an undesirable property in this extension, as we wish for it to be as parallel as possible.

In order to extend Encore with parallelism, the rules must be able to operate on multiple configurations and parallel expressions at a time. The parallel reduction rules (figure 2.9) expand the rules defined in section 2.3 with this trait. OneConfig reduces a single configuration, and ShareConfig reduces multiple configurations at the same time while sharing a common configuration. It is important to note that a configuration that does not reduce may also be an empty configuration. The last rule, ManyPar, reduces multiple elements in a parallel expression in the same step.

## 2.5 Type Soundness

Type soundness is a property that guarantees that a certain level of coherence exists in a programming language; that well-typed programs are free from certain types of behaviors during their execution, and thus there is always some transition in the program state (i.e. the program will never be stuck). Type systems can statically check that the language
primitives are being correctly used. This includes, but is not limited to, checking that
the correct amount of arguments are provided to functions, that the types of the inputs
and outputs of functions are correct, that the called functions exist, etc.

The approach used in this thesis to prove soundness was inspired by the method
proposed by Wright and Felleisen [41], where soundness can be divided into two theo-
rems: progress and preservation. This thesis also relies on the canonical forms concept
introduced by Martin-Löf [42]. The progress theorem indicates that a well-typed term
is not stuck, as it is either a value or it can take an evaluation step. The preservation
theorem indicates that if a well-typed term takes an evaluation step, then the resulting
term is also well-typed.

It is useful have a series of lemmas which ease the proof of the theorems. The first
of these lemmas are the inversion lemmas, one for expressions and another for config-
urations. The former shows how a proof of some expression may have been generated.

Lemma 2.5.1 (Inversion of the typing relation of expressions). We distinguish
all the possible cases that can arise, which is one case per type rule:
1. If $\Gamma \vdash e : R$, then $R = \tau$ for some $\tau \in \text{Basic}$.
2. If $\Gamma \vdash f : R$, then $R = \text{Fut} \tau$ for some $\tau$ such that $f : \text{Fut} \tau \in \Gamma$.
3. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.
4. If $\Gamma \vdash \lambda x.e : R$ then $R = \tau \rightarrow R'$ for some $R'$ with $\Gamma, x : \tau \vdash e : R'$.
5. If $\Gamma \vdash e_1 \ e_2 : R$ then there is some $\tau$ such that $\Gamma \vdash e_1 : \tau \rightarrow R$ and $\Gamma \vdash e_2 : \tau$.
6. If $\Gamma \vdash \text{async} e : R$, then there is some $\tau$ such that $R = \text{Fut} \tau$ and $\Gamma \vdash e : \tau$.
7. If $\Gamma \vdash \text{get} e : R$, then there is some $\tau$ such that $R = \tau$ and $\Gamma \vdash e : \text{Fut} \tau$.
8. If $\Gamma \vdash e_1 \ll e_2 : R$, then there is some $\tau$ and $\tau'$ such that $R = \text{Fut} \tau$, $\Gamma \vdash e_1 : \text{Fut} \tau'$
and $\Gamma \vdash e_2 : \tau' \rightarrow \tau$.
9. If $\Gamma \vdash [] : R$, then there is some $\tau$ such that $R = \text{Ct} \tau$.
10. If $\Gamma \vdash [e] : R$, then there is some $\tau$ such that $R = \text{Ct} \tau$ and $\Gamma \vdash e : \tau$.
11. If $\Gamma \vdash e_1 \odot e_2 : R$, then there is some $\tau$ such that $R = \text{Ct} \tau$, $\Gamma \vdash e_1 : R$ and
$\Gamma \vdash e_2 : R$.
12. If $\Gamma \vdash \{\} : R$, then there is some $\tau$ such that $R = \text{Par} \tau$.
13. If $\Gamma \vdash \{e\} : R$, then there is some $\tau$ such that $R = \text{Par} \tau$ and $\Gamma \vdash e : \tau$.
14. If $\Gamma \vdash e_1 \ll e_2 : R$, then there is some $\tau$ such that $R = \text{Par} \tau$, $\Gamma \vdash e_1 : R$ and
$\Gamma \vdash e_2 : R$.
15. If $\Gamma \vdash e_1 \ll e_2 : R$, then there is some $\tau$ and $\tau'$ such that $R = \text{Par} \tau$, $\Gamma \vdash e_1 : \text{Par} \tau'$
and $\Gamma \vdash e_2 : \tau' \rightarrow \tau$.
16. If $\Gamma \vdash e_1 \ll e_2 : R$, then there is some $\tau$ such that $R = \text{Par} \tau$, $\Gamma \vdash e_1 : R$ and
$\Gamma \vdash e_2 : R$.
17. If $\Gamma \vdash \text{liftf} e : R$, then there is some $\tau$ such that $R = \text{Par} \tau$ and $\Gamma \vdash e : \text{Fut} \tau$.
18. If $\Gamma \vdash \text{extract} e : R$, then there is some $\tau$ such that $R = \text{Ct} \tau$ and $\Gamma \vdash e : \text{Par} \tau$.
19. If $\Gamma \vdash \text{peek} e : R$, then there is some $\tau$ such that $R = \text{Par} \tau$ and $\Gamma \vdash e : R$.
20. If $\Gamma \vdash \text{join} e : R$, then there is some $\tau$ such that $R = \text{Par} \tau$ and $\Gamma \vdash e : \text{Par} R$.

Proof. By induction on the derivation of the typing judgment $\Gamma \vdash e : \tau$. □
Lemma 2.5.2 (Inversion of the typing relation of configurations).
1. If $\Gamma \vdash (\text{fut}\ f \ \text{ok})$, then there is some $\tau$ such that $R = \text{Fut}\ \tau$ and $f:R \in \Gamma$.
2. If $\Gamma \vdash (\text{fut}\ f \ v \ \text{ok})$, then there is some $\tau$ such that $R = \text{Fut}\ \tau$, $R' = \tau$ and $f:R \in \Gamma$.
3. If $\Gamma \vdash (\text{task}\ e\ f \ \text{ok})$, then there is some $\tau$ such that $R = \tau$, $R' = \text{Fut}\ \tau$, and $f:R \in \Gamma$.
4. If $\Gamma \vdash (\text{chain}\ f_1\ e\ f_2 \ \text{ok})$, then there is some $\tau$ and $\tau'$ such that $R = \text{Fut}\ \tau$, $R' = \tau \rightarrow \tau'$, and $f_1:R \in \Gamma$ and $f_2:R'' \in \Gamma$.

Proof. By induction on the derivation of the typing judgment $\Gamma \vdash \text{config}\ \text{ok}$. 

A canonical form is an expression that has been fully evaluated to a value [42]. The canonical forms lemma gives us information about the possible shapes of a value.

Lemma 2.5.3 (Canonical forms). If $\vdash v : \tau$, then
1. if $v$ is a value of type $K$, then $v$ is a constant of correct form.
2. if $v$ is a value of type $\tau \rightarrow \tau'$, then $v$ has the form $\lambda x.e$.
3. if $v$ is a value of type $\text{Fut}\ \tau$, then $v$ has the form $f$.
4. if $v$ is a value of type $\text{Ct}\ \tau$, then either $v$ is an empty container structure with the form $\{\}$, a container structure with a value $[v']$, where $v'$ has type $\tau$, or a container structure with the form $v_1 \circ v_2$, where $v_1$ and $v_2$ have type $\text{Ct}\ \tau$.
5. if $v$ is a value of type $\text{Par}\ \tau$, then either $v$ is an empty parallel structure with the form $\{\}$, a parallel structure with a value $\{v'\}$, where $v'$ has type $\tau$, or a parallel structure with the form $v_1 \mid v_2$, where $v_1$ and $v_2$ have type $\text{Par}\ \tau$.

Proof. By induction on a derivation $\vdash v : \tau$.
1. If $\tau$ is $K$, then the only rule which lets us give a value of this type is $\text{Val}\ c$.
2. If $\tau$ is $\tau \rightarrow \tau'$, then the only rule that can give us a value of this type is $\text{Val}\ \text{Fun}$.
3. If $\tau$ is $\text{Fut}\ \tau$, then by inspection of the typing rules, only $\text{Val}\ f$ gives a value of this type.
4. If $\tau$ is $\text{Ct}\ \tau$, then by inspection of the typing rules, there are three possible cases for $v$:
   (a) If $v = \{\}$, then the only rule that gives us a value of this type is $\text{Val}\ \text{EmptyContainer}$.
   (b) If $v = [v']$, then the only rule that gives us a value of this type is $\text{Val}\ \text{Container}$.
   (c) If $v = v_1 \circ v_2$, then the only rule that gives us a value of this type is $\text{Val}\ \text{ContainerAppend}$.
5. If $\tau$ is $\text{Par}\ \tau$, then by inspection of the typing rules, there are three possible cases for $v$:
   (a) If $v = \{\}$, then the only rule that gives us a value of this type is $\text{Val}\ \text{EmptyPar}$.
   (b) If $v = \{v'\}$, then the only rule that gives us a value of this type is $\text{Val}\ \text{Par}$.
   (c) If $v = v_1 \mid v_2$, then the only rule that gives us a value of this type is $\text{Par}\ \text{Par}$.
A future $f$ depends on another future $f'$ if the configuration that leaves the value in $f$, such as $(\text{task } e f)$ or $(\text{chain } f'' e f)$, contains $f'$ in the expression to evaluate, or is waiting for the result of $f'$ to be available. It is also possible that $f$ may depend on a future variable $f'$ if it is in the value expression of some other future variable, $(\text{fut } f'' e)$. These dependencies are expressed as $f \triangleleft f'$, and, using the $\text{depset}$ function (figure 2.10), are defined as one of three possibilities, depending on whether the future $f$ is associated to a task, a chain or a future with a value configuration.

