The Multiparadigm Programming Language CCFL

Petra Hofstedt

hofstedt@informatik.tu-cottbus.de

Brandenburg University of Technology Cottbus

Abstract: Constraints support an efficient modeling and solution of problems in two ways: The most common application is their use for the description of problems with incomplete knowledge. On the other hand constraints may also serve as particular language constructs for the control of the program evaluation process. This paper presents the *C*oncurrent *C*onstraint *F*unctional *L*anguage CCFL which comprises both aspects.

Real-world problems comprise aspects from several realms. They are, thus, often best implemented by a combination of concepts from different paradigms. This combination is comfortably realised by *multiparadigm programming languages*, an area of research and application which has attracted increased interest in the recent years.

The *C*oncurrent *C*onstraint *F*unctional *L*anguage CCFL is a multiparadigm programming language which combines concepts and constructs from the functional and the constraint-based paradigms. The language enables the description of deterministic computations using a functional programming style and of non-deterministic behaviour based on constraints. Moreover, constraints are used to describe systems of concurrent cooperating processes and even typical parallelization patterns. We discuss language concepts and applications by means of examples.

1 Functional Programming

In CCFL, functions are used to express deterministic computations. CCFL's functional sublanguage inherits notions and concepts from the functional languages HASKELL and OPAL [Opa04, PH06]. It is a lazy language with polymorphic type system. A function consists of a type declaration and a definition allowing the typical constructs such as case-expressions, let-expressions, function application, and some predefined infix operator applications, constants, variables, and constructor terms.

Free Variables In CCFL, expressions are allowed to contain free variables; this also applies to function applications. Function applications with free variables are evaluated using the residuation principle [Smo93], that is, function calls are suspended until the variables are bound to expressions such that a deterministic reduction is possible. For example, a function call $(4 + x)$ with free variable x will suspend. In contrast, a com-

putation `length` [1,*x*,3,2,*y*] of the length of a list containing free variables *x* and *y* is possible because a concrete binding of these variables is not necessary to proceed with the evaluation.

2 Constraint abstractions

Besides for the constraint-based problem description, user-defined constraints (or constraint abstractions) serve two further important purposes: dealing with non-determinism and expressing concurrent computations. We sketch on both in this section.

A constraint abstraction consists of a head and a body which may contain the same elements as a function definition. Additionally, the body can be defined by several alternatives the choice of which is decided by guards, i.e. *ask-constraints*. Each body alternative is a conjunction of *tell-constraints*. A constraint (abstraction) always has result type ***C***.

***ask*- and *tell*-constraints** Within a CCFL rule constraints may have two functionalities: *tell*-constraints generate concurrently working processes which propagate knowledge in form of variable bindings (or constraints in general). These processes communicate over common variables. In contrast, *ask*-constraints do not generate knowledge but check for concrete variable bindings or constraints. *ask*-constraints control the choice of (potentially competing) rules and, thus, allow to express the synchronization of concurrently working processes on the one hand and non-deterministic computations on the other hand.

Example 2.1 *Consider the user-defined constraint* `game` *in Prog. 2.1 which describes a game between two players. For example, a constraint application* `game x y 10` *initiates a game where both players throw the dice 10 times each and they reach the overall values* `x` *and* `y`*, resp.*

The expression `dice x1 & dice y1 & ... & game x2 y2` *(*`m`*−1) in lines 5–7 consists of conjunctively connected* tell*-constraints and they express the rules of the game.*

Guards with ask*-constraints can be found e.g. in the* `member`*-constraint (lines 14 and 15) and they realize the non-determinism in this program as discussed below.*

Concurrent Processes CCFL allows the description of systems of communicating and cooperating processes. The main idea is to express concurrent processes by means of conjunctions of *tell*-constraints.

The constraints in the body of the rules are *tell-constraints.* They create processes which may compute bindings for the incorporated variables. Several *tell*-constraints combined by the &-combinator (as in Prog. 2.1, lines 5–7) generate an according number of processes and these communicate over common variables. *tell*-constraints are either applications of user-defined constraints (e.g. `dice x1`) or they are equality constraints `x =:= fexpr` between a variable *x* and a functional expression `fexpr` (e.g. `x =:= x1 + x2`).

