A Concurrent Lambda-Calculus with Futures

J. Niehren [a], J. Schwinghammer [b,*], G. Smolka [b]

[a] *INRIA Futurs, Lille, France, Mostrare Project*

[b] *Saarland University, Saarbrücken, Germany, Programming Systems Lab*

Abstract

We introduce a new lambda calculus with futures, $\lambda(\mathsf{fut})$, that models the operational semantics of concurrent statically typed functional programming languages with mixed eager and lazy threads such as Alice ML, a concurrent extension of Standard ML. $\lambda(\mathsf{fut})$ is a minimalist extension of the call-by-value λ-calculus that is sufficiently expressive to define and combine a variety of standard concurrency abstractions, such as channels, semaphores, and ports. Despite its minimality, the basic machinery of $\lambda(\mathsf{fut})$ is sufficiently powerful to support explicit recursion and call-by-need evaluation.

We present a static type system for $\lambda(\mathsf{fut})$ and distinguish a fragment of $\lambda(\mathsf{fut})$ that we prove to be uniformly confluent. This result confirms our intuition that reference cells are the sole source of indeterminism. This fragment assumes the absence of so called *handle errors* that violate the single assignment assumption of $\lambda(\mathsf{fut})$'s handled future-construct.

Finally, we present a linear type system for $\lambda(\mathsf{fut})$ by which to prove the absence of handle errors. Our system is rich enough to type definitions of the above mentioned concurrency abstractions. Consequently, these cannot be corrupted in any (not necessarily linearly) well-typed context.

Keywords: functional programming languages, semantics, concurrency.

1 Introduction

Alice ML [1,2] is a concurrent programming language extending on Standard ML (SML) [3], where all synchronisation is based on *futures*, rather than

[*] Corresponding author.
URLs: `www.lifl.fr/~niehren` (J. Niehren), `www.ps.uni-sb.de/~jan` (J. Schwinghammer), `www.ps.uni-sb.de/~smolka` (G. Smolka).

channels [4–6] or joins [7]. The core language of Alice ML features functional programming with static type inference, extended by futures and concurrent threads, that may be eager or lazy and can be mixed. Alice extends further on SML by higher-order modules, packages that integrate dynamic typing, network distribution, and constraint programming [1,8,9]. Many ideas in Alice ML (except those for typing) are inspired by, and inherited from, the concurrent constraint programming language Mozart-Oz [10–12].

The original motivation of this paper has been to formally model the operational semantics of Alice ML. As usual, we have to be less ambitious, so here we restrict ourselves to the basic concepts of Alice ML's core language, concerning futures, concurrency, and typing. We will try to capture their essence rather than their concrete appearance.

Futures [13,14] are a restricted form of logic variables, which carefully separate read and write permissions. Otherwise, they share many characteristics of the logic variables in concurrent logic and concurrent constraint programming [15,16]:

Unknown information. A future is a *placeholder*, referring to a value that is completely described by some expression (possibly containing other futures). As long as this expression has not been evaluated, the value of the future is unknown.

Future value identification. Once the value of a future becomes available, the future is *identified* with it. More complex forms of unification are not needed, in contrast to partial descriptions by logic variables.

Implicit synchronisation. Synchronisation is *implicit* and *data-driven*, by suspending a computation whenever a future with unknown value is accessed, and resuming computation once this value has been determined.

Implicit synchronization by futures provides a convenient mechanism to deal with network latency, since it achieves low coupling between concurrent and possibly distributed computation threads. Calls to a remote site immediately return a future that refers to the result of the call; the actual result may be supplied much later. Since operations may simply continue with a placeholder as long as they do not need its value, the potential for concurrent and distributed computation is maximised by this form of *automatic data-driven synchronisation.*

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 and possibly distributed 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).

In contrast to logic variables, futures allow us to statically determine the data flow. Static data flow is an indispensable prerequisite for static type inference with parametric polymorphism as found 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 type inference in programming languages with unrestricted logic variables such as Oz [17,10] and in π-calculus based extensions of SML such as Pict [18]. The problem for π-calculus based channel approaches was solved with the join-calculus [6,19] and the corresponding programming language JoCaml [7] which extends on CAML [20]. The join-calculus, however, relies on the alternative mechanism of join patterns for synchronisation and does not model futures.

Previous λ-calculi with futures by Felleisen and Flanagan [21] were proposed to model the parallel execution of purely functional programs. They serve to describe a set of parallel threads that communicate through futures. Work by Moreau [22] showed how to extend this to a Scheme-like language with control operators. These calculi model languages that are confluent, and where the construction of cyclic data-structures is not possible.

In this article, we present a new lambda calculus with futures that we call $\lambda(\mathsf{fut})$. It models the operational semantics of concurrent, statically typed, functional programming languages extending SML, as provided by Alice ML. The requirements raised by concurrent computation necessitate a number of technically nontrivial extensions compared to the previous calculi of [21,22].

Indeterminism. Standard concurrency constructs are indeterministic, which is incompatible with previous confluence assumptions. We propose to add indeterminism via reference cells. These are the base components of traditional stores, already available in SML.

Single assignment. For expressing streams, ports, or channels, we will need a particular form of futures that we call *handled futures* which are introduced before any value descriptor is available. Handled futures come with a *handle* that grants once-only permission to write the value of the future. Any further attempt to use the same handle will raise a programming error, that we call a *handle error*. Handled futures have previously been known as Id's I-structures [23] and alternatively as promises [24].

Cyclic data-structures and explicit recursion. It is known that cells enable the construction of cyclic data structures, and that this is difficult to exclude statically by purely syntactical means. Handles have the same property. This raises a number of technical problems, since we cannot simply always replace a future by its value. The same problem is known for explicit recursion [25], which is indeed most naturally expressible by the concept of threads in $\lambda(\mathsf{fut})$.

Mixed eager and lazy theads. As we will see, the machinery we set up is already able to deal with *lazy theads*, whose values are evaluated by need [26,27]. Furthermore, lazy and eager threads can be mixed freely.

We present a static type system for $\lambda(\mathsf{fut})$. The experience with Alice ML's type system shows that it can be smoothly extended by parametric polymorphism in the style of SML [1]. We distinguish a statically typed fragment of $\lambda(\mathsf{fut})$ that we prove to be uniformly confluent. This result confirms our intuition that the reference cell construct is the sole source of indeterminism. Besides well-typedness, it assumes that programs do not exhibit *handle errors*, i.e., no attempts are made to use the same handle twice.

Finally, we present a linear type system for $\lambda(\mathsf{fut})$ inspired by [30], by which to prove the absence of handle errors. Our system is sufficiently rich to type definitions of the above mentioned concurrency abstractions, so that these cannot be corrupted in any (not necessarily linearly) well-typed context.

Outline. We present $\lambda(\mathsf{fut})$ in Section 2 and its static type system in Section 3. A confluent fragment is identified in Section 4. We show how to express diverse concurrency constructs in Section 5. Section 6 presents a linear type system for excluding handle errors and shows that it is sufficiently expressive to type implementations of channels and ports in $\lambda(\mathsf{fut})$.

An extended abstract of this article has appeared as [31]. In comparison, we have added the description of lazy threads, the confluence results, and included proofs.

2 Lambda Calculus with Futures

$\lambda(\mathsf{fut})$ is an extension of the call-by-value λ-calculus by reference cells (as featured by SML and CAML, but with *exchange*), concurrent threads, futures, and handles. We start with a minimalist variant of $\lambda(\mathsf{fut})$ that omits lazy threads, give some examples, and then show how to add lazy threads for free. A static type system for $\lambda(\mathsf{fut})$ will be given in Section 3.

[1] As in SML, the usual value restriction is to be imposed [28]. All more interesting question on Alice's type system are related to its module system [29].

$$x, y, z, f, g \in \mathit{Var}$$

$$c \in \mathit{Const} ::= \textbf{unit} \mid \textbf{cell} \mid \textbf{exch} \mid \textbf{thread} \mid \textbf{handle}$$

$$e \in \mathit{Exp} ::= x \mid c \mid \lambda x.e \mid e_1\, e_2$$

$$v \in \mathit{Val} ::= x \mid c \mid \lambda x.e \mid \textbf{exch}\, v$$

$$C \in \mathit{Config} ::= C_1 \mathrel{|} C_2 \mid (\nu x)C \mid x\,\mathsf{c}\,v \mid x{\Leftarrow}e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\,\bullet$$

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

2.1 Syntax

The syntax of $\lambda(\mathsf{fut})$ has two levels, the layer of λ-calculus expressions $e \in \mathit{Exp}$ for sequential functional computation inside of threads, and the layer of configurations $C \in \mathit{Config}$ for concurrent computation composing multiple sequential threads in parallel.

