A Concurrent Lambda Calculus with Futures

Joachim Niehren[1], Jan Schwinghammer[2], and Gert Smolka[2]

[1] INRIA Futurs, LIFL, Lille, France
[2] Programming Systems Lab, Saarland University, Saarbrücken, Germany

Abstract. We introduce a new concurrent lambda calculus with futures, $\lambda(\mathsf{fut})$, to model the operational semantics of Alice, a concurrent extension of ML. $\lambda(\mathsf{fut})$ is a minimalist extension of the call-by-value λ-calculus that yields the full expressiveness to define, combine, and implement a variety of standard concurrency constructs such as channels, semaphores, and ports. We present a linear type system for $\lambda(\mathsf{fut})$ by which the safety of such definitions and their combinations can be proved: Well-typed implementations cannot be corrupted in any well-typed context.

1 Introduction

The goal of this paper is to model the operational semantics of Alice [23, 2], a concurrent extension of Standard ML (SML) [17] for typed open distributed programming. Alice is the first concurrent extension of SML where all synchronisation is based on futures rather than channels [22, 20, 10]. Many ideas in Alice are inspired by and inherited from the concurrent constraint programming language Mozart-Oz [26, 13, 19].

Futures [5, 12] are a restricted form of logic variables, which carefully separate read and write permissions. In contrast to logic variables, futures grant for *static data flow* that can be predicted at compile time. Otherwise, they behave like the logic variables of concurrent logic and concurrent constraint programming [25, 24]: A future is a transparent placeholder for a value; it disappears once its value becomes available. Operations that need the value of a future block until the value becomes available. Other operations may simply continue with the placeholder, as long as they do not need its value. This form of *automatic data driven synchronisation* is invoked as late as possible, so that the potential for concurrent and distributed computation is maximised.

Static data flow is an indispensable prerequisite for static typing as in SML, CAML, or Haskell. This fact is well-known, as it led to serious problems in several previous approaches to concurrent programming: It prohibited static typing in programming languages with unrestricted logic variables such as Oz [18, 26] and in π-calculus based extensions of SML such as Pict [21]. The problem for π-calculus based channel approaches was solved with the join-calculus [10, 11] and the corresponding programming language JoCaml [8] which extends on CAML [7]. The join-calculus, however, does not model futures on which we focus in this paper.

We introduce a new concurrent lambda calculus with futures $\lambda(\mathsf{fut})$ that models the operational semantics of Alice at high level. $\lambda(\mathsf{fut})$ is a minimalist extension of the call-by-value λ-calculus that yields the full expressiveness to define, implement, and combine a variety of standard concurrency constructs, including channels, semaphores, and ports.

Previous λ-calculi with futures by Felleisen and Flanagan [9] were proposed to model the parallel execution of *otherwise purely functional programs*. They too describe a set of parallel threads that communicate through futures. However, our very different perspective on futures as a *uniform mechanism for introducing concurrency* to Alice necessitates a number of nontrivial extensions:

Indeterminism. Standard concurrency constructs are indeterministic, which is incompatible with confluence properties enjoyed by previous λ-calculi with futures. We propose to add indeterminism via reference cells, as these are already available in SML. Furthermore, we propose *handled futures* for single assignment, similarly to the I-structures of Id [4] and promises of [14]. A handled future comes with a handle that can eventually assign a value to a future. Any attempt to use the same handle twice raises a programming error.

Explicit recursion. Similarly to cells, handles permit the construction of cyclic structures. This raises a number of nontrivial technical problems, some of which are known from call-by-need λ-calculi with explicit recursion. Indeed, we can easily extend $\lambda(\mathsf{fut})$ by lazy threads, so that we obtain an elegant model for call-by-need [3, 15] mixed with call-by-value computation.

Static typing. We have to add a type system, as previous λ-calculi with futures were untyped.

We show that $\lambda(\mathsf{fut})$ can *safely* express concurrency constructs, so that these cannot be corrupted in any well-typed context, in that their usage never raises handle errors. We prove this kind of safety result on basis of a linear type system we introduce, inspired by [27].

We present $\lambda(\mathsf{fut})$ in Sects. 2 and 3. The linear type system for $\lambda(\mathsf{fut})$ that excludes handle errors is given in Sect. 4. We then express diverse concurrency constructs in $\lambda(\mathsf{fut})$ (Sect. 5) and prove their safety (Sect. 6). Finally, we briefly discuss some implementation issues in Sect. 7.

2 Lambda Calculus with Futures

We present the lambda calculus with futures $\lambda(\mathsf{fut})$. We start with an untyped version, discuss its syntax and operational semantics.

2.1 Syntax

Fig. 1 introduces the syntax of $\lambda(\mathsf{fut})$. This calculus extends the call-by-value λ-calculus with cells (as featured by SML and CAML) and by concurrent threads,

$$
\begin{aligned}
x, y, z &\in \mathit{Var} \\
c \in \mathit{Const} &::= \textbf{unit} \mid \textbf{cell} \mid \textbf{exch} \mid \textbf{thread} \mid \textbf{handle} \\
e \in \mathit{Exp} &::= x \mid c \mid \lambda x.e \mid e_1\, e_2 \\
v \in \mathit{Val} &::= x \mid c \mid \lambda x.e \mid \textbf{exch}\, v \\
C \in \mathit{Config} &::= C_1 \mathbin{|} C_2 \mid (\nu x)C \mid x\,\mathsf{c}\,v \mid x \Leftarrow e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\,\bullet
\end{aligned}
$$

Fig. 1. Syntax of $\lambda(\mathsf{fut})$

$$
\begin{array}{ll}
(C_1 \mathbin{|} C_2) \mathbin{|} C_3 \equiv C_1 \mathbin{|} (C_2 \mathbin{|} C_3) & C_1 \mathbin{|} C_2 \equiv C_2 \mathbin{|} C_1 \\
((\nu x)C_0) \mathbin{|} C_1 \equiv (\nu x)(C_0 \mathbin{|} C_1) \quad \text{if } x \notin \mathsf{fv}(C_1) & (\nu x)(\nu y)C \equiv (\nu y)(\nu x)C
\end{array}
$$

Fig. 2. Structural congruence

futures, and handles. Expressions and values of $\lambda(\mathsf{fut})$ model sequential higher-order programming; configurations add the concurrency level.

The expressions e of $\lambda(\mathsf{fut})$ are standard λ-terms with variables $x, y, \ldots$ and constants ranged over by c. We introduce 5 constants, 3 of which are standard: **unit** is a dummy value, **cell** serves for introducing reference cells, and **exch** for atomic exchange of cell values. The new constants **thread** and **handle** serve for introducing threads, futures, and handles. Values v are defined as usual in a call-by-value λ-calculus.

Configurations C are reminiscent of expressions of the π-calculus. They are built from base components by parallel composition $C_1 \mathbin{|} C_2$ and new name operators $(\nu x)C$. We distinguish four types of base components: a thread $x \Leftarrow e$ is a concurrent component whose evaluation will eventually bind the future x to the value of expression e unless it diverges or suspends. We call such variables x *concurrent futures*. Note that a concurrent future x may occur in the expression e whose evaluation computes its future value, i.e., a thread is like a recursive equation $x = e$, but directed from right to left. A cell $x\,\mathsf{c}\,v$ associates a name x to a cell with value v. A handle component $y\,\mathsf{h}\,x$ associates a handle y to a future x, so that y can be used to assign a value to x. We call x a future handled by y, or more shortly a *handled futures.* Finally, a used handle component $y\,\mathsf{h}\,\bullet$ means that y is a handle that has already been used to bind its future.

We define free and bound variables as usual; the only scope bearing constructs are λ-binder and new operators (νx). We identify expressions and configurations up to consistent renaming of bound variables. We write $\mathsf{fv}(C)$ and $\mathsf{fv}(e)$ for the free variables of a configuration and expression, resp., and $e[{}^{e'}\!/x]$ for capture-free substitution of e' for x in e.

We do not want the order of components in configurations to matter. Following the presentation of π-calculus in [16] we use a *structural congruence* $\equiv$ to simplify the statement of the operational semantics. This is the least congruence relation on configurations containing the identities in Fig. 2. The first two axioms render parallel composition associative and commutative. The third rule is known as *scope extrusion* in π-calculus and allows to extend the scope of a local variable. The final identity expresses that the order of restricting names does not matter.

2.2 Operational Semantics

The operational semantics of $\lambda(\mathsf{fut})$ is given in Figs. 3, 4 and 5 by a binary relation $C_1 \to C_2$ on configurations called *reduction*.

The reduction strategy of $\lambda(\mathsf{fut})$ is specified using the evaluation contexts defined in Fig. 3. We base it on standard evaluation contexts F for call-by-value reduction and lift them to threads and configurations. Formally, a context is a term with a single occurrence of the *hole marker* $[\,]$ which is a special constant. Evaluation contexts F are expressions where some subexpression in call-by-value reduction position is replaced by the hole marker. A context E is a thread where a subexpression is left out, and D is a configuration where a subconfiguration is missing. We write $D[C]$, $E[e]$, and $F[e]$ resp. for the object obtained by filling the hole in the context with an expression.

