Fachrichtung 6.2 – Informatik
Naturwissenschaftlich-Technische Fakultät I
– Mathematik und Informatik –
Universität des Saarlandes

A Concurrent λ-Calculus with Promises and Futures

Diplomarbeit

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka
Zweitgutachter Prof. Dr. Andreas Podelski
Betreuung durch Dr. Joachim Niehren und Prof. Dr. Gert Smolka

Jan Schwinghammer

Februar 2002

Hiermit erkläre ich, Jan Schwinghammer, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 21. Februar 2002

Abstract

Concurrency plays an important rôle in programming language design. Logic variables in the form of futures and promises provide a means of synchronization and communication in concurrent computation. Futures and promises, which differ from general logic variables in that a distinction is made between reading and writing them, have been introduced previously. However, no formal operational semantics has been provided for promises.

In order to formally investigate properties of futures and promises in a functional setting, a concurrent λ-calculus extended with futures and promises is presented. It is intended to provide a computation model for the programming language Alice [Ali02]. We prove the calculus confluent, and give a proof showing strong normalization in the simply typed case without promises. Further, we introduce a type system so as to statically enforce proper use of promises in the calculus.

Acknowledgments

I would like to thank my supervisors, Joachim Niehren and Gert Smolka, for proposing this interesting project to me. I am indebted to them for all the ideas they brought in, and also the time they devoted to this project.

Next, I want to thank all the other members of the Programming Systems Lab for creating an enjoyable working atmosphere. I must mention here Andreas Rossberg and Leif Kornstaedt, who were always willing to patiently answer questions on the Alice project, and Patrick Cernko, who helped more than once when I was struggling with information technology.

I am grateful to Manuel Bodirsky, Timo von Oertzen and Tobias Gärtner for being friends and making sure studying is fun. Manuel Bodirsky, Arno Eigenwillig, Tobias Gärtner, Joachim Niehren, Andreas Rossberg and Gert Smolka read and commented on previous versions of this thesis. Finally, I would like to thank my parents for their support throughout my life.

Contents

Chapter 1

Introduction

Concurrency plays an increasingly important rôle in programming, evidenced by the variety of languages offering support for concurrent computation. Java [GJSB00], Concurrent ML [Rep99] and Oz [Smo95b, Smo95a] are examples. The popularity of concurrency is partly due to the rise in distributed and parallel computing, but also due to the use of concurrency in applications such as user-interfaces and the like.

Synchronization and communication of concurrently running threads are fundamental operations in these languages. Different mechanisms are employed, e.g. channels, shared cells or logic variables are used to transmit information between threads, put locks on shared data structures and synchronize on certain events.

In this thesis, we investigate how concurrent computation and a restricted form of logic variables behave in combination with a functional core language. The motivation for doing this is the programming language Alice [Ali02], where this variant of logic variables are used. Also, we are interested in what the basic properties of such languages are from a theoretical point of view.

More specifically, we will introduce a small concurrent functional language containing restricted forms of logic variables, called *promises* and *futures*. We provide an operational semantics and prove several technical results. In particular, the reduction relation of the concurrent language is proved confluent, and for the simply typed fragment we prove a strong normalization theorem.

Concurrent Computation. Concurrency as understood for this thesis may be characterized as interleaving computation. In particular, concurrent computation does not depend on parallelism of the underlying infrastructure. Certain tasks in computing are inherently concurrent in their nature, prime examples being user-interaction, distributed systems, autonomous software agents, and systems providing multiple services to their clients. Concurrent threads provide the right level of abstraction to deal with asynchronous (and slow) requests, and their subsequent synchronization.

Logic Variables and Synchronization. General logic variables stem from the class of logic programming languages such as Prolog [Pro85, SS94, JL87]. Initially, when freshly introduced, they carry no value. Therefore they allow for the stepwise construction of values, using further logic variables for the construction of subvalues if necessary. They are *transient*, in that they are identified with their value as soon as this becomes available. This provides a mechanism for *implicit synchronization* of concurrent threads that share a logic variable: A thread reading the variable automatically suspends while sufficient information is not available.

We will be concerned with *futures* and *promises*, which differ from general logic variables in that a distinction is made between reading and writing them. Bidirectional unification can be replaced by (single-) assignment.

Functional Programming. Functional programming is a paradigm based on well-understood and solid theoretical foundations. We will adopt the common agreement that λ-calculus provides a good foundation to reason about the operational behaviour of functional programs. For a recent, comprehensive account of λ-calculus and functional programming, see [Mit96].

Semantically, one distinguishes *strict*, or *eager*, languages such as Standard ML [MTHM97], from *non-strict* languages, e.g. Haskell [ABB+99] and Miranda [Tur85]. Function application in strict languages is done by first evaluating the arguments. They are strict in the sense that both function and arguments must evaluate to values. On the other hand, non-strict languages evaluate their arguments only if needed in the function body. In [Oka98] it has been demonstrated that *lazy evaluation*, an implementation technique of non-strictness by evaluation on demand, provides an important tool for designing amortized data structures in a purely functional setting.

The absence of side-effects in purely functional languages makes them well-suited base languages for concurrent programming. In fact, several concurrent extensions of functional languages have been developed during the last two decades. CML [Rep99], Concurrent Haskell [JGF96], and Facile [GMP89] are a few examples.

Models of Computation. Commonly, high-level programming languages are explained with respect to a computation model. One goal of this thesis is to provide a computation model for the concurrent functional programming language Alice [Ali02], incorporating logic variables in the form of promise and futures.

There are numerous formal models of basic concurrent computation. Among the best-known are CCS [Mil89] and its successor, the π-calculus [MPW92, SW01], where computation is based solely on the transmission of names via shared channels. Pict is a language based on this model [PT00]. The join-calculus [FG96], underlying the language JoCaml, [CF99] is related, but somewhat closer to actual programming languages.

Concurrent constraint programming [Sar93, Smo95b, VS+96] is based on the idea of concurrently acting agents (actors) which communicate via shared logic variables. These reside in a global constraint store whose information increases monotonically as computation proceeds.

Finally, the models underlying functional programming are untyped and typed λ-calculi, often extended by constants and primitive functions [Plo77]. The formal semantics of λ-calculus with evaluation on demand has been studied, e.g., in [AFM+95, AF97, MOW98].

1.1 Futures and Promises

So what exactly do we mean by futures and promises? A formalization of the concept is deferred to Chapter 3, here we confine ourselves to an informal description.

General logic variables, e.g. as found in Prolog [Pro85, SS94] and Oz [Smo95b, Smo95a], are introduced without being bound to a value, they obtain their final value by participating, possibly more than once, in unification. They provide a simple mechanism for the synchronization of concurrent threads that share a logic variable: A thread reading the variable may suspend due to missing information, and it can resume computation as soon as this information becomes available.

However, logic variables also can become a source of programming errors that are hard to detect since the direction of information flow cannot be statically determined. This is especially true when it comes to concurrency, where deadlocks and indeterminism may result from the use of logic variables. Specifically, it is not at all an easy task to determine which threads can write a particular logic variable and which threads are readers only.

In Oz, there is a second kind of logic variable, called *future*. Futures allow reading only. In fact, futures have already been present in Multilisp [Hal85, FF96] in the context of parallelism, allowing the programmer to explicitly indicate which subexpressions may be evaluated in parallel by annotation with the keyword `future`. The expression `future(`e`)` may thus return a future associated to e, and begin evaluation of e in parallel. As soon as the value denoted by e has been computed, this value and the future are identified.

Similarly, a second kind of "lazy" future annotation is introduced. Expressions can now be declared as either `concur` or `byneed`, where by-need expressions are evaluated only if and when their result is needed.

The expression

$$\texttt{concur}\ x{=}e\ \texttt{in}\ e'$$

evaluates e in a new thread, while evaluation of e' continues as long as no attempt of reading from x is made. Again, once the value e evaluates to is available, future and value are equated in e'. In the case of

$$\texttt{byneed}\ x{=}e\ \texttt{in}\ e'$$

evaluation of e is initiated only after the first attempt to read from x. Thus, byneed futures are somewhat dual to the strictness annotations commonly found in non-strict languages such as Haskell [ABB+99]. They provide a way to obtain lazy evaluation without major adjustments of the framework.

Now viewing a future as the reading side of a logic variable, a *promise* is just the writer, promising to eventually supply a value. For example,

$$\texttt{prom}\ \mathit{bind}_x\ \texttt{for}\ x\ \texttt{in}\ e$$

introduces two new entities: the future x of type σ, and the promise bind_x which can be *fulfilled* by applying it to some value v: The application

$$\mathit{bind}_x\ (v)$$

yields a dummy value, but also, more importantly, binds the future x to v as side-effect.

In summary, there are three ways to introduce a future x. First, as a future "referring" to a concurrent computation, second as a byneed- (or *lazy* future, and third as a promised future. Comparing the different ways to produce futures, the following observation can be made: for futures x created by `concur` or `byneed`, it is immediately (statically) clear who the eventual supplier of x's value is. In contrast, for a promised future, the supplier will be determined only dynamically. Several examples are given in Chapter 2, showing how promises and futures are used in programming.

So naturally the question arises what the benefits are from separating reading and writing of a logic variable. Conceptionally, the future x should be bound only once. Consequently, a promise should be fulfilled only once, i.e. the expression $\mathit{bind}_x\ v$ should not be executed several times. In fact, logic variables are often used in this way in concurrent logic programming (Figure 1.1): Writing x is done in a sequentialized fashion, having the actual writer in a semaphore or the like. However, several concurrent threads may read from x without causing any difficulties.

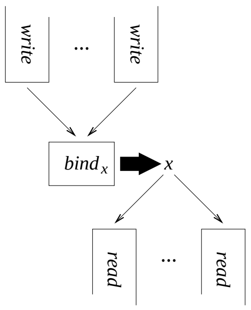

Figure 1.1: Future x and promise

Strictly adhering to the rule of assigning a logic variable at most once will eliminate the usual problems caused by threads competing to write a particular logic variable. As a consequence, the result of computations is now deterministic, since logic variables are no longer true *resources*.

What happens if we try to fulfill a promise that has already been fulfilled? Usually, this is an unintended, and unwanted, state of the computation. There are several design decisions conceivable:

- *Carry on.* As the system is in an abnormal, if not illegal, state, we do not consider this an appropriate solution.
- *Suspend.* One may argue that this is a sensible decision. Moreover, it is very simple. Suspending when trying to fulfill a promise for the second time is the approach we will take over in the next chapter.
- *Throw an exception.* This seems the best solution in practice, as it is then the programmer's responsibility to provide an application-specific error handling. In fact, this is the way the problem is solved in the programming language Alice [Ali02].
- *Don't let it happen.* This is what the intention of using promises tells. In Chapter 5, we will consider how this can be enforced automatically, using a system of linear types.

It would be interesting to see how the third idea, using exceptions, can be formalized. We just remark that it is not immediately clear what properties are required of exceptions in a concurrent language. Certainly this question is of practical importance.

1.2 Futures and Promises in the Programming Language Alice

This work is done in the context of the programming language Alice, the interested reader is referred to [Kor01, Ali02] for more information. Alice is a concurrent extension of the strict functional language Standard ML [MTHM97]. ML is strongly typed, providing polymorphic type inference [Mil78, DM82], a sophisticated, parametric module system and a considerable amount of programming tools and libraries. Alice aims at combining the expressive power of concurrent constraint programming, as found in Oz [Smo95b, Smo95a], with a functional core language. This was first outlined in [Smo98], where it is argued that logic and concurrent constraint programming benefits from being based on a call-by-value functional core language.

Futures and promises in Alice are provided through modules, which are called *structures* as in Standard ML.

Futures. The structure `Future` contains, among others, the values

$$\texttt{concur:}\quad (\mathsf{unit}{\to}\alpha){\to}\alpha$$

and

$$\texttt{byneed:}(\mathsf{unit}{\to}\alpha){\to}\alpha$$

that, given a procedure $f : \mathsf{unit}{\to}\alpha$, immediately return a future while evaluating f`()` in a new thread in the case of `concur`, resp. creating a suspension in the case of `byneed`.

The structure `Future` also contains the exception `Future`. If evaluation of f`()` raises an exception *exn*, then every attempt to access the future will raise `Future(`*exn*`)`. Moreover, it contains further "impure", non-functional operations on futures, such as `await:`$\alpha{\to}\alpha$, `awaitOne:`$\alpha{\times}\beta{\to}\alpha$ and `isFuture:`$\alpha{\to}\mathsf{bool}$ with the obvious behaviours.

Promises. Promises in Alice are provided by the structure `Promise`. This structure defines the type α promise, along with operations

$$\texttt{promise:}\mathsf{unit} \to \alpha\ \mathsf{promise}$$

which creates a new promise,

$$\texttt{future:}\alpha\ \mathsf{promise} \to \alpha$$

which returns the future associated with the promise, and the "application"

$$\texttt{fulfill:}\quad \alpha\ \mathsf{promise} \times \alpha \to \mathsf{unit}$$

The application `fulfill(`p`,`v`)` binds the future associated with p to v (provided it is not the future itself). Any further attempt to fulfill p will raise the exception `Promise`.

Providing promises in this way differs from how we introduced promises in the previous section in two respects. First, in Alice there is the specific type α promise for promises, and application of a promise is done by `fulfill`. This distinguishes writing $y\ v$ for an arbitrary, non-promise variable y from $p\ v$ for a promise p, by clearly indicating the side-effect of binding a future in the type. Second, the associated future is obtained *from* the promise p through `future(`p`)`, whereas in Section 1.1 both future and promise are introduced simultaneously. The effect of

$$\texttt{prom}\ \mathit{bind}_x\ \texttt{for}\ x\ \texttt{in}\ e$$

may thus be achieved by writing

$$\texttt{let}\ \mathit{bind}_x\ \texttt{= promise() in let}\ x\ \texttt{= future(}\mathit{bind}_x\texttt{) in}\ e$$

in Alice, additionally replacing occurences of $\mathit{bind}_x\ v$ with `fulfill(`bind_x`,`v`)`. Note however that the type structure of Alice is rich, using α promise instead of simply typing promises by $\alpha \to$ `()`, and this replacement cannot be done automatically.

Conversely we may regard promises of Alice as pairs $\langle bind_x, x\rangle$, so Alice's promise translates to

$$\texttt{promise()} \rightsquigarrow \texttt{prom}\ bind_x\ \texttt{for}\ x\ \texttt{in}\ \langle bind_x, x\rangle$$
$$\texttt{future}\ e \rightsquigarrow \texttt{snd}\ e$$

and

$$\texttt{fulfill(}e\texttt{,}e'\texttt{)} \rightsquigarrow \texttt{(fst}\ e\texttt{)}\ e'\ .$$

The reason for deviating from promises as found in Alice is motivated by technical considerations: In Section 5.2, a type system is presented that ensures linear use of promises, while allowing the associated futures to be used without restriction. Thus, it makes sense to introduce promise and future simultaneously, but as seperate entities.

1.3 Contributions of the Thesis

In this thesis, we develop the theory of a concurrent computation model with futures and promises, called λ^{FP}. Its sequential core is based on the call-by-value λ-calculus, but it also provides for evaluation on demand through by-need annotations.

In the next part, Chapter 2, several examples on the use of futures and promises are presented. The aim is to demonstrate the usefulness of these concepts in concurrent functional programming.

In Section 3.2, some intuition is developed, and in Section3.3 the calculus λ^{FP} is formally introduced. A type system very similar to that of the simply typed λ-calculus is presented in Section 3.5, and a subject reduction theorem for this language is proved. In Section 3.6, the expressiveness of the language is investigated, with the outcome that promises yield computational completeness even in the simply typed case, as they suffice to define general recursion.

In Chapter 4, we take a closer look at the sublanguage $\lambda^{F}$of promise-free expressions. In particular, we embed the simply typed fragment into the simply typed λ-calculus in Section 4.3, thereby obtaining a strong normalization theorem for the promise-free simply typed language. In Section 4.2, we prove both the simply typed and untyped languages confluent. In fact, they are proven *uniformly* confluent [Nie00], and this property is used in Section 4.4, where complexity-preserving embeddings of call-by-value and call-by-need λ-calculus are given. Uniform confluence then provides a way to relate their respective complexities, which yields a formal proof that the call-by-value strategy needs at least as many computational steps as call-by-need, along the lines of [Nie00]. The results of Section 4.4 have been summarized in [Sch01].

Chapter 5 deals with the full language again. In 5.2, a refined type system based on mode annotations is proposed. It statically enforces proper once-only writing of futures via promises. In fact, in Section 5.3 we prove that reduction on well-typed expressions is uniformly confluent as well. Next, in 5.4 cells are added, obtaining the calculus λ^{FPC}. As demonstrated by the example in Section 2.3, λ^{FPC} may indeed serve as the core language for a concurrent programming language, as it is sufficiently expressive to program channels for many-to-one and many-to-many communication.

1.4 Related Work

There has been much work on the semantics of concurrent programming languages, as well as on formal calculi addressing the related areas of concurrency, mobility and distribution.

The semantics most closely related to the one presented here is probably one given by Berry et al. in [BMT92] for a fragment of Concurrent ML (CML, see [Rep92, Rep99]), another concurrent extension of Standard ML. In [FHJ98], a labelled transition system semantics has been presented for CML which provides a notion of bisimulation equivalence on programs. Following work in [JR00] extends this to cover almost all the essential features of CML, including the dynamic generation of new channels. However, there are no logic variables present in CML; channels are taken as primitives for communication.

Futures have been introduced to functional programming already in Multilisp [Hal85]. In [FF96], Flanagan and Felleisen give a formal semantics at various abstraction levels for a λ-calculus with `let` and futures, making use of abstract machines. A deterministic, sequential operational semantics that simply treats futures as annotations is shown to agree with both a parallel reduction semantics and a placeholder object semantics. A key step in the proof is a Diamond Lemma, similar to the notion of uniform confluence shown to hold for our calculus. This latter work was performed with the intention of providing a basis for implementing futures and proving correctness of certain compiler optimizations in the context of parallelism. Our semantics for futures corresponds roughly to the parallel reduction of [FF96].

Promises as a type-safe mechanism for remote procedure call, being able to deal with exceptions, are described in [LS88]. Closely related are the I-Structures [ANP89] of Id and its successor pH: An I-structure is allocated by `array(`l`,`r`)`, returning an "empty" array with lower and upper index bounds l and r. Such an I-structure `A` can be filled subsequently by constraints of the form `A[`i`] =` v. Multiple writing of location i will result in a run-time error.

Other proposals for integrating logic variables into functional languages have been made. For example, the accumulators of [PE88] provide a quite general way of introducing state to a functional language, which can be updated preserving determinacy of evaluation. Promises can be seen as an instance of this scheme.

Type systems have been successfully used for expressing safety properties, e.g. the system in [Bou97b] was proposed with the intention of proving the absence of deadlock. In the context of concurrent calculi, types have been found useful for proving security and behavioural properties as well. In particular, our linear type system with mode annotations, presented in Chapter 5, is inspired by the various typing disciplines for resource management in π-calculus proposed in the literature, see [SW01] for a recent account.

It turns out that the type system resembles very much the uniqueness annotations [BS96] found in the language Clean. Further, similar type systems have been devised in [TWM95, Mog98] with the rather different motivation of program transformation and optimization. There, the annotations yield information in the sense of a program analysis [NN99].

Boudol's Blue calculus [Bou97a] integrates both the λ-calculus and the π-calculus in a direct way. The calculus shares important properties with the language presented here: (small) β-reduction is strongly normalizing and satisfies the diamond property. However there are differences, e.g. β-reduction in the Blue calculus is split into two parts, substitution of variables and resource fetching.

The theory of uniformly confluent calculi has been developed in [Nie00]. It was put to use to show that call-by-need complexity of λ-terms is bounded from above by call-by-value complexity. These ideas form the basis of Section 4.4 of this thesis.

Chapter 2

Programming with Futures and Promises

In this chapter several small examples are presented, involving the use futures and promises. They are adapted from [Ali02]. These examples shall provide the intuition that guides the formal development in later chapters. Here, the examples are written in an informal way, using a notation close to Standard ML syntax. However, we won't give a formal semantics in this chapter.

2.1 Lazy List Construction

Lazy futures provide a convenient way of implementing possibly infinite lists. As a simple, concrete example, consider the following.

```
fun generate f n =
   byneed ((f n) ::generate f (n+1))
```

The `byneed` keyword delays evaluation of the infinite list

$$[f(n), f(n+1), f(n+2), \dots]$$

which is the result computed by `generate` f n. For example, `generate (fn x => x) 0` computes the list

$$[0, 1, 2, \dots]$$

of natural numbers. Evaluation is done elementwise and only if and when the current element is actually needed. More precisely,

```
generate (fn x => x) 0
```

immediately returns a lazy future of integer list type. Accessing its head, evaluation of this future is initiated, yielding the expected result 0. Likewise, accessing its tail will return a new lazy future with head 1 and another lazy future representing its tail, as expected.

2.2 Streams

A different way of constructing potentially infinite lists is by ending a list with a promised future, which may be replaced by the tail of the list later on. This is a technique useful for dealing with, e.g., input, in particular in combination with concurrency. The channels presented below use essentially this idea. As a simple example of this kind, consider the following.

```
let val p = promise()
    val c = cell p

    fun app n =
       let val p = exchange(c, promise())
           val h = future p
       in
           fulfill(p, n::h)
       end
in
    (app, future p)
end
```

This first introduces a new promise p, locating it in the cell c. Cells are comparable to ML reference cells, with the exception that an atomic exchange operation is provided instead of separate update and dereferencing. A call to the append function will then exchange this

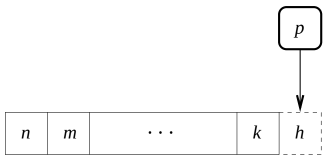

Figure 2.1: Constructing a list iteratively

promise with a freshly created one. Further, by fulfilling the old promise, it will replace the associated future with the new "rest list", consisting of the argument and the future of the newly created promise. For example, after successive calls to append with arguments $n, m, \ldots, k$, the constructed list is depicted in Figure 2.1, with p allowing the construction to continue. Note that whenever access to the next element is not possible, the requesting thread just suspends on the future h until the next call to append yields the next list element. This technique lies at the heart of how channels for many-to-one communication between concurrent threads can be implemented in a language with logic variables.

This example also demonstrates how promises may be used to construct data structures "with holes" for values to be plugged in later. In contrast to purely functional lists, the above append function appends an element onto the tail of a list in constant time.

2.3 Channels

Here, we demonstrate how to implement a (monomorphic) channel with unbounded capacity for buffering in a language with cells. These channels are adequate for many-to-many communication.

