Declarative Laziness in a Concurrent Constraint Language

Alfred Spiessens, Raphaël Collet, and Peter Van Roy

Université Catholique de Louvain,
Place Sainte-Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{fsp,raph,pvr}@info.ucl.ac.be

Abstract. This paper explains how to design and implement an extension for by-need synchronization for a confluent (subset of a) multi-paradigm concurrent constraint language, while keeping the extended language confluent. It reveals the subtleties and pitfalls that can easily lead to the loss of confluence, especially in languages with a powerful unification operator. The authors report on their own experiences, and provide guidelines for similar projects, based on considerations regarding the monotonicity of the constraint store. This paper also explores the boundaries of confluent extensibility for such languages.

1 Introduction

Confluence, or deterministic concurrency, has many advantages for application design and analysis, for security, but also for pedagogical purposes. As Oz is a multi-paradigm language, used for concept-based teaching of programming skills [8], and at the same time as an instrument for research, it was conceived to be very important to have a well-defined deterministic concurrent subset of the language that can guarantee the confluence of all programs written in it. As an initial construct for by-need synchronization turned out to be not confluent, an investigation was done to find the reasons for the loss of confluence and, if possible, also find the cure. We succeeded in both goals, and we report on our most important experiences and conclusions from this investigation in this paper.

The paper is organized as follows. Section 2 defines a concurrent constraint language that is confluent. Sections 3 and 4 give two different language extensions for by-need synchronization. The first is shown non-confluent, while the second respects confluence. Section 5 compares our results with another language, namely Curry. Section 6 then proposes a way to implement our ideas.

2 A Concurrent Language with Unification

This section defines a small concurrent language $\mathcal{L}$, with logic variables and unification on rational trees. A program in $\mathcal{L}$ consists of a set of *threads* that modify a shared *constraint store*. The computation model of $\mathcal{L}$ is very close to Saraswat's concurrent constraints [6]. The language $\mathcal{L}$ is a declarative subset of the multiparadigm programming language Oz [4, 7].

$$
\begin{array}{rl}
\text{store} & \sigma ::= \phi_1 \wedge \cdots \wedge \phi_n \\
\text{constraint} & \phi ::= x{=}y \mid x{=}v \mid \xi : \mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end} \\
\text{partial value} & v ::= l(x_1 \cdots x_n) \mid \xi
\end{array}
$$

Fig. 1. Abstract syntax of constraint stores

2.1 The Constraint Store

The syntax of constraint stores and partial values is given in Fig. 1. A constraint store σ consists of a conjunction of elementary constraints ϕ_i over store entities. The empty store, i.e., without constraint, is written $\top$. The main constraint in our system is equality between logic variables x, y, z, or between a variable and a partial value v. A value is either a *name* ξ (see below), or a *record* $l(x_1 \cdots x_n)$ with $n \geq 0$, where l denotes a literal. A record is also called a partial value because its contained variables x_i may be not constrained. The third kind of constraint is an association between a *name* ξ and a *closure* $\mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end}$. Such associations are always unique. A name is an internal store value without a representation in a program, while a closure represents an abstraction of a statement S.

A store σ *entails* a given constraint ϕ if ϕ is logically implied by the store constraints. We write this as $\sigma \models \phi$. Two stores σ and σ' that entail each other are said to be *equivalent*, which is written $\sigma \equiv \sigma'$. A variable x bound by equality to a value v is said to be *determined*. We write this as $\sigma \models \det(x)$.

A store is *consistent* if there exists a valuation of the variables that satisfies all its constraints, otherwise it is *inconsistent*. The constraints ϕ are chosen so that the consistency of a store is a decidable property. An *inconsistent* store is written $\bot$.

Two operations exist on constraint stores: *ask* and *tell*. Asking a constraint ϕ is waiting until the store entails or disentails ϕ, i.e., $\sigma \models \phi$ or $\sigma \models \neg\phi$. Telling a constraint ϕ to a store σ is updating the store to $\sigma \wedge \phi$. A program *fails* when it tells a constraint that makes the store inconsistent. In that situation, the whole program stops and the store becomes $\bot$. A practical language such as Oz actually does not tell constraints that makes the store inconsistent, but rather uses an exception mechanism. We did not include such a mechanism in our language, because in the presence of concurrency it leads to nondeterminism.

