# Programming Deep Concurrent Constraint Combinators

Christian Schulte

Programming Systems Lab, Universität des Saarlandes
Postfach 15 11 50, 66041 Saarbrücken, Germany
schulte@ps.uni-sb.de

**Abstract.** Constraint combination methods are essential for a flexible constraint programming system. This paper presents deep concurrent constraint combinators based on computation spaces as combination mechanism. It introduces primitives and techniques needed to program constraint combinators from computation spaces. The paper applies computation spaces to a broad range of combinators: negation, generalized reification, disjunction, and implication. Even though computation spaces have been conceived in the context of Oz, they are mainly programming language independent. This point is stressed by discussing them here in the context of Standard ML with concurrency features.

## 1 Introduction

It is widely acknowledged that applications require a constraint programming system to be flexible. Regardless of how many primitive constraints a system offers, combination of primitive constraints into more complex application-specific constraints remains a must. This makes mechanisms for constraint combination key components of a constraint programming system.

Desirable properties of a constraint combination mechanism include that it is *compositional* and *conservative*. Compositional means that constraints obtained by combination can be combined again. Conservative means that the mechanism can be applied to existing constraints without changing them.

This paper's focus is on *deep concurrent constraint combinators* as combination mechanism. The paper introduces primitives from which constraint combinators can be programmed. It presents techniques that are characteristic in programming combinators. As underlying primitives *computation spaces* are proposed. Computation spaces offer two important features: they encapsulate arbitrary, that is *deep*, computations involving constraints and allow for concurrent control of computations. Combinators based on computation spaces are fully compositional: they can be nested arbitrarily. In particular, the constraints that can be combined are not limited to built-in constraints.

Computation spaces and combinator programming techniques are applied to a broad range of combinators, including negation, generalized reification, disjunction, and conditional (implication). In the same way as combinators allow to program new constraints, computation spaces allow to program new combinators: they provide flexibility on the constraint and on the combinator level. The paper introduces and refines the very few operations on computation spaces as the presentation of the paper proceeds. This is complemented by an overview over all operations at the end of the paper in Sect. 7.

Enrico Pontelli, Vítor Santos Costa (Eds.): PADL 2000, LNCS 1753, pp. 215–229, 2000.

Computation spaces are the basic concept that underlies deep-guard combinators found in programming languages like AKL [3,4] and Oz [11]. Even though computation spaces were conceived in the context of Oz, they are a general mechanism independent from the underlying programming language. To stress this point (and to make the paper's program fragments more accessible to a broader audience) this paper chooses Standard ML extended by threads and logic variables as host language [12].

Our experience shows that applications of constraint combinators in finite domain programming are not frequent. They turn out to be of great importance for other constraint domains, like feature or finite set constraints. In particular, they have turned out to be essential in the area of computational linguistics [1], where constraints from different domains are combined naturally.

A second area of application is prototyping new constraints. Starting from already implemented constraints new constraints can be developed by combining them at a high level. After experiments have shown that they are indeed the right constraints, a more efficient implementation can be attempted. This motivation is similar to the motivation for constraint handling rules (CHR) [2]. The difference is that this paper is concerned with primitives to combine constraints, a feature that an implementation of CHRs already requires.

Combinators for constraint programming is not a new idea. Previous approaches include Saraswat's concurrent constraint programming framework [8,7], the cardinality operator by Van Hentenryck and Deville [13], and cc(FD) [14]. The approaches have in common that the combinators considered are not "deep": the constraints that can be combined must be either built-in, or allow a simple reduction to built-in constraints (cardinality combinator). Another difference to the approach taken in this paper is that these approaches offer a fixed set of combinators. This paper's focus is on the primitives and techniques to program combinators. For all combinators but constructive disjunction (as available in cc(FD)) it is shown how to encode them with computation spaces.