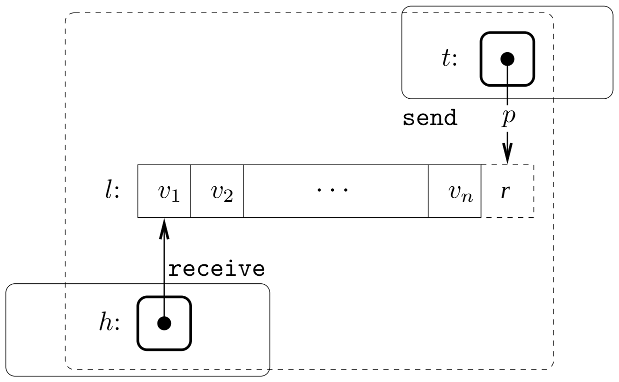

Figure 2.2: Implementation of channels

The channel can be accessed via two procedures; `send`:$\sigma \rightarrow$`unit` will write a given value to the channel, and `receive`:`unit`$\rightarrow \sigma$ will return the next value from the channel. In case there is no such value, `receive` will block until some thread does a `send` on this channel.

The stream is implemented by a finite list, ended not by the usual `nil`-constructor, but by a logic variable representing the elements that are to be written by subsequent `send`s. The basic idea of the implementation are the streams of the preceding Section 2.2. There are references to the *read end* and *write end* of the channel, making use of cells with an (atomic) update operation `exchange`. Figure 2.2 shows an illustration of the implementation, the implementation itself is given in Figure 2.3 on the following page.

A call to `send` will fulfill the old promise, replacing the future ending the buffer by `cons(`v`,`r'`)`, where r' is a fresh future. Its promise will be written into the cell t, which, as invariant, contains at any point of time the promise to bind the end-of-buffer future. Reading from the buffer is done by "moving the head-pointer h." Note that `receive()` will block (more specifically, `tail` l will block), provided h points to a promised future. Note also that these channels are in fact appropriate for many-to-many communication. The implementation is *thread-safe*, since multiple reader threads cannot interfere *in between* the atomic exchange operation in `receive`: the current list l is replaced by a future that is then bound to the tail of l. Any thread attempting to read (via `receive`) from h will block until h contains `tail` l. Using a promised future in this way, it is generally possible to obtain a dereferencing operation from the atomic exchange operator.

2.4 Asynchronous Remote Procedure Call

As final example, we consider Remote Procedure Calls (RPCs) and how futures serve to obtain asynchronicity. In distributed systems, with remote procedure calls there is usually the problem of network latency. Suppose the language supports only synchronous RPCs directly, then `concur` allows to turn this into an asynchronous operation. The idea is to simply *wrap* the call, continuing computation with the future that is immediately returned. To be somewhat more concrete,

```
let val p = promise():σ list
    val r = future p
    val t = cell(p)
    val h = cell(r)
in
    let fun send v =
        let val p′ = promise():σ list
            val r′ = future p′
            val p = exchange(t,p′)
        in p (cons(v,r′)) end

        fun receive _ =
        let val q = promise():σ list
            val l′ = future q
            val l = exchange(h,l′)
        in q (tail l); head l end
    in (send, receive) end
end
```

Figure 2.3: Implementation of channels: $\texttt{channel}_\sigma$

```
let val r = rpc_f(arg)
in
    exp
end
```

will block until the result of the call to f is bound to r, only then evaluation of *exp* proceeds. However, turning this into an asynchronous call as in

```
concur r = rpc_f(arg)
in
    exp
end
```

allows for immediate evaluation of *exp*, blocking only if and when the result of the RPC is in fact needed.

Chapter 3

A Calculus of Futures and Promises

In this chapter, a small language based on λ-calculus is introduced that incorporates promises and futures. It formalizes the concept of "placeholder variables" and the means to fill them later on, in the context of a functional language.

3.1 Notation

In this section, we fix the notation that we use throughout the thesis. Also, we will recall some not so common definitions.

Sets, Relations and Maps

By $\mathbb{N}$ we denote the non-negative integers including 0. $A \cap B$, $A - B$ and $A \cup B$ are intersection, difference and union, resp., of sets. More generally, if $A_i, i \in I$ is a family of sets, then $\bigcup\{A_i \mid i \in I\}$ is their union. $A \times B$ is the cartesian product, and A^n is the n-fold cartesian product $A \times \cdots \times A$. By $A \uplus B$ we denote the disjoint union of sets. We write $=_{\mathsf{df}}$ for definitional equality, and $\equiv$ for structural equality of terms.

We will be concerned with binary relations $\mathcal{R} \subseteq A \times B$ only; we write $(s,t) \in \mathcal{R}$, $\langle s,t \rangle \in \mathcal{R}$ and $s\mathcal{R}t$ interchangeably. For relations $\rightarrow_1 \subseteq A_1 \times A_2$ and $\rightarrow_2 \subseteq A_2 \times A_3$ we denote their composition by $\rightarrow_1\rightarrow_2$. That is, $s_1 \rightarrow_1\rightarrow_2 s_3$ iff there is some $s_2 \in A_2$ such that $s_1 \rightarrow_1 s_2$ and $s_2 \rightarrow_2 s_3$. Also, for a relation $\rightarrow \subseteq A \times A$ on some set S we denote its reflexive transitive closure by $\rightarrow^*$. Its *inverse* is the relation $\leftarrow \subseteq A \times A$ defined by $s \leftarrow s'$ iff $s' \rightarrow s$. We write $s \rightarrow$ if $s \rightarrow s'$ for some s'.

A (partial) function f from A to B, written $f : A \rightarrow B$, is a binary relation between A and B s.t. for each $s \in A$ there is at most one $t \in B$ with $(s,t) \in f$. As usual, we will write functional relationship as $f(s) = t$. The domain $\mathsf{dom}(f)$ of a function $f : A \rightarrow B$ is the subset of A on which f is defined, i.e. $\mathsf{dom}(f) = \{s \in A \mid f(s) = t \text{ for some } t \in B\}$. The range $\mathsf{ran}(f)$ of f is the set $\{f(a) \mid a \in \mathsf{dom}(f)\}$. If A, B are sets, then $[A \rightarrow_{\mathit{fin}} B]$ denotes the set of *finite* (partial) functions from A to B, i.e. those $f : A \rightarrow B$ where $\mathsf{dom}(f)$ has finite cardinality. If $\mathsf{dom}(f) = \{a_1, \ldots, a_n\}$ then f will often be written as the set $\{a_1 \mapsto f(a_1), \ldots, a_n \mapsto f(a_n)\}$.

By $f[a \mapsto b]$ we denote adjunction, i.e. the map which coincides with f on $\mathsf{dom}(f) - \{a\}$ and maps a to b.

Uniform Confluence and Executions

The relation $\to$ on S is *confluent* if whenever $s \to^* s_1$ and $s \to^* s_2$ for some $s, s_1, s_2 \in S$, then there exists $s' \in S$ such that both $s_1 \to^* s'$ and $s_2 \to^* s'$.

Adopting the notions from [Nie00], a relation $\to$ is said to be *uniformly confluent* if $s \to s_1$ and $s \to s_2$, where $s_1 \neq s_2$, implies $s_1 \to s' \leftarrow s_2$ for some $s' \in S$. Two relations $\to_1, \to_2 \subseteq S \times S$ *commute* if whenever $s \to_1 s_1$ and $s \to_2 s_2$ there exists $s' \in S$ such that $s_1 \to_2 s'$ and $s_2 \to_1 s'$. A *partial execution* of $s \in S$ is a (finite or infinite) sequence $(s_i)_i$ such that $s = s_0$ and $s_i \to s_{i+1}$ for all i. By an *execution* we mean a maximal partial execution, i.e. a partial execution that cannot be extended. Again, this may be finite or infinite.

$$\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & \\
s_1 \arrow[dr, dashed] & \neq & s_2 \arrow[dl, dashed] \\
& s' &
\end{tikzcd}$$

Figure 3.1: Uniform confluence.

3.2 A Concurrent λ-Calculus

Before formally introducing the full language in the next section, we will first try to develop some intuition by considering a subset only. The language we consider is an extension of the simply typed λ-calculus. In addition to the usual terms, i.e. variables, abstraction and application, there is an additional construct for the spawning of new threads. The following grammar defines the syntax of expressions.

$$e ::= x \mid \lambda x.e \mid e\,e \mid \texttt{concur}\ e$$

Since we want to model a language with eager evaluation, β-reduction takes the usual form

$$(\lambda x.e)\,v \xrightarrow{\beta} e[{}^{v}\!/\!_{x}]$$

where v is a *value*, i.e. of the form

$$v ::= x \mid \lambda x.e$$

Intuitively, the meaning of `concur` e is that e is evaluated in a new thread. Therefore, we must allow for the concurrent evaluation of *several* expressions. This can be achieved by considering multisets of expressions, instead of single terms. However, `concur` e also immediately returns a *future* x associated with this thread. For this reason we will consider *configurations*, which are finite maps $C : \mathit{Var} \to \mathit{Exp}$, associating a variable (the future) with each thread (an expression). These considerations lead to the following reduction rule.

$$\texttt{concur}\ e \xrightarrow{\mathbf{C}xe} x$$

The *label* $\mathbf{C}xe$ indicates that the current configuration C is extended by the new binding $x \mapsto e$.

Finally, if the result of the evaluation of e is available, the future x should be identified with e. For reasons discussed later, this is done by a *lookup* rather than globally substituting the result for x. The corresponding rule takes the form

$$x\,v \xrightarrow{\mathbf{L}xv'} v'\,v$$

meaning that the future x can be replaced by the result of a concurrently performed computation, provided v' is in fact the value associated with this future, i.e. provided $C(x) = v'$.

To model a *deterministic* evaluation strategy, reduction is not allowed everywhere, but only at positions marked by an *evaluation context*. These contexts are defined by

$$E ::= [\,] \mid E\,e \mid v\,E$$

where the single occurrence of $[\,]$ marks the next redex, and $E[e]$ denotes the term obtained from E by replacing $[\,]$ with e. So in fact, the reduction relation on expressions will be the least relation defined by the above axioms and the inference rule

$$\frac{e \xrightarrow{\alpha} e'}{E[e] \xrightarrow{\alpha} E[e']}$$

Now these reductions on expressions take place in the context of a configuration. For example, the reduction $x\,v \xrightarrow{\mathbf{L}xv'} v'\,v$ is only admissible if $C(x) = v'$, and similarly for `concur` $e \xrightarrow{\mathbf{C}xe} x$ where the domain of C must be extended, i.e. $x \notin \mathsf{dom}(C)$. This interaction between expressions and configurations is expressed by the rule

$$\frac{e \xrightarrow{\alpha} e'}{C[x{\mapsto}e] \xrightarrow{\alpha} C[x{\mapsto}e']}$$

stating that each reduction of a configuration is caused by a reduction of a single expression. Finally, we can define the reduction relation $\rightarrow$ on configurations, which is what we are really interested in. There is a rule for each of the axioms, describing the global effect of this reduction on the configuration. For β-reduction, this is just the trivial

$$\frac{C \xrightarrow{\beta} C'}{C \rightarrow C'}$$

For spawning of new threads, the rule

$$\frac{C \xrightarrow{\mathbf{C}xe} C' \quad x \notin \mathsf{dom}(C)}{C \rightarrow C'[x{\mapsto}e]}$$

formalizes the intuitive explanation of `concur` e given above: a new thread evaluating e is introduced to the configuration, associating the future x to e. Correspondingly, lookup results in the rule

$$\frac{C \xrightarrow{\mathbf{L}xv} C' \quad C(x) = v}{C \rightarrow C'}$$

It is now easy to extend the model with *lazy* futures. We simply add new terms `byneed` e, meaning that a new future x is returned immediately, but evaluation of e is triggered only if and when the actual result of this future is needed. To facilitate this additional control on the evaluation order, we simply extend configurations by a second component containing all the expressions whose evaluation has not been triggered so far. That is, a configuration is a *pair* (C, B) of finite maps associating variables with expressions. There is a new axiom,

$$\texttt{byneed}\; e \xrightarrow{\mathbf{B}xe} x$$

and the rules are supplemented by

$$\frac{C \xrightarrow{\mathbf{B}xe} C' \quad x \notin \mathsf{dom}(C) \cup \mathsf{dom}(B)}{(C, B) \to (C', B[x \mapsto e])}$$

and

$$\frac{C \xrightarrow{\mathbf{L}xx} C'}{(C, B \uplus \{x \mapsto e\}) \to (C'[x \mapsto e], B)}$$

dealing with creation of new lazy futures and their triggering, repectively. In the latter case, observe how lookup is "abused" to force the evaluation of e. The label $\mathbf{L}xx$ expresses that the result of evaluating e is now needed, while leaving the triggering expression as is.

As final extension, promises are added by taking configurations to consist of *three* finite maps (P, C, B), where P maps a future x to its associated promise y. For the new terms $\mathtt{prom}\, e$, there are axioms for introduction and elimination of new promises. Introduction is made formal by

$$\mathtt{prom}\, e \xrightarrow{\mathbf{P}yx} e\, y\, x$$

introducing the future x and the associated promise y. Elimination is

$$y\, v \xrightarrow{\mathbf{F}yv} \mathtt{()}$$

and the two corresponding inference rules on configurations are

$$\frac{C \xrightarrow{\mathbf{P}yx} C' \quad x \neq y \text{ fresh}}{(P, C, B) \to (P[x \mapsto y], C', B)}$$

and

$$\frac{C \xrightarrow{\mathbf{F}yv} C' \quad P(x) = y \quad x \notin \mathsf{dom}(C)}{(P, C, B) \to (P, C[x \mapsto v], B)}$$

The second side-condition in the last rule, requiring $x \notin \mathsf{dom}(C)$, ensures that each promise is used at most once. We will now formally define this calculus.

3.3 The Calculus λ^{FP}

As has been stated before, the language we consider is an extension of the λ-calculus. In addition to the usual terms, we have constructs for concurrent and lazy futures and for promises.

3.3.1 Syntax

The language of types is generated by

$$\sigma \in \mathit{Ty} ::= \mathsf{unit} \mid \alpha \mid \sigma \to \sigma'$$

where we assume an infinite set TyVar of type variables, ranged over by $\alpha, \beta, \ldots$. For simplicity, the only base type considered is unit with the only element of type unit being $\mathtt{()}$. The results of this thesis will, however, easily transfer to a language enriched by other base types with their typical operations, such as booleans and integers. Also, product and sum types should not pose any new problems.

Let *Var* be an infinite set of variables and $x, y, z \in \mathit{Var}$. Then the set of expressions *Exp* is given by the grammar

$$\begin{array}{rll} e \in \mathit{Exp} ::= & \texttt{()} & \textit{(constant)} \\ \mid & x & \textit{(variable)} \\ \mid & \lambda x.e & \textit{(abstraction)} \\ \mid & e\ e & \textit{(application)} \\ \mid & \texttt{concur}\ e & \textit{(spawn)} \\ \mid & \texttt{byneed}\ e & \textit{(delayed)} \\ \mid & \texttt{prom}\ e & \textit{(promise)} \end{array}$$

The first four lines give the usual λ-expressions, while the last three define terms for futures and promises. We will sometimes use the derived syntactical forms `let` x `=` e `in` e', and sequential composition $e; e'$, defined by

$$\begin{array}{rcl} \texttt{let}\ x = e\ \texttt{in}\ e' & =_{\mathsf{df}} & (\lambda x.e')\ e \\ e; e' & =_{\mathsf{df}} & (\lambda y.e')\ e \text{ where } y \text{ is not free in } e' \end{array}$$

Similarly, we consider

$$\begin{array}{c} \texttt{concur}\ x{=}e\ \texttt{in}\ e' =_{\mathsf{df}} (\lambda x.e')\ (\texttt{concur}\ e) \\ \texttt{byneed}\ x{=}e\ \texttt{in}\ e' =_{\mathsf{df}} (\lambda x.e')\ (\texttt{byneed}\ e) \end{array}$$

and

$$\texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in}\ e =_{\mathsf{df}} \texttt{prom}\,(\lambda y.\lambda x.e)$$

as derived constructs. If the name of a variable is not important, an underscore "_" will be used. Also, we will freely use brackets whenever giving examples in concrete syntax.

3.3.2 Reduction

Next, we formally define the operational semantics of λ^{FP}. As there will be several expressions being evaluated in parallel threads in the general case, the semantics must provide some way of representing multisets of expressions. As demonstrated in the preceding section, this is achieved by considering *configurations* which are finite maps with domain the set *Var*, and codomain the set *Exp* of λ^{FP}-expressions satisfying certain additional requirements.

Constants, Values and Variables

Expressions are identified up to consistent renaming of bound variables, so we can always assume all free and bound variables to be distinct.

The set of *values* is the subset $\mathit{Val} \subseteq \mathit{Exp}$ of expressions that consists of constants, variables and abstractions, i.e. λ-terms of the following form.

$$v \in \mathit{Val} \quad ::= \quad \texttt{()} \ \mid\ x \ \mid\ \lambda x.e$$

The sets $\mathsf{fv}(e)$ and $\mathsf{bv}(e)$ of *free* resp. *bound* variables of an expression $e \in \mathit{Exp}$ are defined in the obvious way, as shown in Figure 3.2. A term e is *closed* if $\mathsf{fv}(e) = \emptyset$.

Free variables	$\mathsf{fv}(\mathtt{()}) =_{\mathsf{df}} \emptyset$
	$\mathsf{fv}(x) =_{\mathsf{df}} \{x\}$
	$\mathsf{fv}(e\ e') =_{\mathsf{df}} \mathsf{fv}(e) \cup \mathsf{fv}(e')$
	$\mathsf{fv}(\lambda x.e) =_{\mathsf{df}} \mathsf{fv}(e) - \{x\}$
	$\mathsf{fv}(\mathtt{concur}\ e) =_{\mathsf{df}} \mathsf{fv}(\mathtt{byneed}\ e) =_{\mathsf{df}} \mathsf{fv}(\mathtt{prom}\ e) =_{\mathsf{df}} \mathsf{fv}(e)$
Bound variables	$\mathsf{bv}(\mathtt{()}) =_{\mathsf{df}} \mathsf{bv}(x) =_{\mathsf{df}} \emptyset$
	$\mathsf{bv}(e\ e') =_{\mathsf{df}} \mathsf{bv}(e) \cup \mathsf{bv}(e')$
	$\mathsf{bv}(\lambda x.e) =_{\mathsf{df}} \mathsf{bv}(e) \cup \{x\}$
	$\mathsf{bv}(\mathtt{concur}\ e) =_{\mathsf{df}} \mathsf{bv}(\mathtt{byneed}\ e) =_{\mathsf{df}} \mathsf{bv}(\mathtt{prom}\ e) =_{\mathsf{df}} \mathsf{bv}(e)$

Figure 3.2: Free and bound variables

Configurations

Expressions of a configuration can be distinguished as being either an ordinary computation, a still suspended *byneed* computation, or an unbound future variable that may become bound by *fulfilling* the associated promise, i.e. applying it to a value.

For a triple (P, C, B) of finite maps

$$P \in [\mathit{Var} \to_{\mathit{fin}} \mathit{Var}]$$
$$C, B \in [\mathit{Var} \to_{\mathit{fin}} \mathit{Exp}]$$

we lift the notion of free and bound variables as follows. The set $\mathsf{var}(P, C, B)$ of *variables* of the triple is defined as

$$\mathsf{var}(P, C, B) =_{\mathsf{df}} \mathsf{dom}(P) \cup \mathsf{ran}(P) \cup \mathsf{dom}(C) \cup \mathsf{dom}(B)$$

Next, the set $\mathsf{bv}(P, C, B)$ of its *bound variables* is

$$\mathsf{bv}(P, C, B) =_{\mathsf{df}} \mathsf{var}(P, C, B) \cup \bigcup \{\mathsf{bv}(e) \mid e \in \mathsf{ran}(C) \text{ or } e \in \mathsf{ran}(B)\}$$

Correspondingly, the set $\mathsf{fv}(P, C, B)$ of its *free variables* is defined by

$$\mathsf{fv}(P, C, B) =_{\mathsf{df}} \bigcup \{\mathsf{fv}(e) \mid e \in \mathsf{ran}(C) \text{ or } e \in \mathsf{ran}(B)\} - \mathsf{var}(P, C, B)$$

Thus, variables x, where $x \in \mathsf{var}(P, C, B)$, are bound in the triple (P, C, B). We can now define a *configuration* to be a triple (P, C, B) of functions subject to the following conditions.

Definition 3.3.1. *A triple (P, C, B), where $P \in [\mathit{Var} \to_{\mathit{fin}} \mathit{Var}]$ and $C, B \in [\mathit{Var} \to_{\mathit{fin}} \mathit{Exp}]$ is a* configuration *if the conditions*

(C1) $\mathsf{ran}(P)$, $\mathsf{dom}(C)$ *and* $\mathsf{dom}(B)$ *are pairwise disjoint finite sets of variables*

(C2) P is injective, and $\mathsf{dom}(P) \cap \mathsf{ran}(P) = \mathsf{dom}(B) \cap \mathsf{dom}(P) = \emptyset$

are satisfied.

However, note that we do not require $\mathsf{dom}(P)$ and $\mathsf{dom}(C)$ to be disjoint. Also note that $P(x) = y$ will mean that x is the future associated with the promise y, i.e. x can be bound by applying y. Due to injectivity of P, this choice is purely arbitrary, and one could well use P^{-1} instead. Only the expressions "in C" allow for computations to take place, in the sense made precise by the reduction rules below.

The set of all configurations will be denoted by *Config*. As with expressions, we identify configurations up to consistent renaming of bound variables. Also, we will in the following assume that configurations are *α-standardized*, i.e. all bound variables are different and also differ from all free variables.

Evaluation Contexts and Substitution

There are reduction rules for β-reduction, introduction and elimination of promises and futures, and finally rules dealing with forcing by-need threads. In order to model a standard call-by-value, left-to-right reduction of (sequential) expressions we make use of *evaluation contexts* as used in [WF94]. Such a context E can be seen as a term containing a single occurrence of the distinguished variable $[\,]$. The set of evaluation contexts E is defined by

$$E \quad ::= \quad [\,] \mid E\,e \mid v\,E$$

where e ranges over expressions and v over values. By $E[e]$ we will denote substitution of e for $[\,]$ in E. Note that this is necessarily capture-free, as there are no binding constructs in evaluation contexts. The position of $[\,]$ marks the position of the next redex. For example, consider the term

$$(\lambda x.e)\ ((\texttt{concur}\ e_1)\ e_2) \equiv (\lambda x.e)\ ([\,]\ e_2)[\texttt{concur}\ e_1]$$

which says the next redex is $\texttt{concur}\ e_1$, and neither of e and e_2 may be reduced next.

Generally, substitution $[{}^{v}\!/\!{}_{x}]$ applied to a term e is defined straightforwardly, assuming in particular that all the free variables of v, as well as x, are distinct from the bound variables in e:

$$\begin{aligned}
\texttt{()}\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} \texttt{()}\\
x\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} v\\
y\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} y \ \text{ if } y \neq x\\
(\lambda y.e)\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} \lambda y.e\,[{}^{v}\!/\!{}_{x}]\\
(e\ e')\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} (e\,[{}^{v}\!/\!{}_{x}])\ (e'\,[{}^{v}\!/\!{}_{x}])\\
(\texttt{concur}\ e)\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} \texttt{concur}\ (e\,[{}^{v}\!/\!{}_{x}])\\
(\texttt{byneed}\ e)\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} \texttt{byneed}\ (e\,[{}^{v}\!/\!{}_{x}])\\
(\texttt{prom}\ e)\,[{}^{v}\!/\!{}_{x}] &=_{\mathsf{df}} \texttt{prom}\ (e\,[{}^{v}\!/\!{}_{x}])
\end{aligned}$$

Reduction on Expressions

Borrowing terminology from process calculi, reduction takes the form of labelled transition $\xrightarrow{\alpha}$ between expressions. For $e, e' \in \mathit{Exp}$, the statement $e \xrightarrow{\alpha} e'$ means that e evaluates in one step to e', thereby possibly affecting the configuration via the *action*, or *side-effect*, α.