A nontrivial question is *when* to allow to replace futures by their values. The naive approach to always do so once the value becomes available fails, in that it introduces non-terminating unfolding in the presence of recursion. For instance, consider a thread $x \Leftarrow \lambda y.xy$. The thread's expression contains an occurrence of the future x whose value the thread has computed. Replacing this occurrence of x by its value yields $x \Leftarrow \lambda y.((\lambda y'.xy')\ y)$ which again contains an occurrence of x because of recursion, so the substitution process can be repeated indefinitely.

Alternatively, one might want to permit future substitution in all evaluation contexts. This approach, however, yields confluence problems. Suppose that x is bound to value 5 by some thread $x \Leftarrow 5$ and that another thread is evaluating the expression $(\lambda y.\lambda z.z)\ x$ which contains an occurrence of x in evaluation position. We could thus first replace x by 5 and then β-reduce, resulting in $\lambda z.5$. Or else, we could β-reduce first, yielding $\lambda z.x$. Now the problem is that the occurrence of x has escaped the evaluation context, so that replacing the future by its value is impossible and we are left with two distinct, irreducible terms.

We propose to replace a future only if its value is needed to proceed with the computation of the thread. In order specify this need, we define *future evaluation contexts* E_f and F_f in Fig. 3 that we will use in the rule (FUTURE.DEREF) of the operational semantics in Fig. 5. In the version of $\lambda(\mathsf{fut})$ presented here, the value of futures x is needed in two situations, in function applications xv and for cell exchange $\mathbf{exch}\, x\, v$ in evaluation contexts. Furthermore, note that future evaluation contexts can equally be used to extend $\lambda(\mathsf{fut})$ by lazy threads. The same mechanism has also proved useful to model more implementation oriented issues in [9]. Future evaluation contexts are called *placeholder strict* there.

Every reduction step of $\lambda(\mathsf{fut})$ is defined by an evaluation rule in Fig. 5 which involves either one or two threads of a configuration. These threads can be freely selected according to the two inference rules in Fig. 4: given a configuration C we choose a representation $D[C']$ congruent to C and apply a reduction rule

$$
\begin{array}{ll}
D ::= [\,] \mid C \mid D \mid (\nu x)D & \\
E ::= x{\Leftarrow}F & E_{\mathrm{f}} ::= x{\Leftarrow}F_{\mathrm{f}} \\
F ::= [\,] \mid F\,e \mid v\,F & F_{\mathrm{f}} ::= F[[\,]\,v] \mid F[\mathbf{exch}\,[\,]\,v]
\end{array}
$$

Fig. 3. Evaluation contexts D, E, F and future evaluation contexts E_{f}, F_{f}

$$
\frac{C_1 \equiv C_1' \qquad C_1' \to C_2' \qquad C_2' \equiv C_2}{C_1 \to C_2} \qquad\qquad \frac{C_1 \to C_2}{D[C_1] \to D[C_2]}
$$

Fig. 4. Selection of threads during reduction

to C'. Reduction inside threads $x{\Leftarrow}e$ means to reduce a subexpression e' in an evaluation context F so that $e = F[e']$. Evaluation inside expressions is call-by-value, i.e., all arguments of a function are evaluated before function application. This by the standard call-by-value beta reduction rule (BETA) in Fig. 5.

Besides β-reduction, there are six other reduction rules in Fig. 5. Concurrent futures are created by rule (THREAD.NEW). Evaluating applications **thread** $\lambda y.e$ has the following effects:

– a new concurrent future y is created,
– a new thread $y{\Leftarrow}e$ is spawned which evaluates the expression e concurrently and eventually assigns its value to y,
– the concurrent future y is returned instantaneously in the original thread.

Note that the expression e may also refer to y, i.e., our notion of thread creation incorporates explicit recursion.

As an example consider an application $f\,e$ of some function f, where the evaluation of the argument e takes considerable time, e.g., a communication with a remote process or an expensive internal computation. In this case it may be advantageous to use instead

$$f\,(\mathbf{thread}\ \lambda y.e)$$

which applies f to a fresh future y and delegates the evaluation of e to a concurrent thread $y{\Leftarrow}e$. The point here is that f will only block on y if it really needs the value of its argument, so that the latest possible synchronisation is obtained automatically. The only way we can simulate this effect with channels is by rewriting the function f (even the argument type of f changes). So what futures buy us is maximal concurrency without the need to rewrite existing code.

Rule (FUTURE.DEREF) states when to replaces futures y by their value v. It applies to futures in future evaluation contexts, once the value of the future has been computed by some concurrent thread $y{\Leftarrow}v$ in the configuration.

Rule (HANDLE.NEW) introduces a handled future jointly with a handle. The idea of handled futures appeared before in the form of I-structures [4] and promises [14]. Evaluating applications **handle** $\lambda x.\lambda y.e$ has the following effects:

$$
\begin{array}{lll}
(\textsc{beta}) & E[(\lambda y.e)\, v] \to E[e[{}^{v}\!/\!{}_{y}]] & \\[1ex]
(\textsc{thread.new}) & E[\mathbf{thread}\, v] \to (\nu y)(E[y] \mid y{\Leftarrow}v\, y) & (y \notin \mathsf{fv}(E[v])) \\
(\textsc{future.deref}) & E_{\mathrm{f}}[y] \mid y{\Leftarrow}v \to E_{\mathrm{f}}[v] \mid y{\Leftarrow}v & \\[1ex]
(\textsc{handle.new}) & E[\mathbf{handle}\, v] \to (\nu y)(\nu z)(E[v\, y\, z] \mid z \,\mathsf{h}\, y) & (y, z \notin \mathsf{fv}(E[v])) \\
(\textsc{handle.bind}) & E[z\, v] \mid z \,\mathsf{h}\, y \to E[\mathbf{unit}] \mid y{\Leftarrow}v \mid z \,\mathsf{h}\, \bullet & \\[1ex]
(\textsc{cell.new}) & E[\mathbf{cell}\, v] \to (\nu y)(E[y] \mid y \,\mathsf{c}\, v) & (y \notin \mathsf{fv}(E[v])) \\
(\textsc{cell.exch}) & E[\mathbf{exch}\, y\, v_1] \mid y \,\mathsf{c}\, v_2 \to E[v_2] \mid y \,\mathsf{c}\, v_1 &
\end{array}
$$

Fig. 5. Reduction rules of operational semantics

– a new handled future x is created,
– a new handle y is created,
– a new handle component $y \,\mathsf{h}\, x$ associates handle y to future x,
– the current thread continues with expression e.

Handles can be used only once. According to rule (HANDLE.BIND) an application of handle y to value v reduces by binding the future associated to y to v. This action consumes the handle component $y \,\mathsf{h}\, x$; what remains is a used handle component $y \,\mathsf{h}\, \bullet$. Trying to apply a handle a second time leads to *handle errors*:

$$D[E[y\ v] \mid y \,\mathsf{h}\, \bullet] \qquad \text{(handle error)}$$

We call a configuration C *error-free* if it cannot be reduced to any erroneous configuration, i.e., none of its reducts C' with $C \to^* C'$ is a handle error.

Evaluating $\mathbf{cell}\, v$ with rule (CELL.NEW) creates a new cell y with content v represented through a cell component $y \,\mathsf{c}\, v$. The exchange operation $\mathbf{exch}\, y\, v$ writes v to the cell and returns the previous contents. This exchange is *atomic*, i.e., no other thread can interfere. The cell exchange operation **exch** is strict in its first argument; the definition of strict evaluation contexts E_{f} expresses this uniformly. Observe that cells introduce indeterminism since two threads might compete for access to a cell.

Programs without handle errors and cell exchange are uniformly confluent [19] and thus confluent. So cell exchange by concurrent threads remains the *sole* source of indeterminism in $\lambda(\mathsf{fut})$. Nevertheless, handles are needed together with cells in order to safely express nondeterministic concurrency constructs (see Sect. 5). While handles can be expressed in terms of the other constructs, such an encoding unnecessarily complicates the formal treatment. In order to rule out handle errors by means of the linear type system of Sect. 4 we chose to introduce handled futures as primitive.