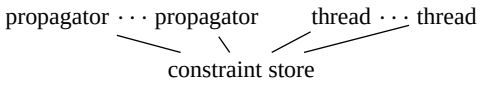
A different approach to combining constraints are *reified* constraints (also known as metaconstraints). Reified constraints reflect the validity of a constraint into a 0/1-variable. Constraints can then be combined by using the 0/1-variable in other constraints. Computation spaces are not intended as a replacement for reified constraints. As is discussed in detail in Sect. 4, a reification combinator based on computation spaces can offer better propagation in cases where reified constructions propagate poorly. And, since the reification combinator is deep, it offers reification for all expressions, including propagators for which the constraint programming system itself does not offer a reified version.

*Plan of the Paper.* Section 2 outlines the computation model and introduces some terminology. The following section introduces computation spaces by discussing a negation combinator. Sections 4 to 6 discuss a generic reification combinator, a disjunction combinator, and a conditional combinator. Section 7 provides a complete overview of computation and relates computation spaces presented in this paper to computation spaces available in Oz. The paper is concluded by Sect. 8.

## 2 Prerequisites

This section introduces the model of computation and notions used in the remainder of the paper.

Computation is performed in a *computation space*. A computation space consists of propagators and threads (to be explained later) connected to a constraint store:




The *constraint store* holds information about values of variables expressed by a conjunction of basic constraints. *Basic constraints* are logic formulae interpreted in a fixed first-order structure. For the purpose of this paper we restrict ourselves to finite domain constraints. A basic finite domain constraint has the form $x \in D$ where $D$ is a finite subset of the positive integers. Other relevant basic constraints are $x = y$ and $x = n$, where $n$ is a positive integer.

More expressive constraints, e.g., $x + y = z$, are not written to the constraint store. Instead, they are imposed by propagators. A *propagator* is a concurrent agent that tries to amplify the store by *constraint propagation*: The propagator amplifies the store by telling new basic constraints to it. A propagator imposing $P$ disappears as soon as it detects that $P$ is entailed by the store's constraints. A propagator imposing $P$ becomes *failed* if it detects that $P$ is inconsistent with the constraints hosted by the store.

In addition to propagators, the computation space contains threads. A thread is a functional evaluator operating on the store. As programming language that defines the set of expressions threads can evaluate, we use Standard ML extended by threads and logic variables. This extension of SML is due to Smolka, for more information we refer the reader to [12].

Threads are used to provide the concurrent execution needed by concurrent constraint combinators. Logic variables are used for synchronization. For example, the evaluation of an application `x ()` might block since `x` can be a logic variable. The application blocks until `x` is bound to a value (in this particular case, a function).

## 3 Getting Started: A Concurrent Negation Combinator

This section familiarizes the reader with computation spaces by showing how to program a concurrent negation combinator from them.

For a given constraint $C$ the negation combinator provides an implementation for the constraint $\neg C$. The negation combinator $\neg C$ executes the propagator for $C$ and:

- disappears, if the propagator for $C$ becomes failed.
- fails, if the propagator for $C$ becomes entailed.

Execution of $C$ by the negation combinator requires *encapsulation* of the computation performed by $C$. Basic constraints that are told by propagation of $C$ must be hidden from any other computation. On the other hand, basic constraints that are told by other computations must be visible to $C$.

We are looking for a method to build a *compositional* negation combinator:

– It must be general enough to deal with statements that post propagators rather than with a single propagator. This supports modularity. Typically, several constraints are composed together by some expression $E$.
– Execution for both expressions and propagators should remain unchanged.

*Local Computation Spaces. Local computation spaces* are used as primitives. The expression $E$ to be executed by the negation combinator is delegated to a local computation space. A local computation space is created by the primitive operation

```
space : (unit -> unit) -> space
```

To execute an expression $E$ in a computation space, the application

```
space(fn () => E)
```

is evaluated which returns the newly created space.

Variables, propagators, and threads are now situated in exactly one space $S$, which we call the entity's *home*. If an expression $E$ is being executed by a thread $T$ whose home is $S$, we refer to $T$ as the *current thread* and to $S$ as the *current space*.