First, labels α are

$$\alpha \in \mathit{Lab} \quad ::= \quad \beta \mid \mathbf{P}yx \mid \mathbf{F}ye \mid \mathbf{C}xe \mid \mathbf{B}xe \mid \mathbf{L}xe$$

where $x, y \in \mathit{Var}$ and $e \in \mathit{Exp}$. The reduction relation $\xrightarrow{\alpha}$ is now defined by the following axioms. β-reduction is

$$(\beta) \qquad E[(\lambda x.e)\ v] \xrightarrow{\beta} E[e[{}^{v}\!/x]]$$

just as in call-by-value λ-calculus.Note that our choice of evaluation contexts E in fact leads to call-by-value standard reduction [Plo75].
The rule for introduction of promises is

$$(\mathit{prom}) \qquad E[\texttt{prom}\ e] \xrightarrow{\mathbf{P}yx} E[e\,y\,x]$$

That is, the expression $\texttt{prom}\ e$ reduces in one step to $e\,y\,x$. A promise y for a future x means that $x{\mapsto}y$ is in P. The variable x can be bound to value v via $y\ v$, and elimination of promises takes the form

$$(\mathit{fulfill}) \qquad E[y\ v] \xrightarrow{\mathbf{F}yv} E[\texttt{()}]$$

New (lazy) futures are introduced by the transitions

$$(\mathit{concur}) \qquad E[\texttt{concur}\ e] \xrightarrow{\mathbf{C}xe} E[x]$$

and

$$(\mathit{byneed}) \qquad E[\texttt{byneed}\ e] \xrightarrow{\mathbf{B}xe} E[x]$$

Next, there is a rule to handle the lookup of values. If a future x comes into function position and its associated value is already available, this occurrence of x is replaced.

$$(\mathit{lookup}) \qquad E[x\ v] \xrightarrow{\mathbf{L}xv'} E[v'\ v]$$

Finally, if a lazy future comes into function position, evaluation of the associated expression is triggered. This will be expressed by a lookup, labelled $\mathbf{L}xx$, as can be seen below.

Action and Reduction on Configurations

We shall now define how the actions change a configuration, actually performing the side-effect that a reduction step $e \xrightarrow{\alpha} e'$ causes. The interplay between reduction on single expressions and their impact on a configuration is stated by the inference rule

$$(\mathit{select}) \qquad \frac{e \xrightarrow{\alpha} e'}{C[x{\mapsto}e] \xrightarrow{\langle\alpha,x\rangle} C[x{\mapsto}e']}$$

This rule, or rather the absence of a similar rule for B, also shows that only expressions of the "C"-component may cause reductions.

Finally, we can define the relation $\rightarrow$ on configurations. There is one rule for each of the actions α. In fact, there are two rules for $\mathbf{L}xe$, corresponding to the fact that forcing is modelled by lookup. As in the case of expressions, reduction takes the form of labelled transition $\xrightarrow{\langle\alpha,x\rangle}$, where $\alpha \in \mathit{Lab}$ and $x \in \mathit{Var}$. The variable x indicates which thread caused the transition.

$$(\beta) \qquad \frac{C \xrightarrow{\langle\beta,z\rangle} C'}{(P,C,B) \xrightarrow{\langle\beta,z\rangle} (P,C',B)}$$

$$(\mathit{Prom}) \qquad \frac{C \xrightarrow{\langle\mathbf{P}yx,z\rangle} C' \quad x \neq y \quad x,y \notin \mathsf{var}(P,C,B)}{(P,C,B) \xrightarrow{\langle\mathbf{P}yx,z\rangle} (P[x\mapsto y],C',B)}$$

$$(\mathit{Fulfill}) \qquad \frac{C \xrightarrow{\langle\mathbf{F}yv,z\rangle} C' \quad P(x) = y \quad x \notin \mathsf{dom}(C)}{(P,C,B) \xrightarrow{\langle\mathbf{F}yv,z\rangle} (P,C'[x\mapsto v],B)}$$

$$(\mathit{Concur}) \qquad \frac{C \xrightarrow{\langle\mathbf{C}xe,z\rangle} C' \quad x \notin \mathsf{var}(P,C,B)}{(P,C,B) \xrightarrow{\langle\mathbf{C}xe,z\rangle} (P,C'[x\mapsto e],B)}$$

$$(\mathit{Byneed}) \qquad \frac{C \xrightarrow{\langle\mathbf{B}xe,z\rangle} C' \quad x \notin \mathsf{var}(P,C,B)}{(P,C,B) \xrightarrow{\langle\mathbf{B}xe,z\rangle} (P,C',B[x\mapsto e])}$$

$$(\mathit{Lookup}) \qquad \frac{C \xrightarrow{\langle\mathbf{L}xv,z\rangle} C' \quad C(x) = v}{(P,C,B) \xrightarrow{\langle\mathbf{L}xv,z\rangle} (P,C',B)}$$

$$(\mathit{Force}) \qquad \frac{C \xrightarrow{\langle\mathbf{L}xx,z\rangle} C}{(P,C,B \uplus \{x\mapsto e\}) \xrightarrow{\langle\mathbf{L}xx,z\rangle} (P,C[x\mapsto e],B)}$$

Note that by condition *(C1)* in Definition 3.3.1, at most one of the last two rules is applicable with respect to x. The *reduction relation* $\rightarrow$ on the set of configurations is simply the union

$$\rightarrow \;=_{\mathsf{df}} \bigcup\{\xrightarrow{\langle\alpha,x\rangle} \;\mid \alpha \in \mathit{Lab} \text{ and } x \in \mathit{Var}\}$$

We just state the following easily verified observations, concerning *well-definedness* of reduction and preservation of *closedness.*

Observation 3.3.2. *If $(P,C,B) \in \mathit{Config}$ and $(P,C,B) \rightarrow (P',C',B')$, then $(P',C',B') \in \mathit{Config}$, i.e. $\rightarrow$ preserves the properties of the maps required by conditions (C1) and (C2) of Definition 3.3.1.*

Observation 3.3.3. *If $\mathsf{fv}(P,C,B) = \emptyset$ and $(P,C,B) \rightarrow (P',C',B')$, then $\mathsf{fv}(P',C',B') = \emptyset$.*

Note that, rather than globally substituting v for x when fulfilling the promise y, we add the binding $x\mapsto v$ to the environment and perform a lookup whenever the value v is needed. This is necessary because it is perfectly possible for configurations to become cyclic, in that v itself contains a free occurrence of x. This gives a first hint at the expressive power of promises and we will get back to this later in Section 3.6. However, promises are the only reason for cycles, as will be stated in Lemma 4.3.2 in the next chapter. Syntax and semantics of λ^{FP}-calculus are summarized at the end of this chapter, in Figure 3.5 on page 34.

Remark. Transitions via $\xrightarrow{\mathbf{P}xy}$, $\xrightarrow{\mathbf{C}xe}$ and $\xrightarrow{\mathbf{B}xe}$ are very similar to *scope extrusion* in π-calculus, e.g. in the reduction

$$(\nu x\, \bar{y}\langle x\rangle.P) \mid y(z).Q \rightarrow \nu x(P \mid Q\{{}^{x}\!/\!{}_{z}\})$$

Here, the scope of the (local) variable x is dynamically extended to also include the process Q. A configuration reminds of canonical forms of π-calculus processes, where the scope of top-level local names is global, i.e. π-processes of the form $(\nu x_1 \ldots x_k)\ (P_1 \mid \cdots \mid P_n)$.

Initial Configuration

Reduction works on configurations. If we want to talk about reduction of terms it is thus necessary to embed the term into a configuration to have something to start with. Given $e \in \mathit{Exp}$, an *initial configuration* of e is

$$(\emptyset, \{x_0 \mapsto e\}, \emptyset)$$

assuming $x_0 \in \mathit{Var}$ is not free in e. Clearly this is a configuration, and if e is closed then so is $(\emptyset, \{x_0 \mapsto e\}, \emptyset)$. Note that by identifying configurations up to α-equivalence, all the initial configurations for e are in fact equal.

If no confusion is likely, we will just write e for the initial configuration of e.

Derived Configuration Semantics

By the rules of labelled transition and its action on configurations, one can derive an equivalent, more direct reduction semantics, consisting of just one level. However, the following demonstrates that the resulting rules are hard to read and understand, since usually only a small part of the configuration is involved in a single reduction step. The notation obscures which parts are really relevant. For example, β-reduction becomes, when lifted to configurations,

$$(\beta') \qquad (P, C \uplus \{y \mapsto E[(\lambda x.e)\ v]\}, B) \rightarrow (P, C \uplus \{y \mapsto E[e[{}^{v}\!/\!{}_{x}]]\}, B)$$

Likewise, (*Concur*) can be rewritten as

$$(\mathit{Concur}') \qquad (P, C \uplus \{y \mapsto E[\mathtt{concur}\ e]\}, B) \rightarrow (P, C \uplus \{y \mapsto E[x], x \mapsto e\}, B)$$

where x is fresh. Deriving the remaining rules is as simple, and not done here as we will not work with this semantics.

Cells

Above, we introduced the language λ^{FP}, which is based on the call-by-value λ-calculus with `concur`- and `byneed`-annotations. Also, it contains assignable logic variables. However, our aim is to provide a computation model for concurrent computation of Alice [Ali02]. For this, λ^{FP} is not sufficient and we will add cells to the language in Section 5.4, obtaining the language λ^{FPC}.

$$\begin{tikzcd}
 & x{\mapsto}\mathtt{concur}\ 2{\cdot}3 + \mathtt{concur}\ 2{\cdot}2 + 2{\cdot}1 \arrow[d] & \\
 & x_3{\mapsto}2{\cdot}3\ ,\ \ x{\mapsto}x_3 + \mathtt{concur}\ 2{\cdot}2 + 2\cdot 1 \arrow[dl] \arrow[ddr] & \\
x_3{\mapsto}6\ ,\ \ x{\mapsto}x_3 + \mathtt{concur}\ 2\cdot 2 + 2\cdot 1 \arrow[dd] \arrow[dddr, bend left] & & \\
 & & x_3{\mapsto}2{\cdot}3\ ,\ \ x_2{\mapsto}2{\cdot}2\ ,\ \ x{\mapsto}x_3 + x_2 + 2{\cdot}1 \arrow[ddl] \arrow[ddd, bend left] \\
x_3{\mapsto}6\ ,\ \ x{\mapsto}6 + \mathtt{concur}\ 2{\cdot}2 + 2{\cdot}1 \arrow[ddd, bend right] & & \\
 & x_3{\mapsto}6\ ,\ \ x_2{\mapsto}2{\cdot}2\ ,\ \ x{\mapsto}x_3 + x_2 + 2{\cdot}1 \arrow[ddl] & \\
 & & x_3{\mapsto}2{\cdot}3\ ,\ \ x_2{\mapsto}4\ ,\ \ x{\mapsto}x_3 + x_2 + 2{\cdot}1 \arrow[ddl, "*"'] \\
x_3{\mapsto}6\ ,\ \ x_2{\mapsto}2{\cdot}2\ ,\ \ x{\mapsto}6 + x_2 + 2{\cdot}1 \arrow[dr, "*"] & & \\
 & x_3{\mapsto}6\ ,\ \ x_2{\mapsto}4\ ,\ \ x{\mapsto}12 &
\end{tikzcd}$$

Figure 3.3: Initial part of a derivation dag

3.4 An Example

To demonstrate (concurrent) evaluation in λ^{FP} to some extent, in this sectionwe will look at an example. This is in no way a practical application of futures and concurrency. However, it allows us to see how our formalizations work. To make the example slightly more interesting, we will use a natural number type `nat`, assuming natural number constants and arithmetic operations.

So suppose we want to compute the summation $\sum_{x=1}^{n} f(x)$, for a function f. Using futures, it is then possible to evaluate several of the $f(x)$ concurrently. In particular, we could write

$$e \equiv \mathtt{concur}\ (2{\cdot}3) + \mathtt{concur}\ (2{\cdot}2) + 2{\cdot}1 \tag{3.1}$$

Already (3.1) shows that there are quite a few interleaved computation sequences possible, thus relaxing deterministic reduction order considerably.

The first step is by rule *(concur)*, and we obtain from the initial configuration e the configuration

$$(\emptyset, \{x_3{\mapsto}2{\cdot}3, x{\mapsto}x_3 + \mathtt{concur}\ 2{\cdot}2 + 2{\cdot}1\}, \emptyset) \tag{3.2}$$

for some fresh variable x_3. Now configuration (3.2) has *two distinct* transitions. First, thread x_3 may compute the value of $2{\cdot}3$, and second, thread x may reduce the redex $\mathtt{concur}\ 2{\cdot}2 + 2{\cdot}1$ by *(concur)*. Figure 3.3 depicts the initial part of the complete derivation graph of e. It is not

$$\frac{}{\Gamma \vdash \texttt{()}{:}\mathsf{unit}}\ (\text{unit}) \qquad \frac{}{\Gamma, x{:}\sigma \vdash x{:}\sigma}\ (\text{var})$$

$$\frac{\Gamma \vdash e : \sigma'{\to}\sigma \quad \Gamma \vdash e'{:}\sigma'}{\Gamma \vdash e\ e' : \sigma}\ (\text{appl}) \qquad \frac{\Gamma, x{:}\sigma \vdash e{:}\sigma'}{\Gamma \vdash \lambda x.e : \sigma{\to}\sigma'}\ (\text{abs})$$

$$\frac{\Gamma \vdash e{:}\sigma}{\Gamma \vdash \texttt{concur}\ e : \sigma}\ (\text{conc}) \qquad \frac{\Gamma \vdash e{:}\sigma}{\Gamma \vdash \texttt{byneed}\ e : \sigma}\ (\text{need})$$

$$\frac{\Gamma \vdash e{:}(\sigma'{\to}\mathsf{unit}){\to}\sigma' \to \sigma}{\Gamma \vdash \texttt{prom}\ e : \sigma}\ (\text{prom})$$

Figure 3.4: Type inference rules for the simply typed λ^{FP}-calculus

difficult to find a path in this graph leading to the expected result

$$(\emptyset, \{x_3{\mapsto}6, x_2{\mapsto}4, x{\mapsto}12\}, \emptyset) \tag{3.3}$$

In fact, from the results of sections 4.2 and 4.3 it follows that both the derivation graph is finite and has the configuration (3.3) as only node of out-degree 0.

3.5 Simple Types for λ^{FP}

In this section, a type system for λ^{FP} is presented. It is a straightforward extension of the type rules for the simply typed λ-calculus, which is then lifted to configurations. Also, it is not hard to prove a subject reduction theorem, stating that typing of configurations is invariant under reduction. However, simple examples show that the single assignment property of promises is in no way enforced in λ^{FP}. Therefore a much more elaborated type system is developed in Chapter 5.

The type rules are listed in Figure 3.4. Here, Γ ranges over *type environments* $x_1{:}\sigma_1 \ldots x_n{:}\sigma_n$, i.e. functional relations between *Var* and *Ty*. The typing judgment $\Gamma \vdash e{:}\sigma$ means that the expression e can be assigned type σ, given type assumptions Γ. Note that by identifying terms up to α-equivalence we can always assume all x_i to be distinct when deriving a type for expression e. In particular, we write $\Gamma, x{:}\sigma$ for $\Gamma \uplus \{x{:}\sigma\}$. We will sometimes write $\Gamma(x) = \sigma$ if $x{:}\sigma \in \Gamma$.

The rules for constants, variables, abstraction and application are well-known from the simply-typed λ-calculus. The rules (conc) and (need) should be obvious. The rule (prom) takes care that promise y and associated future x are consistently used in e, by requiring that the respective types match as $y : \sigma'{\to}\mathsf{unit}$ and $x{:}\sigma'$ in the reduct $e\,y\,x$.

Clearly, for types to be "useful" they should be compatible with reduction. In order to prove a subject reduction theorem for our language, we need some preliminary lemmas, mainly stating properties of substitution.

Lemma 3.5.1. *Let e, e' be arbitrary terms. If $\Gamma, x{:}\sigma' \vdash e{:}\sigma$ and $\Gamma \vdash e'{:}\sigma'$, then $\Gamma \vdash e[^{e'}/_{x}]{:}\sigma$.*

Proof. By induction on the structure of e. This is standard. □

Lemma 3.5.2. *Suppose $\Gamma \vdash E[e]{:}\sigma$ for a context E and term e. Then, for some σ', $\Gamma \vdash e{:}\sigma'$. Moreover, if $\Gamma \vdash e'{:}\sigma'$ then also $\Gamma \vdash E[e']{:}\sigma$.*

Proof. By an easy induction on the structure of the context E. □

Next, we lift typings to configurations, requiring that the types of expressions are globally consistent, in the sense made precise by the following definition.

Definition 3.5.3. *A type environment Γ is a* typing *for the configuration (P, C, B), written $\Gamma \vdash (P, C, B)$, if $\mathsf{var}(P, C, B) \subseteq \mathsf{dom}(\Gamma)$ and*

- *for all $x \in \mathsf{dom}(C)$, $\Gamma(x) = \sigma$ only if $\Gamma \vdash C(x){:}\sigma$,*
- *for all $x \in \mathsf{dom}(B)$, $\Gamma(x) = \sigma$ only if $\Gamma \vdash B(x){:}\sigma$, and*
- *for all $x \in \mathsf{dom}(P)$, both $\Gamma(x) = \sigma$ and $\Gamma(P(x)) = \sigma \to \mathsf{unit}$, for some $\sigma \in Ty$.*

Subject reduction can now be stated as

Lemma 3.5.4 (Subject Reduction). *Suppose $\Gamma \vdash (P, C, B)$, and $(P, C, B) \to (P', C', B')$. Then $\Gamma' \vdash (P', C', B')$ for some $\Gamma' \supseteq \Gamma$.*

Thus, if $\Gamma \vdash C(x){:}\sigma$ then also $\Gamma' \vdash C'(x){:}\sigma$.

Proof. Consider cases for $\to$ and extend Γ to Γ' satisfying $\Gamma' \vdash (P', C', B')$. Lemma 3.5.2 allows to deal with evaluation contexts. For the case $\xrightarrow{\langle\beta,x\rangle}$ by rule (β) we can apply Lemma 3.5.1. □

From this we immediately obtain a subject reduction theorem of simply typed *expressions.*

Proposition 3.5.5. *If $\vdash e{:}\sigma$ for closed $e \in Exp$, $\sigma \in Ty$ and if $(\emptyset, \{x_0 \mapsto e\}, \emptyset) \to^* (P, C[x_0 {\mapsto} v], B)$, then $\Gamma \vdash v{:}\sigma$ for some environment Γ.*

Proof. Let $\Gamma_0 =_{\mathsf{df}} x_0{:}\sigma$. Then $\Gamma_0 \vdash (\emptyset, \{x_0 \mapsto e\}, \emptyset)$, and induction on the number of reduction steps gives Γ such that $\Gamma \vdash (P, C[x_0 {\mapsto} v], B)$ and $\Gamma(x_0) = \Gamma_0(x_0) = \sigma$. This implies $\Gamma \vdash v{:}\sigma$, by definition of typings. □

We conclude this section with two remarks.

Unique types. For the simply typed language, we could have used terms with type annotations, e.g. writing $\lambda x{:}\sigma.e$ and `prom` y `for` $x{:}\sigma$ `in` e, taking the latter as short hand for `prom` $(\lambda y{:}\sigma{\to}\mathsf{unit}.\lambda x{:}\sigma.e)$. Then it is easy to verify that expressions possess at most one type, in the sense that whenever $\Gamma \vdash e{:}\sigma$ and $\Gamma \vdash e{:}\sigma'$ this implies $\sigma = \sigma'$. However, in general this is not the case for typings of configurations, and this property is not even preserved by reduction. To see this, consider the (closed) expression `prom` y `for` $x{:}\mathsf{unit}$ `in` $y\ x;\ x$ which has the unique type unit, and observe that

$$\texttt{prom}\ y\ \texttt{for}\ x{:}\mathsf{unit}\ \texttt{in}\ y\ x;\ x \xrightarrow{\langle\beta,x_0\rangle}\xrightarrow{\langle\beta,x_0\rangle}\xrightarrow{\langle\mathbf{P}yx,x_0\rangle}\xrightarrow{\langle\mathbf{F}yx,x_0\rangle}\xrightarrow{\langle\beta,x_0\rangle} (\{x{\mapsto}y\}, \{x_0{\mapsto}x, x{\mapsto}x\}, \emptyset)$$

by applying reduction rules *(Prom)*, *(Fulfill)* and *(β)* successively. For the resulting configuration, $\Gamma_\sigma =_{\mathsf{df}} x_0{:}\sigma, x{:}\sigma, y{:}\sigma{\to}\mathsf{unit}$ is a valid typing, for *any* type σ.

Type Soundness. The above subject reduction theorem is not sufficient to obtain what is often called *type soundness*, stating that "well-typed programs do not go wrong," meaning that no runtime type errors occur during reduction of well-typed expressions. In fact, the type system guarantees that functions and operators are applied to the right kind of arguments only, nevertheless, well-typed programs may get *stuck*. This problem has to do with promises, as multiple assignments to a future via its associated promise are not ruled out. More concretely, consider the following

$$\texttt{prom}\, y\, \texttt{for}\, x\, \texttt{in}\, y\ \texttt{()};\ y\ \texttt{()}$$

which gets stuck when trying to perform the second of the two assignments y `()`. A stronger type system taking care of the problem will be presented in Chapter 5.

3.6 Computational Completeness and Recursion

We conclude this chapter by looking at the expressivity of λ^{FP}. We begin with the rather obvious result that in the *untyped* calculus all computable functions are definable. Indeed, immediately from the definition of $\rightarrow$ on configurations, we obtain the following proposition with respect to call-by-value reduction $\rightarrow_\beta$ [Plo75] on λ-calculus terms M.

Proposition 3.6.1. *For all M, $M \rightarrow_\beta M'$ if and only if $M \rightarrow M'$ by (β).*

Let us now take a look at the additional expressive power that promises give in the context of recursive function definitions. We will show that we can define a recursion operator in the simply typed language. This is in contrast to the simply typed λ-calculus, where a well-known result states that all definable functions are terminating (see, e.g., [Tho92]).

So why do promises provide a way to define recursive functions? Let us consider the following example. Suppose we want to define the factorial function $\mathsf{fac} : \mathbb{N} \rightarrow \mathbb{N}$ in our language. We assume that the language has been extended by base types B of booleans and N of natural number constants, together with sufficiently many operations on them.