2.3 Examples

The first example illustrates concurrent threads and data synchronization. Let I be the lambda expression $\lambda z.z$. We consider the expression below, and reduce it by a THREAD.NEW step followed by a trivial BETA step. In the previous explanation of thread creation we left such BETA steps implicit.

$$\begin{aligned} x \Leftarrow \underline{(\mathbf{thread}\ (\lambda y.I\,I))}\,(I\,\mathbf{unit}) &\rightarrow (\nu y)(x \Leftarrow y\,(I\,\mathbf{unit})\ \mid\ y \Leftarrow \underline{(\lambda y.I\,I) y}) \\ &\rightarrow (\nu y)(x \Leftarrow y\,(I\,\mathbf{unit})\ \mid\ y \Leftarrow I\,I) \end{aligned}$$

At this point, we can reduce both threads concurrently, i.e., we have a choice of BETA reducing the left or right thread first. We do the former:

$$\begin{aligned} (\nu y)(x \Leftarrow y\,(\underline{I\,\mathbf{unit}})\ \mid\ y \Leftarrow I\,I) &\rightarrow (\nu y)(x \Leftarrow y\,\mathbf{unit}\ \mid\ y \Leftarrow \underline{I\,I}) \\ &\rightarrow (\nu y)(x \Leftarrow y\,\mathbf{unit}\ \mid\ y \Leftarrow I) \end{aligned}$$

In fact, any other reduction sequence would have given the same result in this case. At this point, both threads need to synchronize to exchange the value of y by applying FUTURE.DEREF; this enables a final BETA step:

$$\begin{aligned} (\nu y)(\underline{x \Leftarrow y\,\mathbf{unit}\ \mid\ y \Leftarrow I}) &\rightarrow (\nu y)(x \Leftarrow \underline{I\,\mathbf{unit}}\ \mid\ y \Leftarrow I) \\ &\quad\ (\nu y)(x \Leftarrow \mathbf{unit}\ \mid\ y \Leftarrow I) \end{aligned}$$

The second example illustrates the power of thread creation in $\lambda(\mathsf{fut})$; in contrast to all previous future operators, it can express explicit recursion. Indeed, **thread** can replace a fixed point operator **fix**. Consider for instance:

$$x \Leftarrow (\mathbf{thread}\,\lambda f.\lambda x.(f\ x))\ z$$

Thread creation THREAD.NEW yields a thread assigning a recursive value to f, so that the original thread can FUTURE.DEREF and BETA reduce forever.

$$\begin{aligned} (\nu f)\ (\underline{x \Leftarrow f\ z\ \mid\ f \Leftarrow \lambda x.(f\ x)}) &\rightarrow (\nu f)\ (x \Leftarrow \underline{(\lambda x.(f\ x))\ z}\ \mid\ f \Leftarrow \lambda x.(f\ x)) \\ &\rightarrow (\nu f)\ (\underline{x \Leftarrow f\ z\ \mid\ f \Leftarrow \lambda x.(f\ x)}) \end{aligned}$$

Indeed, rule FUTURE.DEREF simulates the usual UNFOLD rule for fixed point operators.

$$(\text{UNFOLD}) \qquad \mathbf{fix}\ \lambda f.\lambda x.e \rightarrow \lambda x.e[\mathbf{fix}\ \lambda f.\lambda x.e/f]$$

As a final example, consider how handles can introduce cyclic bindings:

$$\begin{aligned} x \Leftarrow \underline{\mathbf{handle}\,\lambda z.\lambda y.y\,z} &\rightarrow^3 (\nu y)(\nu z)(x \Leftarrow \underline{y\,z}\ \mid\ y\,\mathsf{h}\,z) \\ &\rightarrow (\nu y)(\nu z)(x \Leftarrow \mathbf{unit}\ \mid\ z \Leftarrow z\ \mid\ y\,\mathsf{h}\,\bullet) \end{aligned}$$

by HANDLE.NEW and two BETA steps. The final step by HANDLE.BIND binds the future z to itself. This is closely related to what is sometimes referred to as *recursion through the store*, or implicit recursion.

$$\alpha, \beta \in \mathit{Type} ::= \mathsf{unit} \mid \alpha \to \beta \mid \alpha\,\mathsf{ref} \qquad \frac{}{\Gamma \vdash c : \mathit{TypeOf}(c)}$$

$$\mathbf{unit} : \mathsf{unit} \qquad \frac{}{\Gamma, x{:}\alpha \vdash x : \alpha}$$

$$\mathbf{thread} : (\alpha \to \alpha) \to \alpha$$

$$\mathbf{handle} : (\alpha \to (\alpha \to \mathsf{unit}) \to \beta) \to \beta \qquad \frac{\Gamma, x{:}\alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}$$

$$\mathbf{cell} : \alpha \to (\alpha\,\mathsf{ref})$$

$$\mathbf{exch} : \alpha\,\mathsf{ref} \to \alpha \to \alpha \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\,e_2 : \beta}$$

Fig. 6. Typing of $\lambda(\mathsf{fut})$ expressions

3 Typing

Since our intention is to model extensions of the (statically typed) language ML we restrict our calculus to be typed. Types are function types $\alpha \to \beta$, the type $\alpha\,\mathsf{ref}$ of reference cells containing elements of type α, and the single base type unit. Typing of expressions is standard and integrates well with ML-style polymorphism and type inference. On the level of configurations, types are used to ensure a number of well-formedness conditions, and allow us to state a type preservation property during evaluation.

3.1 Typing of Constants and Expressions

According to the operational semantics described in Sect. 2, the constants obtain their natural types. For instance, **thread** has type $(\alpha \to \alpha) \to \alpha$ for any type α since its argument must be a function that maps a future of type α to a value of type α. The operation **thread** then returns the future of type α. The types of the other constants are listed in Fig. 6 and can be justified accordingly.

Let Γ and Δ range over *type environments* $x_1{:}\alpha_1 \ldots x_n{:}\alpha_n$, i.e. finite functional relations between *Var* and *Type*. In writing Γ_1, Γ_2 we assume that the respective domains are disjoint. Writing $\mathit{TypeOf}(c)$ for the type of constant c we have the usual type inference rules for simply typed lambda calculus (Fig. 6).

3.2 Typing of Configurations

Every future in a configuration is either concurrent or handled, i.e., its status is unique. Moreover, the binding of a concurrent future must be unique, and a handle must give reference to a unique future. Since parallel compositions of components are reminiscent of (mutually recursive) declarations the following two configurations are ill-typed:

$$x \Leftarrow e_1 \mid x \Leftarrow e_2 \quad \text{or} \quad y\,\mathsf{h}\,x_1 \mid y\,\mathsf{h}\,x_2 \tag{1}$$

Therefore, in the type system it will be required that the variables introduced by C_1 and C_2 are disjoint in concurrent compositions $C_1 \mid C_2$.

$$\frac{\Gamma, \Gamma_1 \vdash C_1 : \Gamma_2 \quad \Gamma, \Gamma_2 \vdash C_2 : \Gamma_1}{\Gamma \vdash C_1 \mid C_2 : \Gamma_1, \Gamma_2} \qquad \frac{\Gamma, x{:}\alpha \vdash e : \alpha}{\Gamma \vdash x{\Leftarrow}e : (x{:}\alpha)}$$

$$\frac{\Gamma, x{:}\alpha\,\mathsf{ref} \vdash v : \alpha}{\Gamma \vdash x\,\mathsf{c}\,v : (x{:}\alpha\,\mathsf{ref})} \qquad \frac{x, y \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash y\,\mathsf{h}\,x : (x{:}\alpha, y{:}\alpha \to \mathsf{unit})}$$

$$\frac{\Gamma \vdash C : \Gamma'}{\Gamma \vdash (\nu x)C : \Gamma' - x} \qquad \frac{y \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash y\,\mathsf{h}\,\bullet : (y{:}\alpha \to \mathsf{unit})}$$

Fig. 7. Typing rules for components

Types are lifted to configurations according to the inference rules in Fig. 7. The judgment $\Gamma \vdash C : \Gamma'$ informally means that given type assumptions Γ the configuration C is well-typed. The type environment Γ' keeps track of the variables declared by C. In fact, the rules guarantee that $\mathsf{dom}(\Gamma')$ is exactly the set of variables declared by C, and that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$.

To type a thread $x{\Leftarrow}e$ we can use the environment Γ as well as the binding $x{:}\alpha$ that is introduced by the component. Note that writing $\Gamma, x{:}\alpha$ in the premise implies that x is not already declared in Γ. Similarly, when typing a reference cell $x\,\mathsf{c}\,v$ both Γ and the assumption $x{:}\alpha\,\mathsf{ref}$ can be used to derive that the contents v of the cell has type α. The typing rule for handle components $y\,\mathsf{h}\,x$ and $y\,\mathsf{h}\,\bullet$ take care that the types of the handled future x and its handle y are compatible, and that they are not already declared in Γ.

A restriction $(\nu x)C$ is well-typed under assumptions Γ if the configuration C is. The name x is kept local by removing any occurrence $x{:}\alpha$ from Γ', which we write $\Gamma' - x$.

The typing rule for a parallel composition $C_1 \mid C_2$ is reminiscent of the circular assume-guarantee reasoning used in compositional verification of concurrent systems [1]. Recall that the combined environment Γ_1, Γ_2 in the conclusion is only defined if the variables appearing in Γ_1 and Γ_2 are disjoint. So the rule ensures that the sets of variables declared by C_2 and C_1 resp., are disjoint. Note how this prevents the ill-formed configurations in (1) to be typed. Moreover, by typing C_1 in the extended environment Γ,Γ_1 the rule allows variables declared by C_2 to be used in C_1, and vice versa. For example, we can derive

$$\vdash (x{\Leftarrow}y\;\mathbf{unit} \mid y\,\mathsf{h}\,z) : (x{:}\mathsf{unit}, z{:}\mathsf{unit}, y{:}\mathsf{unit} \to \mathsf{unit}) \tag{2}$$

The thread on the left-hand side declares x and can use the assumption $y{:}\mathsf{unit} \to \mathsf{unit}$ about the handle declared in the component on the right. No further assumptions are necessary.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash C_1 : \Gamma'$ and $C_1 \to C_2$ then $\Gamma \vdash C_2 : \Gamma'$.*

Program errors are notorious even for a statically typed programming language. Indeed it turns out that the class of *handle errors* is not excluded by the type system just presented.