Evaluation of `space e` in the space $S_1$ returns a newly created space $S_2$, which is initialized as follows. The constraint store of $S_2$ contains all constraints of $S_1$'s constraint store. A new thread is spawned in $S_2$ to evaluate `e ()`. We refer to this thread as $S_2$'s *root thread*. $S_1$ is called the *parent (space)* of $S_2$.

This construction naturally leads to a tree of computation spaces. The root of the tree we refer to as *toplevel (space)*. Spaces that occur in the subtree rooted at space $S$ (not including $S$) are called subordinated to $S$. A space $S_1$ is superordinated to a space $S_2$, if $S_2$ is subordinated to $S_1$.

With the exception of telling a basic constraint $\phi$, both threads and propagators compute in the same way as in the toplevel space. Telling a basic constraint $\phi$ in a space $S$ means to tell $\phi$ to $S$'s store and to all stores in spaces subordinated to $S$.

*Status of a Space.* In addition to starting an encapsulated computation, the negation combinator needs access to the *status* of the encapsulated computation.

A space $S$ is called *blocked*, if all threads and propagators within $S$ and within spaces subordinated to $S$ cannot reduce. A space $S$ is *stable*, if it is blocked and remains blocked regardless of any tell operations performed in a space superordinated to $S$.

A space becomes *failed* by an attempt to tell a basic constraint to the store that would make it inconsistent. Failing a space $S$ discards all threads and propagators in $S$ and also fails all spaces subordinated to $S$. Note that a space $S$ can be failed by a tell issued by a thread whose home is $S$, as well as by a tell issued in a space superordinated to $S$. Further note that a failed space is also stable.

A space is *solved*, if it is stable, not failed, and does not contain any propagators or threads. Note that the constraint store of a solved space $S$ is entailed by the constraint store of $S$'s parent. This justifies why we sometimes refer to a solved space as *entailed*.

A space that is stable, but neither failed nor solved, is *stuck*. If a space $S$ becomes stuck, it has arrived at a state where it contains propagators or threads that block on variables that are local to $S$ (otherwise $S$ would be blocked, but not yet stable). This

means that constraint propagation within $S$ has not been strong enough to completely drive reduction of all threads and propagators. In other words, a stuck space is usually the result of a programming error.

The operation

```
datatype status = Failed | Solved | Stuck
status : space -> status
```

takes as input a space $S$ and if $S$ is stable, returns $S$'s status. If $S$ is not stable there are two alternative designs: either `status` blocks until $S$ becomes stable, or behaves asynchronously. We choose the asynchronous behavior: if $S$ is not stable, `status` returns a logic variable that is bound to $S$'s status when $S$ becomes stable. Subsequent examples clarify that the asynchronous design is preferable.

*The Combinator.* The concurrent negation combinator takes an expression (as a first-class function `e` of type `unit -> unit`) and creates a space running `e`.

To make the combinator concurrent, a new thread is created that blocks until the created space becomes stable and then takes the appropriate action. For thread creation we use the function

```
spawn : (unit -> unit) -> unit
```

It spawns a new thread for execution of an expression $E$, which is passed as function `fn () =>` $E$. Taking this together, we arrive at:

```
fun not c =
    let val s = space c
    in  spawn(fn () => case status s of
                         Failed => ()
                       | Solved => fail()
                       | Stuck  => raise Error) end
```

Here `fail` is a function that fails the current space (e.g., by attempting to tell the constraint $1 = 2$ to the store).

## 4 A Generic Reification Combinator

As it has been argued in the introduction, reification of constraints is a powerful and natural way to combine constraints. This section presents a generic reification combinator. The reification combinator is shown to sometimes provide stronger constraint propagation than constructions that use reified propagators alone.

*Reification.* The reification of a constraint $C$ with respect to a 0/1-variable $b$ (a finite domain variable with domain $\{0, 1\}$) is the constraint $C \leftrightarrow b = 1$. The idea behind reification is to reflect whether $C$ holds into whether the *control variable* $b$ is 0 or 1.