- Associated to a task configuration: $((\text{task } e f)) \frac{f' \in \text{depset}(e)}{f \triangleleft f'}$
- Associated to a chain configuration: $((\text{chain } f'' e f)) \frac{f' = f'' \lor f' \in \text{depset}(e)}{f \triangleleft f'}$
- Associated to a future configuration with a value: $((\text{fut } f e)) \frac{f' \in \text{depset}(e)}{f \triangleleft f'}$

In order for the expression of a task configuration $(\text{task } e f)$ to advance a step in the progress theorem, it may require the result of some future variable $f'$, which may still not have a value. These dependencies form the basis of dependency trees, which are formed by walking through all the possible future dependency paths. The leaves of any dependency tree will have no dependencies, and their associated future variable will either have an attached value, or an associated task configuration with no future dependencies. This configuration will be able to progress a step because it does not depend on the value of any future variables. The configuration $\text{config}_{dep,ex}$ is shown as an example to help visualize a dependency tree:

$\text{config}_{dep,ex} \equiv (\text{fut } f_1 42) (\text{fut } f_2 f_1)$

$(\text{fut } f_3) (\text{task } ((\text{get } f_1) + (\text{get } (\text{get } f_2))) f_3)$

$(\text{fut } f_4) (\text{task } (f_3 \rightsquigarrow \lambda x. x)) f_4)$

This configuration has the dependency tree shown in figure 2.11. It is easy to see that the different dependency paths of this tree are $f_4 \triangleleft f_3 \triangleleft f_2 \triangleleft f_1$ and $f_4 \triangleleft f_3 \triangleleft f_1$. If $f_1$ had been a future without a value, and no future dependencies, its associated task would have been able to take a reduction step using one of the rules in section 2.3.3. This is because $f_1$ would not be expecting the value of any future variables, and would have no reason to not be able to take a reduction step.

As tasks are the configurations that calculate the values which are placed into their associated futures, the only semantic rules that may not progress without an attached value for some future variable is the Get rule. This rule requires special care, as problems in the progress theorem may arise if the rule is not treated correctly. If there are no cycles in the dependency tree, there must be at least one future variable in the tree with an associated configuration that can reduce, using the progress theorem.

**Theorem 2.5.1 (Progress).** If we have a complete system configuration $\text{config}$ such that $\Gamma \vdash \text{config} : \text{ok}$, then either $\text{config}$ is a terminal configuration, or there exists some configuration $\text{config}'$ such that $\text{config} \rightarrow \text{config}'$. 

16
Proof. By induction on a derivation of config.

Case C-Fut: Here (fut f) ∈ config. By the definition of a complete system configuration, there exists some task configuration (task e f) or some chain configuration (chain f' e f) that is related to (fut f). It is thus possible to apply the progress theorem to this related configuration.

Case C-Task: Here (task e f) ∈ config. By the definition of a complete system configuration, there exists some future configuration (fut f) that is related to (task e f). If the expression e is a value, the appropriate C-Config case of this theorem applies to both the task and its related future configurations. If the expression e is not a value, we have different subcases to consider.

Subcase Val c: Here, e ≡ c. Since e is a basic value, the appropriate C-Config case of this theorem applies to both the task and its related future configurations.

Subcase Val f: Here, e ≡ f'. Since e is a future variable, the appropriate C-Config case of this theorem applies to both the task and its related future configurations.

Subcase Val x: Here, e ≡ x. Impossible, as there are no variables in Γ.

Subcase Val Fun: Here, e ≡ \lambda x.e'. Since e is an abstraction, the appropriate C-Config case of this theorem applies to both the task and its related future configurations.

Subcase Val App: Here, e ≡ e_1 e_2, with ⊢ e_1 : \tau' → \tau and ⊢ e_2 : \tau', by inversion. If e_1 is not a value, then by evaluation context, e_1 e_2 → e'_1 e'_2. If e_1 is a value and e_2 is not, then by evaluation context, e_1 e_2 → e_1 e'_2. If both expressions are values, then the canonical forms lemma tells us that e_1 has the form \lambda x.e'.

Thus, the \beta-red rule applies to the task configuration.

Subcase Val Async: Here, e ≡ async e', with ⊢ e' : \tau, by inversion. The Async rule applies to the task configuration.

Subcase Val Get: Here, e ≡ get e', with ⊢ e' : Fut \tau, by inversion. If e' is not a value, then by evaluation context, get e' → get e''. If e' is a value, then the canonical forms lemma tells us that e' has the form f'. Two possible cases can occur:

1. f' has an attached value. That is, there exists some configuration (fut f' v) in config, with some value v. Thus, the appropriate C-Config case of this theorem applies to this future and the task configurations.

2. f' has no attached value. That is, there exists some configuration (fut f') in config which still has not received its value. It is possible to traverse the dependency tree of f' to find some configuration where the progress theorem can be applied to. There are three possible cases that can occur:

   (a) A leaf is reached, and its future has no attached value. As leaves have no future dependencies, the future has to have an associated task configuration (as chain configurations have at least one dependency). If the task’s expression is a value, the appropriate C-Config case of this theorem applies to the task and its related future configurations. If its expression is not a value, the task can take a reduction step by
applying the progress theorem.
(b) A node is reached where its future has no attached value and its related configuration does not need the value of its dependencies to perform a reduction step. As chain configurations require the result of at least one dependency to take a step, the only other possible related configuration is a task configuration. It is thus possible for the task configuration related to this node to take a reduction step by applying the progress theorem to it.
(c) A node is reached where its future, $f''$, has no attached value and its related configuration needs the value of a dependency to perform a reduction step. If the value of the dependency is available, the appropriate $\mathsf{C-Config}$ case of this theorem applies. If not, it is possible to walk down the dependency tree of $f''$ until a node is found where the progress theorem can be applied to its related configuration.

Subcase $\mathsf{Val\ Chain}$: Here, $e \equiv e_1 \rightsquigarrow e_2$, with $\vdash e_1 : \mathsf{Fut} \tau' \text{ and } \vdash e_2 : \tau' \rightarrow \tau$, by inversion. If $e_1$ is not a value, then by evaluation context, $e_1 \rightsquigarrow e_2 \rightarrow e'_1 \rightsquigarrow e_2$. If $e_1$ is a value and $e_2$ is not, then by evaluation context, $e_1 \rightsquigarrow e_2 \rightarrow e_1 \rightsquigarrow e'_2$. If both expressions are values, then the $\mathsf{Chain}$ rule applies to the task configuration.

Subcase $\mathsf{Val\ EmptyContainer}$: Here, $e \equiv \mathbb{I}$. Since $e$ is an empty container structure, the appropriate $\mathsf{C-Config}$ case of this theorem applies to both the task and its related future configurations.

Subcase $\mathsf{Val\ Container}$: Here, $e \equiv [e']$, with $\vdash e' : \tau$, by inversion. If $e'$ is not a value, then by evaluation context, $[e'] \rightarrow [e'\prime]$. If $e'$ is a value, then the canonical forms lemma tells us that $e'$ has the form $\mathbb{v}$, in which case $e$ is a container value singleton, and, thus, the appropriate $\mathsf{C-Config}$ case of this theorem applies to both the task and its related future configurations.

Subcase $\mathsf{Val\ ContainerAppend}$: Here, $e \equiv e_1 \circ e_2$, with $\vdash e_1 : \mathsf{Ct} \tau \text{ and } \vdash e_2 : \mathsf{Ct} \tau$, by inversion. If $e_1$ is not a value, then by evaluation context, $e_1 \circ e_2 \rightarrow e'_1 \circ e_2$. If $e_1$ is a value and $e_2$ is not, then by evaluation context, $e_1 \circ e_2 \rightarrow e_1 \circ e'_2$. If both expressions are values, then the canonical forms lemma tells us that each expression has the form $\mathbb{v}$, $[v_1]$ or $[v_1] \circ [v'_2]$. Therefore, $e$ is a value with the form $v_1 \circ v_2$, and, thus, the appropriate $\mathsf{C-Config}$ case of this theorem applies to both the task and its related future configurations.

Subcase $\mathsf{Val\ EmptyPar}$: Here, $e \equiv \{\}$. Since $e$ is an empty parallel structure, the appropriate $\mathsf{C-Config}$ case of this theorem applies to both the task and its related future configurations.

Subcase $\mathsf{Val\ Par}$: Here, $e \equiv \{e'\}$, with $\vdash e' : \tau$, by inversion. If $e'$ is not a value, then by evaluation context, $\{e'\} \rightarrow \{e'\prime\}$. If $e'$ is a value, then the canonical forms lemma tells us that $e'$ has the form $\mathbb{v}$, in which case $e$ is a parallel value singleton, and, thus, the appropriate $\mathsf{C-Config}$ case of this theorem applies to both the task and its related future configurations.

Subcase $\mathsf{ParT\ Par}$: Here, $e \equiv e_1 \mid e_2$, with $\vdash e_1 : \mathsf{Par} \tau \text{ and } \vdash e_2 : \mathsf{Par} \tau$, by
inversion. If \( e_1 \) is not a value, then by evaluation context, \( e_1 \mid e_2 \rightarrow e'_1 \mid e_2 \).
If \( e_2 \) is not a value, then by evaluation context, \( e_1 \mid e_2 \rightarrow e_1 \mid e'_2 \). If both expressions are values, then the canonical forms lemma tells us that each expression has the form \{\}, \{v_1\} or \{v_1\} \mid \{v'_1\}. Therefore, \( e \) is a value with the form \( v_1 \mid v_2 \), and, thus, the appropriate C-Config case of this theorem applies to both the task and its related future configurations.

**Subcase** PART Seq: Here, \( e \equiv e_1 \gg e_2 \), with \( \vdash e_1 : Par \tau' \) and \( \vdash e_2 : \tau' \rightarrow \tau \), by inversion. If \( e_2 \) is not a value, then by evaluation context, \( e_1 \gg e_2 \rightarrow e_1 \gg e'_2 \). If \( e_2 \) is a value and \( e_1 \) is not, then by evaluation context, \( e_1 \gg e_2 \rightarrow e'_1 \gg e_2 \). If both expressions are values, then the canonical forms lemma tells us that \( e_1 \) has the form \{\}, \{v_1\} or \{v_1\} \mid \{v'_1\}, and \( e_2 \) has the form \( \lambda x. e'_2 \).
For the first form of \( e_1 \), the SeqE rule applies to the task configuration. For the second form of \( e_1 \), the SeqS rule applies to the task configuration. For the third form of \( e_1 \), the SeqM rule applies to the task configuration.

