Futures and By-need Synchronization for Oz

Michael Mehl, Christian Schulte, Gert Smolka

Programming Systems Lab
DFKI and Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany

May 11, 1998

1 Introduction

We propose a conservative extension of Oz that adds futures and by-need synchronization. Futures are read-only views of logic variables that make it possible to statically limit the scope in which a variable can be constrained. For instance, one can express with futures safe streams that cannot be assigned by their readers. By-need synchronization makes it possible to synchronize a thread on the event that a thread blocks on a future. It is used to express dynamic linking and lazy functions.

We also introduce variable assignment and distinguish between simple and complex variables. The idea is that programs should distinguish between the use of logic variables for concurrency and the use of logic variables for constraint programming. Variable assignment is an asymmetric operation that eliminates a simple variable by binding it to a given unit.

SB's version of Mozart will provide for the distribution of ports, futures, and simple variables. All other stateful units cannot be distributed.

We outline an implementation of futures that does not incur an efficiency penalty. Futures require an extension of the protocols for distributed variables.

The proposed extensions are useful for Oz as is. They also fit into the new model for Oz [2, 3] that starts from functional programming and that introduces logic variables only in subsequent steps for the purposes of top-down construction of stateless data structures, concurrency, and constraint programming.

2 Futures

A future is a read-only view of a logic variable. Every variable is associated with a future that is logically equal to the variable.

The operation “get future”

```
!! : 'a -> 'a
```

checks whether its argument is a variable. If the argument is a variable, it returns the future of this variable, otherwise it returns the argument as is.

Futures must be represented in the constraint store and require changes in the unification algorithm. We will detail the changes in a later section. Here we note the following:

1. A unkinded variable may become a future.
2. A kinded variable cannot become a future.
3. A future cannot become a variable.
4. A future may become another future or a determined unit.
5. Unification of a future with a different future blocks.
6. Unification of a future with a kinded variable blocks.
7. Unification of a future with a determined unit blocks.
8. Unification proceeds until it detects failure or no or only blocked subtasks are left (subtasks are created by record constraints).

3 By-need Synchronization

The operation

```
ByNeed : (unit -> 'a) -> 'a
```

returns a future. As soon as a thread blocks on the returned future, a thread is created in the home space of the future that evaluates the function and that binds the future to the result after the evaluation has terminated.

Futures introduced by ByNeed are special in that the associated logic variable is not visible. This encapsulation avoids semantic complications and provides for optimizations.

The main applications of by-need synchronization are dynamic linking and lazy functions.

Halstead's Multilisp [1] provides by-need synchronization in the same way we propose it above. Multilisp does not have logic variables and tell operations.

4 Simple and Complex Variables

When a variable is created it is always a simple variable. A simple variable becomes complex if it becomes kinded, or if it is constrained in a subordinated space, or if it is unified with a complex variable. Once a variable is complex, it stays complex until it is eliminated. Unification of a complex and a simple variable makes the simple variable complex.

SB's Mozart will provide for the distribution of only three stateful units: ports, futures, and simple variables. It is an error to export a complex variable or to make a distributed variable complex.

An alternative to the above is to fix the distinction between simple and complex upon creation of a variable. In this case one would disallow the unification of a simple with a complex variable. If the default is complex, there is no need to rewrite existing nondistributed programs. However, complex variables are less efficient than simple variables.

5 Variable Assignment

The variable assignment operation

```
:= : 'a * 'a -> unit
```

expects that its left argument is a simple variable whose home is the current space and that is different from the right argument (in particular, they right argument must not be the future of the left argument). In this case variable assignment unifies its two arguments. Otherwise, it raises an error.

Variable assignment is distinguished from tell operations like general unification. Tell operations should only be used for constraint programming. The distinction between variable assignment and tell operations introduces redundancy that makes programs more readable.

6 Unification with Futures

The essence of unification with futures can be presented in a simple formal model that accounts for variables, futures, and constants. The constraint store is represented as a directed acyclic graph. A node is either a constant or a nonconstant. We use x and y to denote nonconstants, c to denote constants, and n to denote nodes. There are 2 types of edges:

1. xfy, which says that y is the future of the variable x.
2. $x \to n$, which says that x is a reference to n.

