Concurrent Constraint Programming Based on Functional Programming

(Extended Abstract)

Gert Smolka

Programming Systems Lab
DFKI and Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
smolka@dfki.de, http://www.ps.uni-sb.de/~smolka/

1 Introduction

We will show how the operational features of logic programming can be added as conservative extensions to a functional base language with call by value semantics. We will address both concurrent and constraint logic programming [9, 2, 18]. As base language we will use a dynamically typed language that is obtained from SML by eliminating type declarations and static type checking. Our approach can be extended to cover all features of Oz [6, 15].

The experience with the development of Oz tells us that the outlined approach is the right base for the practical development of concurrent constraint programming languages. It avoids unnecessary duplication of concepts by reusing functional programming as core technology. Of course, it does not unify the partly incompatible theories behind functional and logic programming. They both contribute at a higher level of abstraction to the understanding of different aspects of the class of programming languages proposed here.

2 The Base Language DML

As base language we choose a dynamically typed language DML that is obtained from SML by eliminating type declarations and static type checking. Given the fact that SML is a strictly statically typed language this surgery is straightforward. To some extent it is already carried out for the definition of the dynamic semantics in the definition of SML [4].

There are several reasons for choosing SML. For the extensions to come it is essential that the language has call by value semantics, sequential execution order, and assignable references. Moreover, variants, records and exceptions are important. Finally, SML has a compact formal definition and is well-known.

Since SML does not make a lexical distinction between constructors and variables, we need to retain constructor declarations. Modules loose their special status since they can be easily expressed with records functions.

Since DML is dynamically typed, the primitive operations of the language will raise suitable exceptions if some of their arguments are ill-typed. Equality in DML is defined for all values, where equality of functions is defined analogous to references.

Every SML program can be translated into an DML program that produces exactly the same results. Hence we can see SML as a statically typed language that is defined on top of DML. There is the interesting possibility of a programming system that based on DML offers a variety of type checking disciplines. Code checked with different type disciplines can be freely combined.

To provide for the extensions to come, we will organize the operational semantics of DML in a style that is rather different from the style used in the definition of SML [4].

3 Values

We distinguish between primitive values and compound values. Primitive values include numbers, nullary constructors, and names. Names represent primitive operations, reference cells, and functions. Compound values are obtained by record and variant construction.

We organize values into a first-order structure over which we obtain sufficiently rich first-order formulas. This set-up gives us a relation $\alpha \models \phi$ that holds if an assignment α satisfies a formula ϕ. An *assignment* is a mapping from variables to values. The details of such a construction can be found in [17].

4 States

A *state* is a finite function σ mapping addresses a to so-called units u. *Units* are either primitive values other than names or representations of records, variants, reference cells, functions, and primitive operations:

$$\begin{gathered}\{l_1 = a_1, \dots, l_k = a_k\} \\ c(a) \\ \texttt{ref}(a) \\ \texttt{fun}(\mathit{Match},\ \mathit{Env}) \\ \texttt{primop}\end{gathered}$$

A match *Match* is a sequence of clauses `(`p_1`=>`e_1 `|` ... `|` p_k`=>`e_k`)`. An environment *Env* is a finite function from program variables to addresses.

For technical convenience we will identify addresses, names and variables occurring in formulas. We say that an assignment α satisfies a state σ and write $\alpha \models \sigma$ if for every address a in the domain of σ the following holds:

1. If $\sigma(a)$ is a primitive value v, then $\alpha(a) = v$.
2. If $\sigma(a)$ is a reference cell, a function, or `primop`, then $\alpha(a) = a$.
3. If $\sigma(a) = \{l_1 = a_1, \dots, l_k = a_k\}$, then $\alpha(a) = \{l_1 = \alpha(a_1), \dots, l_k = \alpha(a_k)\}$.
4. If $\sigma(a) = c(a')$, then $\alpha(a) = c(\alpha(a'))$.

Note that every state is satisfiable and that all assignments satisfying a state σ agree on the domain of σ.

Our states and environments are different from the corresponding notions in the definition of SML. There environments map program variables to values and states map addresses to values. In our set-up environments map program variables to addresses and states map addresses to units. This means that our states represent both stateful and stateless information. Moreover, our states make structure sharing explicit. The sharing information is lost when we move from a state to a satisfying assignment. Given a state σ, the unique restriction of an assignment satisfying σ to the domain of σ is an environment in the sense of the definition of SML.