Fig. 1 introduces the syntax of $\lambda(\mathsf{fut})$. The expressions e of $\lambda(\mathsf{fut})$ are standard λ-terms with variables x, y and constants c. All new operations are introduced by (higher-order) constants. There are 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 futures concomitantly with threads or handles that can bind their values.

Values v are defined as usual in a call-by-value λ-calculus. They consist of variables, constants, abstractions, and partial applications of the curried two-argument operation **exch**.

Configurations C are reminiscent of expressions of the π-calculus. They are built from base components by parallel composition $C_1 \mathrel{|} C_2$ 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*. 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 future*. Finally, a used handle component $y\,\mathsf{h}\,\bullet$ means that y is a handle that has already been used to bind its future.

An original idea we contribute with $\lambda(\mathsf{fut})$ is to consider threads $x{\Leftarrow}e$ as possibly recursive equations $x = e$, but directed from right to left. This is since the

$$C_1 \mid C_2 \equiv C_2 \mid C_1 \qquad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)$$

$$(\nu x)(\nu y)C \equiv (\nu y)(\nu x)C \qquad ((\nu x)C_0) \mid C_1 \equiv (\nu x)(C_0 \mid C_1) \quad \text{if } x \notin \mathsf{fv}(C_1)$$

Fig. 2. Structural congruence

$$\frac{C_1 \to C_2}{C_1 \mid C \to C_2 \mid C} \qquad \frac{C_1 \to C_2}{(\nu x)C_1 \to (\nu x)C_2} \qquad \frac{C_1 \equiv C_1' \quad C_1' \to C_2' \quad C_2' \equiv C_2}{C_1 \to C_2}$$

Fig. 3. Thread selection

concurrent future x may occur in the expression e whose evaluation computes its future value. Such a thread can be created by evaluating **thread** $\lambda x.e$, where x may occur in e. The expression **handle** $\lambda x.\lambda y.e$ introduces a handle component $y \,\mathsf{h}\, x$ with static scope in e; an application $y\,v$ "consumes" the handle y and binds x to v, resulting in a used handle $y \,\mathsf{h}\, \bullet$ and thread $x{\Leftarrow}v$.

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, respectively, and $e[^{e'}/_{x}]$ for capture-free substitution of e' for x in e. Moreover, we use the usual syntactic sugar, writing $\mathsf{let}\ x_1{=}e_1\ \mathsf{in}\ e$ for $(\lambda x_1.e)\,e_1$, and $\lambda\_.e$ for $\lambda x.e$ where x is not free in e, and $e_1;\, e_2$ for $(\lambda\_.e_2)\,e_1$.

2.2 Operational Semantics

The operational semantics of $\lambda(\mathsf{fut})$ is given by a binary relation $C_1 \to C_2$ on configurations called (one-step) *reduction*. We assume that reduction is fair, i.e. that every redex will be eventually reduced in every complete reduction sequence (infinite or terminating).

2.2.1 Configurations

We do not want the order of components in configurations to matter, so we use the *structural congruence* $\equiv$ of the π-calculus in [32]. 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 identity expresses that the order of restricting names does not matter. The final rule is known as *scope extrusion* in π-calculus and is used to extend the scope of a local variable.

Every reduction step of $\lambda(\mathsf{fut})$ involves either one or two threads of a configuration. These threads can be freely selected according to the inference rules stated in Fig. 3: given a configuration C we choose a representation $(\nu x_1)\ldots(\nu x_n)(C' \mid C'')$ congruent to C and replace C' by one of its reducts.

2.2.2 Threads and Expressions

The strategy for reducing threads and expressions in $\lambda(\mathsf{fut})$ is specified using the evaluation contexts defined in Fig. 4. We base it on standard evaluation contexts F for call-by-value reduction and lift them to threads. 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. An evaluation context E is a thread where a subexpression in reduction position is left out. We write $E[e]$, and $F[e]$, respectively, for the object obtained by filling the hole in the context with an expression.

A nontrivial question is *when* to allow replacement of 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.

The alternative to permit future substitution in all evaluation contexts leads to confluence problems. Suppose that x is bound to value 5 by some thread $x{\Leftarrow}5$ and that another thread is evaluating the expression $(\lambda y.\lambda z.y)\ x$ 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. 4 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 a future x is needed in two situations, in function applications xv and for cell exchange $\mathbf{exch}\,x\,v$ in evaluation contexts. Apart from technical considerations, the adoption of future evaluation contexts is motivated by the

$$E ::= x \Leftarrow F \qquad E_{\mathsf{f}} ::= x \Leftarrow F_{\mathsf{f}}$$

$$F ::= [\,] \mid F\,e \mid v\,F \qquad F_{\mathsf{f}} ::= F[[\,]\,v] \mid F[\mathbf{exch}\,[\,]\,v]$$

Fig. 4. Evaluation contexts E, F and future evaluation contexts E_{f}, F_{f}.

(BETA)	$E[(\lambda y.e)\,v] \to E[e[{}^{v}\!/\!_{y}]]$
(THREAD.NEW)	$E[\mathbf{thread}\,v] \to (\nu y)(E[y] \mid y \Leftarrow v\,y) \qquad (y \notin \mathsf{fv}(E[v]))$
(FUTURE.DEREF)	$E_{\mathsf{f}}[y] \mid y \Leftarrow v \to E_{\mathsf{f}}[v] \mid y \Leftarrow v$
(HANDLE.NEW)	$E[\mathbf{handle}\,v] \to (\nu y)(\nu z)(E[v\,y\,z] \mid z\,\mathsf{h}\,y) \quad (y, z \notin \mathsf{fv}(E[v]))$
(HANDLE.BIND)	$E[z\,v] \mid z\,\mathsf{h}\,y \to E[\mathbf{unit}] \mid y \Leftarrow v \mid z\,\mathsf{h}\,\bullet$
(CELL.NEW)	$E[\mathbf{cell}\,v] \to (\nu y)(E[y] \mid y\,\mathsf{c}\,v) \qquad (y \notin \mathsf{fv}(E[v]))$
(CELL.EXCH)	$E[\mathbf{exch}\,y\,v_1] \mid y\,\mathsf{c}\,v_2 \to E[v_2] \mid y\,\mathsf{c}\,v_1$

Fig. 5. Reduction rules of operational semantics

behaviour of the Alice ML implementation [2]. The same mechanism has also proved useful to model more implementation oriented issues in [21]. Future evaluation contexts are called *placeholder strict* there.

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

Concurrent Futures and Threads. Besides β-reduction, there are six reduction rules in Fig. 5. Concurrent futures are created by rule (THREAD.NEW). Evaluating **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 may eventually assign its value to y,
- the concurrent future y is returned instantaneously in the original thread so that it is free to proceed, independent of the evaluation of e.

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

[2] In Alice ML, pattern matching gives rise to further future evaluation contexts.

Rule (FUTURE.DEREF) states when to replace 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.

We illustrate the first three rules at the following example where I is the lambda expression $\lambda z.z$.

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

The THREAD.NEW step for thread creation is followed by a trivial BETA step, that we left implicit in the previous explanation. The result is a configuration with two threads that we can reduce concurrently. We have a choice of BETA reducing the left or right thread first. We do the former:

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

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:

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

Handled Futures. Handles are needed together with cells in order to safely express nondeterministic concurrency abstractions as in Section 5 below. The idea of handled futures appeared before in the form of I-structures [23] and promises [24]. Rule (HANDLE.NEW) introduces a handled future jointly with a handle. Evaluating applications **handle** $\lambda x.\lambda y.e$ has the following effects:

- 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 z to value v reduces by binding the future associated to z to v. This action consumes the handle component $z\,\mathsf{h}\,y$; what remains is a used handle component $z\,\mathsf{h}\,\bullet$ as well as the binding $y{\Leftarrow}v$. Restricting the binding of handled futures to values, rather than forking a thread $y{\Leftarrow}e$ for any expression e, is not essential and merely reflects the call-by-value evaluation of Alice ML.