Multiplicities $\kappa ::= \mathbf{1} \mid \omega$

Linear types $\alpha, \beta \in \mathit{LinType} ::= \mathsf{unit} \mid \alpha \xrightarrow{\kappa} \beta \mid \alpha\,\mathsf{ref}$

Multiplicities of linear types

$$|\mathsf{unit}| \overset{def}{=} \omega, \qquad |\alpha \xrightarrow{\kappa} \beta| \overset{def}{=} \kappa, \qquad |\alpha\,\mathsf{ref}| \overset{def}{=} \omega$$

Typing of constants where $\kappa, \kappa', \kappa''$ arbitrary

$$\textbf{unit} : \mathsf{unit}$$

$$\textbf{thread} : (\alpha \xrightarrow{\kappa} \alpha) \xrightarrow{\kappa'} \alpha \text{ where } |\alpha| = \omega$$

$$\textbf{handle} : (\alpha \xrightarrow{\kappa} (\alpha \xrightarrow{\mathbf{1}} \mathsf{unit}) \xrightarrow{\kappa'} \beta) \xrightarrow{\kappa''} \beta \text{ where } |\alpha| = \omega$$

$$\textbf{cell} : \alpha \xrightarrow{\kappa} (\alpha\,\mathsf{ref})$$

$$\textbf{exch} : \alpha\,\mathsf{ref} \xrightarrow{\kappa} \alpha \xrightarrow{\kappa'} \alpha$$

Operations on type environments

$$\mathsf{once}(\Gamma) \overset{def}{=} \{x \mid x{:}\alpha \text{ in } \Gamma,\ |\alpha| = \mathbf{1}\}$$

$$\Gamma_1 \cdot \Gamma_2 \overset{def}{=} \Gamma_1 \cup \Gamma_2 \text{ provided } \Gamma_1 \cap \Gamma_2 = \{x{:}\alpha \mid x{:}\alpha \in \Gamma_1 \cup \Gamma_2, |\alpha| = \omega\}$$

Fig. 8. Linear types

4 Linear Types for Handles

We refine the type system to prevent handle errors. We see this system as a proof tool to facilitate reasoning about the absence of handle errors; we do *not* want to argue that the linear types should be used in practice. We do also not discuss how to deal with handle errors in a concrete programming language.

Most previous uses of linear type systems in functional languages, such as the uniqueness typing of Clean [6], aimed at preserving referential transparency in the presence of side-effects, and taking advantage of destructive updates for efficiency reasons. In contrast, our system rules out a class of programming errors, by enforcing the single-assignment property for handled futures.

The linear type system will be sufficiently expressive to type a variety of concurrency abstractions (Sects. 5 and 6). Moreover, the linear types of the handles implementing these abstractions will be encapsulated. Thus, users of these abstractions need not know about linear types at all.

We annotate types with *usage information* in the sense of [27]. In our case it is sufficient to distinguish between linear (i.e., exactly one) and nonlinear (i.e., any number of times) uses. Multiplicities $\mathbf{1}$ and ω are ranged over by κ. Moreover, for our purposes of ruling out handle errors we annotate only function types, values of other types can always be used non-linearly (recall that handles have functional types $\alpha \to \mathsf{unit}$). In particular, $\alpha \xrightarrow{\kappa} \beta$ denotes functions from α to β that can be used κ times, and so $\alpha \xrightarrow{\omega} \beta$ corresponds to the usual function type. We write $|\alpha|$ for the multiplicity attached to a type α (see Fig. 8).

For a context Γ we write $\mathsf{once}(\Gamma)$ for the set of variables occuring in Γ with linear multiplicity. If Γ can be split into Γ_1 and Γ_2 that both contain all the variables of Γ with multiplicity ω and a partition of $\mathsf{once}(\Gamma)$ we write $\Gamma = \Gamma_1{\cdot}\Gamma_2$.

$$\frac{\mathsf{once}(\Gamma) = \emptyset}{\Gamma \vdash c : \mathit{TypeOf}(c)} \qquad \frac{\Gamma, x{:}\alpha \vdash e : \alpha \quad |\alpha| = \omega}{\Gamma \vdash x \Leftarrow e : (x{:}\alpha; x{:}\alpha)}$$

$$\frac{\mathsf{once}(\Gamma) = \emptyset}{\Gamma, x{:}\alpha \vdash x : \alpha} \qquad \frac{\Gamma, x{:}\alpha\,\mathsf{ref} \vdash v : \alpha}{\Gamma \vdash x\,\mathsf{c}\,v : (x{:}\alpha\,\mathsf{ref}; x{:}\alpha\,\mathsf{ref})}$$

$$\frac{\Gamma, x{:}\alpha \vdash e : \beta \quad \mathsf{once}(\Gamma) = \emptyset}{\Gamma \vdash \lambda x.e : \alpha \xrightarrow{\omega} \beta} \qquad \frac{x, y \notin \mathsf{dom}(\Gamma) \quad |\alpha| = \omega}{\Gamma \vdash y\,\mathsf{h}\,x : (x{:}\alpha, y{:}\alpha \xrightarrow{\mathbf{1}} \mathsf{unit};\ x{:}\alpha, y{:}\alpha \xrightarrow{\mathbf{1}} \mathsf{unit})}$$

$$\frac{\Gamma, x{:}\alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \xrightarrow{\mathbf{1}} \beta} \qquad \frac{y \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash y\,\mathsf{h}\,\bullet : (y{:}\alpha \xrightarrow{\mathbf{1}} \mathsf{unit}; \emptyset)}$$

$$\frac{\Gamma_1 \vdash e_1 : \alpha \xrightarrow{\kappa} \beta \quad \Gamma_2 \vdash e_2 : \alpha}{\Gamma_1 \cdot \Gamma_2 \vdash e_1\, e_2 : \beta} \qquad \frac{\Gamma \vdash C : \Gamma_1; \Gamma_2}{\Gamma \vdash (\nu x)C : \Gamma_1 - x; \Gamma_2 - x}$$

$$\frac{\Gamma, \Delta_2 \vdash C_1 : \Gamma_1; \Gamma_2 \cdot \Gamma_3 \quad \Delta, \Gamma_2 \vdash C_2 : \Delta_1; \Delta_2 \cdot \Delta_3}{\Gamma \cdot \Delta \vdash C_1 \mid C_2 : \Gamma_1, \Delta_1;\ \Gamma_3 \cdot \Delta_3} \quad \begin{pmatrix} \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta_1) = \emptyset \\ \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Gamma_1) = \emptyset \end{pmatrix}$$

Fig. 9. Linear typing rules for $\lambda(\mathsf{fut})$ expressions and configurations

The types of the term constants are now refined to reflect that handles *must* be used linearly. However, we do not want to restrict access to futures through the rule (FUTURE.DEREF). Hence it must be guaranteed that futures will never be replaced by values of types with linear multiplicity. This is done by restricting the types of **thread** and **handle** by the condition $|\alpha| = \omega$. On the other hand, note that no such restriction is necessary for cells which may contain values of any (i.e., multiplicity $\mathbf{1}$ or ω) type. Intuitively this is sound because cells can be accessed only by the exchange operation. In particular, the contents of a cell (potentially having multiplicity $\mathbf{1}$) cannot be *copied* through cell access.

The type rules for expressions are given in Fig. 9. The rules guarantee that every variable of type α in Γ with $|\alpha| = \mathbf{1}$ appears exactly once in the term: In the rules for constants and variables, the side-condition $\mathsf{once}(\Gamma) = \emptyset$ ensures that Γ contains only variables with use ω. There are two rules for abstraction, reflecting the fact that we have function types with multiplicities $\mathbf{1}$ and ω. The condition $\mathsf{once}(\Gamma) = \emptyset$ in the first abstraction rule allows us to derive a type $\alpha \xrightarrow{\omega} \beta$ only if e does not contain any free variables with multiplicity $\mathbf{1}$. However, with the second rule it is always possible to derive a type $\alpha \xrightarrow{\mathbf{1}} \beta$. Finally, the rule for application splits the linearly used variables of the environment. The annotation κ is irrelevant here, but the type of function and argument must match exactly.

The rules for configurations (Fig. 9) have changed: Judgments are now of the form $\Gamma \vdash C : \Delta_1; \Delta_2$, and the type system maintains the invariants (i) $\Gamma \cap \Delta_1 = \emptyset$ and (ii) $\Delta_2 \subseteq \Delta_1$. The intended meaning is the following.

– As before, Γ contains the type assumptions and Δ_1 is used to keep track of the variables which C provides bindings for. In particular, the use of Δ_1 allows to ensure the well-formedness conditions in configurations (cf. the configurations (1) on page 8) by means of invariant (i).

– Variables with multiplicity **1** declared by C may not be used both by a surrounding configuration *and* within C. The environment $\Delta_2 \subseteq \Delta_1$ lists those variables "available for use" to the outside.

The example configuration (2) on page 9 shows the need for the additional environment Δ_2: Although a binding for the handle y is provided in $y\,\mathsf{h}\,z$, y is already used *internally* to bind its future, in the thread $x{\Leftarrow}y\,\mathbf{unit}$.