The relation $\sigma \models \phi$ is defined to hold if and only if every assignment that satisfies σ also satisfies ϕ. If the free variables of ϕ are all in the domain of σ, then we have either $\sigma \models \phi$ or $\sigma \models \neg\phi$. This means that a state has complete information about the values of its addresses. We require that the first-order language be rich enough so that for every σ there exists a formula ϕ such that

$$\alpha \models \sigma \quad \Longleftrightarrow \quad \alpha \models \phi$$

for all assignments α. Note that σ determines ϕ up to logical equivalence. We use ϕ_σ to denote a constraint with the above property and say that ϕ_σ represents the *logic content of* σ.

5 Thread and Store

The operational semantics of DML distinguishes between a thread and a store. The thread is a functional evaluator that operates on the store. The states of the store are the states defined above. The store should be thought of as an abstract data type that is accessed by the thread only through a number of predefined operations. An example of such an operation is record selection, which takes an address a and a label l and returns an address or an exception packet.

An interesting primitive operation is the equality test $a_1 = a_2$. It returns `true` if $\sigma \models a_1 = a_2$ and `false` if $\sigma \models \neg\, a_1 = a_2$. Note that this definition yields structural equality for records and variants.

We write $\sigma \rightarrow \sigma'$ to say that there is an operation on the store that will replace the state σ with a state σ'. The following *monotonicity property* holds for DML and all extensions we will consider:

$$\sigma \models \phi \wedge \sigma \rightarrow \sigma' \quad \Rightarrow \quad \sigma' \models \phi$$

provided all free variables of ϕ are in the domain of σ.

6 Logic Variables

We now extend DML with logic variables, one of the essentials of logic programming. Logic variables are a means to represent in a state partial information about the values of addresses. Logic variables are modelled with a new unit `lvar`. The definition of states is extended so that a state may map an address also to `lvar` or an address. We only admit states σ whose *dereference relation* $a \rightarrow_\sigma a'$ is terminating, where $a \rightarrow_\sigma a'$ holds iff $\sigma(a) = a'$. The case $\sigma(a) = a'$ may appear when a logic variable is bound. We use $\sigma^*(a)$ to denote the unique normal form of a with respect to the dereference relation $\rightarrow_\sigma$. The definition of the relation $\alpha \models \sigma$ is extended as follows:

1. If $\sigma(a) = \texttt{lvar}$, then there is no constraint on $\alpha(a)$.
2. If $\sigma(a) = a'$, then $\alpha(a) = \alpha(a')$.

States can now represent partial information about the values of their addresses. This means that σ does not necessarily determine the truth value of a formula ϕ whose free variables are in the domain in σ.

Our states contain more information than necessary. For instance, if $\sigma(a) = a_1$ and $\sigma(a_1) = a_2$, then the difference between σ and $\sigma[a_2/a]$ cannot be observed at the level of the programming language. In general, we impose the semantic requirement that for a state σ and addresses a_1 and a_2 such that $\sigma \models a_1 = a_2$ the difference between a_1 and a_2 must not be observable. As it comes to space complexity, it is nevertheless important to model structure sharing.

The existing operations of DML are extended to the new states as follows. If an operation needs more information about its arguments than the state provides, then the operation returns the control value `blocked`. If there is only one thread, then computation will terminate. If there are several threads, the thread will retry the operation in the hope that other threads have contributed the missing information (see next section).

A match (e.g., (`x::xr =>` e_1 `| nil =>` e_2)) blocks until the store contains enough information to commit to one of the clauses or to know that none applies. Of particular interest is the equality test a_1=a_2, which we defined to return `true` if $\sigma \models a_1 = a_2$ and `false` if $\sigma \models \neg\, a_1 = a_2$. Since in the presence of logic variables σ may entail neither $a_1 = a_2$ nor $\neg\, a_1 = a_2$, the equality test a_1=a_2 may block. There is an efficient incremental algorithm [17] that checks for entailment and disentailment of equations $a_1 = a_2$.