Trying to apply a handle a second time leads to a *handle error*:

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

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

Cells. 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. The first (cell) argument must be known, as expressed by the definition of future evaluation contexts E_{f}. Cell exchange is *atomic*, i.e., no other thread can interfere.

Suppose that r is a reference cell. While the primitive for cell access is atomic exchange, it is also possible to define a more conventional access operation $\mathsf{get\_content}(r)$ that does not change its contents. The naive approach of using two exchanges, writing an arbitrary intermediate value before writing back the correct one, is not thread-safe: Other threads might access the cell while it is in the inconsistent intermediate state. The solution is to use a handled future:

$$\mathsf{get\_content}(r) \;\equiv\; \mathbf{handle}(\lambda x \lambda h.\ \mathsf{let}\ v = \mathbf{exch}\,r\,x\ \mathsf{in}\ (h\,v;\ v))$$

The stored value, yet unknown, is exchanged with a future by $v = \mathbf{exch}\,r\,x$. Next, this future is replaced by the value v itself, by an application $h\,v$ of the handle. Finally, the value is returned as result. Other threads accessing the cell inbetween will find the future inside, which guarantees consistency. A similar idea is used in Section 5.1 to obtain a mutual exclusion mechanism.

Explicit Recursion. Threads of $\lambda(\mathsf{fut})$ are more general than previous future operators in that they can spawn recursive equations, binding a future to a value that may contain the future itself. Indeed, **thread** can replace a fixed point operator **fix**. Consider $x \Leftarrow (\mathbf{thread}\,\lambda f.\lambda x.(f\ x))\ z$ for instance. 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.

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

The first FUTURE.DEREF step indeed simulates the UNFOLD rule of the fixed point operator.

Handles introduce yet another mechanism for recursion, similar to cells that enable construction of cyclic heap structures and what is sometimes referred to as *recursion through the store* [33]. Such cyclic bindings are difficult to avoid by purely syntactic means:

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

Reduction starts with HANDLE.NEW and two BETA steps. The final step by HANDLE.BIND binds the future z to itself. Analogously, longer cyclic chains of futures may be constructed.

Futures in Alice ML. There are two main differences between the formalisation of futures in $\lambda(\mathsf{fut})$ just presented, and their implementation in the Alice ML language. First, concurrent futures in $\lambda(\mathsf{fut})$ are slightly more general, by directly permitting recursive use in the spawned expressions.

Second, handled futures are realised as *promises* in Alice ML. Rather than introducing handle and associated future simultaneously, as done by $\lambda(\mathsf{fut})$'s **handle** construct, the expression $\mathsf{promise}\,e$ creates the explicit handle p with type $\alpha\,\mathsf{promise}$. The future can be extracted as $\mathsf{future}\,p : \alpha$, and by applying $\mathsf{fulfill}(p, v)$ it is replaced by the value v. In $\lambda(\mathsf{fut})$, this is simplified by identifying the abstract type $\alpha\,\mathsf{promise}$ with $\alpha \rightarrow \mathsf{unit}$ and replacing $\mathsf{fulfill}$ by ordinary function application. Finally, in Alice ML an exception is raised in the case of handle errors.

The combination of futures and exceptions leads to the concept of a *failed future*: if the expression associated with a future evaluates to an exception packet, the exception is propagated if and when the future is touched to dereference its value, i.e., it is re-raised in the respective thread.

2.3 Extension by Lazy Futures and Threads

Even though eager in principle, it is trivial to extend $\lambda(\mathsf{fut})$ by mixed eager and lazy threads. This is since the base machinery for resolving futures (the FUTURE.DEREF rule) works by need. In contrast to Concurrent Haskell [34], all applications remain eager, only some designated threads become lazy.

We extend $\lambda(\mathsf{fut})$ by a single constant **lazy**, for introducing by-need threads that are suspended until their value is needed. We add one more additional basic component, $x \overset{susp}{\Longleftarrow} e$, to represent a suspended computation in a configuration. The associated transition rules given in Fig. 6 are analogous to the ones for concurrent threads.

$$
\begin{array}{lll}
(\text{LAZY.NEW}) & E[\mathbf{lazy}\, v] \to (\nu x)(E[x] \mid x \overset{susp}{\Longleftarrow} v\, x) & (x \notin \mathsf{fv}(E[v])) \\
(\text{LAZY.TRIGGER}) & E_{\mathsf{f}}[x] \mid x \overset{susp}{\Longleftarrow} e \to E_{\mathsf{f}}[x] \mid x \Leftarrow e &
\end{array}
$$

Fig. 6. Operational semantics of lazy threads

The transition (LAZY.NEW) introduces a new suspended computation $x \overset{susp}{\Longleftarrow} v\, x$ to the configuration; as with concurrent threads, the associated (by-need) future x may be used to evaluate $v\, x$. A suspended computation is triggered if (and when) its value is needed for the first time, i.e., if the associated future x occurs in a future dereference context.

2.4 Are Handled Futures Redundant?

It is natural to ask if handled futures have to be included as primitive construct in $\lambda(\mathsf{fut})$. We conjecture that they cannot be expressed in terms of the remaining constructs of $\lambda(\mathsf{fut})$, unless one is willing to change the termination behaviour of programs.

It is instructive to see what goes wrong with the naive encoding approach: The idea is to introduce a cell for every handled future that contains the value of the future once available, and some distingushed dummy value initially. To access the value of a future, the reader needs to wait until the cell's content becomes distinct from the dummy value. Testing for this can be done by polling, i.e. by accessing the cell's content repeatedly until the dummy value is replaced by a proper value.

The problem with this kind of encoding is that future access may lead to unwanted nontermination; this will happen in contexts where the future's value will never be written. Implementations of handled futures avoid this kind of polling, by the same technique that is used for implementing concurrent futures: All threads waiting for the value of a future are written into a suspension list. When a value is assigned to the future, all threads in that list are notified and re-actived. It is not obvious, however, how to express this implementation trick as a $\lambda(\mathsf{fut})$ program such that termination is preserved in arbitrary contexts.

3 Static Typing

Since our intention is to model extensions of the statically typed language ML we restrict our calculus to be statically typed. We present a simple type system that provides function types $\alpha \to \beta$, the type $\alpha\, \mathsf{ref}$ of reference cells containing

Types $\alpha, \beta \in \mathit{Type} ::= \mathsf{unit} \mid \alpha \to \beta \mid \alpha\,\mathsf{ref}$

Typing of constants

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

$\mathbf{thread} : (\alpha \to \alpha) \to \alpha$ $\qquad$ $\mathbf{exch} : \alpha\,\mathsf{ref} \to \alpha \to \alpha$

$\mathbf{handle} : (\alpha \to (\alpha \to \mathsf{unit}) \to \beta) \to \beta$

Typing of expressions

$$\frac{}{\Gamma \vdash c : \mathit{TypeOf}(c)} \qquad \frac{}{\Gamma, x{:}\alpha \vdash x : \alpha}$$

$$\frac{\Gamma, x{:}\alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\, e_2 : \beta}$$

Fig. 7. Typing of expressions

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.

3.1 Well-Formedness

On the level of configurations, types allow us to state a type preservation theorem during evaluation. Besides, the type system will ensure a number of natural and more basic *well-formedness conditions*: 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, all of the following configurations are ill-formed:

- $x\,\mathsf{h}\,x$, or $x{\Leftarrow}e \mid x'\,\mathsf{h}\,x$, or $x{\Leftarrow}e \mid x\,\mathsf{h}\,x'$, or $x{\Leftarrow}e \mid x\,\mathsf{h}\,\bullet$
- $x{\Leftarrow}e_1 \mid x{\Leftarrow}e_2$, or $y\,\mathsf{h}\,x_1 \mid y\,\mathsf{h}\,x_2$

More precisely, a configuration C is *well-formed* if there is no $C' \equiv C$ such that C' contains one of the above as a subconfiguration.