2.2 The Language $\mathcal{L}$

The syntax of statements S is given in Fig. 2. The letters X, Y, P denote identifiers, and t denotes a term. The lexical scope of an identifier X is restricted by

- a variable declaration (**local** X **in**... **end**),
- a case statement (**case**... **of** $l(\ldots X \ldots)$ **then**... **end**),
- a procedure definition (**proc** $\{P \ldots X \ldots\}$ ... **end**).

statement	$S ::=$	**skip**	(empty statement)
		$S_1\ S_2$	(sequence)
		thread S **end**	(thread creation)
		local X **in** S **end**	(variable declaration)
		$X{=}t$	(unification)
		case X **of** $l(Y_1 \cdots Y_n)$ **then** S_1 **else** S_2 **end**	(pattern matching)
		proc $\{P\ X_1 \cdots X_n\}$ S **end**	(procedure creation)
		$\{P\ X_1 \cdots X_n\}$	(procedure application)
term	$t ::=$	$Y \mid l(Y_1 \cdots Y_n)$	

Fig. 2. Syntax of the language $\mathcal{L}$

A small-step operational semantics of $\mathcal{L}$ is given in Fig. 3. It defines transition rules between *configurations*, which are applicable when a certain condition is satisfied. A configuration $\mathcal{T}/\sigma$ consists of a multiset $\mathcal{T}$ of threads $(T_i)_{1 \leq i \leq n}$[1], together with a constraint store σ. A *thread* is a stack of *semantic statements*. A semantic statement is a statement S where every *free* identifier has been replaced by a store variable. An abstract syntax for threads is

$$T \ ::= \ \langle\rangle \mid \langle S\ T \rangle \ ,$$

where S denotes a semantic statement. The statement in front of a thread is the next statement to execute. A transition between configurations $\mathcal{T}/\sigma$ and $\mathcal{T}'/\sigma'$ is written

$$\frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'} \ \textit{condition} \ .$$

Rules (1) state that threads execute with an interleaving semantics. A thread with no statement is terminated, it eventually disappears. When the store becomes inconsistent, all threads may disappear. Rules (2) define the semantics for the empty statement, the sequence, and thread creation. In (3), a new variable x is introduced by replacing all the occurrences of the declared identifier X in its lexical scope by x. The substitution function is noted $\{X \mapsto x\}$. With (4), a unification simply tells a constraint in the constraint store; u is either a variable or a partial value. The store might become inconsistent. Rules (5) and (6) make the **case** statement block until the variable x becomes determined. The statement then reduces to the first statement if the value of x matches the pattern $l(Y_1 \cdots Y_n)$, or the second statement otherwise. In (7), the statement **proc** creates a closure in the store, with a fresh name ξ, then reduces to an unification. In (8), applying a procedure consists in taking the closure associated to p in the store, and substituting the arguments in the statement S.

[1] For the sake of simplicity, we denote $\mathcal{T}$ as a sequence $T_1 \ldots T_n$, the order being irrelevant in this context.

$$\frac{\mathcal{T}\ \mathcal{U}}{\sigma}\ \Big\|\ \frac{\mathcal{T}'\ \mathcal{U}}{\sigma'} \text{ if } \frac{\mathcal{T}}{\sigma}\ \Big\|\ \frac{\mathcal{T}'}{\sigma'} \qquad \frac{\langle\rangle}{\sigma}\ \Big\|\ \frac{}{\sigma} \qquad \frac{\mathcal{T}}{\bot}\ \Big\|\ \frac{}{\bot} \tag{1}$$

$$\frac{\langle \mathbf{skip}\ T\rangle}{\sigma}\ \Big\|\ \frac{T}{\sigma} \qquad \frac{\langle S_1\ S_2\ T\rangle}{\sigma}\ \Big\|\ \frac{\langle S_1\ \langle S_2\ T\rangle\rangle}{\sigma} \qquad \frac{\langle \mathbf{thread}\ S\ \mathbf{end}\ T\rangle}{\sigma}\ \Big\|\ \frac{T\ \langle S\ \langle\rangle\rangle}{\sigma} \tag{2}$$