The rules for typing thread and handle components now contain the side condition $|\alpha| = \omega$ corresponding to the type restriction of the respective constants. Moreover, the type of y in $y\,\mathsf{h}\,x$ must have multiplicity **1**. Note that in each case we have $\Delta_1 = \Delta_2$, i.e., all the declared variables are available.

In $y\,\mathsf{h}\,\bullet$ the variable y is declared, but not available anymore, i.e. it cannot be used in a surrounding configuration at all. Thus $\Delta_2 = \emptyset$. The rule for restriction keeps declarations local by removing all occurrences of x from Δ_1 and Δ_2.

The rule for parallel composition is the most complex one. Compared to the corresponding inference scheme of the previous section, it splits the linearly used assumptions (in $\Gamma\cdot\Delta$) as well as the linearly used variables available from each of the two constituent configurations ($\Gamma_2\cdot\Gamma_3$ and $\Delta_2\cdot\Delta_3$, resp.). A variable with multiplicity **1** declared by C_1 can then either be used in C_2 (via Γ_2), or is made available to a surrounding configuration (via Γ_3) but not both. The environment of declared variables of $C_1 \mid C_2$ is Γ_1, Δ_1 and therefore contains all the variables declared in C_1 (i.e., those in Γ_1) and C_2 (in Δ_1) as before. By our convention, this ensures in particular that C_1 and C_2 do not contain multiple bindings for the same variable. Finally, the side-condition of the rule is necessary to establish the invariant (i).

Theorem 2 (Subject Reduction). *If* $\Gamma \vdash C_1 : \Delta_1; \Delta_2$ *and* $C_1 \to C_2$ *then* $\Gamma \vdash C_2 : \Delta_1; \Delta_2$.

Error-freeness of well-typed configurations follows by combining the absence of handle errors in the immediate configuration and Subject Reduction as usual.

Corollary 1 (Absence of Handle Errors). *If* $\Gamma \vdash C : \Delta_1; \Delta_2$ *then* C *is error-free.*

5 Concurrency Constructs

We now show how to express various concurrency abstractions in $\lambda(\mathsf{fut})$ which demonstrates the power of handled futures. We use some syntactic sugar, writing $\mathsf{let}\ x_1{=}e_1\ \mathsf{in}\ e$ for $(\lambda x_1.e)\,e_1$, $\lambda\_.e$ for $\lambda x.e$ where x is not free in e, and $e_1;\, e_2$ for $(\lambda\_.e_2)\,e_1$. We also extend $\lambda(\mathsf{fut})$ with products and lists.

Mutual Exclusion. When concurrent threads access shared data it is necessary that they do not interfere in order to prevent any data inconsistencies. We can implement an operation mutex of type $(\mathsf{unit} \to \alpha) \to \alpha$ that applies its argument under mutual exclusion. We use a strict context to synchronize several threads wishing to access a *critical region.*

```
let $r$ = $\mathbf{cell}\,(\lambda y.y)$
in $\lambda a.\mathbf{handle}(\lambda x \lambda \mathit{bind}_x.\ (\mathbf{exch}\, r\, x)\,(\mathbf{unit});$
                                         $(\text{let } v{=}a\,(\mathbf{unit}) \text{ in } \mathit{bind}_x\,(\lambda y.y); v))$
```

Before running the function a that is given as argument (the critical region), a handled future (of type $\mathsf{unit} \rightarrow \mathsf{unit}$) is stored in the reference cell r. This future is bound (to the identity function) only *after* the argument is evaluated. Moreover, *before* this happens, the previous contents of the cell r cannot be an unbound future anymore since the function application $(\mathbf{exch}\, r\, x)\,(\mathbf{unit})$ is strict in its arguments. Consequently, threads cannot interfere when evaluating a.

Ports. We assume that there are pairs and a list data type, and write $v :: l$ for the list with first element v, followed by the list l. A *stream* is a "open-ended" list $v_1{::}\cdots{::}v_n{::}x$ where x is a (handled) future. The stream can be extended arbitrarily often by using the handle of the future, provided each new element is again of the form $v{::}x'$, with x' a handled future. We call the elements $v_1, \ldots, v_n$ on a stream *messages*.

A function newPort that creates a new *port* can be implemented as follows.

```
$\lambda\_.\mathbf{handle}(\lambda s \lambda \mathit{bind}_s.$
  let $\mathit{putr} = \mathbf{cell}\, \mathit{bind}_s$
      $\mathit{put} = \lambda x.\mathbf{handle}(\lambda s \lambda \mathit{bind}_s.(\mathbf{exch}\, \mathit{putr}\, \mathit{bind}_s)\,(x :: s))$
  in $(s, \mathit{put}))$
```

The port consists of a stream s and an operation *put* to put new messages onto the stream. The stream is ended by a handled future, which in the beginning is s itself. Its handle bind_s is stored in the cell *putr* and used in *put* to send the next message to the port. *put* introduces a new handled future before writing the new value to the end of the stream. The new handle is stored in the cell.

Channels. By extending ports with a receive operation of type $\mathsf{unit} \rightarrow \alpha$ we obtain channels, which provide for indeterministic many to many communication. A function newChannel that generates channels is

```
$\lambda\_.\mathbf{handle}(\lambda \mathit{init} \lambda \mathit{bind}_{\mathit{init}}.$
  let $\mathit{putr} = \mathbf{cell}\, \mathit{bind}_{\mathit{init}}$
      $\mathit{getr} = \mathbf{cell}\, \mathit{init}$
      $\mathit{put} = \lambda x.\mathbf{handle}(\lambda n \lambda \mathit{bind}_n.(\mathbf{exch}\, \mathit{putr}\, \mathit{bind}_n)\,(x :: n))$
      $\mathit{get} = \lambda\_.\mathbf{handle}(\lambda n \lambda \mathit{bind}_n.\text{case}(\mathbf{exch}\, \mathit{getr}\, n))$
                                        of $x :: c \Rightarrow \mathit{bind}_n(c); x)$
  in $(\mathit{put}, \mathit{get}))$
```

Given a stream, applying *get* yields the next message on the stream. If the stream contains no further messages, *get* blocks: We assume that the matching against the pattern $x :: c$ is strict. Note how *get* uses a handled future in the same way as the mutual exclusion above to make the implementation thread-safe.

$$\alpha, \beta \in \mathit{LinType} ::= \ \dots \ \mid \ \alpha \times^{\kappa} \beta \ \mid \ \alpha\, \mathsf{list}^{\kappa} \quad \text{such that } |\alpha| = \mathbf{1} \text{ or } |\beta| = \mathbf{1} \implies \kappa = \mathbf{1}$$

$$\frac{\Gamma_1 \vdash e_1 : \alpha \quad \Gamma_2 \vdash e_2 : \beta}{\Gamma_1 \cdot \Gamma_2 \vdash (e_1, e_2) : \alpha \times^{\kappa} \beta} \qquad \frac{\Gamma_1 \vdash e_1 : \alpha \quad \Gamma_2 \vdash e_2 : \alpha\, \mathsf{list}^{\kappa}}{\Gamma_1 \cdot \Gamma_2 \vdash e_1 :: e_2 : \alpha\, \mathsf{list}^{\kappa}}$$

$$\frac{\Gamma_1 \vdash e_1 : \alpha \times^{\kappa} \beta \quad \Gamma_2, x{:}\alpha, y{:}\beta \vdash e_2 : \gamma}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{case}\ e_1\ \mathsf{of}\ (x, y) \Rightarrow e_2 : \gamma} \qquad \frac{\Gamma_1 \vdash e_1 : \alpha\, \mathsf{list}^{\kappa} \quad \Gamma_2, x{:}\alpha, y{:}\alpha\, \mathsf{list}^{\kappa} \vdash e_2 : \beta}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{case}\ e_1\ \mathsf{of}\ x{::}y \Rightarrow e_2 : \beta}$$

Fig. 10. Typing rules for products and lists

6 Proving Safety

The three abstractions defined in the previous section are *safe*, in the sense that no handle errors are raised by using them. For instance, we can always send to the port without running into an error. Intuitively, this holds since nobody can access the (local) handle to the future at the end of the stream s, and the implementation itself uses each handle only once.

The linear type system can be used to make this intuition formal: By the results of Sect. 4 typability guarantees the absence of handle errors. Moreover, all three abstractions obtain "non-linear" types with multiplicity ω. The use of handled futures is thus properly encapsulated and not observable from the types. This suggests to provide concurrency abstractions through safe libraries to users.

Mutual Exclusion. The mutual exclusion mutex can be typed as

$$\vdash \mathsf{mutex} : (\mathsf{unit} \xrightarrow{\kappa} \alpha) \xrightarrow{\omega} \alpha$$

in the linear type system. In fact, in a derivation there is no constraint on the multiplicity κ which can be chosen as either $\mathbf{1}$ or ω. More importantly, the type of mutex itself has multiplicity ω, which allows mutex to be applied any number of times.