The type system given next will require that the variables introduced by C_1 and C_2 are disjoint in concurrent compositions $C_1 \mid C_2$, thereby entailing well-formedness of well-typed configurations: none of the above configurations is typable. That well-formedness is preserved by reduction for typable configurations is therefore an immediate consequence of the subject reduction theorem.

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

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

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

Fig. 8. Typing rules for configurations

3.2 Typing of Expressions and Configurations

According to the operational semantics described in Section 2, the constants obtain their natural types. For instance, **thread** has type $(\alpha \rightarrow \alpha) \rightarrow \alpha$ 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 on the left of Figure 7 and can be justified accordingly.

Let Γ and Δ range over *type environments*, i.e. finite functional relations between *Var* and *Type* which we write as $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$ where all x_i are distinct. In particular, in writing Γ_1, Γ_2 we assume that the respective domains are disjoint. Writing *TypeOf*(c) for the type of constant c we have the usual type inference rules for simply typed lambda calculus, with judgments of the form $\Gamma \vdash e : \alpha$ (Figure 7).

Types are lifted to configurations according to the inference rules in Fig. 8. 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 [35]. 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 respectively, are disjoint. Note how this prevents ill-formed configurations, as on page 13, 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

$$\vdash (x \Leftarrow y\ \mathbf{unit} \mid y\ \mathsf{h}\ z) : (x{:}\mathsf{unit}, z{:}\mathsf{unit}, y{:}\mathsf{unit} \to \mathsf{unit}) \tag{1}$$

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* $\Gamma \vdash C_2 : \Gamma'$.

PROOF SKETCH. The proof of Theorem 1 proceeds by first establishing that typing is preserved by the replacement of well-typed subconfigurations and by structural congruence, and then considering the basic reductions of Fig. 5. This is analogue to the (more complicated) subject reduction proof for linear types which is given in detail in Section 6.3.

Program errors are notorious even for a statically typed programming language. In particular, it turns out that the class of *handle errors* is not excluded by the type system just presented. One approach to address such shortcomings is to refine the type system so that typing provides additional guarantees. Section 6 presents a linear type system that guarantees that every handled future is initialised only once.

4 Uniform Confluence

Uniform confluence is a strong notion of confluence that has been shown for fragments of concurrent calculi [12,36–39]. A number of similar confluence results have been proved previously for calculi with futures [21].

Definition 2 (Uniform Confluence, [12]) *A relation* $\to$ *is* uniformly confluent *if, whenever* $C_1 \leftarrow C \to C_2$ *and* $C_1 \not\equiv C_2$*, there exists* C' *such that* $C_1 \to C' \leftarrow C_2$.

This is depicted on the right. Uniform confluence implies confluence, and if $\rightarrow$ is uniformly confluent then all maximal $\rightarrow$ sequences beginning from C have the same length (which may be finite or infinite). In [12] this fact has been exploited to compare the (time) complexity of different evaluation strategies for lambda calculus. Clearly, the availability of mutable state in the calculus breaks confluence. For instance, the configuration

$$\begin{tikzcd} & C \arrow[dl] \arrow[dr] & \\ C_1 \arrow[dr, dashed] & \not\equiv & C_2 \arrow[dl, dashed] \\ & C' & \end{tikzcd}$$

$$x \Leftarrow \mathbf{exch}\, r\, 1 \mid y \Leftarrow \mathbf{exch}\, r\, 2 \mid r\, \mathsf{c}\, v$$

may lead to distinct irreducible configurations, where the cell r contains either 1 or 2. However, assuming programs are free of handle errors, in the sense that they will never reduce to configurations containing such an error, we can establish uniform confluence for the fragment of $\lambda(\mathsf{fut})$ without reference cells. It is comparatively easy to prove, basically because reduction within each thread is deterministically call-by-value due to the use of evaluation contexts; this is the essence of Lemma 3.

Lemma 3 (Unique Decomposition) *If $e \in Exp$ then either $e \in Val$ or else there exists a* unique *decomposition of e into an evaluation context F and a subexpression, of the form $F[x\, v]$, $F[(\lambda x.e)\, v]$, $F[c\, v]$ where $c \neq$* **exch**, *or $F[\mathbf{exch}\, v\, v']$.*

PROOF. By induction on the structure of e.

Theorem 4 (Uniform Confluence) *The reduction relation $\rightarrow$ of $\lambda(\mathsf{fut})$, restricted to well-typed[3] and handle error-free configurations not containing* **exch**, *is uniformly confluent.*

PROOF. First observe that whenever $C \rightarrow C'$ then there are variables $\overline{x}$ and configurations C_1, C_1' and C_2 such that $C \equiv (\nu\overline{x})(C_1 \mid C_2)$ and $C' \equiv (\nu\overline{x})(C_1' \mid C_2)$; and C_1 and C_1' are lhs and rhs instances of one of the rules of Fig. 5. This can be proved by induction on the derivation of $C \rightarrow C'$.

It suffices to consider all possible combinations of reduction axioms leading to a situation $C_1 \leftarrow C \rightarrow C_2$ with $C_1 \not\equiv C_2$. We treat only some representative cases here.

- Case (BETA) and (BETA): This is established easily, since by unique decomposition (Lemma 3), the reductions to $C_1 \not\equiv C_2$ must originate in different threads, i.e.,

$$C \equiv (\nu\overline{x})(x \Leftarrow F[(\lambda y.e)\, v] \mid x' \Leftarrow F'[(\lambda y'.e')\, v'] \mid C')$$

[3] In fact, only well-formedness is used.

Thus, C_1 and C_2 can be joined by another (BETA) step,

$$C_1 \;\rightarrow\; (\nu\overline{x})(\, x{\Leftarrow}F[e[v/y]] \,\mid\, x'{\Leftarrow}F'[e'[v'/y']] \,\mid C') \;\leftarrow\; C_2$$

- Case (HANDLE.BIND) and (HANDLE.BIND): Again, by unique decomposition (Lemma 3), the reductions must originate in different threads, so that

$$C \equiv (\nu\overline{x})(x{\Leftarrow}F[z\, v] \mid x'{\Leftarrow}F'[z'\, v'] \mid C'')$$

where $C'' \equiv z\,\mathsf{h}\,y \mid C_1' \equiv z'\,\mathsf{h}\,y' \mid C_2'$ for some C_1', C_2', and

$$\begin{aligned} C_1 &\equiv (\nu\overline{x})(\, x{\Leftarrow}F[\mathbf{unit}] \mid y{\Leftarrow}v \mid z\,\mathsf{h}\,\bullet \,\mid x'{\Leftarrow}F'[z'\, v'] \mid C_1') \\ C_2 &\equiv (\nu\overline{x})(x{\Leftarrow}F[z\, v] \mid\, x'{\Leftarrow}F'[\mathbf{unit}] \mid y'{\Leftarrow}v' \mid z'\,\mathsf{h}\,\bullet \,\mid C_2') \end{aligned}$$

By well-formedness of the configuration, there is at most one component that introduces z as handle, and similarly for z'. Therefore, by the assumption of error-freeness of C, it follows that z and z' are indeed distinct handles. Hence, there exists C_3 such that $C'' \equiv z\,\mathsf{h}\,y \mid z'\,\mathsf{h}\,y' \mid C_3$ and such that C_1 and C_2 may be joined by (HANDLE.BIND) steps to obtain

$$C' \equiv (\nu\overline{x})(\, x{\Leftarrow}F[\mathbf{unit}] \mid y{\Leftarrow}v \mid z\,\mathsf{h}\,\bullet \,\mid\, x'{\Leftarrow}F'[\mathbf{unit}] \mid y'{\Leftarrow}v' \mid z'\,\mathsf{h}\,\bullet \,\mid C_3)$$