**Subcase** PART Otherwise: Here, \( e \equiv e_1 \times e_2 \), with \( \vdash e_1 : Par \tau \) and \( \vdash e_2 : Par \tau \), by inversion. If \( e_1 \) is not a value, then by evaluation context, \( e_1 \times e_2 \rightarrow e'_1 \times e_2 \). If \( e_1 \) is a value, then the canonical forms lemma tells us that \( e_1 \) is either \{\}, \{v_1\} or \{v_1\} \mid \{v'_1\}. In the first case, the OtherH rule applies to the task configuration, and in the rest of the cases, the OtherV rule applies to the task configuration.

**Subcase** PART LiftFut: Here, \( e \equiv \text{liftf} e' \), with \( \vdash e' : \text{Fut} \tau \), by inversion. If \( e' \) is not a value, then by evaluation context, \( \text{liftf} e' \rightarrow \text{liftf} e'' \). If \( e' \) is a value, then the canonical forms lemma tells us that \( e' \) has the form \( f \), and, thus, it is possible to apply the LiftFutGet rule to the expression.

**Subcase** PART Extract: Here, \( e \equiv \text{extract} e' \), with \( \vdash e' : Par \tau \), by inversion. If \( e' \) is not a value, then by evaluation context, \( \text{extract} e' \rightarrow \text{extract} e'' \). If \( e' \) is a value, then the canonical forms lemma tells us that \( e' \) has the form \{\}, \{v\} or \{v\} \mid \{v'\}. In the first case, the ExtractE rule applies to the task configuration. In the second case, the ExtractS rule applies to the task configuration. In the third case, the ExtractM rule applies to the task configuration.

**Subcase** PART Peek: Here, \( e \equiv \text{peek} e' \), with \( \vdash e' : Par \tau \), by inversion. If \( e' \) is not a value, then by evaluation context, \( \text{peek} e' \rightarrow \text{peek} e'' \). If \( e' \) is a value, then the canonical forms lemma tells us that \( e' \) is either \{\}, \{v\} or \{v\} \mid \{v'\}.
In the first case, the PeekH rule applies to the task configuration, and in the rest of the cases, the PeekV rule applies to the task configuration.

**Subcase** PART Join: Here, \( e \equiv \text{join} e' \), with \( \vdash e' : Par \tau \), by inversion. If \( e' \) is not a value, then by evaluation context, \( \text{join} e' \rightarrow \text{join} e'' \). If \( e' \) is a value, then the canonical forms lemma tells us that \( e' \) has the form \{\}, \{v\} or \{v\} \mid \{v'\}.
In the first case, the JoinE rule applies to the task configuration. In the second case, the Joins rule applies to the task configuration. In the third case, the JoinM rule applies to the task configuration.

**Case** C-Chain: Here \( \text{chain} f_1 e \), \( f_2 \) \in \text{config} \). If the future \( f_1 \) has an attached value, i.e. there exists some configuration \( \text{fut} f_1 v \) \in \text{config} \), then the appropriate C-Config
case of this theorem applies to the chain and its related future configuration. If the future $f_1$ has no attached value, i.e. there exists some configuration $(\text{fut } f_1) \in \text{config}$, then, as with the Val Get subcase the in C-Task case of this proof, it is possible to traverse the dependency tree of $f_1$ to find some configuration where the progress theorem can be applied to. There are three possible cases that can occur:

1. A leaf is reached, and its future has no attached value. As leaves have no future dependencies, the future has to have an associated task configuration (as chain configurations have at least one dependency). If the task’s expression is a value, the appropriate C-Config case of this theorem applies to the task and its related future configurations. If its expression is not a value, the task can take a reduction step by applying the progress theorem.

2. A node is reached where its future has no attached value and its related configuration does not need the value of its dependencies to perform a reduction step. As chain configurations require the result of at least one dependency to take a step, the only other possible related configuration is a task configuration. It is thus possible for the task configuration related to this node to take a reduction step by applying the progress theorem to it.

3. A node is reached where its future, $f''$, has no attached value and its related configuration needs the value of a dependency to perform a reduction step. If the value of the dependency is available, the appropriate C-Config case of this theorem applies. If not, it is possible to walk down the dependency tree of $f''$ until a node is found where the progress theorem can be applied to its related configuration.

Case C-Config: Here $\text{config}_1 \subseteq \text{config}_2 \subseteq \text{config}$. By the induction hypothesis, for $i \in \{1, 2\}$, either $\text{config}_i$ is a terminal configuration, or there exists some configuration $\text{config}_i'$ such that $\text{config}_i \rightarrow \text{config}_i'$. If both $\text{config}_i$ are not terminal configurations, then there are different subcases to consider:

Subcase 1: One of the configurations is $(\text{fut } f)$ and the other is $(\text{task } (\text{get } f) f')$. Thus, the Get rule applies to these configurations.

Subcase 2: One of the configurations is $(\text{fut } f_1 v)$ and the other is $(\text{chain } f_1 e f_2)$. Thus, the ChainVal rule applies to these configurations.

Subcase 3: One of the configurations is $(\text{fut } f)$ and the other is $(\text{task } v f)$. Thus, the FutVal rule applies to these configurations.

Lemma 2.5.4 (Permutation). If $\Gamma \vdash e : \tau$ and $\Delta$ is a permutation of $\Gamma$, then $\Delta \vdash e : \tau$. Permutation is caused by interchanging any two immediate variables in $\Gamma$ a certain amount of times. This can also be done between variables and future variables, but not between future variables.

Proof. By induction on typing derivations.

Lemma 2.5.5 (Weakening). If $\Gamma \vdash e : \tau$ and $x \notin \text{dom}(\Gamma)$, then $\Gamma, x:\tau' \vdash e : \tau$

Proof. By induction on typing derivations.
Lemma 2.5.6 (Preservation of types under substitution). If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[e'/x] : \tau$

Proof. By induction on the depth of a derivation of the statement $\Gamma, x : \tau' \vdash e : \tau$.

Case Val c: Here, $e = c$, so $e[e'/x] = c$. The desired result is immediate.

Case Val f: Here, $e = f$, so $f[e'/x] = f$. The desired result is immediate.

Case Val x: Here, $e = x$, with $x : \tau \in (\Gamma, x : \tau')$. There are two subcases to consider, depending on whether $z$ is $x$ or another variable.

Subcase 1: If $z = x$, then $z[e'/x] = e'$. The required result is then $\Gamma \vdash e' : \tau'$, which is among the assumptions of the lemma.

Subcase 2: If $z \neq x$, then $z[e'/x] = z$, and the desired result is immediate.

Case Val Fun: Here $e = \lambda y.e''$ with $\tau = \tau_2 \rightarrow \tau_1$ and $\Gamma, x : \tau', y : \tau_2 \vdash e'' : \tau_1$, by inversion. We assume $x \neq y$ and $y \notin FV(e')$. Using permutation on the given subderivation, we obtain $\Gamma, y : \tau_2, x : \tau' \vdash e'' : \tau_1$. Using weakening on the other given subderivation ($\Gamma \vdash e' : \tau'$), we obtain $\Gamma, y : \tau_2 \vdash e' : \tau'$. Now, by the induction hypothesis, $\Gamma, y : \tau_2 \vdash e''[e'/x] : \tau_1$. By Val Fun, $\Gamma \vdash \lambda y. e''[e'/x] : \tau_2 \rightarrow \tau_1$. But this is precisely the needed result, since, by the definition of substitution, $e[e'/x] = \lambda y. e''[e'/x]$.

Case Val App: Here $e = e_1 \cdot e_2$, with $\Gamma, x : \tau' \vdash e_1 : \tau'' \rightarrow \tau$ and $\Gamma, x : \tau' \vdash e_2 : \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e_1[e'/x] : \tau'' \rightarrow \tau$ and $\Gamma \vdash e_2[e'/x] : \tau''$. By Val App, $\Gamma \vdash e_1[e'/x] e_2[e'/x] : \tau$, i.e., $\Gamma \vdash (e_1 \cdot e_2)[e'/x] : \tau$.

Case Val Async: Here $e = \text{async } e''$, with $\Gamma, x : \tau' \vdash e'' : \tau''$ and $\tau = \text{Fut } \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e''[e'/x] : \tau''$. By Val Async, $\Gamma \vdash \text{async } e''[e'/x] : \tau = \Gamma \vdash (\text{async } e'')[e'/x] : \tau$.

Case Val Get: Here $e = \text{get } e''$, with $\Gamma, x : \tau' \vdash e'' : \text{Fut } \tau$, by inversion. By the induction hypothesis, $\Gamma \vdash e''[e'/x] : \text{Fut } \tau$. By Val Get, $\Gamma \vdash \text{get } e''[e'/x] : \tau = \Gamma \vdash (\text{get } e'')[e'/x] : \tau$.

Case Val CHAIN: Here $e = e_1 \rightarrow e_2$, with $\Gamma, x : \tau' \vdash e_1 : \text{Fut } \tau_1$, $\Gamma, x : \tau' \vdash e_2 : \tau_1 \rightarrow \tau_2$ and $\tau = \text{Fut } \tau_2$. By the induction hypothesis, $\Gamma \vdash e_1[e'/x] : \text{Fut } \tau_1$ and $\Gamma \vdash e_2[e'/x] : \tau_1 \rightarrow \tau_2$. By Val CHAIN, $\Gamma \vdash e_1[e'/x] \rightarrow e_2[e'/x] : \tau = \Gamma \vdash (e_1 \rightarrow e_2)[e'/x] : \tau$.

Case Val EmptyContainer: Here $e = \emptyset$, so $\emptyset[e'/x] = \emptyset$. The desired result is immediate.

Case Val Container: Here $e = [e'']$, with $\Gamma, x : \tau' \vdash e'' : \tau''$, and $\tau = \text{Ct } \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e''[e'/x] : \tau''$. By Val Container, $\Gamma \vdash [e''[e'/x]] : \tau = \Gamma \vdash ([e''])[e'/x] : \tau$.

