Functional Programming with Pros and Cons

Benny Höckner and Peter Sauer and Petra Hofstedt

Brandenburg University of Technology, Cottbus
{benny.hoeckner,peter.sauer,petra.hofstedt}@tu-cottbus.de

Abstract. When modelling real world problems different domain aspects are often best supported by language concepts from different programming paradigms. This yields the desire to combine them in multi-paradigm programming languages.
αCCFL is a multi-paradigm language inheriting from the functional programming language HASKELL which is extended by **con**traint**s** for dealing with incomplete information and agents to realize coarse grain computation **pro**cesse**s** and behaviours. We present the language αCCFL and discuss its concept of agents.

1 Motivation

Multi-agent systems aim at the modeling of objects featuring actions and behaviours and being able to act independently, but coordinately at the same time. Typically, every single agent has to make decisions based on incomplete knowledge about his environment and about the other agents. Constraints perfectly support dealing with such incomplete information. Furthermore, they are adequate to model non-determinism, which in turn can be used to emulate independent decisions. Due to the fact that declarative constraints are not suitable to encode agents and their stateful behaviours directly, we decided to combine concepts of agent-oriented and constraint-based programming in a new multi-paradigm language.

Our language αCCFL allows HASKELL-like functional programming to express deterministic calculations. Constraints support non-deterministic computations and concurrency. Agents use functional and constraint-based computations and communicate over a common constraint store. Using a *tell* operation, agents can add knowledge in form of constraints to the store and, thus, make available to other agents. Additionally, the agents use *ask* requests to test whether a constraint currently is entailed by the content of the store, this enables to guard the agent's actions. In this way the agents are forced to only perform actions based on current information from the store.

2 αCCFL – The Language

CCFL (a *C*oncurrent *C*onstraint-based *F*unctional *L*anguage) [3] was designed as an extension of functional programming in a HASKELL-like style by constraints to deal with incomplete information, non-determinism and concurrency.

Listing 2.1 αCCFL: functions and constraints

```
1  data Coin = Heads | Tails
2  headsOrTails :: Coin -> C;
3  headsOrTails x = #True -> x =:= Heads |
4                   #True -> x =:= Tails;
5  count :: Coin -> Int -> Int;
6  count coin val =
7    if (coin == Heads) (val + 10) val;
```

The functional sub-language of CCFL allows all typical constructs such as function definition and declaration, algebraic data types, polymorphism, and higher-order functions. Functional expressions are evaluated with call-by-need semantics. In contrast to HASKELL, expressions are allowed to contain free variables. Functional expressions with free variables are evaluated using the residuation principle, i.e. their evaluation suspends until a deterministic reduction is possible.

Non-deterministic and concurrent computations are described by constraints in CCFL. Atomic constraints are equality constraints on functional expressions or they are integrated from external constraint solvers. User-defined constraints allow to express alternative solutions. Their non-deterministic choice is controlled by guards.

Listing 2.1 shows the definition of an enumeration type *Coin*, a user-defined constraint *headsOrTails* and a function *count*.

The constraint *headsOrTails* describes the non-deterministic tossing of a coin with possible results *Heads* or *Tails*. The result type of constraints is always the predefined type **C**. The alternative branches of the constraint *headsOrTails* are both guarded by the always satisfiable constraint *# True*. Guards for constraint alternatives can, however, in general be any flat constraint (conjunction).

The function *count* just increases a value *val* or lets it unchanged in dependence of the result of tossing a coin.

αCCFL allows us, corresponding to the abstract architecture *Agents with state* [5], to define agents with state and behaviour. The schema of an agent is given by Listing 2.2. An agent describes an object consisting of attributes (Line 2), which correspond to the state of an agent, operations (Line 4), i.e. functions and constraints, and actions (Line 6).

Actions consist of guarded constraints which allow again a non-deterministic choice. Since agents are stateful, an alternative describes the agent behaviour at the current point of time *and* – by means of a **next**-block (or a *stop* action) – a state change and the agent's future behaviour.

Listing 2.3 shows an agent playing pitch-or-toss. Its attributes *points* and *mood* (Line 4) describe the overall point result of the game and the mood of the agent. The constraint *headsOrTails* and the function *count* are as described above, a constraint *change* (Line 10) allows to non-deterministically change the agent's mood.

Listing 2.2 Agent schema in αCCFL

```
1  behaviour AgentName {
2          attributes
3     } where {
4          functions and constraints
5     } actions {
6          stateful agent behaviour
7     }
```

Now about the actions: The agent behaves dependent on his mood. If he is in *Good* mood (Lines 14–19), he tosses the coin which may result in an increased *points* value and a change in mood (both transferred to the future via the **next** statement). Alternatively, if the agent is *Tired*, he either makes a coffee break to change his mood for the future behaviour (Lines 20–24) or he stops the game (Lines 25–26).

Inspired by the work of Saraswat on the concurrent constraint programming paradigm [4], agents are executed as concurrent processes communicating over a common constraint store. The store itself is able to deal with a large number of typical finite domain constraints and is capable of saving constraints over variables as well as lists of variables. Current versions are implemented in JAVA using the constraint library CHOCO [1] in Version 2.