The obvious recursive definition of fac in a language with explicit recursion is of course the following

$$\mathit{fac}\, n =_{\mathsf{df}} \texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * (\mathit{fac}(n-1)) \tag{3.4}$$

To express this in λ^{FP}, promises come into play: They allow to recursively refer to *fac*, without *fac* being defined. Replacing the function name by a variable f and introducing a promise, (3.4) gives

$$\texttt{prom}\, p\, \texttt{for}\, f\, \texttt{in}\, \lambda n.\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * f(n-1)$$

Now, after the definition of *fac* with respect to f has been given, the promise can be fulfilled, binding f to *fac* itself. This gives exactly the required recursive definition. Also, it is not hard to check that this term indeed is well-typed, of type $\mathsf{N}\rightarrow\mathsf{N}$. The complete function definition then might look as follows.

```
prom p for f:N → N
in
   let fac = λn.if n = 0 then 1 else n * f(n − 1);
       _ = p fac
   in fac
```

Having seen this example, the approach can be generalized straightforwardly, leading to the definition of a fixed point operator **fix**, for any given type σ.

$$\begin{array}{ll}\mathbf{fix} =_{\mathsf{df}} & \lambda x.\ \texttt{prom}\ p\ \texttt{for}\ f \\ & \quad\ \texttt{in} \\ & \qquad p\ (x\ f)\texttt{;}\ f\end{array}$$

Typability and fixed point property are stated in the following lemmas.

Lemma 3.6.2. *Let $\sigma \in Ty$. Then* **fix** *is typable. More precisely,*

$$\vdash \mathbf{fix}{:}((\sigma{\to}\sigma) \to \sigma{\to}\sigma) \to (\sigma{\to}\sigma)$$

Lemma 3.6.3. *Suppose $\lambda g.\lambda z.e{:}(\sigma{\to}\sigma) \to \sigma{\to}\sigma$. Then*

$$(P, C[y{\mapsto}E[\mathbf{fix}\ \lambda g.\lambda z.e]], B) \to^{*} (P[f{\mapsto}p], C[f{\mapsto}\lambda z.e[{}^{f}\!/_{g}], y{\mapsto}E[f]], B)$$

where f may occur free in $e[{}^{f}\!/_{g}]$.

Proof. The proof can be done by simply constructing the reduction sequence. The lemma is stated only for terms of the form $\lambda g.\lambda z.e$ in order to guarantee termination of the evaluation of the expression "$x\,f$". $\square$

3.7 Discussion

3.7.1 Summary

In this chapter we introduced an abstract concurrent programming language based on λ-calculus. It contains futures and promises, and is intended to serve as a computation model for the Alice programming language.

Futures allow reading, but not writing. Promises allow to write a future exactly once. Any further attempt to bind the future will result in blocking the respective thread. Thus, this form of logic variables are a resource, and as such a source of indeterminism. In Chapter 5 this will be examined more closely. Associated with each concurrent thread and each suspended computation is a logic variable. However, this thread is the only writer on its respective variable, so the above mentioned problem of multiple attempts to bind the future does not occur.

We then presented a type system and, as a first technical result, proved a subject reduction theorem. Finally, expressiveness of the language was briefly considered. It turned out that recursion is expressible even in the simply typed case. In the next chapter, we will turn our attention to the sublanguage of promise-free expressions.

3.7.2 Variations

One could envisage many different formalizations of the concepts under consideration. Following, some possibilities are discussed.

Threads. Concurrency and logic variables might be treated as orthogonal concepts. More specifically, we could have introduced the new expression `thread` e, along with the axiom

$$\textit{(thread)} \qquad E[\texttt{thread}\ e] \xrightarrow{\mathbf{C}xe} E[\texttt{()}]$$

That is, `thread` e spawns a new thread evaluating e, but does not return a result. It is easily seen that `concur` e becomes definable in terms of `thread` and `prom`, e.g. by writing the result on a future as side-effect.

$$\texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in (thread}\ y\ e\texttt{);}\ x$$

However, it does not seem possible to treat evaluation on demand as simple in this framework.

Term Constants. It would be possible to introduce the syntax of the language as an extension of λ-calculus by viewing `concur`, `byneed` and `prom` as constants. More precisely, we could consider the language as λ-calculus over the signature containing typed term constants

$$\texttt{concur}_\sigma : \sigma{\to}\sigma$$
$$\texttt{byneed}_\sigma : \sigma{\to}\sigma$$

and

$$\texttt{prom}_{\sigma,\sigma'} : ((\sigma{\to}\mathsf{unit}){\to}\sigma{\to}\sigma') \to \sigma'$$

for each type σ, σ', as done in [Mit96] to obtain a λ-calculus over a specific algebraic structure. One would then add directed “equations”to specify the reduction behaviour of these constants. These are exactly the axioms for labelled reduction we used:

$$\texttt{concur}\ e \overset{\mathbf{C}xe}{=} x$$
$$\texttt{byneed}\ e \overset{\mathbf{B}xe}{=} x$$

and

$$\texttt{prom}_{\sigma,\sigma'}\ e \overset{\mathbf{P}yx}{=} e\,y\,x$$

Such a formalisation would have the conceptual advantage that it is not necessary to define the syntax of the language explicitly, but just give the signature and axioms. However, there are two disadvantages related to this approach. First, *syntactical sugar* cannot be defined as easily since the constants depend on types. For example, we cannot simply define

$$\texttt{prom}\ y\ \texttt{for}\ x{:}\sigma\ \texttt{in}\ e =_{\mathsf{df}} \texttt{prom}_{\sigma,?}(\lambda y.\lambda x.e)$$

independently of a type given to e. The second shortcoming is more severe and is connected to the evaluation strategy: considering the constants as values,

$$\texttt{concur}_\sigma\ E$$

would become a valid evaluation context, contrary to our intention that `concur` e is a non-strict application, i.e. that e is not evaluated before the (top-level) reduction of `concur` e has happened. The easy way out, considering the constants as non-values, would lead to an even more questionable behaviour. For example, $(\lambda x.x)$ $\texttt{concur}_\sigma$ would not allow for any reductions.

Lookup Operator The language could be extended by an explicit lookup operator "`?`", implementing "force and wait" (the `await` operator of Alice). For this, the set of expressions contains new terms

$$e ::= \ldots \mid \texttt{?}e$$

and the set of evaluation context becomes

$$E ::= \ldots \mid \texttt{?}E$$

Further, we would add axioms

$$E[\texttt{?}x] \xrightarrow{\mathbf{L}xv} E[\texttt{?}v]$$

for a variable x, and

$$E[\texttt{?}v] \xrightarrow{\varepsilon} E[v]$$

provided $v \notin \mathit{Var}$. Here, ε is the *silent* action, causing no change to the configuration. In the first case the "`?`" is kept because it may be necessary to follow a chain of futures before obtaining the actual value these futures represent. In fact, this process might not even terminate, e.g. if $C(x) = x_1, C(x_1) = x_2, \ldots, C(x_{n-1}) = x_n$ and $C(x_n) = x$. One may guarantee that lookup is well-founded by insisting that $\xrightarrow{\mathbf{F}yx_1}$ is defined only if no *direct* cycles are created, i.e. if not $C(x_1) = x_2, \ldots, C(x_{n-1}) = x_n$ where $P(x_n) = y$. In fact, this corresponds to the specification of promises in the Alice programming language, where an exception is raised in this case.

Having terms for explicit lookup has the advantage that extensions of the language become simpler. For example, when adding integers, because of the lookup operator it suffices to add the new axiom

$$+\, n_1\, n_2 \xrightarrow{\varepsilon} n$$

where $n = n_1 + n_2$. To express addition, one would then write $+\ \texttt{?}e_1\ \texttt{?}e_2$ which takes care that futures are replaced by actual values before performing the addition. Without "`?`" this can only be achieved by explicitly adding the lookup rules

$$E[+\, x\, e] \xrightarrow{\mathbf{L}xv} E[+\, v\, e]$$

and

$$E[+\, n\, x] \xrightarrow{\mathbf{L}xv} E[+\, n\, v]$$

On the other hand, instead of writing function application, it becomes of course necessary to explicitly write $\texttt{?}e_1\ e_2$ to obtain the behaviour of the application $e_1\ e_2$ in λ^{FP}.

Syntactical configurations. Also, it could be nice to consider a purely syntactical calculus "without" the additional concept of configurations, some of the basic rules of which would be

$$\texttt{concur}\ x{=}v\ \texttt{in}\ E[x\ v'] \to \texttt{concur}\ x{=}v\ \texttt{in}\ E[v\ v']$$

and

$$\texttt{byneed}\ x{=}e\ \texttt{in}\ E[x\ v'] \rightarrow \texttt{concur}\ x{=}e\ \texttt{in}\ E[x\ v']$$

In this way, there is no distinction between *expressions* and *configurations.* Concurrent evaluation could be modelled here by extending the set of evaluation contexts E, including e.g. both `concur` $x{=}E$ `in` e and `concur` $x{=}e$ `in` E. In fact, for this to work one then needs context rules such as

$$\texttt{concur}\ x{=}\texttt{concur}\ y{=}e_1\ \texttt{in}\ e_2\ \texttt{in}\ e_3 \rightarrow \texttt{concur}\ y{=}e_1\ \texttt{in}\ \texttt{concur}\ x{=}e_2\ \texttt{in}\ e_3$$

rearranging evaluation contexts. Note the similarities to, e.g., the call-by-let λ-calculus of Maraist et al. in [MOTW95] and the various call-by-need λ-calculi proposed in the literature. However, a problem arises out of the possible interdependence of threads which is introduced by promises. A simple example like

$$\texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in}\ \texttt{concur}\ z_1{=}x\ \texttt{in}\ \texttt{concur}\ z_2{=}z_1\ \texttt{in}\ y\ z_2$$

which reduces to the cyclic

$$\texttt{concur}\ z_1{=}z_2\ \texttt{in}\ \texttt{concur}\ z_2{=}z_1\ \texttt{in}\ \texttt{()}$$

suggests that in order to be complete with respect to the “intuitive” behaviour one needs further context rules such as

$$\texttt{concur}\ x{=}e_1\ \texttt{in}\ \texttt{concur}\ y{=}e_2\ \texttt{in}\ e_3 \rightarrow \texttt{concur}\ y{=}e_2\ \texttt{in}\ \texttt{concur}\ x{=}e_1\ \texttt{in}\ e_3$$

Alternatively, the syntax could be extended to allow mutually recursive definitions within `concur`, e.g. in the example,

$$\texttt{concur}\ z_1{=}z_2 \quad z_2 = z_1\ \texttt{in}\ \texttt{()}$$

These problems are inherent because of recursion in the language. However, they have been dealt with in the literature before, for example in the λ_{need} calculus with recursion in [AF97].

Parallel composition. As a further alternative, one could extend the syntax by some (commutative and associative) parallel composition construct $e \parallel e'$, which is the way this is handled in process calculi such as CCS [Mil89] or π-calculus [Mil99, SW01].

However, in our setting each thread binds a logic variable, namely the future eventually yielding the result of the concurrent computation. In fact, as can be seen from the semantics, parallel composition simply translates to `concur` $\_{=}e$ `in` e', i.e. a concurrent thread where the return value is not used. Thus, we would need to also annotate these variables, e.g. writing

$$e_x \parallel e'_y$$

and this formalization would be quite close to being no more than a syntactic version of the configurations we use.

Promises as pairs. Finally it should be remarked that the reason for formalizing promises by

$$\texttt{prom}\, y \;\texttt{for}\; x \;\texttt{in}\; e$$

or more precisely, by $\texttt{prom}\, e$, thus deviating from promises as found in Alice, is motivated by technical considerations: In Section 5.2, a type system is introduced that ensures linear use of y, but does not restrict the use of x in any way. Therefore, it has advantages treating promise and associated future as different objects.

For the same reason, a formalization of promises as pairs writer/future $\langle y, x \rangle$ causes problems: The first component of this pair needs to be treated linearly, whereas linear use of the second would be too restrictive. In Section 1.2 the relation between these different possibilities has already been indicated.

Types $\alpha \in TyVar$

$\sigma, \sigma' \in Ty \quad ::= \quad \mathsf{unit} \mid \alpha \mid \sigma \to \sigma'$

Syntax $x, y, z \in Var$

$v, v' \in Val \quad ::= \quad \texttt{()} \mid x \mid \lambda x.e$

$e, e' \in Exp \quad ::= \quad v \mid e\, e' \mid \texttt{concur}\ e \mid \texttt{byneed}\ e \mid \texttt{prom}\, e$

Configuration $(P, C, B) \in [Var \to_{fin} Var] \times [Var \to_{fin} Exp]^2$ where

(C1) $\mathsf{ran}(P), \mathsf{dom}(C), \mathsf{dom}(B)$ are p.w. disjoint

(C2) $\mathsf{dom}(P) \cap \mathsf{ran}(P) = \mathsf{dom}(B) \cap \mathsf{dom}(P) = \emptyset$, and P is injective

Contexts $E \quad ::= \quad [\,] \mid E\ e \mid v\ E$

Labels $\alpha \in Lab \quad ::= \quad \beta \mid \mathbf{P}yx \mid \mathbf{F}yv \mid \mathbf{C}xe \mid \mathbf{B}xe \mid \mathbf{L}xe$

Transition

$$
\begin{array}{rl}
(\beta) & E[(\lambda x.e)\ v] \xrightarrow{\beta} E[e[{}^{v}\!/\!_{x}]] \\
(\mathit{prom}) & E[\texttt{prom}\ e] \xrightarrow{\mathbf{P}yx} E[e\,y\,x] \\
(\mathit{fulfill}) & E[y\ v] \xrightarrow{\mathbf{F}yv} E[\texttt{()}] \\
(\mathit{concur}) & E[\texttt{concur}\ e] \xrightarrow{\mathbf{C}xe} E[x] \\
(\mathit{byneed}) & E[\texttt{byneed}\ e] \xrightarrow{\mathbf{B}xe} E[x] \\
(\mathit{lookup}) & E[x\ v] \xrightarrow{\mathbf{L}xv'} E[v'\ v]
\end{array}
$$

Selection

$$
(\mathit{select}) \quad \frac{e \xrightarrow{\alpha} e'}{C[x \mapsto e] \xrightarrow{\langle \alpha, x\rangle} C[x \mapsto e']}
$$

Reduction

$$
(\beta) \quad \frac{C \xrightarrow{\langle \beta, z\rangle} C'}{(P, C, B) \xrightarrow{\langle \beta, z\rangle} (P, C', B)}
$$

$$
(\mathit{Prom}) \quad \frac{C \xrightarrow{\langle \mathbf{P}yx, z\rangle} C' \quad x \neq y \quad x, y \notin \mathsf{var}(P, C, B)}{(P, C, B) \xrightarrow{\langle \mathbf{P}yx, z\rangle} (P[x \mapsto y], C', B)}
$$

$$
(\mathit{Fulfill}) \quad \frac{C \xrightarrow{\langle \mathbf{F}yv, z\rangle} C' \quad P(x) = y \quad x \notin \mathsf{dom}(C)}{(P, C, B) \xrightarrow{\langle \mathbf{F}yv, z\rangle} (P, C'[x \mapsto v], B)}
$$

$$
(\mathit{Concur}) \quad \frac{C \xrightarrow{\langle \mathbf{C}xe, z\rangle} C' \quad x \notin \mathsf{var}(P, C, B)}{(P, C, B) \xrightarrow{\langle \mathbf{C}xe, z\rangle} (P, C'[x \mapsto e], B)}
$$

$$
(\mathit{Byneed}) \quad \frac{C \xrightarrow{\langle \mathbf{B}xe, z\rangle} C' \quad x \notin \mathsf{var}(P, C, B)}{(P, C, B) \xrightarrow{\langle \mathbf{B}xe, z\rangle} (P, C', B[x \mapsto e])}
$$

$$
(\mathit{Lookup}) \quad \frac{C \xrightarrow{\langle \mathbf{L}xv, z\rangle} C' \quad C(x) = v}{(P, C, B) \xrightarrow{\langle \mathbf{L}xv, z\rangle} (P, C', B)}
$$

$$
(\mathit{Force}) \quad \frac{C \xrightarrow{\langle \mathbf{L}xx, z\rangle} C}{(P, C, B \uplus \{x \mapsto e\}) \xrightarrow{\langle \mathbf{L}xx, z\rangle} (P, C[x \mapsto e], B)}
$$

Figure 3.5: Syntax and semantics of λ^{FP}-calculus

Chapter 4

Futures

In this chapter some properties of futures are investigated in detail. There are two main results: The promise-free calculus, called λ^F, is uniformly confluent, and the simply typed version of λ^F is strongly normalizing. Also, we demonstrate that the calculus might be interesting for theoretical applications: By uniform confluence in this functional setting we obtain a comparatively easy formal proof that call-by-value complexity in λ-calculus dominates call-by-need complexity. This considerably simplifies a proof previously given by Niehren in [Nie00], where a uniformly confluent fragment of π-calculus had been used. In fact, uniform confluence is also employed in the strong normalization proof, again demonstrating the importance of the concept when talking about complexity in reduction calculi.

4.1 The Subcalculus λ^F

The subset of expressions for the calculus we are concerned with in this chapter is formally given by the following grammar.

$$\begin{array}{lcll}
e \in \mathit{Exp}^- & ::= & \texttt{()} & \textit{(constant)} \\
 & \mid & x & \textit{(variable)} \\
 & \mid & \lambda x.e & \textit{(abstraction)} \\
 & \mid & e\ e' & \textit{(application)} \\
 & \mid & \texttt{concur}\ e & \textit{(spawn)} \\
 & \mid & \texttt{byneed}\ e & \textit{(delayed)}
\end{array}$$

Essentially, this is the language from the previous chapter, excluding promises. Also, we will denote the restriction of the reduction relation $\rightarrow$ to configurations with Exp^- expressions again by $\rightarrow$, and call the resulting system λ^F.

As is immediate from the reduction rules, the P-part of a configuration (P, C, B) is invariant under reduction of promise-free configurations. Therefore we will take a configuration to be just the pair (C, B) in this chapter for simplicity.

4.2 Uniform Confluence of λ^F

We will begin by proving reduction uniformly confluent, as this is a key property used in the proof of strong normalization that follows. Confluence is a property well-known from λ-

calculus, essentially stating that the order of evaluating subexpressions is irrelevant. On the one hand, our use of call-by-value reduction contexts restricts the number of redexes compared to pure λ-calculi, while on the other hand concurrency spoils deterministic reduction as found in sequential languages with fixed evaluation strategy. The results of this section show, however, that the transition relation on λ^F-configurations is not only confluent, but in fact is *uniformly confluent* [Nie00]. This will be shown to hold for both the untyped and the typed calculus λ^F. Uniform confluence is an interesting concept as it does not only imply *uniqueness* of normal forms but also asserts that all reduction paths leading to a normal form have the *same length.* Hence, it becomes possible to talk about the (time) complexity of expressions in reduction calculi.

Recall that a relation $\rightarrow$ is confluent if whenever $s \rightarrow^* s_1$ and $s \rightarrow^* s_2$ then there exists $s' \in S$ such that both $s_1 \rightarrow^* s'$ and $s_2 \rightarrow^* s'$. It is uniformly confluent if $s_1 \leftarrow s \rightarrow s_2$ for $s_1 \neq s_2$ implies $s_1 \rightarrow s' \leftarrow s_2$ for some s'. The slightly stronger property which we will actually prove is stated diagrammatically in Figure 4.1. It is not hard to show that uniform confluence implies confluence. Also, uniform confluence guarantees that all the executions of a term have the same length, which again may be finite or infinite. This consequence is used in the proof showing strong normalization, given in the next section.

$$\begin{tikzcd}
& (C,B) \arrow[dl, "\langle\alpha_0,x\rangle"'] \arrow[dr, "\langle\alpha_1,y\rangle"] & \\
(C_0,B_0) \arrow[rr, phantom, "\neq"] \arrow[dr, dashed, "\langle\alpha_1,y\rangle"'] & & (C_1,B_1) \arrow[dl, dashed, "\langle\alpha_0,x\rangle"] \\
& (C',B') &
\end{tikzcd}$$

Figure 4.1: Uniform confluence on *Config*.

Proving confluence for the configuration calculus is simplified by the observation that each *term* has at most one redex as subterm, which is proved in the next lemma. Although this does not hold for configurations, we still have that redexes in different threads are independent of each other, so in particular we do not have any *critical pairs*, i.e. there are no overlapping redexes. In fact, the next lemma holds for the full calculus, not only for the restriction λ^F.

Lemma 4.2.1. *Suppose* $(P_2, C_2, B_2) \xleftarrow{\langle\alpha_2,x\rangle} (P, C, B) \xrightarrow{\langle\alpha_1,x\rangle} (P_1, C_1, B_1)$. *Then* $(P_1, C_1, B_1) = (P_2, C_2, B_2)$*, up to renaming of bound variables.*

Proof. By the assumption and the inference rules, we have both

$$C[x{\mapsto}e] \xrightarrow{\langle\alpha_1,x\rangle} C[x{\mapsto}e_1] \text{ and } C[x{\mapsto}e] \xrightarrow{\langle\alpha_2,x\rangle} C[x{\mapsto}e_2]$$

for some $e, e_1, e_2 \in \mathit{Exp}$ such that $e \xrightarrow{\alpha_1} e_1$ and $e \xrightarrow{\alpha_2} e_2$. The proof is now by induction on the structure of e. The cases where $e \in \mathit{Val}$, i.e. where e is a constant, a variable or an abstraction, are trivial, as then the empty evaluation context is the only one that applies and there are no reduction rules at all that match these terms.

- If $e \equiv$ `concur` e', then e itself is reducible via (*concur*). However, this is the only applicable rule, and the empty context is the only applicable context. So necessarily $e_1 \equiv y$, $e_2 \equiv z$, $\alpha_1 \equiv \mathbf{C}ye'$ and $\alpha_2 \equiv \mathbf{C}ze'$, for some variables $y, z \in \mathit{Var}$.. By the side-condition in rule (*Concur*) we know $y, z \notin \mathsf{var}(P, C, B)$. Moreover, $B_1 = B = B_2$, $C_1 = C[x{\mapsto}y, y{\mapsto}e']$ and $C_2 = C[x{\mapsto}z, z{\mapsto}e']$. Therefore the two configurations can differ only in the name of the bound variable y.
- The case $e \equiv$ `byneed` e' is analoguous to the previous one for `concur` e'.
- If $e \equiv$ `prom` e', then the proof is similar.