Case Val ContainerAppend: Here $e = e_1 \circ e_2$, with $\Gamma, x : \tau' \vdash e_1 : \text{Ct } \tau''$, $\Gamma, x : \tau' \vdash e_2 : \text{Ct } \tau''$ and $\tau = \text{Ct } \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e_1[e'/x] : \text{Ct } \tau''$ and $\Gamma \vdash e_2[e'/x] : \text{Ct } \tau''$. By Val ContainerAppend, $\Gamma \vdash e_1[e'/x] \circ e_2[e'/x] : \tau = \Gamma \vdash (e_1 \circ e_2)[e'/x] : \tau$.

Case Val EmptyPar: Here $e = \{\}$, so $\emptyset[e'/x] = \emptyset$. The desired result is immediate.

Case Val Par: Here $e = \{e'\}$, with $\Gamma, x : \tau' \vdash e' : \tau''$, and $\tau = \text{Par } \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e''[e'/x] : \tau''$. By Val Par, $\Gamma \vdash \{e''[e'/x]\} : \tau = \Gamma \vdash \{e''\}[e'/x] : \tau$.

Case Part Par: Here $e = e_1 \mid e_2$, with $\Gamma, x : \tau' \vdash e_1 : \text{Par } \tau''$, $\Gamma, x : \tau' \vdash e_2 : \text{Par } \tau''$ and $\tau = \text{Par } \tau''$, by inversion. By the induction hypothesis, $\Gamma \vdash e_1[e'/x] : \text{Par } \tau''$ and
Lemma 2.5.7 (Subexpression well-typedness). If $\Gamma \vdash e : \tau$, and $e'$ is a subexpression of $e$, then there exists a $\Gamma'$ such that $\Gamma' \supseteq \Gamma$, and some type $\tau'$, such that $\Gamma' \vdash e' : \tau'$.

Proof. By induction on a derivation of $\Gamma \vdash e : \tau$.

1. If $e \equiv (\text{fut} \ e'')$, then there are two possibilities: $e' = e$, or $e'$ is a subexpression of $e''$. In the first case, we are done by assumption. In the second case, with $\Gamma \vdash (\text{fut} \ e'') \text{ ok}$ and $\Gamma \vdash e'' : \tau'$ by inversion, the result follows by the induction hypothesis.

2. If $e \equiv (\text{task} \ e'' \ f)$, then there are two possibilities: $e' = e$, or $e'$ is a subexpression of $e''$. In the first case, we are done by assumption. In the second case, with $\Gamma \vdash (\text{task} \ e'' \ f) \text{ ok}$ and $\Gamma \vdash e'' : \tau'$ by inversion, the result follows by the induction hypothesis.

3. If $e \equiv c$, then the only possible case is where $e' = e$. Thus, we are done by assumption.

4. If $e \equiv f$, then the only possible case is where $e' = e$. Thus, we are done by assumption.

5. If $e \equiv x$, then the only possible case is where $e' = e$. Thus, we are done by assumption.

6. If $e \equiv \lambda x.e''$, then there are two possibilities: $e' = e$, or $e'$ is a subexpression of $e''$. In the first case, we are done by assumption. In the second case, with $\Gamma \vdash \lambda x.e'' : \tau'' \rightarrow \tau'$ and $\Gamma, x: \tau'' \vdash e' : \tau'$ by inversion, the result follows by the induction hypothesis.
7. If \( e \equiv e_1 \cdot e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : \tau'' \rightarrow \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : \tau'' \) by inversion, the result follows by the induction hypothesis.

8. If \( e \equiv \text{async} \; e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{async} \; e'' : \text{Fut} \; \tau' \) and \( \Gamma \vdash e'' : \tau' \) by inversion, the result follows by the induction hypothesis.

9. If \( e \equiv \text{get} \; e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{get} \; e'' : \tau' \) and \( \Gamma \vdash e'' : \text{Fut} \; \tau' \) by inversion, the result follows by the induction hypothesis.

10. If \( e \equiv e_1 \triangleright e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : \text{Fut} \; \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : \tau' \rightarrow \tau'' \) by inversion, the result follows by the induction hypothesis.

11. If \( e \equiv \emptyset \), then the only possible case is where \( e' = e \). Thus, we are done by assumption.

12. If \( e \equiv \{e''\} \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \{e''\} : \text{Ct} \; \tau' \) and \( \Gamma \vdash e'' : \tau' \) by inversion, the result follows by the induction hypothesis.

13. If \( e \equiv e_1 \circ e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : \text{Ct} \; \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : \text{Ct} \; \tau' \) by inversion, the result follows by the induction hypothesis.

14. If \( e \equiv \{\} \), then the only possible case is where \( e' = e \). Thus, we are done by assumption.

15. If \( e \equiv \{e''\} \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \{e''\} : \text{Par} \; \tau' \) and \( \Gamma \vdash e'' : \tau' \) by inversion, the result follows by the induction hypothesis.

16. If \( e \equiv e_1 \mid e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : \text{Par} \; \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : \text{Par} \; \tau' \) by inversion, the result follows by the induction hypothesis.

17. If \( e \equiv e_1 \gg e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : \text{Par} \; \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : \tau' \rightarrow \tau'' \) by inversion, the result follows
By the induction hypothesis.

18. If \( e \equiv e_1 \bowtie e_2 \), then there are three possibilities: \( e' = e \), \( e' \) is a subexpression of \( e_1 \), or \( e' \) is a subexpression of \( e_2 \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash e_1 : Par \tau' \) by inversion, the result follows by the induction hypothesis. In the third case, with \( \Gamma \vdash e_2 : Par \tau' \) by inversion, the result follows by the induction hypothesis.

19. If \( e \equiv \text{liftf} e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{liftf} e'' : Par \tau' \) and \( \Gamma \vdash e'' : Fut \tau' \) by inversion, the result follows by the induction hypothesis.

20. If \( e \equiv \text{extract} e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{extract} e'' : Ct \tau' \) and \( \Gamma \vdash e'' : Par \tau' \) by inversion, the result follows by the induction hypothesis.

21. If \( e \equiv \text{peek} e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{peek} e'' : Par \tau' \) by inversion, the result follows by the induction hypothesis.

22. If \( e \equiv \text{join} e'' \), then there are two possibilities: \( e' = e \), or \( e' \) is a subexpression of \( e'' \). In the first case, we are done by assumption. In the second case, with \( \Gamma \vdash \text{join} e'' : Par \tau' \) and \( \Gamma \vdash e'' : Par (Par \tau') \) by inversion, the result follows by the induction hypothesis.

\[ \square \]

Lemma 2.5.8 (Expression substitution). If \( \Gamma \vdash e : \tau' \) is a subderivation of \( \Gamma \vdash T[e] \text{ ok} \), and the expression \( e' \) satisfies \( \Gamma' \vdash e' : \tau' \), then \( \Gamma \vdash T[e'] \text{ ok} \). Similarly, if \( \Gamma' \vdash e : \tau' \) is a subderivation of \( \Gamma \vdash C[e] : \tau \), and the expression \( e' \) satisfies \( \Gamma' \vdash e' : \tau' \), then \( \Gamma \vdash C[e'] : \tau \).

Proof. By induction on a derivation of \( \Gamma \vdash T[e] \text{ ok} \) or \( \Gamma \vdash C[e] : \tau \).

1. If \( T \equiv \text{(task } C[\_] \text{ f)} \), then \( \Gamma \vdash C[e] : \tau \) for some type \( \tau \), by inversion. By the induction hypothesis we have \( \Gamma \vdash C[e'] : \tau \), for some type \( \tau \). By C-Task we have \( \Gamma \vdash \text{(task } C[e'] \text{ f)} \), which is the required result.

2. If \( C = [\_] \), since \([e] = e\), then \( \tau' = \tau \). Thus, as \( \Gamma \vdash e' : \tau \), then \( \Gamma \vdash [e'] : \tau \), giving us the required result.

3. If \( C = C'[\_] e'' \), then \( \Gamma \vdash C'[e] e'' : \tau \) with \( \Gamma \vdash C'[e] : \tau' \rightarrow \tau \) by inversion. By the induction hypothesis we have \( \Gamma \vdash C'[e'] : \tau' \rightarrow \tau \), and by Val App we have \( \Gamma \vdash C[e'] e'' : \tau \), as required.

4. If \( C = e'' C'[\_] \), then \( \Gamma \vdash e'' C'[e] : \tau \) with \( \Gamma \vdash C'[e] : \tau' \) by inversion. By the induction hypothesis we have \( \Gamma \vdash C'[e'] : \tau' \), and by Val App we have \( \Gamma \vdash e'' C'[e'] : \tau \), as required.

5. If \( C = \text{get } C'[\_] \), then \( \Gamma \vdash \text{get } C'[e] : \tau \) with \( \Gamma \vdash C'[e] : Fut \tau \) by inversion. By the induction hypothesis we have \( \Gamma \vdash C'[e'] : Fut \tau \), and by Val Get we have \( \Gamma \vdash \text{get } C'[e'] : \tau \), as required.

6. If \( C = C'[\_] \rightsquigarrow e'' \), then \( \Gamma \vdash C'[e] \rightsquigarrow e'' : \tau \) with \( \Gamma \vdash C'[e] : Fut \tau' \) by inversion.
By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Fut } \tau'$, and by $\text{Val Chain}$ we have $\Gamma \vdash C'[e'] \rightsquigarrow e'' : \tau$, as required.

7. If $C = e'' \rightsquigarrow C'[\cdot]$, then $\Gamma \vdash e'' \rightsquigarrow C'[e] : \text{Fut } \tau$ with $\Gamma \vdash C'[e] : \tau' \rightarrow \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \tau' \rightarrow \tau$, and by $\text{Val Chain}$ we have $\Gamma \vdash e'' \rightsquigarrow C'[e'] : \text{Fut } \tau$, as required.

8. If $C = [C'][\cdot]$, then $\Gamma \vdash [C'][e] : \text{Ctl } \tau$ with $\Gamma \vdash C'[e] : \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \tau$, and by $\text{Val Container}$ we have $\Gamma \vdash [C'[e']] : \text{Ctl } \tau$, as required.

9. If $C = C'[\cdot] \circ e''$, then $\Gamma \vdash C'[e] \circ e'' : \text{Ctl } \tau$ with $\Gamma \vdash C'[e] : \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \tau$, and by $\text{Val ContainerAppend}$ we have $\Gamma \vdash C'[e'] \circ e'' : \text{Ctl } \tau$, as required.