Operationally, it is important that reification is bidirectional:

**"$\Rightarrow$"** If $C$ holds, $b = 1$ must hold; and if $\neg C$ holds, $b = 0$ must hold.

**"⇐"** If $b = 1$ holds, $C$ must hold; and if $b = 0$ holds, $\neg C$ must hold.

Having 0/1-variables $b$ that reflect validity of constraints allows for powerful means to combine constraints. Common examples for combination are boolean connectives expressed by propagators (see Sect. 5 for an example).

Direction "⇒" can be programmed along the lines of the negation combinator of Sect. 3. Suppose that `s` refers to the space running the expression $E$ to be reified and `b` refers to the 0/1-variable. Then Direction "⇒" is as follows:

```
⟨"⇒"⟩ := case status s of
           Failed => tell(b, 0)
         | Solved => tell(b, 1)
         | Stuck  => raise Error
```

Here `tell(b, `$i$`)` is used to tell the basic constraint `b` $= i$ to the constraint store.

Let us consider the case of Direction "⇐" where `b` is determined to 0. In this case, if the space `s` becomes solved, the current space must be failed. Otherwise, if the space `s` becomes failed, nothing has to be done. This behavior is already realized by the above encoding of Direction "⇒".

*Committing a Space.* Let us consider the case of Direction "⇐" for $b = 1$. The required operational behavior includes two aspects. Firstly, a computation state must be established as if execution of $E$ had not been encapsulated. Secondly, if $E$ has not yet been completely evaluated, its further execution must perform without encapsulation.

These two aspects are dealt with by the operation

```
commit : space -> unit
```

It takes a computation space $S_2$ and merges $S_2$ with the current space $S_1$ (which is $S_2$'s parent) as follows. If $S_2$ is failed, also $S_1$ becomes failed. Otherwise:

1. All constraints of $S_2$'s constraint store are told to $S_1$'s constraint store. By this, the effects of computations performed in $S_2$ are made available in $S_1$.
2. All propagators and threads situated in $S_2$ now become situated in $S_1$. From now on, they execute as if they had been created in $S_1$ in the first place.

Using `commit`, Direction "⇐" of the reification combinator is encoded as follows:

```
⟨"⇐"⟩ := if value b = 1 then commit s else ()
```

Here the function `value` takes a finite domain variable, blocks until it becomes determined, and returns its integer value.

*The Combinator.* The reification combinator is obtained from the implementation of both directions, which must execute concurrently. Concurrent execution is achieved by spawning a thread for each direction.

Taking the two directions together we arrive at a function `reify` that takes a function that specifies the expression to reify as input and returns a 0/1-variable:

```
fun reify e =
    let val s = space e  val b = fdvar(0, 1)
    in spawn(fn () => ⟨"⇒"⟩) ;
       spawn(fn () => ⟨"⇐"⟩) end
```

*Comparison with Propagator-based Reification.* It is instructional to compare space-based reification with propagator-based reification. Suppose we are interested in reifying the conjunction of the two constraints $x + 1 = y$ and $y + 1 = x$ with respect to the variable $b$, where both $x$ and $y$ are finite domain variables. Similar reified constraints occur in computing Hamiltonian paths.

Ideally, the reification mechanism should determine $b$ to 0, since the conjunction is unsatisfiable. Posting the constraints without reification exhibits failure.

Let us first study reification with propagators alone. In order to obtain a reified conjunction, we have to reify each of the conjuncts by introducing two control variables $b_1$ and $b_2$. Altogether we arrive at

$$b_1 = (x + 1 = y) \wedge b_2 = (y + 1 = x) \wedge b \in \{0, 1\} \wedge b_1 \times b_2 = b$$

Neither $b_1$ nor $b_2$ can be determined, thus $b$ cannot be determined.

