Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 277–280
http://www.floc-conference.org/ICLP-home.html

DESIGN AND IMPLEMENTATION OF A CONCURRENT LOGIC PROGRAMMING LANGUAGE WITH LINEAR LOGIC CONSTRAINTS

THIERRY MARTINEZ

EPI Contraintes, INRIA Paris-Rocquencourt, BP105, 78153 Le Chesnay Cedex, France

1. Introduction

Prolog is originally rooted in logic with the elegant mapping: "programs = formulas, execution = proof search". Constraint logic programming extends Prolog to program in a richer structure than mere Herbrand terms. The underlying constraint solving engine requires either to be built-in or to add coroutines mechanisms that are not in the scope of logical reading. Implementations add other non-logical features, like assert/retract, and mutable variables, to mimic imperative programming style.

The concurrent constraint programming language enjoys logical semantics and is expressive enough to describe constraint propagators [Sar93]. Agents tell constraints as messages and are synchronised by asking whether the messages entail some constraints. The execution pursues once the guard is entailed. The suspension is therefore a transient state, captured by using linear-logic implication $\multimap$ [Fag01]. Furthermore, reading constraints as resources in linear logic allows semantics to capture the non-monotonous traits of imperative programming like mutability. LCC enjoys the mapping "programs = linear-logic formulas, execution = logical deduction": observables are the logical consequences of a program, by opposition to the logical resolution in the Prolog settings.

My thesis aims at designing a practical language as close as possible to the linear concurrent constraint (LCC) theory. The main contribution is a new operational semantics which behaves as an angelic scheduler with a tractable algorithmic complexity. This operational semantics is sound and complete with respect to the logical semantics and allows the construction of a rich language over a very simple kernel.

The second section presents the kernel, the third describes the operational semantics and the last section describes the state of the implementation work and its perspectives.

2. Kernel syntax and logical semantics

The four rules of the syntax are given below with their reading in linear logic. We recall that the ! modality introduces unlimited resources.

$$\begin{array}{lll}
[\![\texttt{forall}\ x_1 \dots x_n (x\texttt{.}p\texttt{(}x_1\texttt{,}\dots\texttt{,}x_n\texttt{)} \Rightarrow a)]\!] & = !\forall x_1 \dots x_n (p(x, x_1, \dots, x_n) \multimap [\![a]\!]) & \textit{(ask)} \\
[\![\texttt{exists}\ x\texttt{(}a\texttt{)}]\!] & = \exists x([\![a]\!]) & \textit{(hiding)} \\
[\![x\texttt{.}p\texttt{(}x_1\texttt{,}\dots\texttt{,}x_n\texttt{)}]\!] & = p(x, x_1, \dots, x_n) & \textit{(tell)} \\
[\![a\ a']\!] & = [\![a]\!] \otimes [\![a']\!] & \textit{(parallel composition)}
\end{array}$$

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.277

This syntax is a subset of the syntax of modular LCC agents [Hae07]: constraints are restricted to be single *linear tokens* (*i.e.*, linear-logic predicates without any non-logical axiom) and all asks are persistent (interpreted with the ! modality). The variable which precedes the dot in linear tokens is called the module variable. It is worth noticing that the kernel restricts all the arguments of the constraints guarding asks to be universally quantified. On the opposite, the module variable is never universally quantified.

I proved that modular LCC is as expressive as CHR as far as logical semantics and original operational semantics are concerned [Mar10]. However, CHR implementations trade completeness for committed-choice. Several refinements for controlling the scheduler in CHR have been proposed, taking into account the order of the rules in the program [Duc03], priority annotations [Kon07] or probabilities [Frü02]. None of these refinements are captured by logical semantics. The next section proposes a tractable operational semantics which is correct and complete with respect to the linear-logic reading and such that this kernel is as expressive as the whole modular LCC language.

3. Angelic operational semantics

The observable of interest is the set of all the constraints which are logical consequences of the program. This observable raises naturally in the correctness theorem of the traditional operational semantics [Fag01]. Operationally, it is the set of entailed constraints, leading to observable side-effects.

I propose the concept of derivation nets which generalises Palamidessi's SOS semantics [Bes97] for reducing scheduling non-determinism: derivations are represented as a potentially infinite multihypergraph where vertices are agents and edges are derivation steps. The derivations of the traditional LCC operational semantics correspond to interpretation as Petri-net. Strategies for reducing non-determinism are expressible as vertex sharing. There exist sharings, like the sharing of all ask instances, which allow each edge to be checked in polynomial time. Other tractable sharings should be investigated.

The angelic semantics contrasts with the usual committed-choice execution model. Concurrent languages differ by the expressive power of their guards: single channel matching for π-calculus, multiple-head matching for Join-calculus, multiple-head Prolog checking for CHR. Our CHRat [Fag08] proposition for generalising guards relies on extra-logical CHR propagations for consuming heads only once the entailment is checked. The angelic semantics overcomes this difficulty and allows the kernel to be restricted to the simplest form of guards while providing the most general expressive power: the two linear-logic formulas $\forall\vec{x}(c \otimes c' \multimap a)$ and $\forall\vec{x}(c \multimap (c' \multimap a))$ have the same constraints for consequences since the non-consumption of c' cannot be observed, and computation can be triggered after the consumption of c to check that c' is entailed.