$$\frac{\langle \mathbf{local}\ X\ \mathbf{in}\ S\ \mathbf{end}\ T\rangle}{\sigma}\ \Big\|\ \frac{\langle S\{X{\mapsto}x\}\ T\rangle}{\sigma}\ x \text{ fresh variable} \tag{3}$$

$$\frac{\langle x{=}u\ T\rangle}{\sigma}\ \Big\|\ \frac{T}{\sigma \wedge x{=}u} \text{ if } \sigma \wedge x{=}u \text{ is consistent} \qquad \frac{\langle x{=}u\ T\rangle}{\sigma}\ \Big\|\ \frac{}{\bot} \text{ otherwise} \tag{4}$$

$$\frac{\left\langle \begin{array}{l}\mathbf{case}\ x\ \mathbf{of}\ l(Y_1 \cdots Y_n)\\ \quad \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ T\end{array}\right\rangle}{\sigma}\ \Big\|\ \frac{\langle S_1\{Y_1{\mapsto}y_1, \ldots, Y_n{\mapsto}y_n\}\ T\rangle}{\sigma} \text{ if } \sigma \models x{=}l(y_1 \cdots y_n) \tag{5}$$

$$\frac{\left\langle \begin{array}{l}\mathbf{case}\ x\ \mathbf{of}\ l(Y_1 \cdots Y_n)\\ \quad \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ T\end{array}\right\rangle}{\sigma}\ \Big\|\ \frac{\langle S_2\ T\rangle}{\sigma} \text{ otherwise} \tag{6}$$

$$\frac{\langle \mathbf{proc}\ \{p\ X_1 \cdots X_n\}\ S\ \mathbf{end}\ T\rangle}{\sigma}\ \Big\|\ \frac{\langle p{=}\xi\ T\rangle}{\sigma \wedge \xi : \mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end}}\ \xi \text{ fresh name} \tag{7}$$

$$\frac{\langle \{p\ x_1 \cdots x_n\}\ T\rangle}{\sigma}\ \Big\|\ \frac{\langle S\{X_1{\mapsto}x_1, \ldots, X_n{\mapsto}x_n\}\ T\rangle}{\sigma} \text{ if } \sigma \models p{=}\xi \wedge\ \xi : \mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end} \tag{8}$$

Fig. 3. Small-step semantics of the language $\mathcal{L}$

We assume that the execution is *fair*, i.e., a thread cannot be kept runnable without being executed. The reader can easily check from the rules that a runnable thread stays runnable until execution.

2.3 The Confluence of $\mathcal{L}$

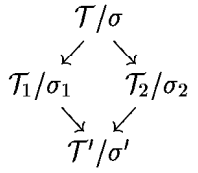
The concurrent nature of $\mathcal{L}$ is such that the language is *confluent*, which means that the "result" of a program (i.e., its final configuration) is always the same, whatever the order of thread reduction. In other words, every program is deterministic.

Some transition rules introduce fresh symbols, namely variables in (3) and names in (7). The confluence should not depend on the choice of those symbols. We thus define an *equivalence between configurations* as follows. Let $\mathcal{T}/\sigma$ and $\mathcal{T}'/\sigma'$ be configurations, and $\mathcal{V}$ be a set of variables. The configurations are said to be *equivalent modulo* $\mathcal{V}$, which we write $\mathcal{T}/\sigma \equiv \mathcal{T}'/\sigma'\ (\mathcal{V})$, if there exists a bijection r such that

- r maps variables to variables, and names to names;
- r is the identity function on $\mathcal{V}$;
- $r(\mathcal{T}) = \mathcal{T}'$ and $r(\sigma) \equiv \sigma'$, where r is used as a replacement on statements and stores.

We define a transition relation that relates configurations up to equivalence. We write $\mathcal{T}/\sigma \longrightarrow \mathcal{T}'/\sigma'$ $(\mathcal{V})$ if there exists a finite execution with the transition rules of $\mathcal{L}$, with $\mathcal{T}/\sigma$ as initial configuration, and whose final configuration is equivalent to $\mathcal{T}'/\sigma'$ modulo $\mathcal{V}$. The confluence property is then defined as follows.