- So suppose $e \equiv e'\ e''$. There are several subcases to consider.
  - Firstly, if both e' and e'' are values, then none of these is a redex. Depending on the shape of e', e is reducible by either of rules (β), (*fulfill*) or (*lookup*). If e' is not a variable, then neither (*fulfill*) nor (*lookup*) applies, and if $e' \equiv y \in \mathit{Var}$, then ($\beta$) does not apply. Now the side-conditions of the respective rules (*Fulfill*) and (*Lookup*) on configurations and disjointness of $\mathsf{ran}(P)$ and $\mathsf{dom}(C), \mathsf{dom}(B)$ show that applicability of these rules is mutually exclusive. In the case of $e \xrightarrow{\mathbf{F}ye''} e_1$ and $e \xrightarrow{\mathbf{F}ye''} e_2$ by (*fulfill*), injectivity of P ensures that the resulting configurations are equal. In the case of $e \xrightarrow{\mathbf{L}yv_1} e_1$ and $e \xrightarrow{\mathbf{L}yv_2} e_2$ by (*lookup*), we can conclude $v_1 \equiv v_2$ from disjointness of $\mathsf{dom}(C)$ and $\mathsf{dom}(B)$. Again, this shows that $(P_1, C_1, B_1) = (P_2, C_2, B_2)$.
  - If e' is not a value, then reduction of e may be caused by reductions of e' only, by definition of evaluation contexts. Thus, the result follows from the inductive hypothesis.
  - Finally, if e' is a value but e'' is not, there is no rule that applies directly to e, and the reductions of e are caused solely by reductions of e''. Again, the result follows by induction.

This concludes the proof. □

4.2.1 Uniform Confluence of Untyped Expressions

We first consider uniform confluence for the calculus λ^F without the restriction to typable terms only. We start with the key lemma, showing that transitions $\xrightarrow{\langle\alpha,x\rangle}$ on configurations that are "independent" of each other do commute.

Lemma 4.2.2. *Let $x_0 \neq x_1$ be distinct variables and suppose there are two transitions*

$$\begin{tikzcd}
(C,B) \arrow[r, "\langle\alpha_1,x_1\rangle"] \arrow[d, "\langle\alpha_0,x_0\rangle"'] \arrow[dr, phantom, "\neq"] & (C_1,B_1) \\
(C_0,B_0) &
\end{tikzcd}$$

Then there exists (C', B') such that the above diagram can be closed to the following

$$\begin{tikzcd}
(C,B) \arrow[r, "\langle\alpha_1,x_1\rangle"] \arrow[d, "\langle\alpha_0,x_0\rangle"'] & (C_1,B_1) \arrow[d, dashed, "\langle\alpha_0,x_0\rangle"] \\
(C_0,B_0) \arrow[r, dashed, "\langle\alpha_1,x_1\rangle"] & (C',B')
\end{tikzcd}$$

Proof. The proof is by a tedious case distinction on the labels α_0, α_1. Most cases are easy to check, so we will consider only some representative ones here.

- First, suppose $\alpha_0 = \beta = \alpha_1$. By the inference rule (β) we have $B_0 = B = B_1$. Further,

$$C(x_0) = E_0[(\lambda x.e_0)\, v_0] \text{ and } C(x_1) = E_1[(\lambda y.e_1)\, v_1]$$

by rule (*select*), and so

$$C_0 = C[x_0 \mapsto E_0[e_0[{}^{v_0}/_{x}]]] \text{ and } C_1 = C[x_1 \mapsto E_1[e_1[{}^{v_1}/_{y}]]]$$

Hence,

$$\begin{aligned}(C_0, B_0) &\xrightarrow{\langle \beta, x_1 \rangle} (C[x_0 \mapsto E_0[e_0[{}^{v_0}/_{x}]],\ x_1 \mapsto E_1[e_1[{}^{v_1}/_{y}]]], B) \\ &\xleftarrow{\langle \beta, x_0 \rangle} (C_1, B_1)\end{aligned}$$

- Next, suppose $\alpha_0 = \mathbf{C}xe_0$ and $\alpha_1 = \mathbf{B}ye_1$. Then, assuming that

  $$C(x_0) = E_0[\texttt{concur}\ e_0] \text{ and } C(x_1) = E_1[\texttt{byneed}\ e_1]$$

  we have $B_0 = B$, $B_1 = B[y \mapsto e_1]$, $C_0 = C[x_0 \mapsto E_0[x],\ x \mapsto e_0]$ and $C_1 = C[x_1 \mapsto E_1[y]]$. Possibly renaming bound variables, we can assume $x \neq y$. Therefore,

  $$\begin{aligned}(C_0, B_0) &\xrightarrow{\langle \mathbf{B}ye_1, x_1 \rangle} (C[x_0 \mapsto E_0[x],\ x_1 \mapsto E_1[y]\ x \mapsto e_0], B_1) \\ &\xleftarrow{\langle \mathbf{C}xe_0, x_0 \rangle} (C_1, B_1)\end{aligned}$$

- Finally, suppose $\alpha_0 = \mathbf{L}xv_0$ and $\alpha_1 = \mathbf{L}yv_1$. The case where $x \neq y$ is similar to the previous ones. Now assume $x = y$, then

  – if $x \in \mathsf{dom}(B)$, $B(x) = e$ and $B' =_{\mathsf{df}} B|_{\mathsf{dom}(B)-\{x\}}$ is the restriction of B to variables distinct from x, then

  $$B_0 = B' = B_1 \text{ and } C_0 = C[x \mapsto e] = C_1$$

  so there is nothing to show.

  – if $x \in \mathsf{dom}(C)$, we have $B_0 = B = B_1$, $v_0 = C(x) = v_1$ and

  $$C(x_0) = E_0[x\,v] \text{ and } C(x_1) = E_1[x\,v']$$

  Thus,

  $$\begin{aligned}(C_0, B_0) &\xrightarrow{\langle \mathbf{L}xv_0, x_1 \rangle} (C[x_0 \mapsto E_0[v_0\,v],\ x_1 \mapsto E_1[v_0\,v']], B) \\ &\xleftarrow{\langle \mathbf{L}xv_0, x_1 \rangle} (C_1, B_1)\end{aligned}$$

The remaining cases are similar. □

However, note that the above reasoning only works as long as we do not consider $e \xrightarrow{\mathbf{F}xv} e'$, by (the λ^{FP}-) rule (*fulfill*). In this case the side-condition ($x \notin \mathsf{dom}(C)$) of (*Fulfill*) may be satisfied in the first transition, but need not hold for the second one. In the next Chapter we will restrict the use of promises, thereby guaranteeing that the side-condition is always satisfied for (*Fulfill*) as well and thus extend the confluence proof.

Proposition 4.2.3 (Uniform Confluence). *Reduction $\rightarrow$ on the set of λ^F-configurations is uniformly confluent. In particular, $\rightarrow$ is confluent.*

Proof. Suppose $(C_0, B_0) \leftarrow (C, B) \rightarrow (C_1, B_1)$ and assume $(C_0, B_0) \neq (C_1, B_1)$. We conclude from Lemma 4.2.1 that there are distinct $x_0 \neq x_1$ in $\mathsf{dom}(C)$ causing the transitions, i.e.

$$(C_0, B_0) \xleftarrow{\langle \alpha_0, x_0 \rangle} (C, B) \xrightarrow{\langle \alpha_1, x_1 \rangle} (C_1, B_1)$$

for some α_0, α_1. By Lemma 4.2.2, there is (C', B') such that $(C_0, B_0) \rightarrow (C', B') \leftarrow (C_1, B_1)$. □

4.2.2 Uniform Confluence of Simply Typed Expressions

Uniform confluence now carries over to the simply typed case immediately: Whenever (C, B) is well-typed and $(C_0, B_0) \leftarrow (C, B) \rightarrow (C_1, B_1)$, then (C_0, B_0), (C_1, B_1) can be given a type, by subject reduction (Lemma 3.5.4). By uniform confluence $(C_0, B_0) \rightarrow (C', B') \leftarrow (C_1, B_1)$ for some (C', B'), and again by subject reduction (C', B') is well-typed.

Proposition 4.2.4 (Uniform Confluence). *Reduction restricted to well-typed configurations of λ^F is uniformly confluent. In particular, it is confluent.*

4.3 Strong Normalization

For terms of the simply typed λ-calculus with constants it is well-known that every reduction sequence is finite (see [Tho92], for instance). As can be seen from the reduction relation,

$$\texttt{concur}\ x{=}e\ \texttt{in}\ e'$$

and similarly

$$\texttt{byneed}\ x{=}e\ \texttt{in}\ e'$$

behave very much like a `let` in λ-calculus, which in turn can be coded into simply typed λ-calculus. Moreover, we consider only *weak reduction* in λ^F, not reducing below, e.g., abstractions. Hence, for expressions that do not contain any promises, it should be intuitively clear that strong normalization holds, just as for the simply typed λ-calculus: There are no infinite reduction sequences. In fact, we will show that this is true. It is worth pointing out again that this certainly does not hold for the full language λ^{FP}, since promises suffice do define recursive (and non-terminating) functions, as shown in Section 3.6 on page 28.

So how should one prove strong normalization of λ^F? The obvious choice is to simulate reduction sequences on configurations by reduction sequences on terms of the simply typed λ-calculus. If one could show that every reduction step possible on the configuration can be matched by some β-reduction step on a simply typed λ-term, this would give a finite upper bound on the number of possible reduction steps, and the result will follow.

However, not all reduction rules of λ^F have a natural analogue in λ-calculus reduction, and so we will try to find simulations that are invariant under, e.g. transition by rule (*lookup*). We then deal with termination of these reductions separately.

In order to apply the simulation technique, it seems necessary to encode a whole configuration into a single λ-term that has sufficiently many redexes for the simulation to work. We proceed as follows. First we give an example showing that a naïve simulation does not work and must be refined. Next, we define a restricted, “canonical” reduction relation $\xrightarrow[can]{}$ for which a very simple simulation already works. We then relate canonical reduction $\xrightarrow[can]{}$ to the original $\rightarrow$ relation, using the uniform confluence property proved in the previous section.

4.3.1 Naïve Simulation

Suppose we want to simulate reduction of the term

$$e =_{\mathsf{df}} (\lambda y.v)\ \texttt{concur}\ e' \tag{4.1}$$

where we assume that v does not contain free occurences of y. It seems very natural to set the term in relation to the λ-term $(\lambda y.v)\ e'$, which is just e where the `concur` annotation is ignored. So, starting with the initial configuration

$$(\{x_0 \mapsto (\lambda y.v)\ \texttt{concur}\ e'\}, \emptyset)$$

we have a transition to

$$(\{x_0 \mapsto (\lambda y.v)\ x, x \mapsto e'\}, \emptyset) \tag{4.2}$$

Now a sensible way to represent *both* threads in a *single* term of the λ-calculus seems to be substituting thread e' for its associated future x in $(\lambda y.v)\ x$, obtaining $(\lambda y.v)\ e'$. By β-reduction (recall that x is a value), one then obtains the configuration $(\{x_0 \mapsto v, x \mapsto e'\})$. Simulating this on the corresponding λ-term, we obtain the term v. A problem occurs now whenever $e' \xrightarrow{\beta} e''$.

$$\begin{tikzcd}
(\lambda y.v)\ \texttt{concur}\ e' \arrow[d] & \sim & (\lambda y.v)\ e' \arrow[d] \\
(\{x_0 \mapsto (\lambda y.v)\ x, x \mapsto e'\}, \emptyset) \arrow[d] & \sim & (\lambda y.v)\ e' \arrow[d] \\
(\{x_0 \mapsto v, x \mapsto e'\}, \emptyset) \arrow[d] & \sim & v \\
(\{x_0 \mapsto v, x \mapsto e''\}, \emptyset) & & \textbf{?}
\end{tikzcd}$$

Looking carefully at example (4.1) on this page we see what the problem with the naïve simulation is: it does not work. Although the "main" thread in the computation might not need the result being computed in the second thread, computation in this thread will continue. In the corresponding λ-term we then do not have a sufficient number of redexes to match this.

Put differently, we should not have performed the first β-reduction, as this led to a λ-term containing only part of the configuration. The reason for it is that the β-reduction step had a future as argument whose value still had to be computed, and so did not correspond to a "canonical" step of by-value reduction. Clearly, disallowing such steps will prevent this problem. We will work out the details of how this works next, before showing that it in fact suffices to consider the restricted reduction relation.

4.3.2 Canonical Reduction

Intuitively, canonical reduction imitates call-by-value reduction, taking the complete configuration into account to prevent reduction steps that led to the failure of the simulation in the above example (4.1).

Ordering and Representability

It should be clear from the operational semantics that we cannot create mutually dependant threads in the restricted language anymore, thus the above idea of *representing* a configuration as a λ-term (by substitution) can be applied. We begin with a lemma stating that our configurations will indeed remain acyclic during reduction, in the following sense.

Definition 4.3.1. *Suppose* $C \cup B = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ *for a configuration* (C, B)*. Then a linear order* $<_{(C,B)}$ *on the variables* $\{x_1, \ldots, x_n\}$ *is* admissible *if* $\mathsf{fv}(e_i) \subseteq \{x_k \mid x_k > x_i\}$ *for all* $1 \leq i \leq n$*.*

If the configuration is clear from the context, we will drop the subscript (C, B), writing just $<$. In (4.2) of the example on the preceding page, we thus have that $x_0 < x$ is admissible, whereas $x < x_0$ is not. Recall that for a term e, the initial configuration is just $(\{x_0 \mapsto e\}, \emptyset)$, for which the empty relation trivially is admissible. The existence of admissible orders is an invariant, as the following lemma shows. In particular, we can assume that for all the configurations we consider in this section, deriving from initial configurations of λ^F-terms, admissible orders exist.

Lemma 4.3.2. *Suppose* $(C, B) \to (C', B')$*, and assume that there is an admissible order* $<_{(C,B)}$*. Then this can be extended to an admissible order* $<_{(C',B')}$ *on* $\mathsf{dom}(C') \cup \mathsf{dom}(B')$*.*

Proof. Consider cases for $\to$. Without loss of generality assume that $x_1 < \cdots < x_n$ is such an ordering on the variables x_i, where $\{x_1, \ldots, x_n\} = \mathsf{dom}(C) \cup \mathsf{dom}(B)$.

- If $(C, B) \xrightarrow{\langle \mathbf{L}yy, x\rangle} (C', B')$ by (*Lookup*), then $C \cup B = C' \cup B'$ and there is nothing to show.
- For $(C, B) \xrightarrow{\langle \beta, x\rangle} (C', B')$ we know $\mathsf{fv}(C(x)) \supseteq \mathsf{fv}(C'(x))$, and for all $y \neq x$ we have $(C \cup B)(y) = (C' \cup B')(y)$. So we let $<_{(C',B')} =_{\mathsf{df}} <_{(C,B)}$.
- If $(C, B) \xrightarrow{\langle \mathbf{L}yv', x\rangle} (C', B')$ by (*Force*), then

  $$C(x) = E[y\ v] \xrightarrow{\mathbf{L}yv'} E[v'\ v] = C'(x)$$

  where $(C \cup B)(y) = v'$, i.e. the free variables of $C'(x)$ are contained in the set $\mathsf{fv}(C(x)) \cup \mathsf{fv}(v')$. However, $C(x) = E[y\ v]$ and admissibility of $<_{(C,B)}$ implies $x <_{(C,B)} y$, and so we may again obtain an admissible order by $<_{(C',B')} =_{\mathsf{df}} <_{(C,B)}$.
- Finally, if $(C, B) \xrightarrow{\langle \alpha, x_i\rangle} (C', B')$ by (*Concur*) or (*Byneed*), then the transition was by

  $$C(x_i) \xrightarrow{\mathbf{C}ye} C'(x_i) \text{ or } C(x_i) \xrightarrow{\mathbf{B}ye} C'(x_i)$$

  for fresh y. In particular, this means $\mathsf{fv}(e) \subseteq \mathsf{fv}(C(x_i))$ and $\mathsf{fv}(C'(x_i)) \subseteq \mathsf{fv}(C(x_i)) \cup \{y\}$. Consequently, the relation $<_{(C',B')}$, obtained by extending the order $<_{(C,B)}$ to

  $$x_1 < \cdots < x_i < y < x_{i+1} < \cdots < x_n$$

  is an admissible order.

$\square$

Observe that the last case depends on the fact that y is fresh. In particular, it would not hold in general for $\xrightarrow{\mathbf{F}ye}$ reductions by the λ^{FP}-rule (*Fulfill*), which may very well introduce cyclic dependencies among the variables.

We will now formally define the representation of a λ^F-term, later lifting this to λ^F-configurations. For an expression $e \in \mathit{Exp}^-$ let $[\![e]\!]$ be the λ-calculus term obtained by simply removing `concur` and `byneed`.

Definition 4.3.3. *A translation $[\![\cdot]\!]$ of well-typed Exp^--terms into the simply typed λ-calculus is given by defining*

$$\begin{aligned}
[\![\texttt{()}]\!] &=_{\mathsf{df}} \texttt{()},\\
[\![x]\!] &=_{\mathsf{df}} x,\\
[\![\lambda x.e]\!] &=_{\mathsf{df}} \lambda x.[\![e]\!],\\
[\![e\ e']\!] &=_{\mathsf{df}} [\![e]\!][\![e']\!]
\end{aligned}$$

and

$$[\![\texttt{concur}\ e]\!] =_{\mathsf{df}} [\![\texttt{byneed}\ e]\!] =_{\mathsf{df}} [\![e]\!] \ .$$

It is not hard to see that for any well-typed term e the translation $[\![e]\!]$ indeed is a well-typed term of the simply typed λ-calculus with constant `()`:`unit`, over base type `unit`. Also, $\mathsf{fv}([\![e]\!]) = \mathsf{fv}(e)$. The translation immediately carries over to contexts E, so that $[\![E[e]]\!] \equiv [\![E]\!][[\![e]\!]]$. Furthermore, it is easily verified that $[\![e[{}^{v}\!/\!{}_{x}]]\!] \equiv [\![e]\!][{}^{[\![v]\!]}\!/\!{}_{x}]$ holds, and thus

$$[\![E[(\lambda x.e)\ v]]\!] \rightarrow_\beta [\![E[e[{}^{v}\!/\!{}_{x}]]]\!] \tag{4.3}$$

The next step is to lift the translation to configurations. We will use the suggestive notation

$$\mathsf{subst}\ M_n/x_n \ldots M_1/x_1\ \mathsf{in}\ M =_{\mathsf{df}} M[{}^{M_1}\!/\!{}_{x_1}]\cdots[{}^{M_n}\!/\!{}_{x_n}] \tag{4.4}$$

for a sequence successive substitutions.

Definition 4.3.4. *Let (C,B) be a configuration, $C \cup B = \{x_1 {\mapsto} e_1, \ldots, x_n {\mapsto} e_n\}$, and assume $x_1 < \cdots < x_n$ wrt. an admissible order $<_{(C,B)}$. The term $\mathsf{rep}_{C,B,<_{(C,B)}}$ defined by*

$$\mathsf{rep}_{C,B,<} =_{\mathsf{df}} \mathsf{subst}\ [\![e_n]\!]/x_n \ldots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1$$

is a representation *for (C,B) wrt. $<_{(C,B)}$.*

Note that by Lemma 4.3.2, there is an admissible order $<$ for every configuration (C,B) that was derived from the initial configuration of a (well-typed) λ^F-term. Further, we just remark that by subject reduction these configurations are also well-typed, and then the representation $\mathsf{rep}_{C,B,<}$ is typable in the simply typed λ-calculus.

However, just considering representations is *not* sufficient, because these may not contain sufficiently many redexes to match reduction steps of the configuration. Example (4.1) on page 40 clearly showed the problem.

Canonical Reduction and Complete Representations

We shall now see how to overcome the problem that occurred with the naïve simulation on page 40. Recall that it was caused by β-reducing a subexpression of the representation where the argument was not fully evaluated.

Definition 4.3.5. Canonical reduction $\xrightarrow[can]{\langle\alpha,x\rangle}$ *is defined by*

$$(C,B) \xrightarrow[can]{\langle\alpha,x\rangle} (C',B')$$

if $(C,B) \xrightarrow{\langle\alpha,x\rangle} (C',B')$ *and there is an admissible order* $<$ *of* (C,B) *such that* $C(x') \in \mathit{Val}$ *for all* $x' \in \mathsf{dom}(C)$ *with* $x' > x$.

As before, we let $\xrightarrow[can]{} =_{\mathsf{df}} \bigcup_{\alpha\in Lab} \xrightarrow[can]{\alpha}$. To demonstrate canonical reduction, reconsider the previous example. The critical reduction was in (4.2),

$$(\{x_0{\mapsto}(\lambda y.v)\ x, x{\mapsto}e'\}, \emptyset) \xrightarrow{\langle\beta,x_0\rangle} (\{x_0{\mapsto}v, x{\mapsto}e'\}, \emptyset)$$

by rule (β), assuming $e' \notin \mathit{Val}$ and x not free in v. However, since necessarily $x_0 < x$ and $e' \notin \mathit{Val}$, the only canonical reduction in this case is

$$(\{x_0{\mapsto}(\lambda y.v)\ x, x{\mapsto}e'\}, \emptyset) \xrightarrow[can]{\langle\beta,x\rangle} (\{x_0{\mapsto}(\lambda y.v)\ x, x{\mapsto}e''\}, \emptyset)$$

assuming $e' \xrightarrow{\beta} e''$. Not surprisingly, this step can be matched in the representation with respect to $<$,

$$\begin{aligned}\mathsf{subst}\ [\![e']\!]/x\ \ ((\lambda y.[\![v]\!])\ x)/x_0\ \mathsf{in}\ x_0 &\equiv (\lambda y.[\![v]\!])\ [\![e']\!] \\ &\to_\beta (\lambda y.[\![v]\!])\ [\![e'']\!] \\ &\equiv \mathsf{subst}\ [\![e'']\!]/x\ \ ((\lambda y.[\![v]\!])\ x)/x_0\ \mathsf{in}\ x_0\end{aligned}$$

by equation (4.3).

It should be remarked that canonical reduction restricts $\to$ reduction, but nevertheless is not just deterministic standard reduction. Canonical reduction is complemented by a notion of complete representation which, intuitively, asserts that the representation contains all the "essential" threads as subterms.

Definition 4.3.6. *Suppose* (C,B) *is a* λ^F*-configuration, with* $C \cup B = \{x_1{\mapsto}e_1, \ldots, x_n{\mapsto}e_n\}$*, and* $x_1 < \cdots < x_n$ *is an admissible order for* (C,B)*. Then* $\mathsf{rep}_{C,B,<}$ *is a* complete representation *if, whenever* $e_i \notin \mathit{Val}$ *and* $x_i \in \mathsf{dom}(C)$*, then*

$$\mathsf{subst}\ [\![e_{i-1}]\!]/x_{i-1} \cdots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1$$

contains x_i *free.*

In particular, any representation of an initial configuration is complete. In the following, we state that this property is invariant under canonical reduction, and hence canonical reduction steps can be appropriately matched on the representations.

Simulation

We next see how to simulate canonical reduction on complete representations.

Lemma 4.3.7. *Suppose (C, B) is a configuration with admissible order $<$ s.t. $\mathsf{rep}_{C,B,<}$ is a complete representation. If $(C, B) \xrightarrow[can]{\langle\beta,x_i\rangle} (C', B')$ by (β), then $<$ is also admissible for (C', B') and*

$$\mathsf{rep}_{C,B,<} \rightarrow_\beta \mathsf{rep}_{C',B',<}$$

where $\mathsf{rep}_{C',B',<}$ is complete.