We say that an operation is *granted* by σ if it does not block on σ. All extensions we will consider will satisfy the *generalized monotonicity condition*:

$$\text{if } \sigma \text{ grants } o \text{ and } \sigma \rightarrow \sigma', \text{ then } \sigma' \text{ grants } o.$$

The operation

```
lvar: unit -> 'a
```

creates a fresh logic variable and returns its address. The operation

```
isvar: 'a -> bool
```

returns `true` if its argument a dereferences to a logic variable (i.e., $\sigma(\sigma^*(a)) = \texttt{lvar}$) and `false` otherwise. The *variable binding operation*

```
<-: 'a * 'a -> 'a
```

expects that its left argument is a logic variable, binds it to its right argument, and returns the right argument. More precisely, if `<-` is applied to (a_1, a_2) and $\sigma^*(a_1) = a_3$, $\sigma(a_3) = \texttt{lvar}$ and $\sigma^*(a_2) = a_4$, we distinguish two cases:

1. If $a_3 = a_4$, then there is no side effect and a_4 is returned.
2. If $a_3 \neq a_4$, then the store is updated to $\sigma[a_4/a_3]$ and a_4 is returned.

The operation

```
wait: 'a -> 'a
```

is an identity function that blocks until its argument is bound to a nonvariable unit. This operation is useful for concurrent programming.

7 Multiple Threads

It is straightforward to extend DML with multiple threads. We use interleaving semantics, that is, the operations threads perform on the store do not overlap in time. Threads can be created with the expression

$$\texttt{spawn}\ e$$

which spawns a new thread evaluating e and returns `()`. Often it is convenient to use the derived form

$$\texttt{thread}\ e$$

which expands to

$$\texttt{let val}\ x\ \texttt{= lvar() in (spawn}\ x\ \texttt{<-}\ e\texttt{);}\ x\ \texttt{end}$$

where x is a program variable that does not occur free in e. For instance, if we want to evaluate the constituents of the application $e(e_1, e_2)$ concurrently, we can simply write

$$\texttt{(thread}\ e\texttt{) (thread}\ e_1\texttt{, thread}\ e_2\texttt{)}$$

since the necessary synchronization comes for free.

The combination of logic variables and reference cells provides for powerful synchronization techniques. For this we need an operation

```
exchange: 'a ref * 'a -> 'a
```

which updates the reference cell given as first argument to hold the second argument and returns the previous content of the cell. Now a function

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

that applies the function given as argument under mutual exclusion can be written as follows:

```
local val r = ref()
in
    fun mutex(a) =
        let val c = lvar()
        in  wait(exchange(r,c));
            let val v = a() in c <- (); v end
        end
end;
```

A function

```
channel: unit -> {put: 'a -> unit, get: unit -> 'a}
```

that returns an asynchronous channel (i.e., a concurrent queue) can be written as follows:

```
fun channel() =
    let val init = lvar()
        val putr = ref init
        val getr = ref init
        fun put(x) =
            let val new = lvar()
                val old = exchange(putr,new)
            in old <- x::new; () end
        fun get() =
            let val new = lvar()
                val x::c = exchange(getr,new)
            in new <- c ; x end
    in {put=put, get=get} end
```

The put function puts items on the channel and the get function gets items from the channel. The get function blocks until there is an item on the channel. The blocking is caused by the match

```
val x::c = exchange(getr,new)
```

To obtain fairness, the simple requirement that every thread that is not blocked will eventually advance suffices. In the two examples above starvation is excluded since the blocked threads are implicitly queued by means of logic variables. Note that both example functions encapsulate the logic variables they introduce. Our simple fairness requirement rests on the generalized monotonicity condition stated above (i.e., the property that a thread can advance cannot be invalidated by the operations performed by other threads). Languages that take channels as concurrency primitive (e.g., Pict [7]) require the more complicated fairness condition that our channels implement with logic variables.

The outlined style of concurrent programming originated with Oz and is explored in [15, 1]. The paper [15] relates to a previous version of Oz that did not have sequential composition. The book [1] is based on the current version of Oz and explores concurrent programming with object-oriented abstractions. The interested reader may also consult [19], which outlines a distributed version of Oz currently under development.

8 Unification

Next we define unification. We say that σ' is obtained from σ by a *narrowing step* if there are addresses a and a' in the domain of σ such that $\sigma(a) = \texttt{lvar}$, $\sigma^*(a') \neq a$, and $\sigma' = \sigma[a'/a]$. Note that the variable binding operation `<-` performs a narrowing step if it succeeds. We say that σ' is obtained from σ by *unification of* a_1 *and* a_2 if σ' can be obtained from σ by a minimal number of narrowing steps such that $\sigma' \models a_1 = a_2$ holds. If there is such a σ', we say that a_1 and a_2 are *unifiable in* σ. If σ' is obtained from σ by unification of a_1 and a_2, then $\phi_{\sigma'}$ is logically equivalent to $\phi_\sigma \wedge a_1 = a_2$. Moreover, a_1 and a_2 are unifiable in σ if and only if $\phi_\sigma \wedge a_1 = a_2$ is satisfiable. This *logical characterisation of unification* is a design principle and will also hold for the constraint extensions introduced in later sections.