Ports and Channels. We only sketch how to extend the linear type system to deal with pairs and lists. The details of such an extension are quite standard (see [27], for instance). Just as with function types these new types are annotated with multiplicities. We can devise the inference rules in Fig. 10 for the new types. If the constructs are given the usual operational semantics, the subject reduction theorem can be extended, as can Corollary 1. For the port abstraction, we derive

$$\vdash \mathsf{newPort} : \mathsf{unit} \xrightarrow{\omega} (\alpha\, \mathsf{list}^{\omega} \times^{\kappa} \alpha \xrightarrow{\omega} \mathsf{unit}) \qquad\qquad (|\alpha| = \omega)$$

for any κ. In particular, both newPort itself and the *put* operation (the second component of the result pair) can be used unrestrictedly. Similarly, for $|\alpha| = \omega$,

$$\vdash \mathsf{newChannel} : \mathsf{unit} \xrightarrow{\omega} ((\alpha \xrightarrow{\omega} \mathsf{unit}) \times^{\omega} (\mathsf{unit} \xrightarrow{\omega} \alpha))$$

can be derived for the implementation of channels.

7 Implementation View

An implementation of futures has to deal with placeholder objects and dereferencing to obtain the value associated with a future. Further, in the case of lazy futures it must perform the triggering of computations. All these aspects are visible on the level of the compiler only; futures are transparent from the programmer's point of view. Therefore, these *touch* operations are introduced by the compiler rather than the programmer.

With an explicit touch operator "**?**" the expression $\mathbf{?}x$ waits for the value of the future x, forces its evaluation if necessary, and returns the value once available. To improve efficiency, a compiler should be able to remove as many redundant touches as possible. That this can be done by strictness analysis was an important achievement of previous work on futures by Felleisen and Flanagan [9]. In this paper, we did not deal with touches and their optimization. We expect that the analysis of [9] can be extended to our calculus, but leave this for future work.

8 Conclusion

We have presented the lambda calculus with futures $\lambda(\mathsf{fut})$ which serves as a semantics for concurrent extensions of ML where all synchronization is based on futures. We assumed call-by-value evaluation, static typing, and state. We have demonstrated how the handled futures of $\lambda(\mathsf{fut})$ provide an elegant, unifying mechanism to express various concurrency abstractions.

We have proved the safety of these implementations on basis of a linear type system. Hence, handle errors cannot arise when using handles only through safe libraries. As a consequence, handled futures can be safely incorporated into a strongly typed ML-style programming language without imposing changes in the type system. An ML extension called Alice [2] along these lines is available.

In future work, we intend to develop a strictness analysis for improving the efficiency of implementations of $\lambda(\mathsf{fut})$. We plan to investigate operational equivalences as a further tool in reasoning about $\lambda(\mathsf{fut})$ programs.

Acknowledgements We would like to thank the anonymous referees for their detailed comments and feedback.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
2. *The Alice Project.* Web site at the Programming Systems Lab, Universität des Saarlandes, http://www.ps.uni-sb.de/alice, 2005.
3. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.
4. Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

5. H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, Aug. 1977.
6. E. Barendsen and S. Smetsers. Uniqueness type inference. In *Proc. PLILP'95*, volume 982 of *LNCS*, pages 189–206. Springer, 1995.
7. E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications With Objective Caml.* O'Reilly, 2000. Available online at http://caml.inria.fr/oreilly-book.
8. S. Conchon and F. L. 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)*, 1999.
9. C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.
10. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL'96*, pages 372–385. ACM Press, 1996.
11. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proc. CONCUR'97*, volume 1243 of *LNCS*, pages 196–212. Springer, 1997.
12. R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
13. S. Haridi, P. V. Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, 1999.
14. B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Notices*, 23(7):260–268, 1988.
15. J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
16. R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification, Proc. Marktoberdorf Summer School*, pages 203–246. Springer, 1993.
17. R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Standard ML Programming Language (Revised).* MIT Press, 1997.
18. M. Müller. *Set-based Failure Diagnosis for Concurrent Constraint Programming.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1998.
19. J. Niehren. Uniform confluence in concurrent computation. *Journal of Functional Programming*, 10(5):453–499, 2000.
20. F. Nielson, editor. *ML with Concurrency: Design, Analysis, Implementation, and Application.* Monographs in Computer Science. Springer, 1997.
21. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.
22. J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
23. A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming.* Intellect, Munich, Germany, 2004.
24. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. POPL'91*, pages 333–352. ACM Press.
25. E. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.
26. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.
27. D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. 7th ICFPCA*, pages 1–11. ACM Press, 1995.

A Proofs

In this section we provide the proof for the subject reduction theorem of the linear type system (Theorem 2). We start with a number of lemmas relating to (linear) variables in contexts and substitution. All of these are fairly standard.

Lemma 1. *If $\Gamma \vdash v : \alpha$ then $\mathsf{once}(\Gamma) \neq \emptyset$ implies $|\alpha| = \mathbf{1}$.*

Proof. The proof is by induction on the type derivation for $\Gamma \vdash v : \alpha$.

- Case v is c: Then always $\mathsf{once}(\Gamma) = \emptyset$.
- Case v is x: Then $\Gamma, x{:}\alpha \vdash x : \alpha$ with $\mathsf{once}(\Gamma) = \emptyset$, and so if $\mathsf{once}(\Gamma, x{:}\alpha) \neq \emptyset$ we have necessarily $|\alpha| = \mathbf{1}$.
- Case v is $\lambda x.e$: This follows immediately from the rules for abstraction. If $\mathsf{once}(\Gamma) \neq \emptyset$ only the second one of the two abstraction rule applies, and so $|\alpha| = \mathbf{1}$.
- Case v is $\mathbf{exch}\, v'$: Suppose $\Gamma \vdash \mathbf{exch}\, v' : \alpha$. Then α is $\beta \xrightarrow{\kappa} \beta$, and we have $\Gamma_1 \vdash \mathbf{exch} : \beta\,\mathsf{ref} \xrightarrow{\kappa'} \beta \xrightarrow{\kappa} \beta$ and $\Gamma_2 \vdash v' : \beta\,\mathsf{ref}$ where $\Gamma_1 \cdot \Gamma_2 = \Gamma$. As in the constant case, $\mathsf{once}(\Gamma_1) = \emptyset$, and since $|\beta\,\mathsf{ref}| = \omega$ the inductive hypothesis implies $\mathsf{once}(\Gamma_2) = \emptyset$. Hence $\mathsf{once}(\Gamma) = \emptyset$. □

Lemma 2. *If $\Gamma \vdash e : \alpha$ and $|\beta| = \omega$ then $\Gamma, x{:}\beta \vdash e : \alpha$.*

Proof. The proof is by a straightforward induction on $\Gamma \vdash e : \alpha$. □

Lemma 3. *Suppose $\Gamma \vdash e : \alpha$.*

1. *If x does not occur in Γ then x is not free in e.*
2. *If $x{:}\beta \in \Gamma$ with $|\beta| = \mathbf{1}$ then there is exactly one free occurrence of x in e.*
3. *If $\Gamma, x{:}\beta \vdash e : \alpha$ and $x \notin \mathsf{fv}(e)$ then $\Gamma \vdash e : \alpha$.*

Proof. Again, by an induction on $\Gamma \vdash e : \alpha$.

- Case e is c: Clearly no variable is free in e in this case, and $\mathsf{once}(\Gamma) = \emptyset$ by the typing rule for constants, so for all $x{:}\beta \in \Gamma$ we have $|\beta| = \omega$. Finally, $\Gamma \vdash c : \mathit{TypeOf}(c)$ for any Γ with $\mathsf{once}(\Gamma) = \emptyset$.
- Case e is y: Necessarily $y{:}\alpha$ in $\Gamma, y{:}\alpha$, and no $x \neq y$ is free in e. Moreover, there is exactly one occurrence of y in $e \equiv y$, and for all $x \neq y$ with $x{:}\beta$ in Γ the condition $\mathsf{once}(\Gamma) = \emptyset$ implies $|\beta| = \omega$. Finally, x not free in e implies $x \neq y$, so $\Gamma - x \vdash y : \alpha$.
- Case e is $e_1\, e_2$: Then $\Gamma_1 \vdash e_1 : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma_2 \vdash e_2 : \alpha'$ where $\Gamma_1{\cdot}\Gamma_2 = \Gamma$. By induction hypothesis, if x not in Γ (hence not in Γ_1 and Γ_2) then x is not free in $e \equiv e_1\, e_2$.
  Moreover, if $x{:}\beta \in \Gamma$ with $|\beta| = \mathbf{1}$ then $x{:}\beta$ in exactly one of Γ_1 and Γ_2. Suppose $x{:}\beta \in \Gamma_1$, then by induction there is exactly one occurrence of x in e_1, and by the previous observation x is not free in e_2. So there is exactly one occurrence in $e_1\, e_2$. The case where $x{:}\beta \in \Gamma_2$ is symmetric.
  The last claim follows immediately by induction.

– Case e is $\lambda y.e_1$: All claims follow immediately from the inductive hypothesis. □

Lemma 4 (Substitution). *Suppose* $\Gamma, x{:}\beta \vdash e : \alpha$ *and* $\Gamma' \vdash v : \beta$. *Then* $\Gamma \cdot \Gamma' \vdash e[{}^{v}\!/\!{}_{x}] : \alpha$.