Proof. By definition, if $C \cup B = \{x_1{\mapsto}e_1, \dots, x_n{\mapsto}e_n\}$ and $x_1 < \cdots < x_n$ then

$$\mathsf{rep}_{C,B,<} \equiv \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1$$

By assumption and the definition of $\xrightarrow[can]{\langle\beta,x_i\rangle}$ we have $x_i \in \mathsf{dom}(C)$ and

$$e_i \xrightarrow{\beta} e_i'$$

By (4.3), $[\![e_i]\!] \rightarrow_\beta [\![e_i']\!]$ and so by the definition of a complete representation,

$$\mathsf{rep}_{C,B,<} \rightarrow_\beta \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![e_i']\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1 \equiv \mathsf{rep}_{C',B',<'}$$

As in the proof of Lemma 4.3.2, $<$ is admissible for (C', B'). Finally, from the definition of canonical reduction it follows that if $e_k \notin \mathit{Val}$, then $C(x_j) = C'(x_j)$ for all $x_j < x_k$ in $\mathsf{dom}(C)$. Hence, completeness of $\mathsf{rep}_{C',B',<}$ follows from completeness of $\mathsf{rep}_{C,B,<}$. ☐

Lemma 4.3.8. *Let (C, B) be a configuration with admissible order $<$ so that $\mathsf{rep}_{C,B,<}$ is complete. If $(C, B) \xrightarrow[can]{\langle\alpha,x_i\rangle} (C', B')$ not by (β), i.e. α is not β, then there exists admissible $<'$ for (C', B') such that*

$$\mathsf{rep}_{C,B,<} \equiv \mathsf{rep}_{C',B',<'}$$

and $\mathsf{rep}_{C',B',<'}$ is complete for (C', B').

Proof. We consider the possible canonical reductions in turn. So suppose (C, B) is a canonical configuration, with admissible order $<$ s.t. $\mathsf{rep}_{C,B,<}$ is a complete representation.

- If $(C, B) \xrightarrow[can]{\langle \mathbf{L}x_jv',x_i\rangle} (C', B')$ by rule (*Lookup*), then $<' =_{\mathsf{df}} <$ is admissible for (C', B') as well, as seen in the proof of Lemma 4.3.2. Also, the reduction is caused by

  $$C(x_i) = E[x_j\ v] \xrightarrow{\langle \mathbf{L}x_jv',x_i\rangle} E[v'\ v] = C'(x_i)$$

  and

  $$\begin{aligned}\mathsf{rep}_{C,B,<} &\equiv \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![v']\!]/x_j \dots [\![E[x_j\ v]]\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1\\ &\equiv \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![v']\!]/x_j \dots [\![E[v'\ v]]\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1\\ &\equiv \mathsf{rep}_{C',B',<'}\end{aligned}$$

  which is a complete representation for (C', B') since $\mathsf{rep}_{C,B,<}$ is.

- Next, suppose the reduction is by (*Concur*), i.e. for some $x_i \in \mathsf{dom}(C)$,
$$C(x_i) = e_i \equiv E[\texttt{concur}\ e] \xrightarrow{\mathbf{C}xe} E[x] \equiv C'(x_i)$$
Using $[\![e_i]\!] \equiv [\![E[\texttt{concur}\ e]]\!] \equiv [\![E]\!][[\![e]\!]] \equiv ([\![E[x]]\!])[{}^{[\![e]\!]}/x]$, we obtain
$$\begin{aligned}\mathsf{rep}_{C,B,<} &\equiv \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![e_i]\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1 \\ &\equiv \mathsf{subst}\ [\![e_n]\!]/x_n \dots [\![e]\!]/x\ [\![E[x]]\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1 \\ &\equiv \mathsf{rep}_{C',B',<'}\end{aligned}$$
where $<'$ is the admissible extension of $<$ defined in the proof of Lemma 4.3.2. Completeness of $\mathsf{rep}_{C',B',<'}$ follows from completeness of $\mathsf{rep}_{C,B,<}$.

- The case $\xrightarrow[can]{\langle \mathbf{B}xe,x_i\rangle}$ by (*Byneed*) is analogous.

- Finally, suppose $(C, B) \xrightarrow[can]{\langle \mathbf{L}x_jx_j,x_i\rangle} (C', B')$, by rule (*Force*). By definition, there is $x_i \in \mathsf{dom}(C)$ causing the reduction, i.e.
$$C(x_i) = e_i \equiv E[x_j\ v] \xrightarrow{\mathbf{L}x_jx_j} e_i = C'(x_i)$$
$x_i < x_j$ by admissibility and in particular x_j occurs free in
$$\mathsf{subst}\ [\![e_{j-1}]\!]/x_{j-1} \dots [\![e_i]\!]/x_i \dots [\![e_1]\!]/x_1\ \mathsf{in}\ x_1$$
Thus, $\mathsf{rep}_{C',B',<'} \equiv \mathsf{rep}_{C,B,<}$ is complete.

This concludes the proof. □

Strong Normalization of Canonical Reduction

We are now in the position to prove the main result of this section, the strong normalization property of λ^F with respect to canonical reduction $\xrightarrow[can]{}$.

Proposition 4.3.9 (Strong Normalization of $\xrightarrow[can]{}$).
Let $e \in Exp^-$ be a well-typed term. Then every $\xrightarrow[can]{}$-reduction sequence beginning with the initial configuration $(\{x_0 \mapsto e\}, \emptyset)$ is finite.

Proof. For any configuration (C, B), let $(\hat{C}, \hat{B})$ be the configuration where all future variables are replaced by the respective expression. Formally, without loss of generality assume $x_1 < \dots < x_n$ is an admissible order on $\mathsf{var}(C, B)$, and let $e_i = (C \cup B)(x_i)$. Then, if
$$\hat{e_i} =_{\mathsf{df}} e_i[{}^{e_{i+1}}/x_{i+1}] \cdots [{}^{e_n}/x_n] \ ,$$
$(\hat{C}, \hat{B})$ is obtained from (C, B) by replacing e_i with $\hat{e_i}$. Also, let $|(C, B)|$ denote the *size* of the configuration, given by the sum
$$\sum_{x \in \mathsf{dom}(C)} |C(x)| + \sum_{x \in \mathsf{dom}(B)} |B(x)| \ ,$$

where the length $|e|$ of e is defined in the obvious way. Moreover, for any configuration (C, B), the number $l(C, B)$ of possible consecutive $\underset{can}{\longrightarrow}$-reductions by rule (*Lookup*) is finite: Let $l(e_i)$ denote the maximal number of possible lookups for variables in e_i. A very crude estimate on the number of variables occurring in e_i shows that clearly

$$l(e_i) \leq |e_i|(1 + \sum_{x_j > x_i} l(e_j))$$

and by induction one obtains the bound

$$l(e_{n-k}) \leq 2^k |e_{n-k}| \cdots |e_{n-1}|$$

Thus, $\sum_{x_i \in \mathsf{dom}(C)} l(e_i) < \infty$. Finally, define $\phi(C, B) \in \mathbb{N}^3$ by

$$\phi(C, B) =_{\mathsf{df}} (\left|(\hat{C}, \hat{B})\right|, |\mathsf{dom}(B)|, l(C, B))$$

Then, whenever $(C, B) \underset{can}{\longrightarrow} (C', B')$, not by rule (β), one easily checks that

$$\phi(C, B) >_{\text{lex}} \phi(C', B')$$

with respect to the lexicographic order on $\mathbb{N}^3$. In fact, we have the following relationship on the components of $\phi(C, B)$ and $\phi(C', B')$, depending on the rule applied to infer the reduction.

(*Concur, Byneed*)	$>$	$\leq$	$\leq$
(*Force*)	$=$	$>$	$\leq$
(*Lookup*)	$=$	$=$	$>$

Now assume there is some infinite reduction sequence

$$(C_1, B_1) \underset{can}{\longrightarrow} (C_2, B_2) \underset{can}{\longrightarrow} \ldots$$

then well-foundedness of $>_{\text{lex}}$ implies that any such infinite reduction sequence must make use of infinitely many $\xrightarrow[can]{\langle\beta,x\rangle}$ reductions, by rule (β). Using the fact that for initial λ^F-configurations there exists a complete representation, then by Lemma 4.3.7 and Lemma 4.3.8 we can then find admissible orders $<_i$ such that for all i,

$$M_i =_{\mathsf{df}} \mathsf{rep}_{C_i, B_i, <_i}$$

is a complete representation of (C_i, B_i), and for some infinite subsequence $M_{i_1} \rightarrow_\beta M_{i_2} \rightarrow_\beta \ldots$, contradicting the strong normalization property of the simply typed λ-calculus. Hence, reduction on configurations must be strongly normalizing as well. □

4.3.3 Strong Normalization of →

Although working with canonical reduction so far, the results carry over to the reduction relation → as well. Here, we show how the two relate.

Lemma 4.3.10. *For any configuration (C, B) with admissible variable order,*

$$(C, B) \rightarrow \quad \textit{iff} \quad (C, B) \underset{can}{\longrightarrow}$$

Proof. Clearly any $\xrightarrow[can]{}$-reduction also is a $\rightarrow$-reduction. Conversely, suppose (C, B) is $\xrightarrow[can]{}$-irreducible. Then, for all $x_i \in \mathsf{dom}(C)$, $C(x_i)$ is not $\rightarrow$-reducible or else $C(x_j) \notin \mathit{Val}$ for all $x_j > x_i$ in $\mathsf{dom}(C)$. The second case is absurd, considering $i = n$, where x_n is maximal wrt. $<$. Hence, $C(x_i)$ is $\rightarrow$-irreducible for all $x_i \in \mathsf{dom}(C)$, and so (C, B) must be $\rightarrow$-irreducible as well, by rule (*select*). □

Lemma 4.3.11. *The length of any (complete) $\rightarrow$-execution of a configuration with admissible order equals the length of any (complete) $\xrightarrow[can]{}$-execution.*

Proof. By the preceding Lemma 4.3.10, any (complete) $\xrightarrow[can]{}$-execution is a (complete) $\rightarrow$-execution. The result now follows from Lemma 4.3.12. □

Lemma 4.3.12. *The length of all (complete) $\rightarrow$-executions of a configuration (C, B) is equal.*

Proof. This is an instance of Proposition 2.4 of [Nie00], a general result about uniformly confluent calculi. □

Immediately from Proposition 4.3.9 and Lemma 4.3.11 we now obtain strong normalization for $\rightarrow$.

Proposition 4.3.13 (Strong Normalization of $\rightarrow$).
Let $e \in Exp^-$ be a well-typed term. Then every $\rightarrow$-reduction sequence beginning with the initial configuration e is finite.

4.4 An Application: Call-by-Need vs. Call-by-Value Complexity Revisited

Call-by-value and call-by-name are well-known reduction strategies in λ-calculi. Consider an application $(\lambda x.M)\, M'$, then using a call-by-value strategy the actual parameter M' is first evaluated before being substituted for the formal parameter x in the function body M. In contrast, under a call-by-name strategy x is replaced immediately by M' in M. Both strategies are not *optimal* in that they may perform unnecessary work, either if x does not occur in M at all, or else if there are multiple occurrences of x. Call-by-need improves on the by-name strategy by replacing x not directly by M' but rather by a reference to M', and replacing M' by its value as soon as this has been computed for the first time. Therefore, it is "obvious" that evaluation of any term M under call-by-need takes at most as many computational steps as needed using call-by-value. However, a formal *proof* is not at all straightforward, the problem being the different scheduling of corresponding computation steps.

Our language is expressive enough to encode call-by-value and call-by-need λ-calculi such that the complexity is preserved, while concurrency allows for the reordering of computational steps that is necessary for what might be phrased *transforming call-by-need into call-by-value.*

For a configuration (C, B) we define its *complexity* by

$$\mathcal{C}(C, B) =_{\mathsf{df}} \sup \{ m \mid m \text{ is the number of } \xrightarrow{\langle\beta,x\rangle}\text{-reductions in a partial execution of } (C, B) \}$$

As in the preceding section on strong normalization, the crucial point for the following is that all (complete) executions of a configuration have the same length, by uniform confluence of (untyped) λ^F-reduction $\rightarrow$. Thus, for all configurations (C, B) the complexity $\mathcal{C}(C, B)$ equals the length of *any* of its complete executions, which may be finite or infinite.

Syntax $M \;::=\; x \mid \lambda x.M \mid M\,M$

$\qquad\quad V \;::=\; \lambda x.M$

Contexts $E \;::=\; [\,] \mid E\,M \mid V\,E$

Reduction $E[(\lambda x.M)\,V] \;\to_{\text{val}}\; E[M[V/x]]$

Encoding

$$\begin{aligned}
[\![x]\!]^{val} &=_{\mathsf{df}} x\\
[\![\lambda x.M]\!]^{val} &=_{\mathsf{df}} \lambda x.[\![M]\!]^{val}\\
[\![M\;N]\!]^{val} &=_{\mathsf{df}} [\![M]\!]^{val}\,(\texttt{concur}\;[\![N]\!]^{val})
\end{aligned}$$

Figure 4.2: The call-by-value λ-calculus λ_{val} with standard reduction

4.4.1 Encoding the Call-by-Value λ-calculus

We define an embedding $[\![\cdot]\!]^{val}$ of the call-by-value λ-calculus with standard reduction [Plo75], which is presented in Figure 4.2. Basically, the embedding relaxes the order in which subexpressions are reduced. Intuitively, allowing arguments to be evaluated concurrently will permit to match β-reduction steps of call-by-need and call-by-value strategy. By uniform confluence, this reordering does not change complexity. For the encoding, we observe that $[\![M]\!][{}^{[\![V]\!]}/x] = [\![M[{}^{V}/x]]\!]$. Thus, the encoding naturally extends to contexts E so that $[\![E]\!]^{val}\;[[\![M]\!]^{val}] = [\![E[M]]\!]^{val}$. For a λ-term M, let

$$\mathcal{C}^{val}(M) =_{\mathsf{df}} \sup\,\{m \mid m \text{ is the length of a partial execution of } M\}$$

be its *by-value complexity*. Since only β-reduction steps contribute to the complexity of a configuration, $\mathcal{C}^{val}(M) = \mathcal{C}([\![M]\!]^{val})$ is intuitively clear. It can be shown formally using a simulation similar to that used in the preceding section.

Lemma 4.4.1. *For all closed λ-terms M, $\mathcal{C}^{val}(M) = \mathcal{C}([\![M]\!]^{val})$.*

4.4.2 Implementing Call-by-Need

We proceed in a similar fashion for the call-by-need strategy. It is *defined* using `byneed` annotations, analogous to the embedding of by-value:

$$[\![M_1\;M_2]\!]^{need} =_{\mathsf{df}} [\![M_1]\!]^{need}\,(\texttt{byneed}\;[\![M_2]\!]^{need})$$

The encoding is summarized in Figure 4.3. This is reasonably close to formal models of call-by-need proposed previously in the literature, e.g. the λ_{need}-calculus in [AFM+95]. Note, however, that our evaluation contexts are slightly more restrictive than those of the λ_{need}-calculus, with the result that lookup is "lazy". We leave it open whether λ_{need}-calculus can be encoded s.t. time complexity is preserved.

In analogy to $\mathcal{C}^{val}$, the byneed complexity $\mathcal{C}^{need}(M)$ of a λ-term M is defined by

$$\mathcal{C}^{need}(M) =_{\mathsf{df}} \mathcal{C}([\![M]\!]^{need})$$

Encoding
$$\begin{array}{rcl} [\![x]\!]^{need} & =_{\mathsf{df}} & x \\ [\![\lambda x.M]\!]^{need} & =_{\mathsf{df}} & \lambda x.[\![M]\!]^{need} \\ [\![M\ N]\!]^{need} & =_{\mathsf{df}} & [\![M]\!]^{need}\ (\texttt{byneed}\ [\![N]\!]^{need}) \end{array}$$

Figure 4.3: Implementing the call-by-need strategy

4.4.3 Relating Call-by-Need and Call-by-Value

We are now in the position to sketch the proof of the result relating call-by-value and call-by-need complexity, using the encodings of the previous sections. Let $[\![\cdot]\!]$ be the map that replaces (sub-) expressions `byneed` e with `concur` e. Let $\mathcal{S}_{\text{val}}^{\text{need}}$ be the relation consisting of all pairs of configurations $\langle (C, B), (C', B')\rangle$ s.t. B' is empty and for all $x \mapsto e$ in $C \cup B$ there is $x \mapsto e'$ in C' with $e' \equiv [\![e]\!]$. So in particular,

$$\langle [\![M]\!]^{need},\ [\![M]\!]^{val}\rangle \in \mathcal{S}_{\text{val}}^{\text{need}}$$

for all λ-terms M.

It is easy to check that $\mathcal{S}_{\text{val}}^{\text{need}}$ indeed is a *lengthening simulation* in the sense that whenever

$$\langle (C_1, B_1),\ (C_2, B_2)\rangle \in \mathcal{S}_{\text{val}}^{\text{need}} \text{ and } (C_1, B_1) \xrightarrow{\langle\beta,x\rangle} (C_1', B_1')$$

then there exists (C_2', B_2') such that

$$(C_2, B_2) \xrightarrow{\langle\beta,x\rangle} (C_2', B_2') \text{ and } \langle (C_1', B_1'),\ (C_2', B_2')\rangle \in \mathcal{S}_{\text{val}}^{\text{need}}$$

and if

$$(C_1, B_1) \rightarrow (C_1', B_1') \text{ not by } (\beta)$$

there exists (C_2', B_2') such that

$$(C_2, B_2) \rightarrow^* (C_2', B_2') \text{ for some } \langle (C_1', B_1'),\ (C_2', B_2')\rangle \in \mathcal{S}_{\text{val}}^{\text{need}}$$

Hence, $\langle (C_1, B_1),\ (C_2, B_2)\rangle \in \mathcal{S}_{\text{val}}^{\text{need}}$ implies $\mathcal{C}(C_1, B_1) \leq \mathcal{C}(C_2, B_2)$. From this, Lemma 4.4.1 and the definition of $\mathcal{C}^{\text{need}}$ we then obtain

Proposition 4.4.2. *For all terms M of the λ-calculus, $\mathcal{C}^{need}(M) \leq \mathcal{C}^{val}(M)$.*

Chapter 5

Linear Types for Promises

In this section, we investigate properties of the full calculus λ^{FP} again, including promises. A linear type system is presented that enforces the single-assignment discipline that we want to hold for promises. We show that uniform confluence extends to configurations well-typed with respect to the system.

5.1 Promises, Single-assignment and Confluence

Recall that a promise y is created, along with its associated future x, by

$$\texttt{prom}\, y\, \texttt{for}\, x\, \texttt{in}\, e$$

formalized by the axiom $\texttt{prom}\, e \xrightarrow{\mathbf{P}yx} eyx$. Then, y may be applied to some value v, thereby binding x to v. This was formalized by the transition rule

$$\textit{(fulfill)} \qquad\qquad E[y\ v] \xrightarrow{\mathbf{F}yv} E[\texttt{()}]$$

in Chapter 3. This rule is applicable to $E[y\ v]$ in a configuration provided $x \notin \mathsf{dom}(C)$, for the unique x such that $y = P(x)$. The intention is to bind x at most *once*, viewing it as a read-only logic variable that initially carries no value. Consequently, y should be applied no more than once, which is ensured by the side-condition $x \notin \mathsf{dom}(C)$.

Note the difference between `concur`- and `byneed`-futures on the one hand and promises on the other. While $\texttt{concur}\ x{=}e\ \texttt{in}\ e'$ and $\texttt{byneed}\ x{=}e\ \texttt{in}\ e'$ introduce x along with the description e of its value, $\texttt{prom}\, y\, \texttt{for}\, x\, \texttt{in}\, e$ introduces x without value description, but rather with the promise to supply a value in due time.

Put differently, "possession" of a promise y should be seen as the permission to write the logic variable, while x allows only for reading its contents. Generally, if several concurrently running threads have permission to bind x then this is a source of (unwanted) indeterminism. For example, consider

$$\texttt{prom}\, y\, \texttt{for}\, x\, \texttt{in}\, (\texttt{concur}\, x_2{=}y\ v_2\ \texttt{in}\ y\ v_1) \qquad (5.1)$$

which eventually results in the configuration

$$(\{x{\mapsto}y\}, \{x_1{\mapsto}y\ v_1,\ x_2{\mapsto}y\ v_2\}, \emptyset)$$

which may evolve to either of the (possibly distinct) configurations

$$(\{x{\mapsto}y\}, \{x{\mapsto}v_1, x_1{\mapsto}\texttt{()}, x_2{\mapsto}y\ v_2\}, \emptyset)$$

and

$$(\{x{\mapsto}y\}, \{x{\mapsto}v_2, x_1{\mapsto}y\ v_1, x_2{\mapsto}\texttt{()}\}, \emptyset)$$

So the future x could end up being bound to either of v_1, v_2. Perhaps worse, one of the concurrent threads $y\ v_1$, $y\ v_2$ is sure to get stuck as soon as x has been bound by the other thread. The simple example (5.1) already demonstrates that "improper" use of promises spoils confluence and introduces indeterminism.

In contrast, as long as it is at most once attempted to apply the function y in the course of reduction, this will capture exactly the single-assignment behaviour we expect of promises, and it will render reduction of programs confluent. But this usage of promises requires discipline when programming. Usually this is hard, if not impossible, to maintain. It would be much better if there was a way to enforce this discipline, and indeed in the next section a refined type system is presented that takes care of proper usage of promises.

5.2 Linear Types for Promises

In this section, a linear type system is presented. It refines the type system for simple types from Section 3.5 by mode annotations, introducing constraints on the usage of promises. The fragment of the language that is well-typed with respect to this type system will be shown to treat promises properly, in that all the promises are attempted to be assigned at most once. Clearly this property is not decidable in general, there is an easy reduction from the halting problem of the simply typed λ-calculus with explicit recursion operator[1]: If e_2 is an expression using a promise assignment twice, such as (5.1), then $(\lambda x.e_2)\,e$ is admissible if and only if e is not terminating, so this gives a reduction from the (complement of) the set of terminating λ-terms with explicit recursion.

We proceed as follows: First the linear type system is presented, along with some examples demonstrating its usage. Next, a Subject Reduction lemma is proved that shows invariance of typing under reduction. Finally we state that well-typing indeed implies proper use of promises.