- Case (FUTURE.DEREF) and (FUTURE.DEREF): There must be configurations C_1' and C_2' and decompositions of C,

$$\begin{aligned} C &\equiv (\nu\overline{x})(\, x{\Leftarrow}E_{\mathrm{f}}[y] \mid y{\Leftarrow}v \,\mid C_1'), & C_1 &\equiv (\nu\overline{x})(\, x{\Leftarrow}E_{\mathrm{f}}[v] \mid y{\Leftarrow}v \,\mid C_1'), \\ C &\equiv (\nu\overline{x})(\, x'{\Leftarrow}E_{\mathrm{f}}'[y'] \mid y'{\Leftarrow}v' \,\mid C_2'), & C_2 &\equiv (\nu\overline{x})(\, x'{\Leftarrow}E_{\mathrm{f}}'[v'] \mid y'{\Leftarrow}v' \,\mid C_2') \end{aligned}$$

By the well-formedness assumption, $x \neq x'$, for otherwise also $y = y'$ and $v = v'$, contradicting the assumption $C_1 \not\equiv C_2$. Thus, assuming $y \neq y'$, there exists C'' such that $C \equiv (\nu\overline{x})(x{\Leftarrow}E_{\mathrm{f}}[y] \mid y{\Leftarrow}v \mid x'{\Leftarrow}E_{\mathrm{f}}'[y'] \mid y'{\Leftarrow}v' \mid C'')$ and C_1 and C_2 can be joined by further (FUTURE.DEREF) steps to

$$(\nu\overline{x})(\, x{\Leftarrow}E_{\mathrm{f}}[v] \mid y{\Leftarrow}v \,\mid\, x'{\Leftarrow}E_{\mathrm{f}}'[v'] \mid y'{\Leftarrow}v' \,\mid C'')$$

Similarly, if $y = y'$ then $v = v'$, $C \equiv (\nu\overline{x})(x{\Leftarrow}E_{\mathrm{f}}[y] \mid x'{\Leftarrow}E_{\mathrm{f}}'[y] \mid y{\Leftarrow}v \mid C'')$ for some C'', and C_1 and C_2 can be joined by (FUTURE.DEREF) steps to

$$(\nu\overline{x})(\, x{\Leftarrow}E_{\mathrm{f}}[v] \mid x'{\Leftarrow}E_{\mathrm{f}}'[v] \mid y{\Leftarrow}v \,\mid C'')$$

- The remaining cases are similar; in particular those reductions $C_1 \leftarrow C \rightarrow C_2$ caused by different rules tend to be more straightforward.

$$\begin{aligned}\mathsf{mutex} \equiv\ & \text{let } t = \lambda a.(a\ \mathbf{unit}) \text{ in let } r = \mathbf{cell}\ t \\ & \text{in } \lambda a.(\mathbf{handle}\ \lambda x \lambda h.\ \text{let } v = (\mathbf{exch}\, r\, x)\, a \text{ in } h\, t; v)\end{aligned}$$

Fig. 9. Mutual exclusion in $\lambda(\mathsf{fut})$

5 Concurrency Constructs

We now show how to express various concurrency abstractions in $\lambda(\mathsf{fut})$ which demonstrates the expressive power of handled futures.

5.1 Mutual Exclusion

When concurrent threads access shared data it is often necessary that they do not interfere, in order to prevent any data inconsistencies. Mutual exclusion is a technique for avoiding such inconsistencies. The idea is that at most one thread can access a *critical region* in which to do its actions. We implement an operation mutex of type $(\mathsf{unit} \to \alpha) \to \alpha$ in Fig. 9. As its argument, it receives an action of type $\mathsf{unit} \to \alpha$ which it applies under mutual exclusion in some critical region. At every time point, there is a unique permission t provided by the mutex that is needed to trigger actions a passed to the mutex. The trigger resides in the cell r of the mutex when available. Otherwise, the cell contains a future that will get bound to t when it is given back.

$$x \Leftarrow \mathsf{mutex}\ \mid y_1 \Leftarrow x\ a_1\ \mid y_2 \Leftarrow x\ a_2$$

Future evaluation contexts are important for the application $(\mathbf{exch}\, r\, x)\, a$ where the trigger $\mathbf{exch}\, r\, x$ needs to be known before it can trigger the action a.

5.2 Ports and Channels

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 an "open-ended" list $v_1 :: \cdots :: v_n :: x$ where x is a (handled) future. Thus, 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 $\mathsf{newPort} : \mathsf{unit} \to (\alpha\mathsf{list} \times \alpha {\to} \mathsf{unit})$ that creates a new *port* can be implemented as follows.

$$\begin{aligned}\mathsf{newPort} \equiv\ & \lambda\_.\mathbf{handle}(\lambda s \lambda bind_s. \\ & \quad \mathsf{let}\ putr = \mathbf{cell}\, bind_s\end{aligned}$$

$$
\begin{array}{l}
\mathsf{newChannel} \equiv \lambda\_.\mathbf{handle}(\lambda \mathit{init} \lambda \mathit{bind}_{\mathit{init}}.\\
\qquad\qquad \mathsf{let}\ \mathit{putr} = \mathbf{cell}\, \mathit{bind}_{\mathit{init}}\\
\qquad\qquad\qquad \mathit{getr} = \mathbf{cell}\, \mathit{init}\\
\qquad\qquad\qquad \mathit{put} \ \ = \lambda x.\mathbf{handle}(\lambda n \lambda \mathit{bind}_n.(\mathbf{exch}\, \mathit{putr}\, \mathit{bind}_n)\ (x :: n))\\
\qquad\qquad\qquad \mathit{get} \ \ = \lambda\_.\mathbf{handle}(\lambda n \lambda \mathit{bind}_n.\mathsf{case}(\mathbf{exch}\, \mathit{getr}\, n))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{of}\ x :: c\ \Rightarrow \mathit{bind}_n(c); x)\\
\qquad\qquad \mathsf{in}\ (\mathit{put}, \mathit{get}))
\end{array}
$$

Fig. 10. Channels in $\lambda(\mathsf{fut})$

$$
\begin{array}{l}
\qquad \mathit{put} \ = \lambda x.\mathbf{handle}(\lambda s \lambda \mathit{bind}_s.(\mathbf{exch}\, \mathit{putr}\, \mathit{bind}_s)\ (x :: s))\\
\mathsf{in}\ (s, \mathit{put}))
\end{array}
$$

The port consists of a stream $s : \alpha\mathsf{list}$ and an operation $\mathit{put} : \alpha \rightarrow \mathsf{unit}$ to append new messages to the stream. The stream is ended by a handled future, which in the beginning is just the future s itself. Its handle bind_s is stored in the cell $\mathit{putr} : (\alpha \rightarrow \mathsf{unit})\ \mathsf{ref}$ 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.

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 $\mathsf{newChannel} : \mathsf{unit} \rightarrow (\alpha \rightarrow \mathsf{unit}) \times (\mathsf{unit} \rightarrow \alpha)$ that generates channels for α values is listed in Fig. 10.

Given a channel, applying $\mathit{get} : \mathsf{unit} \rightarrow \alpha$ 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 dereferencing and mutual exclusion above to make the implementation thread-safe.

6 Linear Types for Single Assignment by Handles

We refine the type system by linear types, which serve as a proof tool to facilitate reasoning about the absence of handle errors (we do not want to argue that linear types should be used in programming practice). It can be used to prove the safety of libraries (for instance of concurrency abstractions) implemented in $\lambda(\mathsf{fut})$ with respect to the usage of handles.

Most previous uses of linear type systems in functional languages, such as the uniqueness typing of Clean, aimed at preserving referential transparency in the presence of side-effects, and taking advantage of destructive updates for efficiency reasons [40,41]. In contrast, our system rules out a class of programming errors, by enforcing the single-assignment property for handled futures.

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

unit : unit

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

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

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

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

Operations on type environments

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

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

Fig. 11. Linear types

The linear type system is sufficiently expressive to type the concurrency abstractions of Section 5 and others. 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.

6.1 *Multiplicities and Linear Typing of Expressions and Configurations*

We annotate types with *usage information* in the sense of [30]. In our case it suffices to distinguish between linear (i.e., exactly one) and nonlinear (i.e., any number of times) uses, where "use" means a safe approximation of the number of applications to a term. 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 be duplicated without restriction (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 applied 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. 11).