Theorem 1 (Confluence). *Let $\mathcal{T}/\sigma$, $\mathcal{T}_1/\sigma_1$ and $\mathcal{T}_2/\sigma_2$ denote configurations, and $\mathcal{V}$ be a set of variables.*
If $\mathcal{T}_1/\sigma_1 \longleftarrow \mathcal{T}/\sigma \longrightarrow \mathcal{T}_2/\sigma_2(\mathcal{V})$, then there exists a configuration $\mathcal{T}'/\sigma'$ such that $\mathcal{T}_1/\sigma_1 \longrightarrow \mathcal{T}'/\sigma' \longleftarrow \mathcal{T}_2/\sigma_2(\mathcal{V})$.
The diagram on the right depicts this property.



From this property, we can easily characterize the executions of a program. For instance, if a program fails in one execution, it always fails. In that case, every execution will reach the final configuration $\{\}/\bot$. Note that this configuration is clearly not equivalent to a program that terminates with some threads still blocked (not runnable). Such a program can be qualified as *partially terminated*. This means that if an "external agent" tells some constraints in the program's store, the program might execute further, and possibly reach a new partial termination, which is unique by confluence.

2.4 Functional Programming in $\mathcal{L}$

Our language is expressive enough to reproduce the behavior of functional programs. The idea is simply to translate a functional program into $\mathcal{L}$, functions becoming procedures, and expressions being expressed in elementary operations. The following simple example gives an idea of this translation. The "bar" operator `X|Xr` is a simple notation for a record like `cons(X Xr)`.

```
fun {Append Xs Ys}
   case Xs of X|Xr then
      X|{Append Xr Ys}
   else Ys end
end
```

```
proc {Append Xs Ys Zs}
   case Xs of X|Xr then
      local Zr in
         Zs=X|Zr
         {Append Xr Ys Zr}
      end
   else Zs=Ys end
end
```

3 A Flawed Definition of By-need Synchronization

In this section we present the definition of by-need synchronization described in [3]. It is implemented in Mozart [4] at least until the current version (1.2.5). We will show that it does not respect the confluence of the language, and investigate why.

statement	$S ::= \ldots$	(syntax rules defined in Fig. 2)
	| `{ByNeed` P X`}`	(execution of P when X is needed)

Fig. 4. Syntax extension of the language $\mathcal{L}$ with `ByNeed`

3.1 Naive Definition and Semantics

The `ByNeed` construct allows a computation to be associated to a logic variable, which represents the result of the computation. The computation will be performed as soon as its result becomes needed. Figure 4 shows the added statement.

The operational semantics are set up in a way as to assure the following rules in the computation. We use x and p as the variable and its associated calculation, respectively.

- `{`p x`}` will be calculated as soon as a statement needs the variable x for its reduction.
- A statement needs x if it cannot reduce without x being determined.
- The unification of x with a determined variable needs x. This will protect x from being unified with a value before p itself has ended. Only the proper invocation of `{`p x`}` will be allowed to bind x.
- The unification of x with another by-need variable needs x. This rule ensures that, if `{`p`}` would itself return a by-need variable, the latter would immediately be calculated before being assigned to x.
- `{`p x`}` will be calculated at most once for every application of `{ByNeed` p x`}`.

The reader will notice that this definition of `ByNeed` is indeed inspired by functional programming. It is modeled after a typical "let" construct [1, 2, 5] allowing for the declaration of a value with a predefined expression, and in the mean time protecting the variable from being overwritten by another value. This was translated into our programming language, which provides functions as syntactic sugar for procedures with at least one parameter (see Sect. 2.4). However, most constructs in our language stem from concurrent constraint programming. It is one of these differences with other multi-paradigm languages [1, 2] that would turn out to be important in unexpected ways.