5.2.1 Linear Types

We distinguish types σ and annotated types σ^1. Intuitively, if a value is of annotated, or *linear*, type σ^1, it may be applied only once, whereas use of values of type σ is not restricted in any way.

$$\sigma ::= \mathsf{unit} \mid \alpha \mid \sigma_1^{\mu_1} \to \sigma_2^{\mu_2}$$
$$\mu ::= \epsilon \mid 1$$

The type rules are presented in Figure 5.1. A judgment is of the form $\Gamma \vdash e{:}\sigma^\mu$, where the type environment Γ is a finite set written as the sequence

$$\Gamma = x_1{:}\sigma_1^{\mu_1}, \ldots, x_n{:}\sigma_n^{\mu_n}$$

[1]Recall that recursion can be expressed using promises.

$$\frac{}{\Gamma, x{:}\sigma^\mu \vdash x{:}\sigma^\mu}\ \text{(var)} \qquad \frac{\Gamma \vdash e{:}\sigma}{\Gamma \vdash e{:}\sigma^1}\ \text{(weak)}$$

$$\frac{\Gamma \vdash e{:}\sigma^\mu}{\Gamma \vdash \texttt{concur}\ e : \sigma^\mu}\ \text{(conc)} \qquad \frac{}{\Gamma \vdash \texttt{()}{:}\mathsf{unit}^\mu}\ \text{(unit)}$$

$$\frac{\Gamma \vdash e{:}\sigma^\mu}{\Gamma \vdash \texttt{byneed}\ e : \sigma^\mu}\ \text{(need)} \qquad \frac{\Gamma' \subseteq \Gamma, x{:}\sigma_1^{\mu_1} \quad \Gamma' \vdash e{:}\sigma_2^{\mu_2}}{\Gamma \vdash \lambda x.e : (\sigma_1^{\mu_1} \to \sigma_2^{\mu_2})^\mu}\ \text{(abs)}$$

$$\frac{\begin{array}{c}\mathbf{once}(\Gamma) \supseteq \mathbf{once}(\Gamma_1) \uplus \mathbf{once}(\Gamma_2) \\ \Gamma_1 \vdash e_1{:}(\sigma_1^{\mu_1} \to \sigma_2^{\mu_2})^\mu \quad \Gamma_2 \vdash e_2{:}\sigma_1^{\mu_1} \quad \Gamma_i \subseteq \Gamma\end{array}}{\Gamma \vdash e_1\ e_2 : \sigma_2^{\mu_2}}\ \text{(appl)}$$

$$\frac{\Gamma \vdash e{:}(\sigma_1^{\mu_1} \to \mathsf{unit}^1)^1 \to \sigma_1^{\mu_1} \to \sigma^\mu}{\Gamma \vdash \texttt{prom}\, e : \sigma^\mu}\ \text{(prom)}$$

Figure 5.1: Rules for the linear mode system

$x{:}\sigma^1$ in Γ will mean the variable may be used only once, and this is enforced essentially by splitting these *resources* in the antecedent of rule (appl). For a given type environment Γ, the set **once**(Γ) consists of the variables of linear type in Γ,

$$\mathbf{once}(\Gamma) =_{\mathsf{df}} \{x{:}\sigma^\mu \in \Gamma \mid \mu = 1\}$$

We will always additionally assume that the following well-formedness condition holds

$$\Gamma \vdash e{:}\sigma^\mu \text{ implies } \mu = 1, \text{ whenever } \mathbf{once}(\Gamma) \neq \emptyset \qquad \text{(wfc)}$$

Also, there is a rule handling conversion of types σ into σ^1. One might devise a similar rule for the conversion back,

$$\frac{\Gamma \vdash e{:}\sigma^1 \quad \mathbf{once}(\Gamma) = \emptyset}{\Gamma \vdash e{:}\sigma}\ \text{(gen)}$$

but this turns out to be unsound in that types are not preserved under reduction. A simple example for this is

$$\texttt{prom}\,(\lambda y.\lambda x.y)$$

which is a closed term, and so could be given type $(\sigma \to \mathsf{unit}^1)$, using rule (gen). However, this term needs to be given a linear type, otherwise y could be applied more than once, e.g. in

$$(\lambda z.(\ldots z \ldots z \ldots))(\texttt{prom}\,\lambda y.\lambda x.y)$$

Note that the only way to introduce a promise y is by assigning the (linear) type $(\sigma^\mu \to \mathsf{unit}^1)^1$.

The type system is similar to those presented in [TWM95, Mog98], which can be seen as instances of the general framework of type and effect systems [NN99]. However, these systems were introduced to obtain usage information along with type checking, in the sense of program analyses. In particular, programs are typable in the underlying system if and only if they are typable in the annotated system.

In contrast, with the system above we consider only those programs well-typed where usage of promises can be proved linear by the annotated system. This is closer in spirit to the uniqueness typings in the language Clean [BS96].

Examples of Linear Typings

Before passing on to the technicalities needed for the further development we explain the rules of Figure 5.1 by means of a few examples. These show how to handle type derivations in the new system.

Example 1. Consider again the simple confluence counter-example (5.1) from the previous section. This is not typable any more, using the rules given in Figure 5.1. For suppose to the contrary that

$$\Gamma \vdash \texttt{prom}\ p\ \texttt{for}\ x\ \texttt{in}\ (\texttt{concur}\ x'{=}p\ v_2\ \texttt{in}\ p\ v_1){:}\sigma^{\mu}$$

for some Γ, some σ^{μ}. Consider a shortest derivation. In case this ends by (weak), then also $\Gamma \vdash \ldots{:}\sigma^{\mu'}$. Hence, wlog. we can assume that the last rule in the derivation is (prom). By the same reasoning,

$$\Gamma \vdash \lambda p.\lambda x.\texttt{concur}\ x'{=}p\ v_2\ \texttt{in}\ p\ v_1{:}\sigma^{\mu}$$

and then for some $\Gamma' \subseteq \Gamma, p{:}(\sigma' \to \mathsf{unit}^1)^1, x{:}\sigma'$,

$$\Gamma' \vdash \texttt{concur}\ x'{=}p\ v_2\ \texttt{in}\ p\ v_1{:}\sigma^{\mu'}$$

Again, for a shortest derivation we can thus assume it ends by (conc), and, unravelling the definition of $\texttt{concur}\ldots\texttt{in}\ldots$, we know there are $\Gamma_1, \Gamma_2 \subseteq \Gamma'$,

$$\mathbf{once}(\Gamma_1) \cap \mathbf{once}(\Gamma_2) = \emptyset$$

and both

$$\Gamma_1 \vdash p\ v_1{:}\sigma_1^{\mu_1} \text{ and } \Gamma_2, x'{:}\sigma_1^{\mu_1} \vdash p\ v_2{:}\sigma^{\mu''}$$

However, one easily checks that $p \notin \Gamma_1$ implies $\Gamma \not\vdash p\ v_1$, and likewise for $p \notin \Gamma_2$ (see Lemma 5.2.2, stated below on page 56). Thus, the above term is not typable any more.

Example 2. Consider the fixpoint operator **fix**, defined in Section 3.6 on page 29. In the simply typed case, **fix** could be given the type

$$\vdash \mathbf{fix}{:}((\sigma{\to}\sigma) \to (\sigma{\to}\sigma)) \to (\sigma{\to}\sigma)$$

Unfortunately, this type cannot be derived with the rules from Figure 5.1 on the preceding page because **fix** contains a promise. However, it still *is* typable. In fact, by rules (var) and (app),

$$\Gamma_1 \vdash x\ f{:}\sigma \to \sigma$$

for the environment $\Gamma_1 =_{\mathsf{df}} x{:}(\sigma{\to}\sigma) \to (\sigma{\to}\sigma), f{:}\sigma \to \sigma$. Likewise,

$$p{:}((\sigma \to \sigma) \to \mathsf{unit}^1)^1 \vdash p{:}((\sigma \to \sigma) \to \mathsf{unit}^1)^1$$

by (var), and so

$$\Gamma_1, p{:}((\sigma \to \sigma) \to \mathsf{unit}^1)^1 \vdash p\ (x\ f) : \mathsf{unit}^1 \tag{5.2}$$

by rule (app). Similarly, by rules (var) and (weak), we obtain

$$\Gamma_1 \vdash x{:}(\sigma{\rightarrow}\sigma) \rightarrow (\sigma{\rightarrow}\sigma) \text{ and } \Gamma_1 \vdash f{:}(\sigma \rightarrow \sigma)^1$$

Therefore, $\Gamma_1, u{:}\mathsf{unit}^1 \vdash x\ f{:}(\sigma \rightarrow \sigma)^1$ by (app), and from (abs) we conclude

$$\Gamma_1 \vdash \lambda u.x\ f : (\mathsf{unit}^1 \rightarrow (\sigma{\rightarrow}\sigma)^1) \tag{5.3}$$

Using rule (app) on (5.2) and (5.3) yields

$$\Gamma_2 \vdash p\ (x\ f);\ x\ f : (\sigma \rightarrow \sigma^1)$$

for $\Gamma_2 =_{\mathsf{df}} \Gamma_1, p{:}((\sigma \rightarrow \sigma) \rightarrow \mathsf{unit}^1)^1$, and so

$$\vdash \mathbf{fix}{:}((\sigma{\rightarrow}\sigma) \rightarrow (\sigma{\rightarrow}\sigma)) \rightarrow (\sigma{\rightarrow}\sigma)^1$$

by rules (prom) and (abs).

Typing of Configurations

As in the case for simple types, typings are lifted from terms to whole configurations. Obviously, because of linearity there are more constraints on the configuration that must be satisfied.

Definition 5.2.1. *A type environment* Γ *is a* well-typing *for the configuration* (P, C, B), *written* $\Gamma \vdash (P, C, B)$, *if* $\mathsf{var}(P, C, B) \subseteq \mathsf{dom}(\Gamma)$, *and for all* $x_i \in \mathsf{dom}(C) \cup \mathsf{dom}(B)$ *there are* $\Gamma_i \subseteq \Gamma$ *s.t. all of the following hold:*

(L0) whenever $x_i{:}\sigma_i^{\mu_i} \in \Gamma$ *for* $x_i \in \mathsf{dom}(C) \cup \mathsf{dom}(B)$, *then*

$$\Gamma_i \vdash (C \cup B)(x_i){:}\sigma_i^{\mu_i}$$

(L1) for all $x \in \mathsf{dom}(P)$, $\Gamma(x) = \sigma^\mu$ *implies* $\Gamma(P(x)) = (\sigma^\mu \rightarrow \mathsf{unit}^1)^1$

(L2) if $\mathbf{once}(\Gamma_i)$ *is non-empty, then* $\mu_i = 1$ *and whenever* $\mu_i = 1$ *then* x_i *occurs free at most once in* $\mathsf{ran}(C) \cup \mathsf{ran}(B)$; *and*

(L3) $\mathbf{once}(\Gamma_i) \cap \mathbf{once}(\Gamma_j) \neq \emptyset$, *for some* $i \neq j$ *implies*

- $(C \cup B)(x_i)$ *is irreducible and* x_i *is not free in* $\mathsf{ran}(C) \cup \mathsf{ran}(B)$, *or*
- $(C \cup B)(x_j)$ *is irreducible and* x_j *not free in the image* $\mathsf{ran}(C) \cup \mathsf{ran}(B)$ *of* $(C \cup B)$.

Conditions *(L0)* and *(L1)* are the analogue of the consistency conditions on type environments from Chapter 3.5. Condition *(L2)* guarantees that futures associated with linear expressions have linear types, and *(L3)* guarantees that each linear variable occurs in at most one thread. However, due to possible lookups of values, the condition takes the slightly relaxed above form.

5.2.2 Subject Reduction

As in the simply typed case, we need some preliminary lemmas before stating a subject reduction theorem. The following are easily proved by induction on a shortest derivation.

Lemma 5.2.2. *Suppose $\Gamma, x{:}\sigma^1 \vdash e{:}\sigma_1^{\mu_1}$. Then x occurs free in e at most once. Also, if $\Gamma \vdash e{:}\sigma^\mu$ and Γ does not contain x, then e does not contain x free.*

Lemma 5.2.3. *Assume, for some $\Gamma^{(1)}$, $\Gamma^{(2)}$ s.t. $\mathbf{once}(\Gamma^{(1)}) \cap \mathbf{once}(\Gamma^{(2)}) = \emptyset$, $\Gamma^{(1)} \vdash e{:}\sigma^\mu$ holds. If $\Gamma^{(3)} \subseteq \Gamma^{(2)}, x{:}\sigma^\mu$ so that $\Gamma^{(3)} \vdash e_1{:}\sigma_1^{\mu_1}$, then $\Gamma^{(1)} \cup \Gamma^{(3)} \vdash e_1[{}^{e}\!/\!_{x}]{:}\sigma_1^{\mu_1}$.*

Lemma 5.2.4. *Suppose $\Gamma \vdash E[e]{:}\sigma^\mu$. Then also*

$$\Gamma' \vdash e{:}\sigma_1^{\mu_1}, \ \textit{where } \Gamma' \subseteq \Gamma - (\mathbf{once}(\Gamma) - \mathsf{fv}(e))$$

for some σ_1, μ_1. Moreover, for fresh variable x

$$\Gamma'' \vdash E[x]{:}\sigma^\mu, \ \textit{where } \Gamma'' \subseteq \Gamma - (\mathbf{once}(\Gamma) \cap \mathsf{fv}(e)), x{:}\sigma_1^{\mu_1} \ .$$

The last lemma allows to deal with evaluation contexts, splitting the free *linear* variables of a term $E[e]$ in the environments Γ', Γ''. We can use these lemmas to derive the following assertion about β-reduction.

Lemma 5.2.5. *Suppose $\Gamma \vdash E[(\lambda x.e)\ v]{:}\sigma^\mu$. Then for some $\Gamma' \subseteq \Gamma$, $\Gamma' \vdash E[e[{}^{v}\!/\!_{x}]]{:}\sigma^\mu$.*

Proof. By Lemma 5.2.4 there are $\Gamma_1, \Gamma_2 \subseteq \Gamma$ s.t. $\mathbf{once}(\Gamma_1) \cap \mathbf{once}(\Gamma_2) = \emptyset$ and

$$\Gamma_1, y{:}\sigma_1^{\mu_1} \vdash E[y]{:}\sigma^\mu \text{ and } \Gamma_2 \vdash (\lambda x.e)\ v : \sigma_1^{\mu_1}$$

By the rules (app) and (abs), given in Figure 5.1 on page 53, there are $\Gamma_3 \subseteq \Gamma_2, x{:}\sigma_2^{\mu_2}$ and $\Gamma_4 \subseteq \Gamma_2$ so that $\mathbf{once}(\Gamma_3) \cap \mathbf{once}(\Gamma_4) = \emptyset$, and

$$\Gamma_3 \vdash e{:}\sigma_1^{\mu_1} \text{ and } \Gamma_4 \vdash v : \sigma_2^{\mu_2}$$

By Lemma 5.2.3,

$$\Gamma_4, \Gamma_3 \vdash e[{}^{v}\!/\!_{x}]{:}\sigma_1^{\mu_1}$$

and again by Lemma 5.2.3,

$$\Gamma_1, \Gamma_3, \Gamma_4 \vdash E[e[{}^{v}\!/\!_{x}]]{:}\sigma^\mu$$

□

With the help of the preceding lemmas, we can now prove a Subject Reduction lemma for linearly typed configurations, the proof of which is somewhat involved, though not difficult.

Lemma 5.2.6. *Assume $\Gamma \vdash (P, C, B)$ and $(P, C, B) \to (P', C', B')$. Then, for some $\Gamma' \supseteq \Gamma$, $\Gamma' \vdash (P', C', B')$.*

Proof. We consider cases according to the rule by which $(P, C, B) \to (P', C', B')$ has been inferred.

- Suppose $(P, C, B) \xrightarrow{\langle \beta, x_i \rangle} (P', C', B')$ by (β). By definition,

$$C(x_i) = E[(\lambda x.e)\ v] \xrightarrow{\beta} E[e[{}^{v}\!/\!{}_{x}]] = C'(x_i) \ .$$

By assumption on Γ, there exists $\Gamma_i \subseteq \Gamma$ satisfying *(L0)-(L3)*. In particular,

$$\Gamma_i \vdash E[(\lambda x.e)\ v]{:}\sigma^{\mu}$$

where $x_i{:}\sigma^{\mu}$ in Γ, and by Lemma 5.2.5

$$\Gamma_i' \vdash E[e[{}^{v}\!/\!{}_{x}]]{:}\sigma^{\mu}$$

for some $\Gamma_i' \subseteq \Gamma_i$. From this it follows that conditions *(L0)-(L3)* are satisfied, and thus $\Gamma \vdash (P', C', B')$ as required.

- In the case where $(P, C, B) \xrightarrow{\langle \mathbf{L}xx, x_i \rangle} (P', C', B')$ by rule (*Force*), $\Gamma \vdash (P', C', B')$ trivially holds.

- If $(P, C, B) \xrightarrow{\langle \mathbf{C}x_k e, x_i \rangle} (P', C', B')$ by rule (*Concur*), then there is $x_i \in \mathsf{dom}(C)$ and fresh variable x_k s.t.

$$C(x_i) = E[\texttt{concur}\ e] \xrightarrow{\mathbf{C}x_k e} E[x_k] = C'(x_i)$$

By assumption, there is $\Gamma_i \subseteq \Gamma$ so that

$$\Gamma_i \vdash E[\texttt{concur}\ e]{:}\sigma^{\mu}$$

and by Lemma 5.2.4 there are subsets $\Gamma_i' \subseteq \Gamma_i$ and $\Gamma_i'' \subseteq \Gamma_i, x_k{:}\sigma_1^{\mu_1}$ s.t. $\mathbf{once}(\Gamma_i') \cap \mathbf{once}(\Gamma_i'') = \emptyset$ and both

$$\Gamma_i' \vdash \texttt{concur}\ e{:}\sigma_1^{\mu_1} \text{ and } \Gamma_i'' \vdash E[x_k]{:}\sigma^{\mu}$$

If we let $\Gamma_k =_{\mathsf{df}} \Gamma_i'$ and consider Γ_i'' instead of Γ_i, then it is easily seen that $\Gamma, x_k{:}\sigma_1^{\mu_1}$ satisfies conditions *(L0)-(L3)* and so $\Gamma, x_k{:}\sigma_1^{\mu_1} \vdash (P', C', B')$

- The case for $\xrightarrow{\langle \mathbf{B}x_k e, x_i \rangle}$ by (*Byneed*) is analogous.

- If $(P, C, B) \xrightarrow{\langle \mathbf{L}x_j v, x_i \rangle} (P', C', B')$, by (*Lookup*), then there are $x_i, x_j \in \mathsf{dom}(C)$ s.t. $C(x_j) = v'$ and

$$C(x_i) = E[x_j\ v] \xrightarrow{\mathbf{L}x_j v} E[v'\ v] = C'(x_i)$$

Moreover, there are $\Gamma_i, \Gamma_j \subseteq \Gamma$ chosen with respect to *(L0)-(L3)* so that

$$\Gamma_i \vdash E[x_j\ v]{:}\sigma_1^{\mu_1} \text{ and } \Gamma_j \vdash v'{:}\sigma_2^{\mu_2}$$

In the case where $\mathbf{once}(\Gamma_j)$ is empty we are done, for then $\Gamma_i, \Gamma_j \vdash E[v'\ v]{:}\sigma_1^{\mu_1}$ by Lemma 5.2.3 and *(L0)-(L3)* are satisfied for $\Gamma' =_{\mathsf{df}} \Gamma$.

So suppose $\mathbf{once}(\Gamma_j) \neq \emptyset$. By *(L2)*, $\Gamma(x_j) = \sigma_2^1$ and x_j occurs at most once free in the configuration. As it indeed *does* occur free in $C(x_i)$ and $C(x_i)$ is not irreducible, then

$$\mathbf{once}(\Gamma_i) \cap \mathbf{once}(\Gamma_j) = \emptyset$$

by *(L3)*. Thus we know that by Lemma 5.2.3

$$\Gamma_i' \vdash E[v'\ v]{:}\sigma_1^{\mu_1}$$

for some $\Gamma_i' \subseteq \Gamma_i \cup \Gamma_j$. Also, as x_j has a linear type and occurs in $C(x_i)$, $\mathbf{once}(\Gamma_i)$ is nonempty, by Lemma 5.2.2. Hence, conditions *(L0)-(L2)* hold for Γ and (P', C', B'). Now

$$\mathbf{once}(\Gamma_i') \cap \mathbf{once}(\Gamma_j) \neq \emptyset$$

but $C(x_j) = v'$ is irreducible and the only occurrence of x_j has been replaced. So finally suppose $\mathbf{once}(\Gamma_k) \cap \mathbf{once}(\Gamma_i')$ is nonempty, i.e.

$$\begin{aligned}\emptyset &\neq \mathbf{once}(\Gamma_i') \cap \mathbf{once}(\Gamma_j)\\ &= (\mathbf{once}(\Gamma_i) \cap \mathbf{once}(\Gamma_k)) \cup (\mathbf{once}(\Gamma_j) \cap \mathbf{once}(\Gamma_k))\end{aligned}$$

So $\mathbf{once}(\Gamma_i) \cap \mathbf{once}(\Gamma_k)$ is nonempty or $\mathbf{once}(\Gamma_j) \cap \mathbf{once}(\Gamma_k)$ is nonempty. Hence, $(C \cup B)(x_k)$ must be irreducible and condition *(L3)* for Γ and (P', C', B') follows from *(L3)* for (P, C, B).

- Next suppose $(P, C, B) \xrightarrow{\langle \mathbf{P} y x, x_i \rangle} (P', C', B')$, by (*Prom*). So there is $x_i \in \mathsf{dom}(C)$ and $x, y \notin \mathsf{var}((P, C, B))$ such that

  $$C(x_i) = E[\mathtt{prom}\, e] \xrightarrow{\mathbf{P} y x} E[e\, y\, x] = C'(x_i)$$

  By assumption there exists Γ_i satisfying *(L0)-(L3)* and

  $$\Gamma_i \vdash E[\mathtt{prom}\, e]{:}\sigma^1$$

  By Lemma 5.2.4 there are $\Gamma_1, \Gamma_2 \subseteq \Gamma_i$ s.t. $\mathbf{once}(\Gamma_1) \cap \mathbf{once}(\Gamma_2) = \emptyset$ and

  $$\Gamma_1, z{:}\sigma_2^{\mu_2} \vdash E[z]{:}\sigma^1 \text{ and } \Gamma_2 \vdash \mathtt{prom}\, e : \sigma_2^{\mu_2}$$

  for fresh z. Hence

  $$\Gamma_2 \vdash e : (\sigma_1^{\mu_1} {\to} \mathsf{unit}^1)^1 \to \sigma_1^{\mu_1} \to \sigma_2^{\mu_2}$$

  for some $\sigma_1^{\mu_1}$, by rule (prom). So letting $\Gamma' =_{\mathsf{df}} \Gamma, y{:}(\sigma_1^{\mu_1} {\to} \mathsf{unit}^1)^1, x{:}\sigma_1^{\mu_1}$ and $\Gamma_2' =_{\mathsf{df}} \Gamma_2, y{:}(\sigma_1^{\mu_1} {\to} \mathsf{unit}^1)^1, x{:}\sigma_1^{\mu_1}$, by (app) and Lemma 5.2.3 we obtain

  $$\Gamma_1, \Gamma_2' \vdash E[e\, y\, x]{:}\sigma^1$$

  By Lemma 5.2.2, y occurs free at most once and so $\Gamma' \vdash (P', C', B')$.