The *unification operation*

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

expects that its two arguments a_1 and a_2 be unifiable. If this is the case, it narrows the state accordingly and returns a_2. Otherwise, it returns an exception packet.

Our states combine first-order constraints with higher-order functions and reference cells. Unification only concerns the part of a state that represents first-order constraints. Investigations of unification and constraint solving that relate to the unification defined here can be found in [3, 17].

9 Choices

An essential feature of logic programming is a built-in mechanism for search. To add this feature to DML, we introduce choice expressions of the form

$$\texttt{choice}\ e_1 \mid \ldots \mid e_k$$

A choice is evaluated by replacing it with one of its alternatives e_i. To make this practical, the choices are tried from left to right employing chronological backtracking as in Prolog. We arrange things such that a speculative computation terminates with failure if a unification operation fails. If there is only one thread, this gives us the search mechanism of pure Prolog.

If there are multiple threads, we require that a choice is only committed once all other threads are either blocked or can only advance by committing a choice.

10 Spaces

The outlined Prolog-like search is not satisfactory in a concurrent setting since search is done at the top level and cannot be encapsulated into concurrent agents. It also fails to provide means for programming search engines like all solution search. This long standing problem of logic programming is solved by Oz with a new concept called spaces. A space is a box consisting of a store and threads. Computation in a space is speculative and does not have a direct effect outside. Computation in a space proceeds until the space becomes either failed or stable. Stability means that no thread can advance except by committing a choice. There is an operation that blocks until a space is failed or stable and then reports the result. For stable spaces there are two possibilities: either there is a pending choice or not. If there is no pending choice, the space can be merged with the parent space to obtain the result of the speculative computation. If there is a pending choice, the space can be cloned and be committed to the respective alternatives.

Spaces turn out to be a simple and flexible means for programming search engines. A first version is described in [11,14]. A recent paper on spaces and their use is [10].

11 Finite Domain Constraints

Finite domain constraints are constraints over integers that in conjunction with constraint programming yield a powerful tool for solving combinatorial problems like scheduling [2,18,12]. To include them in our framework, we introduce a new unit `lvar`(D) that represents a logic variable that is constrained to take a value in D, where D must be a finite set of integers. Variable binding and unification are adapted so that they respect finite domain constraints. The primitive operations of DML treat finite domain variables like unconstrained variables. There is a new primitive operation

```
fdvar: findom -> int
```

that returns a fresh logic variable constrained to the finite domain given as argument. Unification is extended to handle constrained logic variables according to their logical meaning. For instance, the expression

```
let val x = fdvar[1,2] val y = fdvar[0,2] in x == y end
```

is equivalent to the expression 2.

More expressive constraints like $2*x = y$ are realized with concurrent agents called *propagators*. For instance, if the store knows that $x \in \{1,\ldots,10\}$ and $y \in \{1,\ldots,9\}$, a propagator for the constraint $2*x = y$ can narrow the domains of x and y to $x \in \{1,2,3,4\}$ and $y \in \{2,4,6,8\}$. This form of inference is called *constraint propagation*. In general, there will be many propagators that communicate through the store. The power of a constraint programming system depends on the class of propagators it offers. Depending on the constraints they realize, propagators often use nontrivial algorithms. A ubiquitous constraint is "$x_1, \ldots, x_k$ are all different". For instance, if the store knows

$$x \in \{1,2,3\} \quad y \in \{1,2,3\} \quad z \in \{1,2,3\} \quad u \in \{1,2,3,4,5\} \quad v \in \{1,3,4\}$$

a propagator for "x, y, z, u, v are all different" can narrow the domains to

$$x \in \{1,2,3\} \quad y \in \{1,2,3\} \quad z \in \{1,2,3\} \quad u \in \{5\} \quad v \in \{4\}$$

which determines the values of u and v. There is a complete propagation algorithm for the all different constraint that has quadratic complexity in the number variables and possible values [8].