3.2 Counterexample for Confluence

We found a counterexample that proved `ByNeed` to introduce non-confluence in the language, when we examined the following procedure and its applications.

```
proc {ReadOnly X Y}  % make Y a read-only version of X
   Y={ByNeed proc {$ Z} {Wait X} Z=X end}
end
```

Since every attempt to unify `Y` to a value (or another read-only variable) will start the computation, and synchronize on `X` becoming bound, this was an obvious application for the `ByNeed` function. The following application defies confluence.

```
local X Y Z in
   thread X=Y end
   thread Y=2 end
   thread X=1 end
   thread X={ReadOnly Z} end
end
```

Depending on the order of execution, this example will fail or succeed. Let us look at it in detail:

1. Suppose these concurrent statements are executed in the order of their definition. First both free variables X and Y are unified. This statement just adds the equality constraint to the store. Then Y=2 unifies Y with the constant 2, resulting in both X and Y having the value 2. The next statement X=1 will then fail because it would introduce inconsistency into the store. The last statement will not be executed due to the failure.
2. When the concurrent statements are executed in the reverse order, something different happens. First X becomes a by-need variable, to be bound to the eventual value of Z. Next X=1 triggers the computation of the value of X, causing an indefinite waiting for Z to become determined. The statement Y=2 binds Y to 2. At last, X=Y waits indefinitely for X to become determined via Z. Since Z is local within **local ... end**, no constraint can be added to the store afterwards, that would bind Z to a value. This means that both executions are not confluent.

The fact that ByNeed can be used to make unification block was quickly generalized to the following observation: *Any language construct that can make our unification operator block, will introduce non-confluence in the language.* This is shown in the following generalized example in pseudo-code.

```
local X Y Z in
   thread setup X and Z to make X=Z block end
   thread X=Y tell C1 end
   thread Y=Z tell C2 end
end
```

The execution of the first thread prevents the unification of X and Z to reduce. Therefore the second thread can still tell its constraint *C1*, but the third thread blocks. Changing the order of execution of the threads results in either *C1*, or *C2*, or both to be told to the store. If *C1* and *C2* are chosen to be incompatible, the computation can also fail due to inconsistency. In Sect. 5 we explain why this observation cannot be done in the subset of Curry, described in [2].

Our unification operator is constraint-oriented. That means that the unification of free variables x and y adds the constraint $x{=}y$ to the constraint store, and does not have to wait for x or y to become determined. It is also a very rich and powerful monotonic unification operator, that unifies partial values into the union of the information they carry. It causes a failure if the partial values x and y are incompatible.

3.3 The Reason for the Loss of Confluence

The deep reason for the loss of confluence is the loss of monotonicity. This is best understood in the context of the *ask* and *tell* operators of CCP (see [6] and Sect. 2.1). Before the introduction of `ByNeed`, unification was a simple *tell* operation, adding the equality constraint to the store and never blocking. `ByNeed` seems to have somehow turned unification into an *ask* operation, since it now can block. The unification of a by-need variable x with a needed variable y (or a partial value) transfers the need to x and triggers the evaluation of the expression associated with x. But in a monotonic setting, an operation should not block with a constraint store with more information available than in another store it does not block with.

To ensure confluence, monotonicity is to be kept in all the state transitions from a free to a determined variable. Since unification (now an *ask* operation) blocks for by-need variables, it should also block for free variables.

4 A Good Definition of By-need Synchronization

In this section we give the revised definition and semantics of by-need synchronization, and give its interpretation in terms of constraints, to show that confluence is indeed respected this time.

4.1 Revised Definition and Semantics

We use a new constraint, need(x), to express that the determinacy of x is needed by the program, together with a primitive statement `{WaitNeed` x`}` to ask that constraint. Figure 5 gives the syntax extension for this new primitive, and its operational semantics. The relation $\text{need}_\sigma(S, x)$ defines when a statement S needs a variable x in the store σ. We assume that this relation is *stable* as defined in Def. 1 below. With this primitive we build a confluent by-need construct, that we call `OnDemand` to avoid confusion with the previous one.

```
proc {OnDemand P X}
   thread {WaitNeed X} {P X} end
end
```

A by-need computation is now simply a thread that waits for a variable to be needed. In order to ensure monotonicity, determined variables are always needed, i.e., $\text{det}(x) \Rightarrow \text{need}(x)$. The need constraint allows to define three possible states for a variable, namely *free*, *needed* and *determined*. Those three states are presented in Fig. 6.