The store is a monoton one, which means that we only can add new information. We cannot delete former information. Therefore, the store needs to be consistent. There has to be always a valid assignment for all the variables respectively to the stored constraints. To guarantee this, the store refuses tell operations trying to add information to the store, which would lead to an inconsistent store. Furthermore we are going for a transactional semantic for the actions. This means, that all tell operations occurring in an action are collected and communicated at once. If at least one operation would lead to an inconsistent store, the whole constraint conjunction is rejected by the store. We then have the possibility to choose another action, if available, or to stop the agent completely. This prevents one dysfunctional agent from disturbing all the other agents.

3 Interpretation

The front-end of the current version of the αCCFL interpreter is generated by the compiler generator tool SABLECC [2] and therefore the back-end implementation is based on the visitor design pattern.

The interpreter consists further of a type-checker, which is able to detect type errors, but does not support type inference. Though it does supports the use of type variables to define polymorphic data types and functions. Additionally the interpreter supports features like imports, higher-order functions, partial application, η-conversion, etc. and uses lazy-evaluation to evaluate functional expressions.

Agents including their personal environments are collected and executed by a scheduler (see Fig. 1). The scheduler decides in which order the agents are

Listing 2.3 A pitch-or-toss agent

```
1  data Mood = Good | Tired
2  data Coin = Heads | Tails
3  behaviour PitchOrToss {
4        points :: Int, mood :: Mood
5    } where {
6        headsOrTails :: Coin -> C;
7        ...
8        count :: Coin -> Int -> Int;
9        ...
10       change :: Mood -> C;
11       change x = #True -> x =:= Good |
12                  #True -> x =:= Tired;
13   } actions {
14       mood =:= Good ->
15         with y :: Int, mood' :: Mood, points' :: Int
16         in headsOrTails y &
17            points' =:= count y points &
18            change mood'
19         next {points <- points', mood <- mood'}
20       mood =:= Tired ->
21         with mood' :: Mood
22         in --... make a coffee break &
23            change mood'
24         next {mood <- mood'}
25       mood =:= Tired ->
26         stop
27   }
```

executed, depending on the chosen strategy. Currently, the scheduler just executes one agent after another. An agent is able to communicate with the store via an abstract class, which offers ask- and tell-methods. All agent attributes are stored within the store and therefore needed to decide which action to perform next. An action can be chosen, if the corresponding guard is fulfilled. If a multiple number of actions might be executed, then one of them is chosen non-deterministically. After an action has finished, the agent updates his attributes by propagating its store or in case of *stop*, stops itself. Note that values which are send to the store must have been evaluated eagerly (*Eager Evaluator*).

As mentioned in Sect. 2, constraints are also used to describe concurrent computations. This holds for our example, too. Therefore take a closer look at the constraints in the Lines 16 - 18. The evaluation of the constraints *headsOrTails y* and *change mood'* can be performed simultaneously. For this purpose another scheduler exists which takes care of the order of execution. Since the residuation principle is used the evaluation of the constraint in Line 17 is delayed until the free variable y is bound.

The prototypical interpreter currently offers a graphical user interface, so the user is able to define start values for the agent attributes, as well as, to pick an action or constraint alternative manually.

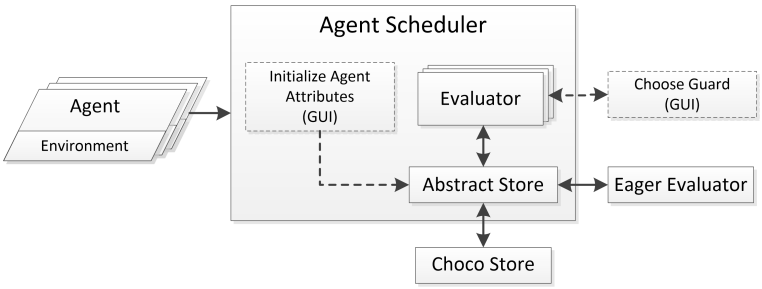

Fig. 1. Execution of the agents and communication with the store

4 Conclusion

In this paper we introduced the language αCCFL as a functional constraint-based language with actions to describe the behaviours for agents. The states of the agents, i.e. their attributes, are stored in form of constraints in a shared constraint store. Calculations are described in a functional programming manner. Non-deterministic behaviour is modeled using user-defined constraints. Time-dependent calculations are encapsuled within actions. The state is updated with the **next**-block.

Our tasks for the future comprise the enabling of the execution of multiple agents and porting the language to real agents, which have the ability to collect real world data like sensor data. The interpreter therefore must support suspension of agents, for the case that guards are currently not entailed. Another interesting point will be the use of a distributed store, which allows for a better load balance by trying to utilize every constraint sub-store evenly and, furthermore, this eases the use in distributed systems.

References

1. Choco. `http://www.emn.fr/z-info/choco-solver/`.
2. SableCC. `http://sablecc.org/`.
3. P. Hofstedt. *Multiparadigm Constraint Programming Languages*. Springer, 2011.
4. V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In D. S. Wise, editor, *POPL*, pages 333–352. ACM Press, 1991.
5. M. Wooldridge. Intelligent agents: The key concepts. In V. Marík, O. Stepánková, H. Krautwurmova, and M. Luck, editors, *Multi-Agent-Systems and Applications*, volume 2322 of *Lecture Notes in Computer Science*, pages 3–43. Springer, 2001.