10. If $C = e'' \circ C'[\cdot]$, then $\Gamma \vdash e'' \circ C'[e] : \tau$ with $\Gamma \vdash C'[e] : \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \tau$, and by $\text{Val Container}$ we have $\Gamma \vdash e'' \circ C'[e'] : \text{Ctl } \tau$, as required.

11. If $C = \{C'[\cdot] \}, \text{then } \Gamma \vdash \{C'[e]\} : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \tau$, and by $\text{Par Par}$ we have $\Gamma \vdash \{C'[e']\} : \text{Par } \tau$, as required.

12. If $C = C'[\cdot] \mid e''$, then $\Gamma \vdash C'[e] \mid e'' : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] \mid e'' : \text{Par } \tau$, and by $\text{ParT Par}$ we have $\Gamma \vdash C'[e'] \mid e'' : \text{Par } \tau$, as required.

13. If $C = e'' \mid C'[\cdot]$, then $\Gamma \vdash e'' \mid C'[e] : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash e'' \mid C'[e'] : \text{Par } \tau$, and by $\text{ParT Par}$ we have $\Gamma \vdash e'' \mid C'[e'] : \text{Par } \tau$, as required.

14. If $C = e'' \triangleright C'[\cdot]$, then $\Gamma \vdash e'' \triangleright C'[e] : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \tau'' \rightarrow \tau'$ by inversion. By the induction hypothesis we have $\Gamma \vdash e'' \triangleright C'[e'] : \text{Par } \tau$, and by $\text{ParT Seq}$ we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, as required.

15. If $C = C'[\cdot] \triangleright e''$, then $\Gamma \vdash C'[e] \triangleright e'' : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Fut } \tau'$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] \triangleright e'' : \text{Par } \tau$, and by $\text{ParT Seq}$ we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, as required.

16. If $C = C'[\cdot] \times e''$, then $\Gamma \vdash C'[e] \times e'' : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, and by $\text{Val Otherwise}$ we have $\Gamma \vdash C'[e'] \times e'' : \text{Par } \tau$, as required.

17. If $C = \text{lift } C'[\cdot]$, then $\Gamma \vdash \text{lift } C'[e] : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Fut } \tau'$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Fut } \tau$, and by $\text{ParT LiftFut}$ we have $\Gamma \vdash \text{lift } C'[e'] : \text{Par } \tau$, as required.

18. If $C = \text{extract } C'[\cdot]$, then $\Gamma \vdash \text{extract } C'[e] : \text{Ctl } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, and by $\text{ParT Extract}$ we have $\Gamma \vdash \text{extract } C'[e'] : \text{Ctl } \tau$, as required.

19. If $C = \text{peek } C'[\cdot]$, then $\Gamma \vdash \text{peek } C'[e] : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, and by $\text{ParT Peek}$ we have $\Gamma \vdash \text{peek } C'[e'] : \text{Par } \tau$, as required.

20. If $C = \text{join } C'[\cdot]$, then $\Gamma \vdash \text{join } C'[e] : \text{Par } \tau$ with $\Gamma \vdash C'[e] : \text{Par } \text{Par } \tau$ by inversion. By the induction hypothesis we have $\Gamma \vdash C'[e'] : \text{Par } \tau$, and by
Theorem 2.5.2 (Preservation). If \( \Gamma \vdash \text{config ok} \) and \( \text{config} \rightarrow \text{config}' \), then there exists a \( \Gamma' \) such that \( \Gamma' \supseteq \Gamma \) and \( \Gamma' \vdash \text{config'} ok \).

Proof. By induction on a derivation of \( \text{config} \rightarrow \text{config}' \).

Case OneConfig: Here, \( \text{config} \equiv \text{config}_1 \text{config}_2 \) and \( \text{config}' \equiv \text{config}_1 \text{config}_2 \).

There are multiple subcases to be considered:

Subcase \( \beta\text{-red} \): Here, \( \text{config}_1 \equiv T[\lambda \alpha.e] v \), \( \text{config}_1' \equiv T[e[v/x]] \), and \( \Gamma' = \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash \lambda \alpha.e : \tau' \rightarrow \tau \) and \( \Gamma \vdash v : \tau' \), by inversion. There is only one typing rule for abstractions, \( \text{Val Fun} \), from which we know \( \Gamma, x : \tau' \vdash e : \tau \). By the substitution lemma, we have \( \Gamma \vdash e[v/x] : \tau \). Using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}_1 \text{ok} \), as required.

Subcase \( \text{Async} \): Here, \( \text{config}_1 \equiv T[\text{async} e] \), \( \text{config}_1' \equiv (\text{fut} f) (\text{task} e f) T[f] \) and \( \Gamma' = \Gamma, f : \text{Fut} \tau \).

By Lemma 2.5.7, we have \( \Gamma \vdash \text{async} e : \tau \), for some \( \tau \), and by inversion \( \tau = \text{Fut} \tau' \) for some \( \tau' \). By C-Fut, \( \Gamma' \vdash (\text{fut} f) \text{ok} \), and by C-Task, \( \Gamma' \vdash (\text{task} e f) \text{ok} \). As futset((fut f)) = f and futset((task e f)) = \varnothing, by using C-Config, we can conclude that \( \Gamma' \vdash \text{config}_1 \text{ok} \).

Subcase \( \text{Get} \): Here, \( \text{config}_1 \equiv (\text{fut} f v) T[\text{get} f] \), with \( \text{fut} \text{Fut} \tau \in \Gamma \) and \( \Gamma \vdash v : \tau \), by inversion, and \( \text{config}_1' \equiv (\text{fut} f v) T[v] \), with \( \Gamma' = \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash \text{get} f : \tau \), for some \( \tau \). As this type is equal to \( v \), by using Lemma 2.5.8, we have that \( \Gamma \vdash T[v] \text{ok} \). By C-FutV, we have \( (\text{fut} f v) \text{ok} \).

As futset((fut f v)) = f and futset(T[v]) = \varnothing, by using C-Config, we can conclude that \( \Gamma \vdash \text{config}_1 \text{ok} \).

Subcase \( \text{Chain} \): Here, \( \text{config}_1 \equiv T[f_1] (\text{chain} f_1 \ (\lambda \alpha.e \ f_2) T[f_2] \), and \( \Gamma' = \Gamma, f_2 : \text{Fut} \tau \).

By Lemma 2.5.7 and inversion, we have \( \Gamma \vdash f_1 \sim \lambda \alpha.e : \text{Fut} \tau \) and \( \Gamma \vdash \lambda \alpha.e : \tau' \rightarrow \tau \). As this first type is equal to \( f_2 \), by using Lemma 2.5.8, we have that \( \Gamma \vdash T[f_2] \text{ok} \). By C-Fut, \( \Gamma \vdash (\text{fut} f_2) \text{ok} \), and by C-Chain \( \Gamma' \vdash (\text{chain} f_1 \ (\lambda \alpha.e \ f_2) \text{ok} \).

As futset((fut f_2)) = f_2, futset((chain f_1 e f_2)) = \varnothing, and futset(T[v]) = \varnothing, by using C-Config, we can conclude that \( \Gamma \vdash \text{config}_1 \text{ok} \).

Subcase \( \text{ChainVal} \): Here, \( \text{config}_1 \equiv (\text{fut} f v) (\text{chain} f_1 \ e f_2) \), with \( f_1 : \text{Fut} \tau \in \Gamma \), \( f_2 : \text{Fut} \tau \in \Gamma \), \( \Gamma \vdash e : \tau' \rightarrow \tau \), and \( \Gamma \vdash v : \tau' \), by inversion. Also, \( \text{config}_1 \equiv (\text{task} (e v) f_2) \), with \( \Gamma' = \Gamma \).

By Val App, \( \Gamma \vdash (e v) : \tau \). Thus, by C-Task, we can conclude that \( \Gamma \vdash \text{config}_1 \text{ok} \).

Subcase \( \text{FutVal} \): Here, \( \text{config}_1 \equiv (\text{fut} f) (\text{task} v f) \), with \( f : \text{Fut} \tau \in \Gamma \) and \( \Gamma \vdash v : \tau \), by inversion, and \( \text{config}_1' \equiv (\text{fut} f v) \), with \( \Gamma' = \Gamma \). By C-FutV, \( \Gamma \vdash \text{config}_1 \text{ok} \), as required.

Subcase \( \text{SeqLF} \): Here, \( \text{config}_1 \equiv T[\text{liftf} f] \gg e] \) and \( \text{config}_1' \equiv \text{liftf} (f \sim e) \), with \( \Gamma' = \Gamma \). By Lemma 2.5.7, we have \( \Gamma \vdash (\text{liftf} f) \gg e : \text{Par} \tau \), for some \( \tau \), with \( \Gamma \vdash f : \text{Fut} \tau' \) and \( \Gamma \vdash e : \tau' \rightarrow \tau \), by inversion. By Val Chain, we have that \( \Gamma \vdash f \sim e : \text{Fut} \tau \), which with \text{ParT LiftFut} gives us...
\( \Gamma \vdash \text{lift} (f \rightsquigarrow e) : \text{Par } \tau \). Using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}' \text{ ok} \), as required.

**Subcase OtherV:** Here, \( \text{config}_1 \equiv T[e_1 \times e_2] \) and \( \text{config}'_1 \equiv T[e_1] \), with \( \Gamma' \equiv \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash e_1 \times e_2 : \text{Par } \tau \), for some \( \tau \), with \( \Gamma \vdash e_1 : \text{Par } \tau \), by inversion. Thus, by using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}'_1 \text{ ok} \), as required.

**Subcase OtherH:** Here, \( \text{config}_1 \equiv T[e_1 \times e_2] \) and \( \text{config}'_1 \equiv T[e_2] \), with \( \Gamma' \equiv \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash e_1 \times e_2 : \text{Par } \tau \), for some \( \tau \), with \( \Gamma \vdash e_2 : \text{Par } \tau \), by inversion. Thus, by using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}'_1 \text{ ok} \), as required.