The direction of an edge is from left to right.

A constraint store is now a finite set of edges such that:

1. The edges form a DAG whose nodes have either no or exactly one departing edge.
2. If there is an edge xfy, then there is no edge departing from y.

A unification task is a pair $n_1 = n_2$ of 2 nodes. The execution of a unification task is described by rewriting rules for constraint stores, which are as follows:

1. $x_1fy_1,\ \ x_2fy_2,\ \ x_1 = x_2 \quad \Rightarrow \quad x_1 \to x_2,\ \ y_1 \to y_2,\ \ x_2fy_2$
2. $x_1fy_1,\ \ x_2fy_2,\ \ x_1 = y_2 \quad \Rightarrow \quad x_1 \to y_2,\ \ y_1 \to y_2,\ \ x_2fy_2$
3. $xfy,\ \ x = c \quad \Rightarrow \quad x \to c,\ \ y \to c$
4. $n = n \quad \Rightarrow \quad \emptyset$
5. $c_1 = c_2 \quad \Rightarrow \quad \mathsf{failure} \qquad$ if $c_1 \neq c_2$
6. $x \to n_1,\ \ x = n_2 \quad \Rightarrow \quad x \to n_1,\ \ n_1 = n_2$
7. $x = c \quad \Rightarrow \quad x \to c$ if there is no link departing from x.

The addition of determined records does not bring anything new as it comes to futures. The addition of kinded variables relies on the assumption that the unification of a future and a kinded variable blocks.

The assumption that the unification of a future and a kinded variable blocks is more rigid that necessary. The alternative is to propagate constraints from the future of a kinded variable to the other partner, be it a variable or a determined unit. The question is whether this alternative has a straightforward implementation and whether there are interesting applications.

The next extension is deep unification. Here it seems easiest to make unification of a complex variable and a future blocking.

7 Implementation

Futures can be put into Mozart without loss of efficiency. When a variable is created, a word

$$\boxed{\texttt{V|--}}\ \texttt{-->}\ \texttt{home}$$

is allocated. When the variable is eliminated, the word is updated to a transparent reference

$$\boxed{\texttt{R|--}}\ \texttt{-->}\ \texttt{unit}$$

When the variable acquires a suspension, a constraint, or a future, the word is updated to

$$\boxed{\texttt{VF|--}}\ \texttt{-->}\ \boxed{\texttt{F|type | ...}}$$

where the right-hand side is a block of words. The first word of the block is tagged with `F` and represents the future of the variable. When such a variable is eliminated, the two words tagged with `VF` and `F` are updated to references to the same unit.

Note that it is ok to bind a future to a variable provided the variable is updated to be a future. We can also say that it is ok to unify a future and a variable, provided we make the result a future.

The presence of futures requires an extension of the protocols for distributed variables so that the unification of two futures is avoided. Let us refer to variables and futures jointly as quasi-variables.

The protocol extensions for distributed quasi-variables are as follows. As before the protocols respect a ranking of quasi-variables, where bindings go from higher rank to lower rank. If a distributed future x is asked to bind itself to a lower ranking quasi-variable y, the following happens:

1. x locks itself.
2. x sends a request to y to turn itself into a future.
3. y answers with `ack` or `nack`. `ack` means that y was a variable and did update itself into a future. `nack` means that y is not a variable anymore and hence cannot fulfill the request.

4. When x gets the answer `ack`, it binds itself to y as usual and unlocks itself.

5. When x gets the answer `nack`, it sends `nack` to the requester of the binding and unlocks itself.

6. Bind requests that arrive during the time x is locked are queued and are processed once x becomes unlocked again.

7. Deadlock is excluded since a locked variable always waits for an answer from a variable with lower rank.

8. Messages from x to the requester of the binding must arrive in order.

References

[1] H. Halstaedt, Robert. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 1985.

[2] G. Smolka. Concurrent constraint programming based on functional programming. In C. Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

[3] G. Smolka. Concurrent constraint programming based on functional programming, Slides of an Invited Talk at ETAPS'98, 1998. http://www.ps.uni-sb.de/ smolka/drafts/etaps98.ps.