Proof. By induction on $\Gamma, x{:}\beta \vdash e : \alpha$.

– Case e is c: By the rule for constants, $\mathsf{once}(\Gamma) = \emptyset$ and $|\beta| = \omega$. Clearly $\Gamma \vdash e : \alpha$ by the constant rule. By Lemma 1 $|\beta| = \omega$ implies $\mathsf{once}(\Gamma') = \emptyset$, and so by repeated applications of Lemma 2 and the fact $e[{}^{v}\!/\!{}_{x}] \equiv c$ we obtain $\Gamma \cdot \Gamma' \vdash e[{}^{v}\!/\!{}_{x}] : \alpha$.
– Case e is y: If $x = y$ then $\mathsf{once}(\Gamma) = \emptyset$ and $\alpha = \beta$ by the type rule for variables. But then $e[{}^{v}\!/\!{}_{x}] \equiv v$ and repeated applications of Lemma 2 yield the desired result.
  If $x \neq y$ then by the rule for variables $|\beta| = \omega$. So by Lemma 1 this yields $\mathsf{once}(\Gamma') = \emptyset$, and Lemma 2 and $e[{}^{v}\!/\!{}_{x}] \equiv y$ gives $\Gamma \cdot \Gamma' \vdash e[{}^{v}\!/\!{}_{x}] : \alpha$.
– Case e is $e_1\, e_2$: If $|\beta| = \omega$ then there are Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 \cdot \Gamma_2$ and we have $\Gamma_1, x{:}\beta \vdash e_1 : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma_2, x{:}\beta \vdash e_2 : \alpha'$. By induction, $\Gamma_1 \cdot \Gamma' \vdash e_1[{}^{v}\!/\!{}_{x}] : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma_2 \cdot \Gamma' \vdash e_2[{}^{v}\!/\!{}_{x}] : \alpha'$. By $|\beta| = \omega$ Lemma 1 implies $\mathsf{once}(\Gamma') = \emptyset$. Hence, $\Gamma \cdot \Gamma' \vdash (e_1\, e_2)[{}^{v}\!/\!{}_{x}] : \alpha$.
  If $|\beta| = \mathbf{1}$ then $\Gamma_1 \vdash e_1 : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma_2 \vdash e_2 : \alpha'$ where $\Gamma_1 \cdot \Gamma_2$ is defined and $x{:}\beta$ occurs in exactly one of Γ_1 and Γ_2. Suppose $x{:}\beta \in \Gamma_1$. Then by induction hypothesis, $(\Gamma_1 - x) \cdot \Gamma' \vdash e_1[{}^{v}\!/\!{}_{x}] : \alpha' \xrightarrow{\kappa} \alpha$. Also, by the first part of Lemma 3 we know $e_2[{}^{v}\!/\!{}_{x}] \equiv e_2$. Using Lemma 2 on $\Gamma_2 \vdash e_2 : \alpha'$ (with the non-linear variables of Γ') we obtain $\Gamma \cdot \Gamma' \vdash (e_1\, e_2)[{}^{v}\!/\!{}_{x}] : \alpha$. The case where $x{:}\beta \in \Gamma_2$ is symmetric.
– Case e is $\lambda y.e_1$: Suppose $\Gamma, x{:}\beta \vdash \lambda y.e_1 : \alpha_1 \xrightarrow{\kappa} \alpha_2$. By bound renaming, we can assume that y is different from x and all the variables declared in Γ and Γ'. By either rule for abstraction we get $\Gamma, x{:}\beta, y{:}\alpha_1 \vdash e_1 : \alpha_2$. By induction, $(\Gamma, y{:}\alpha_1) \cdot \Gamma' \vdash e_1[{}^{v}\!/\!{}_{x}] : \alpha_2$. Now if $\kappa = \omega$ then the condition in the abstraction rule implies $\mathsf{once}(\Gamma) = \emptyset$ and $|\beta| = \omega$. Hence by Lemma 1 $\mathsf{once}(\Gamma') = \emptyset$ as well. So for any κ we can derive $\Gamma \cdot \Gamma' \vdash (\lambda y.e_1)[{}^{v}\!/\!{}_{x}] : \alpha_1 \xrightarrow{\kappa} \alpha_2$. □

Lemma 5 (Context). *If* $\Gamma \vdash F[e] : \alpha$ *then there exist* Γ_1, Γ_2 *and* β *such that* $\Gamma = \Gamma_1 \cdot \Gamma_2$ *and* $\Gamma_1, x{:}\beta \vdash F[x] : \alpha$ *and* $\Gamma_2 \vdash e : \beta$.

Proof. The proof is by induction on the evaluation context F.

– Case F is $[\,]$: Choose $\beta = \alpha$, Γ_1 the empty environment and $\Gamma_2 = \Gamma$.
– Case F is $F'\, e'$: By the application rule, $\Gamma' \vdash F'[e] : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma'' \vdash e' : \alpha'$ for some Γ' and Γ'' with $\Gamma = \Gamma' \cdot \Gamma''$.
  By induction, exists β, Γ_1' and Γ_2 with $\Gamma' = \Gamma_1' \cdot \Gamma_2$ and $\Gamma_1', x{:}\beta \vdash F'[x] : \alpha' \xrightarrow{\kappa} \alpha$ and $\Gamma_2 \vdash e : \beta$. So the result follows by choosing $\Gamma_1 = \Gamma'' \cdot \Gamma_1'$ since then by the type rule for application $\Gamma_1, x{:}\beta \vdash F'[x]\, e' : \alpha$.
– Case F is $v\, F'$: This case is analogous to the previous case. □

Note that this result also holds for strict evaluation contexts F_{f} since these form a subset of evaluation contexts.

Lemma 6. *If $\Gamma \vdash C : \Delta_1; \Delta_2$ then*

1. $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta_1) = \emptyset$
2. $\Delta_2 \subseteq \Delta_1$
3. *if x is not in Γ and Δ_1, and $|\alpha| = \omega$, then $\Gamma, x{:}\alpha \vdash C : \Delta_1; \Delta_2$*
4. *if x is not free in C then $\Gamma - x \vdash C : \Delta_1; \Delta_2$.*

Proof. The proof is by an easy induction on $\Gamma \vdash C : \Delta_1; \Delta_2$. □

Note that without the side condition $\mathsf{dom}(\Gamma \cdot \Delta) \cap \mathsf{dom}(\Gamma_1, \Delta_1) = \emptyset$ in the rule for parallel composition the first claim of Lemma 6 will be false in general. To see this, observe that

$$y{:}\mathsf{unit} \xrightarrow{\mathbf{1}} \mathsf{unit} \vdash x \Leftarrow y\ \mathbf{unit} : (x{:}\mathsf{unit};\ x{:}\mathsf{unit})$$

and

$$\vdash y \,\mathsf{h}\, x' : (x'{:}\mathsf{unit}, y{:}\mathsf{unit} \xrightarrow{\mathbf{1}} \mathsf{unit};\ x'{:}\mathsf{unit}, y{:}\mathsf{unit} \xrightarrow{\mathbf{1}} \mathsf{unit})$$

and so without the side condition

$$y{:}\mathsf{unit} \xrightarrow{\mathbf{1}} \mathsf{unit} \vdash (x \Leftarrow y\ \mathbf{unit} \mid y \,\mathsf{h}\, x') : \Delta; \Delta$$

where $\Delta = x{:}\mathsf{unit}, x'{:}\mathsf{unit}, y{:}\mathsf{unit} \xrightarrow{\mathbf{1}} \mathsf{unit}$ contains y as well.

Lemma 7 (Congruence). *If $\Gamma \vdash C_1 : \Delta_1; \Delta_2$ and $C_1 \equiv C_2$ then $\Gamma \vdash C_2 : \Delta_1; \Delta_2$.*

Proof. By induction of the derivation of $C_1 \equiv C_2$. We consider only the cases of the four axioms listed in Fig. 2, the cases of the congruence rules are straightforward.

– Commutativity: Follows since the rule for parallel composition is symmetric.
– Associativity: Follows from the rule for parallel composition and the fact that all operations on contexts are associative.
– New names: If $\Gamma \vdash (\nu x)(\nu y)C_1 : \Delta_1; \Delta_2$ then $\Gamma \vdash C_1 : \Delta_1'; \Delta_2'$ where $\Delta_i = \Delta_i' - y - x = \Delta_i' - x - y$. Hence also $\Gamma \vdash (\nu y)(\nu x)C_1 : \Delta_1; \Delta_2$.
– Scope extrusion: By bound renaming, we can assume x does not appear in any of the contexts. Suppose $\Gamma \cdot \Delta \vdash ((\nu x)C_1) \mid C_2 : \Gamma_1, \Delta_1;\ \Gamma_3, \Delta_3$, with
  - $\Gamma, \Delta_2 \vdash (\nu x)C_1 : \Gamma_1; \Gamma_2 \cdot \Gamma_3$
  - $\Delta, \Gamma_2 \vdash C_2 : \Delta_1; \Delta_2 \cdot \Delta_3$ and
  - $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta_1) = \emptyset$ and $\mathsf{dom}(Delta) \cap \mathsf{dom}(\Gamma_1) = \emptyset$.