4.2 The Revised Definition Respects the Confluence of the Language

Let us analyze the new rules for by-need synchronization, in terms of constraints.

statement	$S ::= \ldots$	(syntax rules defined in Fig. 2)
	$\mid$ {WaitNeed X}	(wait for X to be needed)
constraint	$\phi ::= \ldots$	(syntax rule defined in Fig. 1)
	$\mid$ need(x)	(variable x is needed)

$$\frac{\langle \{\texttt{WaitNeed}\ x\}\ T \rangle \,\|\, T}{\sigma \,\|\, \sigma} \text{ if } \sigma \models \text{need}(x) \tag{9}$$

$$\frac{S \,\|\, S}{\sigma \,\|\, \sigma \wedge \text{need}(x)} \text{ if need}_\sigma(S, x) \text{ and } \sigma \not\models \text{need}(x) \tag{10}$$

Fig. 5. Syntax and operational semantics of `WaitNeed` and need()

1. {WaitNeed x} is a simple *ask* operation, just waiting for the constraint need(x) to be entailed by the store.
2. A statement S needing x to become determined for its reduction, will simply tell need(x). (Remember that once x is needed, it stays needed forever.)

Here the rules of the operational semantics no longer imply that a statement should block in order to trigger the on-demand computation associated with a variable. This is indeed the crucial difference with the previous version, and it is enabled by the introduction of a monotonic "needed" state for variables, visualized in Fig. 6. The unification of a needed variable with a free variable can now make the free variable needed, (performing its "monotonic" duty) without having to block for this *transfer of need* to be assured. The `ReadOnly` function from the example in Sect. 3.2 can no longer make unification block.

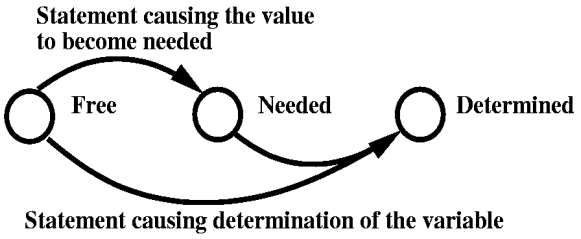


Fig. 6. State transitions of a variable, with the need constraint

The confluence of the language is ensured if the relation $\text{need}_\sigma(S, x)$ is stable.

Definition 1 (Stable need). *We say that the need relation* $\text{need}_\sigma(S, x)$ *is* stable *if, for every two stores* σ, σ' *such that* $\sigma' \models \sigma$,

$$\text{need}_\sigma(S, x) \text{ and } \sigma \not\models \text{need}(x) \text{ implies } \neg\text{reduce}_\sigma(S) \tag{11}$$

$$\text{need}_\sigma(S, x) \text{ implies } \text{need}_{\sigma'}(S, x) \text{ or } \sigma \models \text{need}(x) \tag{12}$$

This property of the relation guarantee that a statement that needs a variable x cannot reduce before need(x) is in store (11), even when the store is evolving monotonically (12). A simple case-analysis of all semantic rules, by checking the conditions for reduction, now reveals that every statement in the language respects confluence.

The following need relation, defined as in [8], is an example of a stable need relation, but others are possible.

$$\begin{aligned} \text{need}_\sigma(S,x) \text{ iff} \quad & \neg\text{reduce}_\sigma(S) \\ \text{and} \quad & \exists\sigma' : \sigma' \models \sigma \text{ and } \text{reduce}_{\sigma'}(S) \\ \text{and} \quad & \forall\sigma' : \sigma' \models \sigma \text{ and } \text{reduce}_{\sigma'}(S) \text{ implies } \sigma \models \text{det}\, x \end{aligned}$$

4.3 A Few More Remarks

Need-Triggered Execution. In contrast with "function-oriented" multi-paradigm languages, by-need synchronization unifies the variable to its eventual value *inside* the procedure it was associated with by the application of `OnDemand`. But of course the decision to unify its parameter to a value is up to the procedure. This provides a more general mechanism for any kind of computation synchronizing on the need of a variable.

Efficient Implementation. The fact that `OnDemand` itself does not *tell* any information to the store before *asking* for the need(x) condition, indicates that there is no need for an "on-demand" state in this new definition. No difference is detectable in the store before and after the application of `OnDemand`. This observation drastically simplifies the implementation of `OnDemand`, as described in Sect. 6.