Let us now study the behavior of the reification combinator developed in this section. It is applied as

```
b = reify(fn () => (x + 1 = y ; y + 1 = x))
```

Both constraints are posted in the same local space $S$. Exactly like posting them in the toplevel space, constraint propagation leads to failure of $S$. Indeed, the reification combinator determines $b$ to 0.

This shows that using spaces for reification can yield better constraint propagation than using per propagator reification. Per propagator reification encapsulates the propagation of each propagator. This in particular *disables* constraint propagation in reified conjunctions. This is a major disadvantage, since reified conjunctions occur frequently as building block in other reified constructions as for example disjunction.

On the other hand the generic reification combinator offers weak propagation in case the control variable is determined to be 0. Instead of propagation, constraints told by other propagators are tested only. Whenever a reified propagator is available, it is preferable to use it directly.

So the reification combinator can be best understood as offering additional techniques but not as a replacement of reified propagators.

## 5 Disjunction

This section shows how to program disjunctive combinators that resolve their alternatives by propagation rather than by search. Disjunctive combinators occur frequently in a variety of application domains, a well-known example is scheduling. For examples that use disjunctive combinators in the domain of computational linguistics see [ 1].

Let us consider a disjunction

$$E_1 \vee \cdots \vee E_n$$

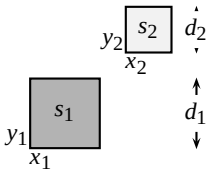
that is composed of $n$ expressions $E_i$. We refer to the $E_i$ as the disjunction's *alternatives*. A straightforward operational semantics is as follows:

1. Discard failed alternatives ($\bot \vee C$ is logically equivalent to $C$).
2. If a single alternative $E$ remains, reduce the disjunction to $E$ (a disjunction with a single alternative $C$ is equivalent to $C$).
3. If all alternatives have failed, fail the current space (a disjunction with no alternatives is equivalent to $\bot$).

This operational semantics can be directly encoded by the reification operator as introduced in Sect. 4. The well-known encoding reifies each alternative $E_i$ with respect to a 0/1-variable $b_i$. The disjunction itself is encoded by

$$b_1 + \cdots + b_n \geq 1$$

The suggested operational semantics is driven by failure only. However, it can be beneficial to also take entailment of alternatives into account. As an example consider the placement of two squares $s_1$ and $s_2$ such that they do not overlap. A well known modeling of this constraint is




$$x_1 + d_1 \leq x_2 \vee x_2 + d_2 \leq x_1 \vee y_1 + d_1 \leq y_2 \vee y_2 + d_2 \leq y_1$$

where the meaning of the variables $x_i$, $y_i$, and $d_i$ is sketched to the right. The squares do not overlap, if the relative position of $s_1$ with respect to $s_2$ is either left, right, above, or below. As soon as one of the relationships is established, the squares are guaranteed to not overlap.

Suppose $s_1$ is placed left to $s_2$. Since the first and second alternatives are mutually exclusive (so are the third and fourth), the first and second reified propagator disappears. However, the third and fourth remain.

Assume a constraint store $C$ and a disjunction $C_1 \vee C_2$ where $C_1$ is entailed by $C$ (that is, $C \rightarrow C_1$ is valid). Under this condition, $C_1 \vee C_2$ is logically equivalent to $\top \vee C_2$, which in turn is equivalent to $\top$. This justifies extending the operational semantics of the disjunctive combinator as follows:

4. If an alternative is entailed, reduce by discarding all alternatives.

Taking entailment into account has the following advantages: execution can be more efficient since computations that cannot contribute are discarded early. The computation space in which the disjunctive combinator is executed can possibly become solved sooner. In our compositional setup this might allow for earlier reduction of other combinators and by this provide better propagation.

*Discarding a Space.* For programming a disjunctive combinator with entailment we need to discard a computation space. The primitive

```
discard : space -> unit
```

discards a computation space $S$ by failing it. The operational semantics is exactly as if creating a new thread in $S$ that executes `fail ()`.