  So $\Gamma, \Delta_2 \vdash C_1 : \Gamma_1'; \Gamma_2' \cdot \Gamma_3'$ such that $\Gamma_1' - x = \Gamma_1$ and $\Gamma_2' \cdot \Gamma_3' - x = \Gamma_2 \cdot \Gamma_3$. Since x not in Δ, Γ_2 we also have $\Delta, \Gamma_2' \vdash C_2 : \Delta_1; \Delta_2 \cdot \Delta_3$ by Lemma 6 (3), and so $\Gamma \cdot \Delta \vdash (C_1 \mid C_2) : \Gamma_1', \Delta_1;\ \Gamma_3', \Delta_3$. Hence, $\Gamma \cdot \Delta \vdash (\nu x)(C_1 \mid C_2) : \Gamma_1, \Delta_1;\ \Gamma_3, \Delta_3$.
  The direction from right to left is similar, using Lemma 6 (4). □

Lemma 8 (Context). *Suppose $\varGamma \vdash D[C_1] : \varDelta_1; \varDelta_2$. Then there are $\varGamma', \varDelta'_1$ and $\varDelta'_2$ such that*

1. *$\varGamma' \vdash C_1 : \varDelta'_1; \varDelta'_2$, and*
2. *whenever $\varGamma' \vdash C_2 : \varDelta'_1; \varDelta'_2$ then $\varGamma \vdash D[C_2] : \varDelta_1; \varDelta_2$.*

Proof. By induction on the context D. ☐

Theorem 3 (Subject Reduction). *If $\varGamma \vdash C_1 : \varDelta_1; \varDelta_2$ and $C_1 \to C_2$ then $\varGamma \vdash C_2 : \varDelta_1; \varDelta_2$.*

Proof. Consider cases for reduction (cf. Fig. 5).

– Case (BETA): So $x \Leftarrow F[(\lambda y.e)\, v]$ reduces to $x \Leftarrow F[e[^v/_y]]$. By the typing rule for threads,

$$\varGamma, x{:}\alpha \vdash F[(\lambda y.e)\, v] : \alpha$$

with $|\alpha| = \omega$ and $\varDelta_1 = \varDelta_2 = x{:}\alpha$. By Lemma 5, $\varGamma_1, x{:}\alpha, y{:}\beta \vdash F[y] : \alpha$ and $\varGamma_2, x{:}\alpha \vdash (\lambda y.e)\, v : \beta$ where $\varGamma = \varGamma_1 \cdot \varGamma_2$. Hence $\varGamma'_2, x{:}\alpha, y{:}\beta' \vdash e : \beta$ and $\varGamma''_2, x{:}\alpha \vdash v : \beta'$ for some β' and $\varGamma_2 = \varGamma'_2 \cdot \varGamma''_2$. By Lemma 4, $\varGamma_2, x{:}\alpha \vdash e[^v/_y] : \beta$ and then

$$\varGamma, x{:}\alpha \vdash F[e[^v/_y]] : \alpha$$

So $\varGamma \vdash x \Leftarrow F[e[^v/_y]] : \varDelta_1; \varDelta_2$ by the typing rule for threads.

– Case (THREAD.NEW): So $x \Leftarrow F[\mathbf{thread}\, v]$ reduces to the configuration $C_2 \equiv (\nu y)(E[y] \mid y \Leftarrow v\, y)$ where E is $x \Leftarrow F$ and $x \neq y \notin \mathsf{fv}(F[v])$. By Lemma 5, $\varGamma_1, x{:}\alpha, y{:}\beta \vdash F[y] : \alpha$ and $\varGamma_2, x{:}\alpha \vdash \mathbf{thread}\, v : \beta$ where $\varGamma = \varGamma_1 \cdot \varGamma_2$ and $\varDelta_1 = \varDelta_2 = x{:}\alpha$. Hence, $\varGamma_2, x{:}\alpha \vdash v : \beta \xrightarrow{\kappa} \beta$ with $|\beta| = \omega$ by the rules for application and constants. We obtain $\varGamma_2, x{:}\alpha, y{:}\beta \vdash y \Leftarrow v\, y : (y{:}\beta; y{:}\beta)$ by the application and thread rules, and

$$\varGamma \vdash (x \Leftarrow F[y] \mid y \Leftarrow v\, y) : (x{:}\alpha, y{:}\beta; x{:}\alpha, y{:}\beta)$$

by thread and composition. By restriction $\varGamma \vdash (\nu y)(E[y] \mid y \Leftarrow v\, y) : \varDelta_1; \varDelta_2$ follows.

– Case (FUTURE.DEREF): By the definition, $(x \Leftarrow F_{\mathrm{f}}[y] \mid y \Leftarrow v)$ reduces to and $(x \Leftarrow F_{\mathrm{f}}[v] \mid y \Leftarrow v)$. By the parallel composition rule and Lemma 5,

$$\varGamma_1, y{:}\beta, x{:}\alpha \vdash F_{\mathrm{f}}[y] : \alpha \text{ and } \varGamma_2, x{:}\alpha, y{:}\beta \vdash v : \beta$$

where $|\alpha| = |\beta| = \omega$, $\varGamma = \varGamma_1 \cdot \varGamma_2$ and $\varDelta_1 = \varDelta_2 = x{:}\alpha, y{:}\beta$. By Lemma 1, $\mathsf{once}(\varGamma_2) = \emptyset$, so $\varGamma_2 \subseteq \varGamma_1$ and by Lemma 4 $\varGamma_1, x{:}\alpha, y{:}\beta \vdash F_{\mathrm{f}}[v] : \alpha$. So

$$\varGamma \vdash (x \Leftarrow F_{\mathrm{f}}[v] \mid y \Leftarrow v) : \varDelta_1; \varDelta_2$$

– Case (HANDLE.NEW): Similar to the case for (THREAD.NEW).

– Case (HANDLE.BIND): So $(x \Leftarrow F[z\,v] \mid z \mathsf{h}\, y)$ is reduced to $(x \Leftarrow F[\mathbf{unit}] \mid y \Leftarrow v \mid z \,\mathsf{h}\, \bullet)$. Then $\Delta_2 = x{:}\alpha, y{:}\beta$ and $\Delta_1 = \Delta_2, z{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$, and

$$\Gamma_1, y{:}\beta, z{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}, x{:}\alpha \vdash F[z\,v] : \alpha \text{ and } \Gamma_2 \vdash z \,\mathsf{h}\, y : \Delta'; \Delta'$$

for $\Gamma = \Gamma_1 \cdot \Gamma_2$ with $x, y \notin \Gamma_2$ and $\Delta' = y{:}\beta, z{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$. By Lemma 5 and Lemma 4, also

$$\Gamma_1', y{:}\beta, x{:}\alpha \vdash F[\mathbf{unit}] : \alpha \text{ and } \Gamma_1'', y{:}\beta, x{:}\alpha \vdash v : \beta$$

for $\Gamma_1 = \Gamma_1' \cdot \Gamma_1''$. Hence

$$\Gamma_1 \vdash (x \Leftarrow F[\mathbf{unit}] \mid y \Leftarrow v) : \Delta_2; \Delta_2$$

and

$$\Gamma \vdash (x \Leftarrow F[\mathbf{unit}] \mid y \Leftarrow v \mid z \,\mathsf{h}\, \bullet) : \Delta_1; \Delta_2$$

– Case (CELL.NEW): Similar to (THREAD.NEW) and (HANDLE.NEW).
– Case (CELL.EXCH): Similar to the case (HANDLE.BIND). □

The theorem now follows from Lemma 7 and Lemma 8 which show the soundness of the rules in Fig. 4.

Corollary 2 (Absence of Handle Errors). *If $\Gamma \vdash C : \Delta_1; \Delta_2$ then C is error-free.*

Proof. Suppose C has an error, i.e.,

$$C \;\equiv\; D[E_{\mathrm{f}}[y\ v] \mid y \,\mathsf{h}\, \bullet]$$

Further, suppose $\Gamma \vdash C : \Delta_1; \Delta_2$, so there exists Γ', Δ_1' and Δ_2' such that

$$\Gamma' \vdash (z \Leftarrow F_{\mathrm{f}}[y\ v] \mid y \,\mathsf{h}\, \bullet) : \Delta_1'; \Delta_2'$$

In fact, $\Delta_2' = z{:}\alpha$ and $\Delta_1' = \Delta_2', y{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$.

By parallel composition and thread, for some Γ_1 and Γ_2 such that $\Gamma' = \Gamma_1 \cdot \Gamma_2$ we have

$$\Gamma_1, z{:}\alpha \vdash F_{\mathrm{f}}[y\ v] : \alpha$$

Now $y \in \mathsf{fv}(F_{\mathrm{f}}[y\ v])$ since evaluation contexts do not involve any bindings. By Lemma 3, y must occur in $\Gamma_1 \subseteq \Gamma'$. By Lemma 6, $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Delta_1') = \emptyset$, a contradiction to $y{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit} \in \Delta_1'$.

Hence, C cannot be typable whenever it has an error. By Subject Reduction (Theorem 3), this proves that $\Gamma \vdash C : \Delta_1; \Delta_2$ implies C is error-free. □