For a context Γ we write $\mathsf{once}(\Gamma)$ for the set of variables occuring in Γ with linear multiplicity. We write $\Gamma = \Gamma_1 \cdot \Gamma_2$ if Γ can be "split" into Γ_1 and Γ_2, in the sense that Γ_1 and Γ_2 consist of a partition of $\mathsf{once}(\Gamma)$, and each contains

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

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

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

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

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

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

Fig. 12. Linear typing rules for $\lambda(\mathsf{fut})$ expressions and configurations

all the variables of Γ with multiplicity ω. Table 11 defines this formally.

The types of 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. We achieve this 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[4].

The type rules for expressions are given in Fig. 12. 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 $\alpha \xrightarrow{\omega} \beta$ (whose values are freely copyable) 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; this prevents duplication of linear values. The annotation κ is irrelevant here, but the type of function and argument must match exactly.

[4] The derived dereferencing operation $\mathsf{get\_content}(r)$ permits duplication of the contents. However, in this case the cell r is forced to have type $\alpha\,\mathsf{ref}$ where $|\alpha| = \omega$.

The rules for configurations (Fig. 12) have changed: Judgements 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 ill-formed configurations on page on page 13) by means of invariant (i).
- Variables with multiplicity $\mathbf{1}$ 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 (1) on page 15 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 $\mathbf{1}$. 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. 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 Section 3, 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 $\Delta_2 \cdot \Delta_3$, respectively). A variable with multiplicity $\mathbf{1}$ 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, Γ_1 and Δ_1 have disjoint domains which in particular ensures 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 5 (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 6 (Absence of Handle Errors) *If* $\Gamma \vdash C : \Delta_1; \Delta_2$ *then* C *is error-free.*

$$\begin{array}{lll} \mathit{action}_\kappa(\alpha) = \mathsf{unit} \xrightarrow{\kappa} \alpha & \mathit{trigger}_\kappa(\alpha) = \mathit{action}_\kappa(\alpha) \xrightarrow{\omega} \alpha & \\ \Gamma(r) = \mathit{trigger}_\kappa(\alpha)\ \mathsf{ref} & \Gamma(h) = \mathit{trigger}_\kappa(\alpha) \xrightarrow{\mathbf{1}} \mathsf{unit} & \Gamma(t) = \mathit{trigger}_\kappa(\alpha) \\ \Gamma(x) = \mathit{trigger}_\kappa(\alpha) & \Gamma(a) = \mathit{action}_\kappa(\alpha) & \Gamma(v) = \alpha \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma(r,a,x) \vdash (\mathbf{exch}\, r\, x)\, a : \alpha} \qquad \dfrac{\cdots}{\Gamma(t,h,v) \vdash h\ t\,;v : \alpha}}{\Gamma(r,t,a,x,h) \vdash \underbrace{\text{let } v = (\mathbf{exch}\, r\, x)\, a;\ \text{in } h\, t\,; v}_{e} : \alpha}}{\Gamma(r,t,a,x) \vdash \lambda h.e : \Gamma(h) \xrightarrow{\kappa} \alpha}}{\Gamma(r,t,a) \vdash \lambda x.\lambda h.e : \Gamma(x) \xrightarrow{\kappa} \Gamma(h) \xrightarrow{\kappa} \alpha}}{\Gamma(r,t,a) \vdash \mathbf{handle}\ \lambda x.\lambda h.e : \alpha}}{\cdots}}{\vdash \mathsf{mutex} : \mathit{action}_\kappa(\alpha) \xrightarrow{\omega} \alpha}$$

Fig. 13. Safety proof for mutual exclusion protocol

The proofs of Theorem 5 and Corollary 6 are given in Section 6.3 below.

6.2 Proving Safety

The abstractions defined in Section 5 are *safe*, in the sense that no handle errors are raised by using them. For instance, we can always send to a 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 Corollary 6, typability guarantees the absence of handle errors. Moreover, all the abstractions we have given 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.

As the derivation in Fig. 13 shows, the mutual exclusion mutex may be typed as

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

in the linear type system. Both $\kappa = 1$ and $\kappa = \omega$ are possible. More importantly, the type of mutex itself has multiplicity ω, which allows mutex to be copied and applied any number of times.

In the derivation, we write $\Gamma(x_1,\ldots,x_n)$ for the environment $\Gamma(x_1),\ldots,\Gamma(x_n)$. Let us consider the case where $\kappa = \omega$. In this case, the single once-only type introduced by the derivation is the type of the handle $\Gamma(h)$. This once-only type is removed from the type environment when lambda abstracting over h. This permits to abstract over x with multipicity ω, since $\mathsf{once}(\Gamma(r,t,a)) = \emptyset$. In the case $\kappa = 1$, we cannot apply this rule, but we can abstract over x with the other rule for multiplicity 1.

For the list and pair types, the linear type system has to be extended accordingly; the details of such an extension are quite standard (see [30], for instance). Just as with function types these new types $\alpha \times^\kappa \beta$ and $\alpha\,\mathsf{list}^\kappa$ are annotated with multiplicities, satisfying the additional constraint that whenever $|\alpha| = \mathbf{1}$ or $|\beta| = \mathbf{1}$ then also $\kappa = \mathbf{1}$.

The subject reduction theorem can be extended, as can Corollary 6. For the port abstraction, we may then 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 $\mathsf{newPort}$ itself and the *put* operation (the second component of the result pair) can be used unrestrictedly. Similarly, if $|\alpha| = \omega$,

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

can be derived for the implementation of channels.

6.3 Proof of Subject Reduction

In the remainder of this section we sketch the proof of Theorem 5. The following lemmas relate to (linear) variables in contexts and substitution. All of these are fairly standard properties of linear type systems.

Lemma 7 *Suppose* $\Gamma \vdash e : \alpha$.

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

PROOF. The claims can be proved by induction on the derivation $\Gamma \vdash e : \alpha$.

Lemma 8 (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 the structure of e.

- Case e is of the form $c \in \mathit{Const}$: By the typing rules for constants, also $\Gamma \vdash e : \alpha$, and $\mathsf{once}(\Gamma) = \emptyset$ and $|\beta| = \omega$. By Lemma 7(1), $|\beta| = \omega$ implies $\mathsf{once}(\Gamma') = \emptyset$, and so by repeated applications of Lemma 7(2) and the fact $e[{}^{v}\!/x] \equiv c$ we obtain $\Gamma \cdot \Gamma' \vdash e[{}^{v}\!/x] : \alpha$.
- Case e is $y \in \mathit{Var}$: 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 7(2) yield the desired result.
  If $x \neq y$ then by the rule for variables $|\beta| = \omega$. So by Lemma 7(1) this yields $\mathsf{once}(\Gamma') = \emptyset$, and Lemma 7(2) and $e[{}^{v}\!/x] \equiv y$ show $\Gamma \cdot \Gamma' \vdash e[{}^{v}\!/x] : \alpha$.
- 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 7(1) implies $\mathsf{once}(\Gamma') = \emptyset$. Hence, $\Gamma \cdot \Gamma' \vdash (e_1\, e_2)[{}^{v}\!/x] : \alpha$ by Lemma 7(2).
  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, $(\Gamma_1 - x) \cdot \Gamma' \vdash e_1[{}^{v}\!/x] : \alpha' \xrightarrow{\kappa} \alpha$. By Lemma 7(4) we know $e_2[{}^{v}\!/x] \equiv e_2$. Using Lemma 7(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 occurring 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 $\mathsf{once}(\Gamma, x{:}\beta) = \emptyset$, in particular, $|\beta| = \omega$. Hence by Lemma 7(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 9 *Suppose* $\Gamma \vdash C : \Delta_1; \Delta_2$*. Then the following hold.*

(1) $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta_1) = \emptyset$.
(2) $\Delta_2 \subseteq \Delta_1$.
(3) If $x \notin \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Delta_1)$ *and* $|\alpha| = \omega$*, then* $\Gamma, x{:}\alpha \vdash C : \Delta_1; \Delta_2$.
(4) If $x \notin \mathsf{fv}(C)$ *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 the side condition $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Gamma_1) = \emptyset$ in the rule for parallel composition is essential for Lemma 9(1).

Lemma 10 (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 on the derivation of $C_1{\equiv}C_2$. The congruence rules are straightforward; associativity and commutativity follow from the parallel composition rule by observing that the operations Γ, Δ and $\Gamma{\cdot}\Delta$ on contexts are both associative and commutative. The case of name restriction is immediate. The left-to-right direction in the case of scope extrusion follows with Lemma 9(3); the direction from right to left follows by Lemma 9(4).