- Finally, suppose $(P, C, B) \xrightarrow{\langle \mathbf{F}yv, x_i \rangle} (P', C', B')$, by (*Fulfill*). Then

  $$C(x_i) = E[y\ v] \xrightarrow{\mathbf{F}yv} E[()] = C'(x_i)\ ,$$

  assuming that for $x_k \notin \mathsf{dom}(C)$ unique x_k with $y = P(x_k)$. By *(L1)*,

  $$\Gamma(x_k) = \sigma^\mu \text{ and } \Gamma(y) = (\sigma^\mu \to \mathsf{unit}^1)^1$$

  Also, there exists $\Gamma_i \subseteq \Gamma$ satisfying

  $$\Gamma_i \vdash E[y\ v]{:}\sigma_2^{\mu_2}$$

  and *(L0)*-*(L3)*, so by Lemma 5.2.4 there are $\Gamma_1, \Gamma_2 \subseteq \Gamma_i$ s.t. $\mathbf{once}(\Gamma_1) \cap \mathbf{once}(\Gamma_2) = \emptyset$ and

  $$\Gamma_1 \vdash v{:}\sigma^\mu \text{ and } \Gamma_2 \vdash E[()]{:}\sigma_2^{\mu_2}$$

  If we define $\Gamma_i' =_{\mathsf{df}} \Gamma_1$ and $\Gamma_k =_{\mathsf{df}} \Gamma_2$ then conditions *(L0)*-*(L3)* are satisfied, and so $\Gamma \vdash (P', C', B')$. This concludes the proof.

□

Corollary 5.2.7. *Each promise y introduced in the reduction of a well-typed term $\vdash e{:}\sigma^\mu$ is fulfilled at most once.*

Proof. Suppose there is a first application of promise y,

$$e \to^* (P, C, B) \xrightarrow{\langle \mathbf{F}yv, x_i \rangle} (P', C', B)$$

Then $C(x_i) = E[y\ v]$ and both, $P(x) = y$, $x \notin \mathsf{dom}(C)$ and

$$C'(x_i) = E[()] \text{ and } C'(x) = v$$

By Subject Reduction, Lemma 5.2.6, $\Gamma \vdash (P, C, B)$, and therefore $\Gamma_i \vdash E[y\ v]$ for some $\Gamma_i \subseteq \Gamma$. Observing that $\Gamma_i(y) = \Gamma(y)$ is a linear type and applying Lemma 5.2.2 we see that there is exactly this single occurrence of y in $C(x)$. Moreover, by condition *(L3)* on typings, whenever y additionally occurs in some $(C \cup B)(x_j)$, then $(C \cup B)(x_j)$ is irreducible and will stay so (as it cannot come into evaluation contexts through lookups again). So this is the only application of y in the evaluation of e. □

5.3 Uniform Confluence of Linearly Typed λ^{FP}

One of the main goals of this chapter is to show that proper use of promises results in a well-behaved language. Under the assumption, asserted by Corollary 5.2.7, that for each promised future the respective assignment function y is applied at most once, we can indeed easily conclude confluence. We do this by extending the proof of Lemma 4.2.2.

Lemma 5.3.1. *Let $x_0 \neq x_1$ be distinct variables and suppose there are two transitions*

$$\begin{tikzcd}
(P, C, B) \arrow[r, "\langle\alpha_1,x_1\rangle"] \arrow[d, "\langle\alpha_0,x_0\rangle"'] & (P_1, C_1, B_1) \\
(P_0, C_0, B_0) & \arrow[from=1-1, phantom, "\neq"]
\end{tikzcd}$$

Then there exists (C', B') such that the above diagram can be closed to the following

$$\begin{tikzcd}
(P, C, B) \arrow[r, "\langle\alpha_1,x_1\rangle"] \arrow[d, "\langle\alpha_0,x_0\rangle"'] & (P_1, C_1, B_1) \arrow[d, dashed, "\langle\alpha_0,x_0\rangle"] \\
(P_0, C_0, B_0) \arrow[r, dashed, "\langle\alpha_1,x_1\rangle"] & (P', C', B')
\end{tikzcd}$$

Proof. We extend the proof from Lemma 4.2.2, considering the missing combinations of reductions. Of course, the interesting case is that $\alpha_0 = \mathbf{F}yv_0$ and $\alpha_1 = \mathbf{F}yv_1$ is prevented.

- Suppose $\alpha_0 = \mathbf{F}y_0v_0$ and $\alpha_1 = \mathbf{F}y_1v_1$. If $y_0 = P(x) = y_1$ for some $x \in \mathsf{dom}(P)$, then by Corollary 5.2.7 the promise y_0 is applied at most once. So in fact, we must have $x_0 = x_1$, contradicting the assumption that x_0 and x_1 are distinct. Thus, no such transitions are possible.

  If $P(x) = y_0 \neq y_1 = P(x')$, then $x \neq x'$. Hence, $P_0 = P = P_1$, $B_0 = B = B_1$,and if

  $$C(x_0) = E_0[y_0\, v_0] \text{ and } C(x_1) = E_1[y_1\, v_1]$$

  then $C_0 = C[x_0{\mapsto}E_0[\texttt{()}]]$ and $C_1 = C[x_1{\mapsto}E_1[\texttt{()}]]$. Hence

  $$\begin{aligned}(P_0, C_0, B_0) &\xrightarrow{\mathbf{F}y_1v_1} (P, C[x_0{\mapsto}E_0[\texttt{()}],\, x_1{\mapsto}E_1[\texttt{()}],\, x{\mapsto}v_0,\, x'{\mapsto}v_1], B)\\ &\xleftarrow{\mathbf{F}y_0v_0} (P_1, C_1, B_1)\end{aligned}$$

- Now consider the case $\alpha_0 = \mathbf{C}xe$ and $\alpha_1 = \mathbf{F}yv$. Then $B_0 = B = B_1$, $P_0 = P = P_1$, and

  $$C_0 = C[x_0{\mapsto}E_0[x],\, x{\mapsto}e] \text{ and } C_1 = C[x_1{\mapsto}E_1[\texttt{()}],\, x'{\mapsto}v]$$

  provided $y = P(x')$, $C(x_0) = E_0[\texttt{concur}\ e]$, and $C(x_1) = E_1[y\, v]$. Therefore,

  $$\begin{aligned}(P_0, C_0, B_0) &\xrightarrow{\mathbf{F}yv} (P, C[x_0{\mapsto}E_0[x],\, x{\mapsto}e,\, x_1{\mapsto}E_1[\texttt{()}],\, x'{\mapsto}v], B)\\ &\xleftarrow{\mathbf{C}xe} (P_1, C_1, B_1)\end{aligned}$$

The remaining cases are similar. □

These considerations suffice to obtain the following Uniform Confluence result.

Proposition 5.3.2 (Uniform Confluence). *Reduction $\rightarrow$ on the set of well-typed configurations is uniformly confluent. In particular, it is confluent.*

Proof. The result follows from Lemma 4.2.1 on page 36 and the above Lemma 5.3.1, exactly as in the proof of Lemma 4.2.3. □

Types $\sigma ::= \ldots \mid \sigma\ \mathsf{ref}$

Syntax $e ::= \ldots \mid \texttt{cell}\ e \mid \texttt{exchange(}e_1\texttt{,}e_2\texttt{)}$

Labels $\alpha ::= \ldots \mid \mathbf{S}xe \mid \mathbf{E}xee'$

Type rules

$$\frac{\Gamma \vdash e{:}\sigma}{\Gamma \vdash \texttt{cell}\ e{:}\sigma\ \mathsf{ref}}\ \text{(cell)}$$

$$\frac{\Gamma \vdash e_1{:}\sigma\ \mathsf{ref} \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash \texttt{exchange(}e_1\texttt{,}e_2\texttt{)}{:}\sigma}\ \text{(exch)}$$

Configuration $(S, P, C, B) \in [\mathit{Var} \rightarrow_{\mathit{fin}} \mathit{Val}]^2 \times [\mathit{Var} \rightarrow_{\mathit{fin}} \mathit{Exp}]^2$

Contexts $E ::= \ldots \mid \texttt{cell}\ E \mid \texttt{exchange(}E\texttt{,}e\texttt{)} \mid \texttt{exchange(}v\texttt{,}E\texttt{)}$

Reduction

$(\mathit{lookup'})$ $\quad E[\texttt{exchange(}x\texttt{,}v'\texttt{)}] \xrightarrow{\mathbf{L}xv} E[\texttt{exchange(}v\texttt{,}v'\texttt{)}]$

(cell) $\quad E[\texttt{cell}\ v] \xrightarrow{\mathbf{S}xv} E[x]$

(exch) $\quad E[\texttt{exchange(}x\texttt{,}v\texttt{)}] \xrightarrow{\mathbf{E}xvv'} E[v']$

$$(\mathit{Cell})\quad \frac{C \xrightarrow{\langle \mathbf{S}xv,y\rangle} C' \quad x \notin \mathsf{var}(S,P,C,B)}{(S,P,C,B) \xrightarrow{\langle \mathbf{S}xv,y\rangle} (S[x{\mapsto}v],P,C',B)}$$

$$(\mathit{Exch})\quad \frac{C \xrightarrow{\langle \mathbf{E}xvv',y\rangle} C'}{(S[x{\mapsto}v'],P,C,B) \xrightarrow{\langle \mathbf{E}xvv',y\rangle} (S[x{\mapsto}v],P,C',B)}$$

Figure 5.2: Syntax, type and reduction rules for reference cells

5.4 Extending the Language

As the results of the previous sections show, so far we do not have full "concurrent expressivity", in that reduction is necessarily confluent, provided promises are used properly, i.e. are applied at most once. In particular, so far it is not possible to program many-to-one or many-to-many communication as this necessarily introduces indeterminism.

To this end, we simply extend the language with reference cells, obtaining the language λ^{FPC}. A cell is a *mutable* piece of data, so this extension indeed will take us outside of the confluent setting. In contrast to, e.g., ML `ref`-types, we do not directly allow for separate update and dereferencing but only provide an atomic exchange operation. This is fairly standard in concurrent resp. parallel systems. However, in a language with promises this is not really a restriction since update and dereferencing can be easily coded.

The syntax is extended by two new constructs dealing with cells.

$$\texttt{cell}\ v,$$

introduces a new cell carrying the initial value v, and

$$\texttt{exchange(}x\texttt{,}v\texttt{)},$$

which updates the cell x to v while returning the old value of x. Again, we may use derived forms,

$$\texttt{let } x\texttt{=cell(}e\texttt{) in } e' =_{\mathsf{df}} (\lambda x.e')\ (\texttt{cell } e)\ .$$

Further, the new type constructor ref is introduced. The type rules ensure that reference cells carrying values of type σ are given type σ ref. Configurations are extended by an additional component $S \in [\mathit{Var} \to_{\mathit{fin}} \mathit{Val}]$ that deals with the bindings of cells to their current contents. Evaluation contexts extend in the obvious way to these new constructs by setting

$$E ::= \dots \mid \texttt{cell } E \mid \texttt{exchange(}E\texttt{,}e\texttt{)} \mid \texttt{exchange(}v\texttt{,}E\texttt{)}$$

To deal correctly with lookup of values bound to a future, as well as forcing of byneed futures, there is an additional axiom

$$(\mathit{lookup}') \qquad E[\texttt{exchange(}x\texttt{,}v'\texttt{)}] \xrightarrow{\mathbf{L}xv} E[\texttt{exchange(}v\texttt{,}v'\texttt{)}]$$

This axiom reflects the fact that exchange is strict in its first argument. In fact, a similar treatment would be necessary when considering, e.g. integer addition. The axioms for cells are

$$(\mathit{cell}) \qquad E[\texttt{cell } v] \xrightarrow{\mathbf{S}xv} E[x]$$

and

$$(\mathit{exch}) \qquad E[\texttt{exchange(}x\texttt{,}v\texttt{)} \xrightarrow{\mathbf{E}xvv'} E[v']$$

The corresponding inference rules for the reduction relation on configurations are then

$$(\mathit{Cell}) \qquad \frac{C \xrightarrow{\langle \mathbf{S}xv, y\rangle} C' \quad x \notin \mathsf{var}((S,P,C,B))}{(S,P,C,B) \xrightarrow{\langle \mathbf{S}xv, y\rangle} (S[x{\mapsto}v], P, C', B)}$$

and

$$(\mathit{Exch}) \qquad \frac{C \xrightarrow{\langle \mathbf{E}xvv', y\rangle} C'}{(S[x{\mapsto}v'], P, C, B) \xrightarrow{\langle \mathbf{E}xvv', y\rangle} (S[x{\mapsto}v], P, C', B)}$$

A summary of the new constructs along with reduction and type inference rules are given in Figure 5.2.

Although possible in principle, here we will not consider cells in combination with linear types. We just remark that Definition 5.2.1 carries over to extended configurations, replacing (P,C,B) by (S,P,C,B) etc., and also Lemmas 5.2.2-5.2.4 hold again. In fact, Subject Reduction is valid as well, and so is Corollary 5.2.7, giving a safety property for the extended language with respect to proper use of promises. It should be observed, however, that confluence does not hold any more.

As shown by the example in Section 2.3, monomorphic channels $\texttt{channel}_\sigma$ for transmission of elements of type σ (with `send` and `receive`) can indeed be implemented, provided lists and products are added to the language. Indeed, $\texttt{channel}_\sigma$ is typable with respect to the linear type system presented in the preceding sections. Unfortunately, the type for $\texttt{channel}_\sigma$ itself will be linear, and so both `send` and `receive` can be used at most once, which does not lead to any interesting applications. However, we can still infer the expected simple types, i.e.

$$\vdash \texttt{channel}_\sigma{:}(\sigma \rightarrow \mathsf{unit}) \times (\mathsf{unit} \rightarrow \sigma)$$

when extending the inference rules given in Figures 3.4 and 5.2 by rules for list and binary products.

Chapter 6

Discussion and Outlook

6.1 Summary

In this work we introduced a formal model of concurrency with logic variables in the form of futures and promises. The (sequential) core language is an eager functional language, modelled by the call-by-value λ-calculus.

We investigated various properties of this calculus, which suggest that futures and promises form a well-behaved extension of λ-calculus in that it inherits well-known properties: First, the simply typed language λ^F is strongly normalizing. Second, although reduction is clearly not deterministic, we established a powerful confluence result by proving both the typed and untyped promise-free calculus λ^F uniformly confluent.

Moreover, we showed its applicability to more theoretical problems by providing very simple and natural complexity-preserving embeddings of call-by-value- and call-by-need λ-calculus. Concurrency then gave a means of comparing the respective encodings, thus allowing for a formal proof that the complexity of call-by-need reduction is bounded above by call-by-value reduction.

In order to enforce "proper" use of promises, a refined type system has been devised. It allowed to prove all the well-typed expressions of the full calculus λ^{FP} uniformly confluent.

Finally, we discussed an extension made in order to obtain full "concurrent expressiveness," by which we mean the ability to perform many-to-one communication via channels. To this end, cells with an atomic exchange operation have been added. Despite the extension which rendered reduction indeterministic, we indicated that it is possible to guarantee the proper use of promises, using the linear type system.

6.2 Open Problems and Further Work

Several questions remain open, and there are interesting directions for future work. We consider some of these briefly in this section.

Formal Language Definition. First, the work done in this thesis needs to be assessed more closely with respect to its practical applicability. It will be particularly worthwile to incorporate futures and promises, as well as the other extensions of Alice [Ali02], into the formal definition of Standard ML [MTHM97], or rather the alternative definition [HS00] by Harper et al. which is more appropriate for concurrent extensions. This project seems very natural to do, and indeed, one of the reasons to choose SML as base language for Alice was the existence of a formal semantics of the former.

Such work would include an investigation of impure features of both SML and Alice, and how these fit into the concurrent framework. For example, there are several possible design decisions conceivable of how to best treat failure and exception handling in concurrent threads. To demonstrate this, consider the Alice expression

$$\texttt{let val } x \texttt{ = concur (fn \_ => raise } \mathit{exn}\texttt{) in } \mathit{exp} \texttt{ end}$$

The exception *exn* is raised in a second thread, and the future x is replaced by a *failed* future. As soon as x is accessed, the exception is passed on. However, when using threads *imperatively*, i.e. not using the return value, as in

$$\texttt{let val \_ = concur (fn \_ => raise } \mathit{exn}\texttt{) in } \mathit{exp} \texttt{ end}$$

the programmer may be completely unaware of the exception being raised. Therefore, different solutions might be better suited.

Besides exceptions, Alice introduces several impure features necessary for actual programming with futures and promises, such as `isFuture`, `isFailed`, which have not been considered here.

Polymorphic Types and Type Inference. Further questions that were not discussed here include polymorphic and recursive types, as well as type inference. These form an essential part of the design of ML and, certainly, of ML's popularity.

For the join-calculus, underlying the language JoCaml, type inference in the style of Hindley-Milner provides no additional difficulties [FLMR97]. It is well-known, however, that polymorphic references cause problems. A simple and practical solution has been proposed in [Wri95] by restricting type generalizations. Note that essentially the same restrictions apply to promises as well: If x is given the polymorphic type $\forall\alpha.\alpha$, then

$$\texttt{prom } y \texttt{ for } x \texttt{ in } y\ (\lambda z.\texttt{not } z);\ x\ \texttt{1}$$

would be well-typed, but clearly will result in a type conflict at runtime.

Moreover, it remains to be seen how polymorphism and linear modes go together. A promising starting point for an investigation of these issues might be to consider the more general framework of (annotated) type and effect systems [NN99]. Using a two-level approach, it might even be possible to infer principal types.

Expressiveness of Linear Types. A different line of work concerning the linear type system is its expressiveness. For linear types to be of practical use, the class of typable expressions (using promises safely) needs to be extended to cover, e.g., the channels from Section 2.3. Also, there should be some possibility to hide the linear annotations, for example when occurring in library implementations.

So far, we do not know whether and how this can be achieved. Considering some statement of the form "there exists a fresh resource p", similar to the way existential types $\exists\alpha.\tau$ are used in polymorphic λ-calculus [Mit96], might be an idea.

Operational Equivalence. A notion of bisimulation equivalence, possibly capturing observational equivalence of our calculus, would be important. A notion of program equivalence is necessary to justify rewriting of fragments, as usually done, e.g. in compiler optimizations. Besides, it has some applications in specification and subsequent correctness proofs. Work along these lines has been done for Facile in [PGM90], and for a quite comprehensive fragment of CML in [FHJ98, JR00].

The need for such an investigation may become more plausible when noticing that promises ruin equational reasoning. In fact, just as with the I-structures of [ANP89], it is clear that the program fragment

$$\texttt{let}\ z{=}(\texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in}\ y)\ \texttt{in}\ (z, z)$$

semantically differs from

$$(\texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in}\ y, \texttt{prom}\ y\ \texttt{for}\ x\ \texttt{in}\ y),$$

showing that we lose referential transparency of the core language when adding promises.

λ^{FP}, Process Calculus and by-need Models. On the purely theoretical side, the exact relation of our calculus to the established process calculi is not clear. Reusing the δ-calculus and its encoding into π_0, the applicative core of π-calculus, as shown in [Nie99], it may be possible to also translate λ^{FP} into π-calculus.

Also, as indicated in Section 4.4, it would be interesting to see how the λ_{need} calculus of [AFM+95] can be encoded in λ^{FP} in a way that preserves complexity. Finally, we conjecture that the proof of strong normalization given in Chapter 4.3 carries over to the simply typed call-by-let calculus of [MOTW95] without major adaptations.

Finally, as discussed at the end of Chapter 3, there are many promising variations of the calculus. Certainly investigating some of them will yield new insights. In particular, a language with an explicit lookup operation "$\texttt{?}\,e$" seems interesting since it avoids the need to define new lookup axioms for every language extension. Moreover, the `await` found in Alice exactly implements such an operator, so this is clearly relevant for a programming model of Alice.

Bibliography

[ABB$^+$99] L. Augustsson, D. Barton, B. Boutel, F. W. Burton, J. Fasel, K. Hammond, R. Hinze, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Report on the Non-Strict Funtional Language Haskell 98*, February 1999.

[AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[AFM$^+$95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 233–246, January 1995.

[Ali02] *The Alice Project.* Web site at the Programming Systems Lab, Universität des Saarlandes, `http://www.ps.uni-sb.de/alice`, 2002.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[BMT92] Dave Berry, Robin Milner, and David Turner. A semantics for ML concurrency primitives. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–129. ACM, January 1992.

[Bou97a] Gérard Boudol. The π-calculus in direct style. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, January 1997.

[Bou97b] Gérard Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science, Proceedings of ASIAN '97, the Asian Computing Science Conference*, volume 1345 of *LNCS*, pages 239–253. Springer, 1997.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, December 1996.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, October 1999.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional prograias. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[FF96] Cormac Flanagan and Matthias Felleisen. The semantics of futures. Technical Report TR94-238, Rice University, February 1996.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, 1996.

[FHJ98] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for Core CML. *Journal of Functional Programming*, 8(5):447–491, September 1998.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implict typing à la ML for the join-calculus. In *Proc. of the 8th International Conference on Concurrency Theory*. Springer-Verlag, 1997.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[Hal85] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HS00] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM, January 1996.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JR00] A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. In *15th Symposium on Logic in Computer Science*, pages 311–321. IEEE, June 2000.

[Kor01] Leif Kornstaedt. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*. Elsevier, September 2001.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 23(7):260–268, June 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, May 1999.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[Mog98] T. A. Mogensen. Types for 0, 1 or many uses. In *Proceedings of IFL'97*, volume 1467 of *LNCS*, pages 112–123. Springer, 1998.

[MOTW95] John Maraist, Martin Odersky, David Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11th International Conference on the Mathematical Foundations of Programming Semantics*, April 1995.

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–40, 41–77, 1992.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Standard ML Programming Language (Revised).* MIT Press, 1997.

[Nie99] Joachim Niehren. Uniform confluence in concurrent computation (unabridged). Technical report, Programming Systems Lab. Universität des Saarlandes, Germany, June 1999.

[Nie00] Joachim Niehren. Uniform confluence in concurrent computation. *Journal of Functional Programming*, 10(5):453–499, September 2000.

[NN99] F. Nielson and H. R. Nielson. Type and Effect Systems. In E. R. Olderog and B. Steffen, editors, *Correct System Design*, number 1710 in LNCS, pages 114–136. Springer-Verlag, 1999.

[Oka98] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[PE88] Keshav Pingali and Kattamuri Ekanadham. Accumulators: New logic variable abstractions for functional languages. Technical Report TR88-952, Cornell University, Computer Science Department, December 1988.

[PGM90] S. Prasad, A. Giacalone, and P. Mishra. Operational and algebraic semantics of FACILE: A symmetric integration of concurrent and functional programming. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 765–780. Springer, July 1990.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[Plo77] G. Plotkin. LCF considerd as a programming language. *Theoretical Computer Science*, 5:223–225, 1977.

[Pro85] Special issue on Prolog. *Communications of the Association for Computing Machinery*, 28(12), 1985.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[Rep92] John H. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, January 1992.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Sch01] Jan Schwinghammer. On futures, call-by-need and call-by-value complexity. In *Proceedings of GI Informatiktage 2001*, November 2001. To appear.

[Smo95a] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *LNCS*, pages 251–292. Springer, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.

[Smo98] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *LNCS*, pages 1–11. Springer, 1998.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Tho92] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1992.

[Tur85] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorpic Types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 1–16. Springer, 1985.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7th International Conference on Functional Programming and Computer Architecture*, pages 1–11. ACM, June 1995.

[VS$^+$96] Pascal Van Hentenryck, Vijay Saraswat, et al. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701–726, December 1996.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, December 1995.