The implementation of the disjunctive combinator can be simplified by the following observation: it is sufficient to discard all failed alternatives but the last one. If a

single alternative remains, commit to it, regardless of whether the alternative is failed or not. Committing a failed space fails the current space (see Sect. 4). In the following the discussion is limited to a binary disjunctive combinator only. Generalization to the *n*-ary case is straightforward.

A function `or` that takes two alternatives `a1` and `a2` (again encoded as first-class functions) decomposes naturally into three parts: space creation for encapsulated execution of the alternatives, a concurrent controller, and last but not least the part that implements the reduction rules as discussed before. This yields the following encoding:

```
fun or(a1, a2) =
    let val (s1, s2) = (space a1, space a2)
        fun reduce(s1, s2) = ⟨Reduction⟩
    in ⟨Controller⟩ end
```

The concurrent controller blocks until either `s1` or `s2` becomes stable. This indeterminate choice is encoded by:

```
first : 'a * 'b -> bool
```

The application `first(x,y)` takes logic variables `x` and `y` as input and blocks until at least one of `x` and `y` becomes determined. If it returns `true` (`false`), `x` (`y`) is determined. Now the concurrent controller can be programmed from `first` which is applied to the status of both `s1` and `s2` as follows:

```
⟨Controller⟩ := if first(status s1, status s2)
                then reduce(s1, s2) else reduce(s2, s1)
```

The concurrent controller guarantees the invariant that the first space to which `reduce` is applied, is stable.

Finally, reduction is programmed as follows:

```
⟨Reduction⟩ :=
if failed s1 then commit s2
else if solved s1 then (discard s1 ; discard s2)
else if failed s2 then commit s1
     else if solved s2 then (discard s1 ; discard s2)
     else raise Error
```

where `failed` (`solved`) returns `true`, if applied to a failed (solved) space. Both `failed` and `solved` can be obtained straightforwardly from `status`. The part of `reduce` that does not have a gray background executes immediately, since the concurrent controller ensures that `s1` is stable. The gray part synchronizes on stability of `s2`.

Without the nested `if`-statements that test whether `s1` or `s2` are solved, the programmed disjunctive combinator implements the reduction rules 1 to 3. As it has been argued, this simplified version of disjunction can be expressed by the reification combinator introduced in Sect. 4.

The other direction, that is programming `reify` by using `or`, is also possible with the additional use of `not` as introduced in Sect. 3. The reification of expression *E* with respect to `b` can be programmed as follows:

```
or (fn () => (tell(b,1) ; E),
    fn () => (tell(b,0) ; not(fn () => E)))
```

Programming reification from disjunction has the disadvantage that the expression $E$ is executed twice. This points out a deficiency in the designs of AKL and early versions of Oz, where neither spaces nor reification but disjunction was provided.

## 6 Conditional

This section shows how to program conditionals that use arbitrary expressions as conditions (so-called *deep guards*). In particular it presents how to use continuations that allow to share variables between the condition and the body of a conditional. We also study how to apply the same ideas to parallel conditionals and disjunctions.

A conditional consists of three constituents, all of which are expressions: a guard $G$, a body $B$, and an else-constituent $E$. A common suggestive syntax would be

```
cond G then B else E
```

The part $G$ `then` $B$ is called the *clause* of the conditional.

Programming a conditional from computation spaces is straightforward. The program used for programming `not` (see Sect. 3) can be adapted as follows:

```
fun cond(g,b,e) =
    let val s = space g
    in  case status s of
            Failed => e()
          | Solved => b()
          | Stuck  => raise Error end
```

where `g`, `b`, and `e` are functions that specify the guard, body, and else-constituent of the conditional. In contrast to the concurrent negation combinator, the conditional is sequential. It does not spawn a new thread to synchronize on stability of the guard's space.