Lemma 11 (Context)

(1) Suppose $\Gamma \vdash F[e] : \alpha$. *Then there exist* x, Γ_1, Γ_2 *and* β *such that* $\Gamma = \Gamma_1{\cdot}\Gamma_2$ *and* $\Gamma_1, x{:}\beta \vdash F[x] : \alpha$ *and* $\Gamma_2 \vdash e : \beta$.
(2) Suppose $\Gamma \vdash C_1 \mid C : \Delta_1; \Delta_2$. *Then there are* Γ', Δ_1' *and* Δ_2' *such that* $\Gamma' \vdash C_1 : \Delta_1'; \Delta_2'$, *and whenever* $\Gamma' \vdash C_2 : \Delta_1'; \Delta_2'$ *then* $\Gamma \vdash C_2 \mid C : \Delta_1; \Delta_2$.
(3) Suppose $\Gamma \vdash (\nu x)C_1 : \Delta_1; \Delta_2$. *Then there are* Γ', Δ_1' *and* Δ_2' *such that* $\Gamma' \vdash C_1 : \Delta_1'; \Delta_2'$, *and if* $\Gamma' \vdash C_2 : \Delta_1'; \Delta_2'$ *then* $\Gamma \vdash (\nu x)C_2 : \Delta_1; \Delta_2$.

Note that the first part holds in particular for future evaluation contexts F_{f}, since these form a subset of evaluation contexts.

PROOF. The first part is by induction on the evaluation context F; parts (2) and (3) follow immediately from the typing rules.

PROOF of Subject Reduction (Theorem 5). It suffices to consider the case where C_1 and C_2 are the left-hand side and right-hand side, respectively, of one of the basic reductions of Fig. 5. The theorem then follows by Lemmas 10 and 11(2,3) that show soundness of the rules in Fig. 3.

- Case (BETA): So $x{\Leftarrow}F[(\lambda y.e)\, v]$ reduces to $x{\Leftarrow}F[e[{}^{v}\!/\!_{y}]]$. By assumption, $\Gamma \vdash x{\Leftarrow}F[(\lambda y.e)\, v] : \Delta_1; \Delta_2$ so that by the typing rule for threads,

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

  with $|\alpha| = \omega$ and $\Delta_1 = \Delta_2 = x{:}\alpha$. By Lemma 11(1), $\Gamma_1, x{:}\alpha, y{:}\beta \vdash F[y] : \alpha$ and $\Gamma_2, x{:}\alpha \vdash (\lambda y.e)\, v : \beta$ for some Γ_1, Γ_2 such that $\Gamma = \Gamma_1 \cdot \Gamma_2$. Hence $\Gamma_2', x{:}\alpha, y{:}\beta' \vdash e : \beta$ and $\Gamma_2'', x{:}\alpha \vdash v : \beta'$ for some β' and $\Gamma_2 = \Gamma_2' \cdot \Gamma_2''$. By Lemma 8, $\Gamma_2, x{:}\alpha \vdash e[{}^{v}\!/\!_{y}] : \beta$ and therefore $\Gamma, x{:}\alpha \vdash F[e[{}^{v}\!/\!_{y}]] : \alpha$. Thus,

$$\Gamma \vdash x{\Leftarrow}F[e[{}^{v}\!/\!_{y}]] : \Delta_1; \Delta_2$$

  by the typing rule for threads.
- Case (THREAD.NEW): So $C_1 \equiv x{\Leftarrow}F[\textbf{thread}\, v]$ reduces to the configuration $C_2 \equiv (\nu y)(x{\Leftarrow}F[y] \mid y{\Leftarrow}v\, y)$, where $y \notin \mathsf{fv}(F[v])$ and $y \neq x$. By assumption, $\Gamma \vdash C_1 : \Delta_1; \Delta_2$. So by the typing rule for thread components and Lemma 11, $\Gamma_1, x{:}\alpha, y{:}\beta \vdash F[y] : \alpha$ and $\Gamma_2, x{:}\alpha \vdash \textbf{thread}\, v : \beta$ for some Γ_1 and Γ_2 such

that $\Gamma = \Gamma_1 \cdot \Gamma_2$, and $\Delta_1 = \Delta_2 = x{:}\alpha$. Hence, $\Gamma_2, x{:}\alpha \vdash v : \beta \xrightarrow{\kappa} \beta$ with $|\beta| = \omega$, by the rules for application and constants. We obtain $\Gamma_2, x{:}\alpha, y{:}\beta \vdash y \Leftarrow v\, y : (y{:}\beta; y{:}\beta)$ by Lemma 7 and the application and thread rules, and

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

by thread and composition rules. By the type rule for scope restriction $\Gamma \vdash (\nu y)(x \Leftarrow F[y] \mid y \Leftarrow v\, y) : \Delta_1; \Delta_2$ follows.

- Case (FUTURE.DEREF): By the definition, $C_1 \equiv (x \Leftarrow F_{\mathrm{f}}[y] \mid y \Leftarrow v)$ reduces to the configuration $C_2 \equiv (x \Leftarrow F_{\mathrm{f}}[v] \mid y \Leftarrow v)$. Moreover, $\Gamma \vdash C_1 : \Delta_1; \Delta_2$ by assumption. So by the typing rules for parallel composition and threads and by Lemma 11(1), there are Γ_1 and Γ_2 such that

$$\Gamma_1, y{:}\beta, x{:}\alpha \vdash F_{\mathrm{f}}[y] : \alpha \quad \text{and} \quad \Gamma_2, x{:}\alpha, y{:}\beta \vdash v : \beta$$

where $|\alpha| = |\beta| = \omega$, $\Gamma = \Gamma_1 \cdot \Gamma_2$ and $\Delta_1 = \Delta_2 = x{:}\alpha, y{:}\beta$. By Lemma 7(1), $\mathsf{once}(\Gamma_2) = \emptyset$, so $\Gamma_2 \subseteq \Gamma_1$ and by the substitution lemma (Lemma 8) also $\Gamma_1, x{:}\alpha, y{:}\beta \vdash F_{\mathrm{f}}[v] : \alpha$. So

$$\Gamma \vdash (x \Leftarrow F_{\mathrm{f}}[v] \mid y \Leftarrow v) : \Delta_1; \Delta_2$$

- Case (HANDLE.NEW): Similar to the case for (THREAD.NEW).
- Case (HANDLE.BIND): So $C_1 \equiv (x \Leftarrow F[z\, v] \mid z\, \mathsf{h}\, y)$ reduces to configuration $C_2 \equiv (x \Leftarrow F[\mathbf{unit}] \mid y \Leftarrow v \mid z\, \mathsf{h} \bullet)$. By assumption, $\Gamma \vdash C_1 : \Delta_1; \Delta_2$. By the typing rules this means $\Delta_2 = x{:}\alpha, y{:}\beta$ and $\Delta_1 = \Delta_2, z{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$, and there exist contexts Γ_1, Γ_2 such that

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

and $\Gamma = \Gamma_1 \cdot \Gamma_2$ with $y, z \notin \mathsf{dom}(\Gamma_2)$ and $\Delta' = y{:}\beta, z{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$. By Lemma 11(1) and Lemma 8 there are Γ_1' and Γ_1'' such that

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

and $\Gamma_1 = \Gamma_1' \cdot \Gamma_1''$. Hence $\Gamma_1 \vdash (x \Leftarrow F[\mathbf{unit}] \mid y \Leftarrow v) : \Delta_2; \Delta_2$, which entails

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

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

PROOF of Corollary 6. Suppose C has an error, i.e., there exists a subconfiguration $C' \equiv E_{\mathrm{f}}[y\ v] \mid y\, \mathsf{h} \bullet$ of C. Further, suppose $\Gamma \vdash C : \Delta_1; \Delta_2$, so by Lemma 11(2,3) there exist 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 $\Delta_1' = \Delta_2', y{:}\beta \xrightarrow{\mathbf{1}} \mathsf{unit}$ for some type β.