Program 2.1 A simple game of dice

```
 1  fun game :: Int → Int → Int → C
 2  def game x y n =
 3    case n of 0 → x =:= 0 & y =:= 0 ;
 4              m → with x1, y1, x2, y2 :: Int
 5                   in dice x1 & dice y1 &
 6                      x =:= x1 + x2 & y =:= y1 + y2 &
 7                      game x2 y2 (m−1)
 8
 9  fun dice :: Int → C
10  def dice x = member [1,2,3,4,5,6] x
11
12  fun member :: List a → a → C
13  def member l x  =
14     l =:= y:ys → x =:= y |
15     l =:= y:ys → case ys of []    → x =:= y ;
16                             z:zs → member ys x
```

Equality constraints are interpreted as strict. That is, the constraint s =:= t is satisfied, if both expressions can be reduced to the same ground data term [HAB+06]. While a satisfiable equality constraint `x =:= fexpr` produces a binding of the variable `x` to the functional expression `fexpr` and terminates with result value `Success`, an unsatisfiable equality is reduced to the value `Fail` representing an unsuccessful computation.

Non-deterministic Computations The atoms of the guard of a user-defined constraint are *ask-constraints.* If a guard of a rule with matching left-hand side is entailed by the current accumulated bindings and constraints, the concerning rule alternative may be chosen for further derivation. In case that the guard fails or cannot be decided yet, this rule alternative is suspended. If all rule alternatives suspend, the computation waits (possibly infinitely) for a sufficient instantiation of the concerning variables.

Example 2.2 *Consider the* `member`*-constraint in Prog. 2.1. The* ask*-constraints (lines 14, 15) of the guards of both alternatives are the same, i.e.* `l =:= y : ys`*, while the bodies differ. Thus, the evaluation of a constraint* `member` $[y_1,y_2,...,y_n]$ `x` *non-deterministically generates a constraint which binds the variable* `x` *to one list element* y_i*.*

For *ask*-constraints, we distinguish between bound-constraints `bound x` checking, whether a variable `x` is bound to a non-variable term (not used in our examples), and match-constraints `x =:= d` $x_1 \ldots x_n$ which test for a matching of the root symbol of a term bound to the variable `x` with a certain constructor `d`. The variables $x_1 \ldots x_n$ are fresh.

Example 2.3 *For Prog. 2.1 the constraint abstraction* `member` *is the only source of non-*

Program 3.1 Functional `map` and constraint-based `farm`

```
 1  fun map :: (a → b) → List a → List b
 2  def map f l =
 3    case l of []      → [];
 4              x : xs → (f x) : (map f xs)
 5
 6  fun farm :: (a → b) → List a → List b → C
 7  def farm f l r =
 8    case l of []      → r =:= [];
 9              x : xs → with rs :: List b
10                       in r =:= (f x) : rs & farm f xs rs
```

determinism. It non-deterministically chooses a value from a list such that a constraint application `dice x` *calling the* `member`*-constraint simulates the dice. The* tell*-constraints* `dice x1` *and* `dice y1` *(line 5) produce values which are consumed by the equality constraints* `x =:= x1 + x2` *and* `y =:= y1 + y2`*, resp. Note that their computation is suspended until the arguments are (sufficiently) instantiated to apply the built-in arithmetic function* +*.*

3 Parallel Programming

The differentiation between functions as computational core and constraints as coordinational core of CCFL allows an explicit control of concurrency and even to express typical parallelization schemes. The following examples are taken from [HL09].

Example 3.1 *Consider Prog. 3.1 defining a function* `map` *and a constraint abstraction* `farm`*. Both have the same general structure, i.e. a function* `f` *is applied to every element of a given list* `l` *and the results are composed into a new list. However, there is one fundamental difference: Since* `map` *is a function, it is evaluated sequentially. In contrast,* `farm` *is a user-defined constraint. Its evaluation yields two concurrently working processes generated from the constraint conjunction in line 10.*

While this uncontrolled form of parallelization as demonstrated in Example 3.1 may yield a huge number of, possibly computationally light-weight, concurrent processes, a selective control of the degree of parallelization of computations is possible in CCFL, too.