12 Feature Constraints

Feature constraints are constraints over records that have applications in computational linguistics and knowledge representation. There is a parameterized primitive operation

`tellfeature#`*label*`: record * 'a -> unit`

that constrains its first argument to be a record that has a field with the label *label* and with the value that is given as second argument. For instance,

```
tellfeature#age(x,y)
```

posts the constraint that x is a record of the form `{age=y,...}`.

To accommodate feature constrains, we use units of the form

$$\texttt{lvar}(w, \{l_1 = a_1, \ldots, l_k = a_k\})$$

that represent logic variables that are constrained to records as specified. A record satisfies the above specification if it has at most w fields and at least a field for every label l_i with a value that satisfies the constraints for the address a_i. The metavariable w stands for a nonnegative integer or ∞, where $k \leq w$.

There is also a primitive operation

```
tellwidth: record * int -> unit
```

that constrains its first argument to be a record with as many fields as specified by the second argument.

The operation `tellfeature` is in fact a unification operation. It narrows the store in a minimal way so that the logic content of the new state is equivalent to the logic content of the old state conjoined with the feature constraint told. For instance, the expression

```
let val x = lvar()
in  tellwidth(x,1); tellfeature#a(x,7); x end
```

is equivalent to the expression `{a=7}`.

Feature constraints and the respective unification algorithms are the subject of [17]. Feature constraints are related to Ohori's [5] inference algorithm for polymorphic record types.

13 Conclusion

The main point of the paper is the insight that logic and concurrent constraint languages can be profitably based on functional core languages with call by value semantics. This avoids unnecessary duplication of concepts. SML wins over Scheme since it has richer data structures and factored out reference cells.

Our approach does not unify the theories behind functional and logic programming. It treats the extensions necessary for concurrent constraint programming at an abstract implementation level. To understand and analyse concurrent constraint programming, more abstract models are needed (e.g., [9, 2, 13, 15, 16]).

It seems feasible to extend the SML type system to logic variables and constraints. Such an extension would treat logic variables similar to reference cells. Feature constraints could possibly be treated with Ohori's polymorphic record types [5].

The approach presented here is an outcome of the Oz project. The development of Oz started in 1991 from logic programming and took several turns. Oz subsumes all concepts in this paper but has its own syntax and is based on a relational rather than a functional core. The relational core makes Oz more complicated than necessary. The insights formulated in this paper can be used to design a new and considerably simplified version of Oz. Such a new Oz would be more accessible to programmers experienced with SML and would be a good vehicle for teaching concurrent constraint programming.

Acknowledgments

Thanks to Martin Müller and Christian Schulte for comments on a draft of the paper, and thanks to all Oz developers for inspiration.

References

1. M. Henz. *Objects for Concurrent Constraint Programming.* Kluwer Academic Publishers, Boston, Nov. 1997.
2. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.
3. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 1988.
4. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, MA, 1997.
5. A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Prog. Lang. Syst.*, 17(6):844–895, 1995.
6. Oz. The Oz Programming System. Programming Systems Lab, DFKI and Universität des Saarlandes: `http://www.ps.uni-sb.de/oz/`.
7. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* The MIT Press, Cambridge, MA, 1997.

8. J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 362–367, 1994.
9. V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.
10. C. Schulte. Programming constraint inference engines. In G. Smolka, editor, *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloss Hagenberg, Linz, Austria, Oct. 1997. Springer-Verlag.
11. C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium*, pages 505–520, Ithaca, New York, USA, Nov. 1994. The MIT Press, Cambridge, MA.
12. C. Schulte, G. Smolka, and J. Würtz. Finite domain constraint programming in Oz, a tutorial, 1998. Programming Systems Lab, DFKI and Universität des Saarlandes: `ftp://ftp.ps.uni-sb.de/oz/documentation/FDTutorial.ps.gz`.
13. G. Smolka. A foundation for concurrent constraint programming. In J.-P. Jouannaud, editor, *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, Berlin, Sept. 1994.
14. G. Smolka. The definition of Kernel Oz. In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, Berlin, 1995.
15. G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
16. G. Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4), Dec. 1996. Electronic Section.
17. G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.
18. P. Van Hentenryck, V. Saraswat, et al. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701–726, Dec. 1997. ACM 50th Anniversary Issue. Strategic Directions in Computing Research.
19. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5), Sept. 1997.