**Subcase PeekV:** Here, \( \text{config}_1 \equiv T[\text{peek} (e_1 | \{v\} | e_2)] \) and \( \text{config}'_1 \equiv T[\{v\}] \), with \( \Gamma' \equiv \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash \text{peek} (e_1 | \{v\} | e_2) : \text{Par } \tau \), for some \( \tau \), with \( \Gamma \vdash \{v\} : \text{Par } \tau \), by inversion. By using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}'_1 \text{ ok} \), as required.

**Subcase PeekH:** Here, \( \text{config}_1 \equiv T[\text{peek } \{\} ] \) and \( \text{config}'_1 \equiv T[\{\} ] \), with \( \Gamma' \equiv \Gamma \).

By Lemma 2.5.7, we have \( \Gamma \vdash \text{peek } \{\} : \text{Par } \tau \), for some \( \tau \), with \( \Gamma \vdash \{\} : \text{Par } \tau \), by inversion. By using Lemma 2.5.8, we have that \( \Gamma \vdash \text{config}'_1 \text{ ok} \), as required.

**Subcase ManyPar:** Here, \( T[e_1 | e_2] \in \text{config}_1 \) and \( T[e'_1 | e'_2] \in \text{config}'_1 \). By the induction hypothesis on \( \text{config}_1 \), there exists some \( \Gamma' \) such that \( \Gamma' \supseteq \Gamma \) and \( \Gamma' \vdash \text{config}'_1 \text{ ok} \). The only semantic rules that introduce new elements into \( \Gamma \) are \text{Async} and \text{Chain}. In both of these cases, the new variables are fresh. As the new variables are created in the same reduction step (which could be seen as “at the same time”), the order of their placement at the end of \( \Gamma \) should not matter. Thus, the variables created by the reduction step of \( T[e_1] \), represented as \( \Gamma_1 \), are arbitrarily placed before those created by the reduction step of \( T[e_2] \), represented as \( \Gamma_2 \), and \( \Gamma' \equiv \Gamma, \Gamma_1, \Gamma_2 \).

**Case ShareConfig:** Here, \( \text{config} \equiv \text{config}_1 \text{ config}_2 \text{ config}_3 \) and \( \text{config}' \equiv \text{config}_1 \text{ config}_2 \text{ config}_3 \). By the induction hypothesis, for \( i \in \{2, 3\} \), there exists some \( \Gamma_i \) such that \( \Gamma_i \supseteq \Gamma \) and \( \Gamma_i \vdash \text{config}_i \text{ ok} \). \( \text{config}_i \equiv \text{config}_1 \text{ config}_2 \text{ config}_3 \) is not considered, as it does not take a step in \text{ShareConfig}. The only semantic rules that introduce new elements into \( \Gamma \) are \text{Async} and \text{Chain}. In both of these cases, the new variables are fresh. It is possible to view \( \Gamma_i \) as \( \Gamma_i = \Gamma, \Gamma_i' \), where \( \Gamma_i' \) are the new variables concatenated to \( \Gamma \). As the new variables are created in the same reduction step (which could be seen as “at the same time”), the order of their placement at the end of \( \Gamma \) should not matter. Thus, the variables created by \( \text{config}_2 \) are arbitrarily placed before those of \( \text{config}_3 \), and \( \Gamma' = \Gamma, \Gamma_2', \Gamma_3' \).

### 2.6 Deadlock-freedom

Deadlocks occur when it is not possible to take any reduction steps because multiple expressions are waiting for each other to finish. They appear in this system when there is a circular dependency among future variables. Simple examples of this could be the following correctly defined configuration:
Configuration 2.1 represents a simple deadlock between two tasks, and configuration 2.2 is a self-deadlocked task.

In order to avoid this, an explicit ordering on future variables is introduced into the context $\Gamma$. This order is expressed as $f_1 \prec f_2 \prec \cdots \prec f_n$, where $f_i \prec f_{i+1}$ symbolizes that the future $f_i$ was created before $f_{i+1}$. The binary relation $\prec$ is transitive, that is, if $f_1 \prec f_2$ and $f_2 \prec f_3$, then $f_1 \prec f_3$, but not reflexive, so the relation $f_i \prec f_i$ cannot occur.

The task that leaves a value in $f_i$ may depend on any number of future variables in the set $\{f_1, \ldots, f_{i-1}\}$. No variable of this set can depend on $f_i$, as doing so may lead to a deadlock.

The order of the future variables is the order of their creation, as expressed in the progress theorem proof. An example of this future variable order is $\Gamma \equiv f:\text{Fut}, f':\text{Fut}',$ which indicates that the ordering is $f \prec f'$. That is, $f$ was created before, or at the same time as, $f'$.

With parallelism, multiple future variables might be created at the same time. If this were to happen, the order that these futures are introduced into the context $\Gamma$ does not matter. This is because $\Gamma$ is the same to all configurations in a reduction step, so if any future variable depends on another for its result, it must already exist in $\Gamma$.

Defining a deadlock-freedom invariant for the type soundness proofs would ensure that as long as this invariant is true, there would be no deadlocks during execution. An invariant that expresses the desired property is if $f \prec f'$, then $f' \in \Gamma$ and $f' \prec f$.

Proof. By induction on a derivation of $\text{config} \longrightarrow \text{config}'$. \hfill $\square$
\begin{align*}
(\text{Env } \emptyset) & \quad \emptyset \vdash \emptyset \\
(\text{Env } x) & \quad x \notin \text{dom}(\Gamma) \\
& \quad \Gamma, x : \tau \vdash \emptyset \\
(\text{Val } c) & \quad \Gamma \vdash c : \tau \\
(\text{Val } f) & \quad f : \text{Fut } \tau \in \Gamma \\
& \quad \Gamma \vdash f : \text{Fut } \tau \\
(\text{Val } x) & \quad x : \tau \in \Gamma \\
& \quad \Gamma \vdash x : \tau \\
(\text{Val Fun}) & \quad \Gamma, x : \tau \vdash \lambda x.e : \tau \\
(\text{Val App}) & \quad \Gamma \vdash e_1 : \tau \rightarrow \tau' \\
& \quad \Gamma \vdash e_2 : \tau' \\
& \quad \Gamma \vdash e_1 \, e_2 : \tau \\
(\text{Val Get}) & \quad \Gamma \vdash \text{get } e : \tau' \\
(\text{Val Chain}) & \quad \Gamma \vdash e_1 : \text{Fut } \tau' \\
& \quad \Gamma \vdash e_2 : \tau' \rightarrow \tau \\
& \quad \Gamma \vdash e_1 \Rightarrow e_2 : \text{Fut } \tau \\
(\text{Val EmptyContainer}) & \quad \Gamma \vdash [] : \text{Ct } \tau \\
(\text{Val Container}) & \quad \Gamma \vdash [e] : \text{Ct } \tau \\
(\text{Val ContainerAppend}) & \quad \Gamma \vdash e_1 : \text{Ct } \tau \\
& \quad \Gamma \vdash e_2 : \text{Ct } \tau \\
& \quad \Gamma \vdash e_1 \circ e_2 : \text{Ct } \tau \\
(\text{Val EmptyPar}) & \quad \Gamma \vdash \{\} : \text{Par } \tau \\
(\text{Val Par}) & \quad \Gamma \vdash \{e\} : \text{Par } \tau \\
(\text{Par Par}) & \quad \Gamma \vdash e_1 : \text{Par } \tau \\
& \quad \Gamma \vdash e_2 : \text{Par } \tau \\
& \quad \Gamma \vdash e_1 \mid e_2 : \text{Par } \tau \\
(\text{Par Seq}) & \quad \Gamma \vdash e_1 : \text{Par } \tau' \\
& \quad \Gamma \vdash e_2 : \tau' \rightarrow \tau \\
& \quad \Gamma \vdash e_1 \bowtie e_2 : \text{Par } \tau \\
(\text{Par Otherwise}) & \quad \Gamma \vdash e_1 : \text{Par } \tau \\
& \quad \Gamma \vdash e_2 : \text{Par } \tau \\
& \quad \Gamma \vdash e_1 \times e_2 : \text{Par } \tau \\
(\text{Par LiftFut}) & \quad \Gamma \vdash \text{lift } e : \text{Par } \tau \\
(\text{Par Extract}) & \quad \Gamma \vdash e : \text{Fut } \tau \\
& \quad \Gamma \vdash \text{extract } e : \text{Ct } \tau \\
(\text{Par Peek}) & \quad \Gamma \vdash \text{peek } e : \text{Par } \tau \\
(\text{Par Join}) & \quad \Gamma \vdash e : \text{Par } (\text{Par } \tau) \\
& \quad \Gamma \vdash \text{join } e : \text{Par } \tau
\end{align*}

\textbf{Figure 2.1: Type rules}
\[ \begin{align*}
&(C\text{-Fut}) & f \in \text{dom}(\Gamma) & \quad (C\text{-FutV}) & f : \text{Fut} \tau \in \Gamma & \quad (C\text{-Task}) & f : \text{Fut} \tau \in \Gamma \\
\Gamma \vdash (\text{fut} f) \text{ ok} & \quad \Gamma \vdash (\text{fut} f v) \text{ ok} & \quad \Gamma \vdash (\text{task} e f) \text{ ok} \\
& f_1 : \text{Fut} \tau_1 \in \Gamma & \quad f_2 : \text{Fut} \tau_2 \in \Gamma & \quad \Gamma \vdash e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash (\text{chain} f_1 e f_2) \text{ ok} \\
\Gamma \vdash \text{config}_1 \text{ ok} & \quad \Gamma \vdash \text{config}_2 \text{ ok} & \quad \text{futset}(\text{config}_1) \cap \text{futset}(\text{config}_2) = \emptyset \\
\Gamma \vdash \text{config}_1 \urcorner \text{config}_2 \text{ ok}
\end{align*} \]

**Figure 2.2:** Configuration type system

\[
\begin{align*}
\text{futset}(e) & = \emptyset \\
\text{futset}((\text{fut} f)) & = \{f\} \\
\text{futset}((\text{fut} f v)) & = \{f\} \\
\text{futset}((\text{task} e f)) & = \emptyset \\
\text{futset}((\text{chain} f_1 e f_2)) & = \emptyset \\
\text{futset}((\text{config}_1 \urcorner \text{config}_2)) & = \text{futset}((\text{config}_1)) \cup \text{futset}((\text{config}_2))
\end{align*} \]