The Definitive Loss of a Confluent Read-Only. The operation `OnDemand` can no longer be used to build read-only variables that cause unification to block. A "read-only" variable build as in Sect. 3.2 will be forced to become determined upon unification with a value. In fact it becomes clear that our language will never be able to protect variables by causing their unification to block, while respecting confluence.

5 Related work

The language Curry [1, 2] is a good example to compare to. Curry is an integrated functional logic language. It combines features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Curry is confluent.

The definitions of laziness and unification in Curry are a bit different from our language. In order to show those differences, we can compare the Curry constraint on the left with the Oz constraint on the right:

```
let x = <expr1>                local X Y in
    y = <expr2>                   X={OnDemand proc {$ Z} Z=<expr1> end}
in x =:= y                        Y={OnDemand proc {$ Z} Z=<expr2> end}
                                  X=Y
                               end
```

Though they look similar, they behave differently. The Curry constraint `x =:= y` forces both expressions to be evaluated, then it checks for the satisfiability of the equality. The Oz constraint `X=Y` does not force the by-need computations, since both `X` and `Y` are simply free. Both will be forced only when `X` (or `Y`) becomes needed.

What makes Curry confluent is the way logic variables relate to "normal" variables. A variable declared in a `let` construct is associated to an expression that is evaluated lazily. So a variable is always declared *together* with an expression. A logic variable is a special case, where the associated expression does not give a value, but rather a *black hole*. Logic variables are typically declared as in

```
let x = x in x =:= 42
```

In Curry a logic variable is a variable whose *complete* evaluation does not lead to a value. By the way the language is defined, it is not possible to associate a lazy computation to a logic variable. This is why unification forces its both arguments to be evaluated.

6 Implementation

We now describe how to extend an existing implementation of Oz, which our language $\mathcal{L}$ was a subset of. The constraint store of Oz is implemented as a graph, where nodes are variables and partial values. Each equivalence class of variables has a union-find structure, i.e., each variable node (except one) has an outgoing edge to another variable node in the same equivalence class, and those nodes form a tree where the edges are directed to the root node. The latter is the class representative. It may have an outgoing edge to a value node, meaning that the variable is bound to the given value.

When a thread blocks on a variable (asking for a constraint on that variable), a reference to the thread is put in a *suspension list* associated to the class representative of the variable. When a constraint is put on the variable, all the threads in the suspension list are woken up, and given to the thread scheduler. The suspension list allows to synchronize threads on constraints.

We simply extend this suspension list so that it also handles the constraint need(x). Each class representative now has a *needed* state that tells whether the variable is needed or not. When a thread blocks on need, and the variable is not needed yet, the thread is simply put it the suspension list. As soon as the variable becomes needed, the threads in the suspension list are woken up.

If a thread blocks on determinacy, we set the variable in the *needed* state and schedule the threads in the suspension list. When a variable x is bound to a value, we simply schedule its suspension list.

The implementation is in progress and will be part of a future release of Mozart [4].

7 Conclusion

We have tried and succeeded in extending a deterministic (subset of a) multi-paradigm CCP language with by-need synchronization, while respecting confluence. For constraint-oriented multi-paradigm languages, such an extension is not straightforward, and should not be based naively on its counterpart in function-oriented multi-paradigm languages. We showed that the semantics of the unification operator makes a subtle difference that can have an important influence on the confluence of the extended language. Finally, we gave an example of how reasoning from constraints and monotonicity should guide the design for confluence-preserving extensions of multi-paradigm CCP languages.

Acknowledgements

This research was partly funded by the MILOS project of the Walloon Region of Belgium (convention 114856).

References

1. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
2. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
3. Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, May 1998. DRAFT.
4. Mozart Consortium (DFKI, SICS, UCL, UdS). The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.
5. Simon Peyton Jones, editor. *Haskell 98 language and libraries: The revised report.* Cambridge University Press, 2003. Also published as the January 2003 Special Issue of the Journal of Functional Programming.
6. Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
7. Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
8. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* 2002. Work in progress. Expected publishing date 2003.