A common desire is to introduce variables $\overline{x}$ locally in the guard $G$ of the conditional and use them in the body. Thus the conditional should synchronize on entailment of $\exists\overline{x}G$. In our current setup, the bindings computed for $\overline{x}$ in $G$ are not accessible. An inefficient and thus unsatisfactory solution would be to execute the guard expression again together with the body.

A more satisfactory solution is to let the guard pass the variables to the body. This can be accommodated by extending computation spaces as follows. The root thread in a space computes a result (of some type `'a`). Committing the space gives access to that result. That is, space creation and committing of spaces is extended as follows:

```
space  : (unit -> 'a) -> 'a space
commit : 'a space -> 'a
```

Note that this extension of `space` and `commit` does not require a modification of the programs presented so far in this paper (in these cases `'a` is just `unit`).

A space can be committed before the root thread has terminated and has computed the result. Therefore `commit` returns a logic variable that is bound to the root thread's result as soon as it terminates.

In the context of a programming language with first-class functions the sharing of variables between guard and body is achieved straightforwardly by letting the guard return as result a function for the body:

```
let x̄ in (G ; fn () => B) end
```

Here $B$ can refer to variables declared in the `let`-expression. Without first-class functions, the variables would be stored in an appropriate data structure.

Programming the conditional from the extended primitives is now straightforward.

```
fun cond(c, e) =
    let val s = space c
    in case status s of
           Failed => e()
         | Solved => let val b = commit s in b() end
         | Stuck  => raise Error end
```

*Parallel Conditional.* A common combinator is a *parallel* conditional that features more than a single clause with a committed choice operational semantics: As soon as the guard of a clause becomes entailed, commit the conditional to that clause (that is, continue with reduction of the clause's body). Additionally, discard all other guards. If the parallel conditional also features an else-constituent $E$, reduce the conditional with $E$ if all guards have failed.

Encoding the parallel conditional from computation spaces follows closely the program for the disjunction presented in Sect. 5. In fact, the setup of the computation spaces for guard execution and the concurrent controller can remain unchanged. The function that implements the reduction rules is as follows:

```
⟨Reduction⟩ :=
let val b = if solved s1 then
               (discard s2 ; commit s1)
            else if solved s2 then
               (discard s1 ; commit s2)
            else raise Error
in b() end
```

The encoding is simplified in that it does not consider the straightforward handling of an else-constituent.

*Clauses for Disjunction.* The disjunctive combinator presented in Sect. 5 can be extended to employ clauses as alternatives. This extension is straightforward but two issues require some consideration. Firstly, when to start execution of a clause's body? Secondly, for which clause employ reduction by entailment?

Execution of the parallel conditional evaluates a clause's body $B$ only after the clause's guard $G$ has become entailed. This in particular ensures that the root thread has terminated and has computed $B$ as its result. A disjunctive combinator, in contrast, can already commit to a clause $C$ if its guard $G$ is not yet stable, provided the clause is the last remaining.

Nevertheless, it is desirable that evaluation of the $C$'s body $B$ starts only after $G$ has been completely evaluated. The semantics of `commit` ensures this: It returns a logic variable that is bound to the root thread's result as soon as it terminates. Since function application synchronizes (see Sect. 2), evaluation of the body synchronizes on termination of the root thread.

As discussed in Sect. 5 it is beneficial to consider both failure and entailment of alternatives for the disjunctive combinator. Reduction by entailment is justified by the fact that if an alternative $A$ is entailed it becomes logically equivalent to $\top$. This justification does apply to a clause only if its body is known to be logically equivalent to $\top$ as well. A possible solution is to tag clauses appropriately (as $\top$-clause). Reduction by entailment is then applied to $\top$-clauses only.

## 7 Computation Spaces: Summary and Comparison

The signature of all space operations that are necessary to program concurrent constraint combinators is as follows:

```
type 'a space
datatype status = Failed | Solved | Stuck

val space   : (unit -> 'a) -> 'a space
val status  : 'a space -> status
val commit  : 'a space -> 'a
val discard : 'a space -> unit
```

*Search Engines.* A different use of computation spaces is to use them for programming search engines. Search requires two further concepts: cloning and choice points.