**Figure 2.3:** Future set function needed for the configuration type system (figure 2.2)

\[
\begin{align*}
C & ::= [ ] \mid C e \mid v C \\
& \quad \mid \text{get} C \mid C \rightarrow e \mid v \rightarrow C \\
& \quad \mid [C] \mid C \circ e \mid v \circ C \mid \{C\} \\
& \quad \mid (C | e) \mid (e | C) \\
& \quad \mid e \geq C \mid C \geq v \\
& \quad \mid C \times e \\
& \quad \mid \text{lift} f C \\
& \quad \mid \text{extract} C \\
& \quad \mid \text{peek} C \\
& \quad \mid \text{join} C \\
T & ::= (\text{task} C f)
\end{align*} \]

**Figure 2.4:** Evaluation and task context

\[
\begin{align*}
\text{(Expressions)} & \quad \text{(C-contexts)} & \quad \text{(T-contexts)} \\
& \quad e \equiv e' & \quad C[e] \equiv C[e] & \quad T[e] \equiv T[e'] \\
\text{(CommutativeConfig)} & \quad \text{config} \text{ config}' \equiv \text{ config'} \text{ config} \\
\text{(AssociativeConfig)} & \quad (\text{config config'}) \text{ config}'' \equiv \text{ config} (\text{config config'})
\end{align*} \]

**Figure 2.5:** Structural congruence
<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(PeekM)</strong></td>
<td>( \text{peek} (e_1</td>
</tr>
<tr>
<td><strong>(PeekS)</strong></td>
<td>( \text{peek} (\text{peek} e) = \text{peek} e )</td>
</tr>
<tr>
<td><strong>(LiftFutGet)</strong></td>
<td>( \text{lift} f = { \text{get} f } )</td>
</tr>
</tbody>
</table>

**Figure 2.6:** Parallel combinator equivalences

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(SeqE)</strong></td>
<td>( { } \gg e' \rightarrow { } )</td>
</tr>
<tr>
<td><strong>(SeqS)</strong></td>
<td>( { e } \gg e' \rightarrow { e' e } )</td>
</tr>
<tr>
<td><strong>(SeqM)</strong></td>
<td>( (e_1</td>
</tr>
<tr>
<td><strong>(ExtractE)</strong></td>
<td>( \text{extract} { } \rightarrow [] )</td>
</tr>
<tr>
<td><strong>(ExtractS)</strong></td>
<td>( \text{extract} { e } \rightarrow [e] )</td>
</tr>
<tr>
<td><strong>(ExtractM)</strong></td>
<td>( \text{extract} (e_1</td>
</tr>
<tr>
<td><strong>(JoinE)</strong></td>
<td>( \text{join} { } \rightarrow { } )</td>
</tr>
<tr>
<td><strong>(JoinS)</strong></td>
<td>( \text{join} { e } \rightarrow { e } )</td>
</tr>
<tr>
<td><strong>(JoinM)</strong></td>
<td>( \text{join} (e_1</td>
</tr>
</tbody>
</table>

**Figure 2.7:** Recursive parallel combinator definitions
\[ T[(\lambda x.e) v] \rightarrow T[e[v/x]] \]

(\text{fresh } f)

\[ T[async e] \rightarrow (fut f)(task e f) T[f] \]

(GET)

\[ (fut f v) T[get f] \rightarrow (fut f v) T[v] \]

(\text{fresh } f_2)

\[ T[f_1 \rightsquigarrow \lambda x.e] \rightarrow (fut f_2)(\text{chain } f_1 (\lambda x.e) f_2) T[f_2] \]

(CHAINVAL)

\[ (fut f_1 v) (\text{chain } f_1 e f_2) \rightarrow (fut f_1 v)(\text{task } (e v) f_2) \]

(FUTVAL)

\[ (fut f) (\text{task } v f) \rightarrow (fut f v) \]

(SEQLF)

\[ T[(\text{lift } f) \gg e] \rightarrow T[\text{lift } (f \rightsquigarrow e)] \]

(OTHERV)

\[ T[e_1 | \{v\} | e_2 \gg e_3] \rightarrow T[e_1 | \{v\} | e_2] \]

(OTHERH)

\[ T[\{\} \gg e] \rightarrow T[e] \]

(PEEKV)

\[ T[\text{peek } (e_1 | \{v\} | e_2)] \rightarrow T[\{v\}] \]

(PEEKH)

\[ T[\text{peek } \{\}] \rightarrow T[\{\}] \]

(STRUCT)

\[ e \equiv e' \quad e' \rightarrow e'' \quad e'' \equiv e''' \]

\[ e \rightarrow e'''' \]

\text{Figure 2.8: Operational semantics}

(\text{ONECONFIG})

\[ \text{config}_1 \rightarrow \text{config}_1' \]

\[ \text{config}_1 \text{ config}_2 \rightarrow \text{config}_1' \text{ config}_2 \]

\text{SHARECONFIG)

\[ \text{config}_1 \text{ config}_2 \rightarrow \text{config}_1' \text{ config}_2' \quad \text{config}_1 \text{ config}_3 \rightarrow \text{config}_1' \text{ config}_3' \]

\[ \text{config}_1 \text{ config}_2 \text{ config}_3 \rightarrow \text{config}_1' \text{ config}_2' \text{ config}_3' \]

(MANYPAR)

\[ T[e_1] \rightarrow T[e_1'] \quad T[e_2] \rightarrow T[e_2'] \]

\[ T[e_1 | e_2] \rightarrow T[e_1' | e_2'] \]

\text{Figure 2.9: Parallel reduction rules}
\begin{align*}
\text{depset}(c) &= \emptyset \\
\text{depset}(f) &= \{f\} \\
\text{depset}(x) &= \emptyset \\
\text{depset}(\lambda x.e) &= \text{depset}(e) \\
\text{depset}(e_1 e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}(\text{async } e) &= \text{depset}(e) \\
\text{depset}(\text{get } e) &= \text{depset}(e) \\
\text{depset}(e_1 \leftarrow e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}([]) &= \emptyset \\
\text{depset}([e]) &= \text{depset}(e) \\
\text{depset}(e_1 \diamond e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}(\{\}) &= \emptyset \\
\text{depset}(\{e\}) &= \text{depset}(e) \\
\text{depset}(e_1 | e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}(e_1 \gg e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}(e_1 \prec e_2) &= \text{depset}(e_1) \cup \text{depset}(e_2) \\
\text{depset}(\text{liftf } e) &= \text{depset}(e) \\
\text{depset}(\text{extract } e) &= \text{depset}(e) \\
\text{depset}(\text{peek } e) &= \text{depset}(e) \\
\text{depset}(\text{join } e) &= \text{depset}(e)
\end{align*}

Figure 2.10: Future dependency set function definition

\begin{center}
\begin{tikzpicture}
\node (f4) at (0,0) {$f_4$};
\node (f3) at (1,-1) {$f_3$};
\node (f2) at (2,-2) {$f_2$};
\node (f1) at (3,-3) {$f_1$};
\draw (f1) -- (f3) -- (f2) -- (f4);
\end{tikzpicture}
\end{center}

Figure 2.11: Dependency tree of \texttt{config}_{dep\_ex}
Chapter 3

Additional Constructs

The proposed semantics have been kept to a bare minimum core language, which has shown to be simple, yet expressive enough to solve complicated problems. This simplicity, while useful for proving properties of the language extension and building other elements upon it, makes it hard for a programmer to directly use the proposed syntax. This chapter provides suggestions that could improve the legibility and expressiveness of the parallel combinators extension, which would be of particular use to the users and implementors of the language.

3.1 Core Language Extensions

One possible way to extend the parallel combinators extension is to enrich the proposed core language. The following constructs are a few examples that could be particularly useful.

3.1.1 Let Bindings

This extension greatly increases the readability of programs by helping the user avoid repetition. The let binding names an expression, which is then used inside another expression. The following shows the equivalence of this construct if it were built using the existing core, as well as its type rule and operational semantics if it were included in the language.

\[
\text{let } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) e_1
\]

\[
\Gamma \vdash e_1 : \tau' \quad \Gamma, x:\tau' \vdash e_2 : \tau \\
\frac{}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}
\]

\[
T[\text{let } x = v \text{ in } e] \rightarrow T[e[v/x]]
\]

3.1.2 Wildcards

Encore is a language with side effects, so at times one might want to run a computation and ignore its result. In these cases, choosing an explicit variable name is unneeded,
as it will not be used if the value is unwanted. Wildcard variable binders, written as an underscore instead of providing a variable name, provide the ability to assign computational results to a character that will never be used.

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda_x.e : \tau} \quad T[(\lambda_x.e) v] \rightarrow T[e]
\]

### 3.1.3 Sequencing

Encore, as a non-functional language with side effects, may sequence multiple expressions. The intention of this extension is to evaluate the computations of a sequence and only use the result of the last expression. Sequencing could be expressed by only using wildcards in the presented subset of Encore, with the inclusion of let bindings to help increase the program legibility, yet results in a verbose form of expressing this simple idea. Another possibility is to introduce syntax that expresses sequencing. The reason the Otherwise combinator was designated the \(<\) symbol was to use \(=\) as the syntax for this extension, a commonly used symbol to express sequencing in other programming languages.

\[
e_1 ; e_2 \equiv \text{let } _- = e_1 \text{ in } e_2
\]

\[
\frac{\Gamma \vdash e_1 : \tau'}{\Gamma \vdash e_1 ; e_2 : \tau} \quad C ::= \cdots | C ; e \ | v ; C
\]

### 3.2 Combining Combinators

The power of the proposed parallel combinators not only comes from the abstraction they provide to parallel computations, but also to their ability of forming more complex and powerful combinators. This section presents a few combinators that have surfaced from example programs built during the course of this thesis.