Also by the type rules for parallel composition and threads, there are Γ_1 and Γ_2 such that $\Gamma' = \Gamma_1 \cdot \Gamma_2$ and $\Gamma_1, z{:}\alpha \vdash F_\mathsf{f}[y\ v] : \alpha$. Since evaluation contexts do not capture variables, $y \in \mathsf{fv}(F_\mathsf{f}[y\ v])$. Thus, by Lemma 7(4), y must occur in $\Gamma_1 \subseteq \Gamma'$. By Lemma 9(1), $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Delta_1') = \emptyset$, 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 5), this proves that $\Gamma \vdash C : \Delta_1; \Delta_2$ implies handle error-freeness.

7 Conclusions and Future Work

We have presented the lambda calculus with futures $\lambda(\mathsf{fut})$ which serves as a semantics for concurrent extensions of ML. In its full power, $\lambda(\mathsf{fut})$ models the operational semantics of Alice ML where all synchronization is based on futures. A particular advantage of $\lambda(\mathsf{fut})$ is that it can naturally model mixed eager and lazy computation, all this in a statically typed framework.

$\lambda(\mathsf{fut})$ can be used to implement 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 to the type system.

At least two questions remain open. The first one is how to perform static analysis for $\lambda(\mathsf{fut})$, in order to reduce the cost of futures in programs where they are not used. 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. These operations can be modeled in a refinement of $\lambda(\mathsf{fut})$ with *touch* operations, as proposed in the lambda calculus with futures by Felleisen and Flanagan [21]. Touch operations are introduced systematically by the compiler. To improve efficiency, a compiler should be able to remove as many redundant touches as possible. We leave the extension of the techniques of [21] to $\lambda(\mathsf{fut})$ for future work.

Another open question is whether handled futures are redundant or not (as we conjectured). In order to well-define this question, an appropriate notion of program equivalence has to be developed.

Acknowledgements. We would like to thank the anonymous referees for their detailed comments on the various preliminary versions of this work. Thanks to Andreas Rossberg for many helpful discussions and suggestions.

References

[1] A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, G. Smolka, Alice Through the Looking Glass, Vol. 5 of Trends in Functional Programming, Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, 2006, Ch. 6, pp. 97–96.

[2] A. Rossberg, G. Smolka, G. Tack, *The Alice Project*, web site at the Programming Systems Lab, Saarland University, `http://www.ps.uni-sb.de/alice` (2006).

[3] R. Milner, M. Tofte, R. Harper, D. B. MacQueen, The Standard ML Programming Language (Revised), MIT Press, 1997.

[4] J. H. Reppy, Concurrent Programming in ML, Cambridge University Press, 1999.

[5] F. Nielson (Ed.), ML with Concurrency: Design, Analysis, Implementation, and Application, Monographs in Computer Science, Springer, 1997.

[6] C. Fournet, G. Gonthier, The reflexive chemical abstract machine and the join-calculus, in: Proc. POPL'96, ACM Press, 1996, pp. 372–385.

[7] S. Conchon, 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, pp. 22–29.

[8] A. Rossberg, Generativity and dynamic opacity for abstract types, in: Proc. 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03), ACM Press, 2003, pp. 241–252.

[9] G. Tack, L. Kornstaedt, G. Smolka, Generic pickling and minimization, Electronic Notes in Theoretical Computer Science 148 (2) (2006) 79–103.

[10] G. Smolka, The Oz programming model, in: J. van Leeuwen (Ed.), Computer Science Today, Vol. 1000 of LNCS, Springer, 1995, pp. 324–343.

[11] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, G. Smolka, Efficient logic variables for distributed computing, ACM Transactions on Programming Languages and Systems 21 (3) (1999) 569–626.

[12] J. Niehren, Uniform confluence in concurrent computation, Journal of Functional Programming 10 (5) (2000) 453–499.

[13] H. Baker, C. Hewitt, The incremental garbage collection of processes, ACM Sigplan Notices 12 (1977) 55–59.

[14] R. H. Halstead, Jr., Multilisp: A Language for Concurrent Symbolic Computation, ACM Transactions on Programming Languages and Systems 7 (4) (1985) 501–538.

[15] E. Shapiro, The family of concurrent logic programming languages, ACM Computing Surveys 21 (3) (1989) 413–510.

[16] V. A. Saraswat, M. Rinard, P. Panangaden, Semantic foundations of concurrent constraint programming, in: Proceedings POPL'91, ACM Press, 1991, pp. 333–352.

[17] M. Müller, Set-based failure diagnosis for concurrent constraint programming, Ph.D. thesis, Saarland University, Saarbrücken (1998).

[18] B. C. Pierce, D. N. Turner, Pict: A programming language based on the pi-calculus, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 2000, pp. 455–494.

[19] C. Fournet, C. Laneve, L. Maranget, D. Rémy, Implicit typing à la ML for the join-calculus, in: Proc. CONCUR'97, Vol. 1243 of LNCS, Springer, 1997, pp. 196–212.

[20] E. Chailloux, P. Manoury, B. Pagano, Developing Applications With Objective Caml, O'Reilly, 2000, available online at `http://caml.inria.fr/oreilly-book`.

[21] C. Flanagan, M. Felleisen, The semantics of future and an application, Journal of Functional Programming 9 (1) (1999) 1–31.

[22] L. Moreau, The semantics of scheme with future, in: International Conference on Functional Programming, ACM Press, 1996, pp. 146–156.

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

[24] B. Liskov, L. Shrira, Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems, SIGPLAN Notices 23 (7) (1988) 260–267.

[25] Z. M. Ariola, J. W. Klop, Lambda calculus with explicit recursion, Information and Computation 139 (2) (1997) 154–233.

[26] Z. M. Ariola, M. Felleisen, The call-by-need lambda calculus, Journal of Functional Programming 7 (3) (1997) 265–301.

[27] J. Maraist, M. Odersky, P. Wadler, The call-by-need lambda calculus, Journal of Functional Programming 8 (3) (1998) 275–317.

[28] A. K. Wright, Simple imperative polymorphism, Lisp and Symbolic Computation 8 (4) (1995) 343–355.

[29] A. Rossberg, The missing link - dynamic components for ML, in: 11th International Conference on Functional Programming, ACM Press, 2006.

[30] D. N. Turner, P. Wadler, C. Mossin, Once upon a type, in: Proc. 7th ICFPCA, ACM Press, 1995, pp. 1–11.

[31] J. Niehren, J. Schwinghammer, G. Smolka, A concurrent lambda calculus with futures, in: B. Gramlich (Ed.), Frontiers of Combining Systems, Vol. 3717 of Lecture Notes in Artificial Intelligence, Springer, 2005, pp. 248–263.

[32] R. Milner, The polyadic π-calculus: A tutorial, in: F. L. Bauer, W. Brauer, H. Schwichtenberg (Eds.), Logic and Algebra of Specification, Proc. Marktoberdorf Summer School, Springer, 1993, pp. 203–246.

[33] P. J. Landin, The mechanical evaluation of expressions, Computer Journal 6 (4) (1964) 308–320.

[34] S. P. Jones, A. Gordon, S. Finne, Concurrent Haskell, in: Proc. POPL'96, ACM Press, 1996, pp. 295–308.

[35] M. Abadi, L. Lamport, Conjoining specifications, ACM Transactions on Programming Languages and Systems 17 (3) (1995) 507–534.

[36] N. Kobayashi, B. C. Pierce, D. N. Turner, Linearity and the Pi-Calculus, ACM Transactions on Programming Languages and Systems 21 (5) (1999) 914–947.

[37] A. Jeffrey, J. Rathke, A fully abstract may testing semantics for concurrent objects, in: Proc. Lics 2002, 17th Annual Symposium on Logic in Computer Science, 2002, pp. 101–112.

[38] G. Boudol, The π-calculus in direct style, in: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1997, pp. 228–241.

[39] D. Caromel, L. Henrio, A Theory of Distributed Objects, Springer, 2005.

[40] E. Barendsen, S. Smetsers, Uniqueness type inference, in: Proceedings PLILP'95, Vol. 982 of LNCS, Springer, 1995, pp. 189–206.

[41] P. Wadler, Linear types can change the world!, in: M. Broy, C. B. Jones (Eds.), Programming Concepts and Methods, North Holland, 1990, pp. 546–566.