Example 3.2 *Prog. 3.2 shows a data parallel farm skeleton* `pfarm` *with granularity control. Here, the number of processing elements* `noPE` *determines the number of generated processes. The abstraction* `pfarm` *calls* `nfarm` *which splits the list to be processed into* `noPE` *sub-lists and generates an according number of processes for list processing. These are distributed across the parallel computing nodes by the run-time system of* CCFL*.*

Program 3.2 Farm parallelization patterns

```
1  fun nfarm :: Int → (a → b) → List a → List b → C
2  def nfarm n f l r =
3    with rs :: List (List b)
4    in let parts = partition n l;
5           pf    = map f
6       in farm pf parts rs & r =:= concat rs
7
8  fun pfarm :: (a → b) → List a → List b → C
9  def pfarm f l r = nfarm noPE f l r
```

CCFL does not feature specialized data structures to support data parallelism in contrast to other approaches [CLJ+07, Nit05]. Instead, the user provides a regular splitting of the data structure controlling the granularity of parallelism in this way[1], while the runtime system is responsible for an equitable distribution of the data (and tasks) onto the processing nodes. Thus, the step from data to task parallel skeletons is smooth in CCFL.

More examples, in particular on parallel processing, and a brief sketch of semantics and implementation details (including further references) of our language can be found in the extended version of this paper (attached to the proceedings and in electronic form).

4 Conclusion

Constraints support an efficient modelling and solution of problems in two forms: First, they are used to model and solve problems with incomplete knowledge. But secondly, they also allow to guide the program evaluation process. The *multiparadigm programming language* CCFL presented in this paper comprises both aspects. It is a successful approach on the integration of the functional and constraint-based paradigms allowing a comfortable modelling of systems of concurrent processes and typical parallelization patterns on the one hand, and of deterministic and non-deterministic behavior on the other hand.

Related Work GOFFIN [CGKL98] is a constraint functional language with a similar structure like CCFL. However, there are fundamental differences, e.g. in the nature of constraint abstractions, the *ask*-constraint's functionalities, and in expressing parallelism. Moreover, in [Hof09] we discuss the extension of CCFL by e.g. arithmetic constraints.

Concentrating on functional programming approaches, there are, e.g. the functional languages EDEN [LOMP05] and ERLANG [AVWW07] which, similar to CCFL, allow concurrent computation of processes. However, both use explicit notions for the generation of

[1] Thus, in our approach the number of processing elements *noPE* plays a role not only in the machine space but also on the level of the problem description.

processes and their communication. CONCURRENT HASKELL [PGF96] supports threads via the IO monad. DATA PARALLEL HASKELL [CLJ+07] targets multicore architectures and allows nested data-parallel programming based on a built-in type of parallel arrays.

Acknowledgment This work has been partially supported by a postdoctoral fellowship of the author, No. PE 07542, from the Japan Society for the Promotion of Science (JSPS).

References

[AVWW07] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 2007.

[CGKL98] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock. GOFFIN: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.

[CLJ+07] M.M.T. Chakravarty, R. Leshchinskiy, S.L. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Workshop on Declarative Aspects of Multicore Programming – DAMP*, pages 10–18. ACM, 2007.

[HAB+06] M. Hanus, S. Antoy, B. Braßel, H. Kuchen, F.J. Lopez-Fraguas, W. Lux, J.J. Moreno Navarro, and F. Steiner. Curry: An Integrated Functional Logic Language. Technical report, 2006. Version 0.8.2 of March 28, 2006.

[HL09] P. Hofstedt and F. Lorenzen. Constraint Functional Multicore Programming. In S. Fischer, E. Maehle, and R. Reischuk, editors, *Informatik 2009. GI Jahrestagung*, volume 154 of *Lecture Notes in Informatics (LNI)*, pages 367, 2901–2915. GI, 2009.

[Hof09] P. Hofstedt. Multiparadigm Constraint Programming Languages, 2009. Habilitation thesis. Technische Universität Berlin.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[Nit05] T. Nitsche. *Data Distribution and Communication Management for Parallel Systems*. PhD thesis, Technische Universität Berlin, 2005.

[Opa04] The OPAL Project. `http://uebb.cs.tu-berlin.de/~opal/`, 2004. last visited 2010-04-24.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages – POPL*, pages 295–308, 1996.

[PH06] P. Pepper and P. Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer, 2006.

[Smo93] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming. Selected Research*, pages 405–419. The MIT Press, 1993.