#### 3.2.1 Prune Combinator

The inspiration for this combinator comes from Orc’s pruning combinator. While this combinator was originally part of the proposed syntax, it was removed when realized that it could be expressed by using the **peek** combinator.

\[
e_1 \ll e_2 \equiv e_1 \ (\text{async} \ (\text{peek} \ e_2))
\]

\[
\frac{\Gamma \vdash e_1 : \text{Fut} (\text{Par} \ \tau') \rightarrow \text{Par} \ \tau \quad \Gamma \vdash e_2 : \text{Par} \ \tau'}{\Gamma \vdash e_1 \ll e_2 : \text{Par} \ \tau}
\]

\[
\text{fresh } f
\]

\[
T[\lambda x.e_1 \ll e_2] \rightarrow (\text{fut } f) (\text{task} (\text{peek } e_2) f) T[e_1[f/x]]
\]
As with Orc’s variant of this combinator, this operation executes two expressions in parallel, where the first expression will be suspended if it depends on a value provided by the second expression, and will continue once that value has been fulfilled. What is particularly different from Orc’s pruning combinator is that this version uses Encore’s futures as a way to execute both expressions in parallel. This works by letting the first expression execute, and only block if it awaits for the future provided by the second expression to be fulfilled.

### 3.2.2 Bind Combinator

This combinator provides a seemingly simple, yet very powerful function: it is capable of taking every expression that will be calculated by one parallel expression, and use its values to initiate separate parallel executions in another expression, and ultimately flattening this result.

\[
e_1 \gg e_2 \equiv \text{join}(e_1 \gg e_2)
\]

\[
\frac{\Gamma \vdash e_1 : \text{Par} \tau' \quad \Gamma \vdash e_2 : \tau' \rightarrow \text{Par} \tau}{\Gamma \vdash e_1 \gg e_2 : \text{Par} \tau}
\]

\[
T[e_1 \gg \lambda x.e_2] \rightarrow T[\text{join}(e_1 \gg e_2)]
\]

### 3.2.3 Imitating Maybe Types

Encore currently does not provide a notion of maybe types, which are also known as option types. Maybe types express the possibility of existence of a certain value. Using Haskell’s syntax, if an expression has the type \(\text{Maybe} \ Int\), the values of that expression can be \(\text{Just } v\) (where \(v\) is a number), or \(\text{Nothing}\).

Parallel combinators can be used to express this concept, although it is much less elegant than using real maybe types:

\[
\text{nothing} \equiv \{\}
\]

\[
\text{just } e \equiv \{e\}
\]

\[
\frac{\Gamma \vdash \text{nothing} : \text{Par} \tau}{T[\text{nothing}] \rightarrow T[\{\}]} \quad \frac{\Gamma \vdash e : \tau}{T[\text{just } e] \rightarrow T[\{e\}]}
\]

The empty nothing constructor of the maybe type can be represented with \(\{\}\), and the constructor encapsulating some datum can be represented with \(\{e\}\).

### 3.3 Syntactic Sugar for Parallel Combinators

In order to facilitate even further writing programs with this syntax, it should be possible to express parallel combinators with a reduced syntax. Instead of using the parallel
combinator to continuously combine parallel structures, as has been done throughout the thesis \(\{e_1\} \cdot \cdot \cdot \{e_n\}\), the following syntax is proposed: \(\{e_1, \ldots, e_n\}\).

To use this syntax, it is also important to incorporate the lifted futures, also giving the opportunity to easily write them with the rest of elements in a parallel structure. Thus, the proposed syntax to represent liftf \(e\) would be \(\{e\}\).

With these two proposals, it is easy to provide a desugaring function that would transform the provided shorthand into its core representation.

### 3.4 Exceptions

Exceptions can occur not only from troublesome computations, such as divisions by zero or arithmetic overflows, but also from problems that may arise in the machine where the computation is being executed, which may include not being able to open a file or memory corruption. Depending on the severity of the exception, these errors can cause the program to abort. Exceptions provide a way of signaling and recovering from an error condition during the execution of a program.

As explained previously, one alternative to controlling certain exceptions from occurring is through the use of a maybe type. The problem with this solution is that not much information is provided with the error. One possible solution would be to use something similar to the Either monad in Haskell, which also provides attached data when an error takes place.

Using a maybe or an either monad might be cumbersome, as it forces the user to continuously check the value of the data that is being carried around in the program. This could be solved by raising an exception during run time, which would provide certain information about the error, and using an error handler around the expression to be evaluated. The error handler contains an alternative expression to be executed in the occurrence of an exception. There are multiple possibilities for the semantics of the error handler, which include catching multiple types of errors and treating them with a single expression, or allowing the execution of a different expression for each caught error.

In the context of parallel combinators, where an error can occur in any element of the parallel structure, choosing one representation of exceptions over another may make it difficult for users to treat errors, as the user might be seeing an unexpected behavior from their program. Questions that one could ask oneself when exploring options are: If one element of a parallel structure raises an error, could this element not be ignored? If an error is truly exceptional, should the whole parallel structure not fail? What should happen to the rest of the computations in a parallel structure if an exception is raised?

Various programming languages have various constructs to provide support for exceptions, yet I believe that a try-in-unless approach [43] would be a very elegant addition to the language semantics. The proposed syntax for this construct could be the following:

\[
\text{try } x = e \text{ in } e' \text{ unless } E_1 \Rightarrow e_1 | \cdots | E_n \Rightarrow e_n
\]

What this syntax states is that the value of the expression \(e\) is bound to the variable \(x\),
and then the expression $e'$ is evaluated. If an exception, $E_i$, is raised during the evaluation of $e$, then its corresponding expression, $e_i$, is evaluated. Thus, if a single element in a parallel structure raises an exception, the whole construct would be discarded and have to be treated. In order to avoid this, it would be possible to use a fine-grained try-in-unless approach on each of the individual expressions in a parallel structure.
Chapter 4

Conclusions and Future Work

The main contribution of this thesis is the theoretical framework which integrates parallel computation combinators with futures in the Encore programming language. Its simplicity makes it a suitable starting point for implementing the parallel combinator extension not only in Encore, but also in any language with similar semantics. However, due to the theoretical nature of this thesis, it would not be suitable to directly translate its content into a language implementation. Details, such as the data structures that are used or the mechanisms to distribute tasks, have been abstracted away in order to improve the clarity of the semantics. This will provide the implementors of this extension with a high degree of flexibility when integrating parallel combinators into Encore.

The subtleties of proving the soundness of the extension proved to be much more difficult than originally thought. Consequently, parts of the formalization, such as the integration with active objects, were not developed due to time restrictions. This is not to say that there is little left to do, as the work provided here can be extended into multiple directions. This chapter contains examples of how to continue working on the ideas and results produced by this thesis.

4.1 Formalizing Exceptions

Exceptions were discussed in section 3.4, yet they still need to be extended into the semantics to be fully considered a part of the language. The proposed syntax for exceptions is easy to understand at a high level, yet may show unexpected behavior if it is not fully developed and incorporated into the formal semantics. The process of inclusion of this formalization into the language semantics, in addition to proving its soundness, might help find other alternatives to expressing exceptions in the parallel combinators.

4.2 Avoiding Intermediate Parallel Structures

Certain parallel combinators, such as the sequence combinator, create intermediate parallel structures when they are used. This process is costly at run time, both in space
and time, yet can be optimized through techniques known as operation fusion or de-
forestation [44–46]. These techniques try to combine functions in such a way that the
creation of intermediate structures is completely avoided. A simple example of fusion
on a sequence combinator could be the following:

\[
\text{parstruct } e_1 e_2 \equiv \text{parstruct } \lambda x. e_2(e_1 x)
\]

On the left hand side of the equivalence, the expression \(\text{parstruct } e_1\) creates an
intermediate parallel structure, \(e'_1\), which is then used for sequencing with \(e_2\): \(e'_1 e_2\).
The right hand side of the equivalence only creates a single parallel structure, which is
the result of sequencing the original parallel structure with the function composition of
the expressions \(e_1\) and \(e_2\).

4.3 Dealing with Unneeded Tasks and Futures

If one were to naively implement what is proposed in this thesis, the implementation
would be very inefficient. One of the main reasons for this is due to how tasks of
unneeded futures are handled in this thesis: they simply continue running. The \text{peek}
combinator, in particular, may potentially eliminate the need for the majority of the
computations in a parallel combinator. These unwanted futures and their associated
tasks stay in the complete system configuration, wasting hardware resources to compute
calculations that will not be used. It would therefore be interesting to explore different
options to remove unneeded tasks and futures during the evaluation of an expression.
One possibility would be to formalize and integrate high-level calculi that could express
memory management strategies in this extension, as well as prove its correctness in the
system [47].

4.4 (Efficiently) Implementing Parallel Combinators

The formalization of the parallel combinators provides no suggestions on how the ex-
tension should be implemented by integrating it into Encore. Furthermore, an efficient
implementation of parallel combinators is not trivial, as there is a myriad of different op-
tions to consider, such as deciding what data structures should be used for representing
parallel structures or if different data structures should be used in different situations,
how should tasks be distributed and managed, what optimizations could be performed
on the parallel combinators to reduce the amount of space or time they take to compute
results, etc. A significant amount of research would have to be conducted to ensure good
strategies for the implementation of the parallel combinators extension.

4.5 Creating Benchmarks for the Parallel Combinators

This option could be combined with the implementation of parallel combinators, as it is
interesting not only to have an implementation of this extension but also to compare it
with other solutions to a set of problems. By benchmarking the parallel combinators, it would not only be possible to compare their performance with solutions written in other programming languages, but also their expressiveness and legibility.

4.6 Exploring More Combinators

Only a few core combinators have been provided by this thesis, yet it would be beneficial to explore more complex functions that would be useful to programmers. The combinators already explored have been tested with a pen-and-paper approach to solving a few problems, a solution that does not scale well as the interaction between combinators cannot quickly be tested. Once parallel combinators have been implemented and used, it is very likely that more combinators will surface, providing more possibilities to improve their usability.

4.7 Integrating the Semantics with Active Objects

A prominent feature of Encore is the use of active objects, yet the proposed semantics in this thesis make no use of them. Another direction for the future work of this thesis would be to combine parallel combinators with the semantics of active objects, bringing the formalization of this extension closer to the formalization of Encore [22].
Bibliography