A search engine takes a specification of the search problem (as function) and runs it in a computation space. The primitive

```
val choose : unit -> int
```

creates a choice point. A thread that executes `choose()` blocks. If a stable space contains a thread blocking on `choose()`, the `status`-operation is extended to return `Choice`. The search engine uses

```
val select : 'a space -> unit
```

to select whether `1` or `2` is returned by the blocking `choose` operation. The last additional primitive

```
val clone : 'a space -> 'a space
```

creates a copy of stable space. A search engine can use `clone` to implement backtracking. The programming of search engines from computation spaces is detailed in [ 9,10].

*Spaces in Oz.* As has already been argued in the introduction, the choice of SML as host language for spaces is mostly to stress language independence. In the following we relate the computation spaces as presented in this paper to computation spaces in Oz, where they have been originally conceived.

Spaces as presented here are almost identical to spaces in Oz with the exception of the additional concept of a *root variable*. Each space in Oz is created initially with a logic variable as root variable. The function supplied with space creation is applied to that root variable. Thus the root variable in Oz roughly corresponds to the result computed by the root thread in this paper. This paper does not introduce the root variable since it is not needed for programming combinators.

*Combinators in Oz.* The latest Mozart implementation of Oz (version 1.1.0) switched from a native C++-based implementation of combinators to a space-based implementation. Information on techniques for native implementation of combinators can be found in [4,6,5]. The main motivation to switch was to simplify the implementation. The goal is to decrease the necessary maintenance effort which has been considerable with the native implementation.

First experiments suggest that the space-based implementation is competitive to the native C++-implementation as it comes to runtime and memory requirements. Space consumption of both approaches is approximately the same. Native combinators are approximately twice as fast for programs where execution time is dominated by reduction of combinators (for example, appending two lists, where a deep guard conditional is used to decide whether the first input list is empty or not). For examples where the runtime is dominated by constraint propagation, both approaches offer approximately the same execution speed.

## 8 Conclusion

In this paper, we have presented computation spaces as primitives for deep concurrent constraint combinators. We have shown how to program negation, generalized reification, disjunction, and conditional from computation spaces.

The paper displays the simplicity of the approach: all combinators are obtained from a single concept with very few (four) primitive operations. By the choice of Standard ML with concurrency extensions as host language we have demonstrated that our approach is mainly language independent.

Computation spaces are provided by the Mozart implementation of Oz, which is available from `www.mozart-oz.org`.

**Acknowledgements**

Thanks to Denys Duchier, Tobias Müller, and Gert Smolka for fruitful discussions on combinators and computation spaces. Leif Kornstaedt, Tobias Müller, Andreas Rossberg, Gert Smolka, and the anonymous referees provided helpful comments. Tobias brought to my attention that the example in Sect. 4 occurs in computing Hamiltonian paths.

## References


1. Denys Duchier and Claire Gardent. A constraint-based treatment of descriptions. In H. C. Bunt and E. G. C. Thijsse, editors, *Third International Workshop on Computational Semantics (IWCS-3)*, pages 71–85, Tilburg, NL, January 1999.
2. Thom Frühwirth. Constraint handling rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, 1995.
3. Seif Haridi, Sverker Janson, and Catuscia Palamidessi. Structural operational semantics for AKL. *Future Generation Computer Systems*, 8:409–421, 1992.
4. Sverker Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994. SICS Dissertation Series 14.
5. Michael Mehl. *The Oz Virtual Machine: Records, Transients, and Deep Guards*. Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany, 1999.
6. Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.
7. Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, USA, 1993.
8. Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, USA, January 1990. ACM Press.
9. Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
10. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Linz, Austria, October 1997. Springer-Verlag.
11. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
12. Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.
13. Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 383–403. The MIT Press, Cambridge, MA, USA, 1993.
14. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, October 1998.