Since the kernel forces asks to universally quantify over all the arguments appearing in guards, LCC agents of the form $x.p(v) \Rightarrow A$ (for any variables x and v and sub-agent A) should be translated in the kernel syntax. A natural translation is `forall` $v'(x.p(v') \Rightarrow$ `exists` $k(eq.\text{check}(v, v', k)\ (k.\text{true}() \Rightarrow A)))$ where the token $eq.\text{check}(v, v', k)$ refers to an agent implementing value comparison (that we suppose defined in the standard library). Since there is no observable side-effects between the consumption of $x.p(v')$ and the execution of A, angelic semantics ensures that consumptions of $x.p(v')$ for $v \neq v'$ are not observed. Therefore the proposed translation preserves the semantics. However, trying to consume all

the tokens $x.p(v')$ is inefficient compared to usual argument indexing mechanisms present in CHR implementations for example.

An illustration of the expressive power of angelism is that the indexing of linear tokens with respect to their arguments is user-implementable, as soon as tokens are indexed with respect to their module variable (which is a weaker hypothesis for the implementation since the module variable is distinguished and is never universally quantified). Suppose an agent M implementing maps (*e.g.* with hash-tables, AVL or other custom structures) associating a fresh variable x_v to each value v: for a map m, the agent M is supposed to react to the tokens $m.\text{get}(v, x)$ by unifying x with x_v. Indexing tokens $x.p(v)$ with respect to v in the map m is performed by the agent **forall** $v(x.p(v) \Rightarrow$ **exists** $x_v(m.\text{get}(v, x_v)\ x_v.\text{p}()))$. Then, agents of the form **exists** $x_v(m.\text{get}(v, x_v)\ (x_v.p() \Rightarrow A))$ have the same logical consequences as $x.p(v) \Rightarrow A$: consuming $x_v.p()$ supposes that $x.p(v)$ has been consumed, while $x.p(v)$ can still be consumed by other asks, preventing $x_v.p()$ to appear in the store.

4. The SiLCC project and perspectives

The implementation aims to build a compiler for the kernel and to reconstruct the full LCC language on top of it. A prototype has been implemented with a library for full LCC over Herbrand domain [1]. Transient asks (without the ! modality) are encoded with token consumption and complex guards are decomposed to elementary asks. *E.g.*, the ask **forall** $x(m.\text{p}(x)\ m.\text{q}(x) \to a)$ is compiled to the following kernel agent:

exists $t(t.\text{transient}()$
forall $x(m.\text{p}(x) \Rightarrow$
forall $y(m.\text{q}(y) \Rightarrow$
exists $k(eq.\text{check}(x, y, k)$
$(k.\text{true}() \Rightarrow t.\text{transient}() \Rightarrow a)))))$

where the token $t.\text{transient}()$ translates the non-persistency of the original ask. More evolved syntactic sugars have been implemented for sequentiality, conditionals, pattern-matching on Herbrand terms and records, *etc.*, so that usual programming idioms can be expressed easily on top of the mono-paradigm simple kernel.

LCC can express sequentiality and non-monotonous traits for imperative programming, closures and modules. Asks allow LCC agents to wait for some logical consequence, therefore LCC enjoys a reflexive mechanism allowing LCC agents to observe (the consequences of) their own accessible stores, since these stores are proved to be equal to the set of logical consequences by the correctness and completeness theorem of the operational semantics. However, the canonical encoding of constraint propagators as LCC agents have terminal stores for observable of interest: terminal stores cannot be reflectively observed by any agent, for the mere fact that they are terminal. We should investigate more involved encoding of constraint propagators such that relevant observables would be the accessible stores. Moreover, constraint programming involves search tree exploration: how to express search is still open. Our work on formalising search strategies as pattern-matching [Mar09] initiates a better understanding of the distinction between search trees and search heuristics in the settings of a modeling language. We still have to investigate how to encode trees and heuristics in LCC, possibly with similar control mechanism encoding as for sequentiality.

[1] The compiler together with a reasonable documentation and examples are available for download: `http://contraintes.inria.fr/~tmartine/silcc`

References

[Bes97] E. Best, F.S. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In *Proceedings of Coordination*, *Lecture Notes in Computer Science*, vol. 1282, pp. 256–273. Springer-Verlag, 1997.

[Duc03] Gregory J. Duck, Peter J. Stuckey, Marìa Garcìa de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP'03, International Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden*, pp. 79–90. ACM Press, 2003.

[Fag01] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, 2001. doi:10.1006/inco.2000.3002.

[Fag08] François Fages, Cleyton Mario de Oliveira Rodrigues, and Thierry Martinez. Modular CHR with ask and tell. In Thom Frühwirth and Tom Schrijvers (eds.), *Proceedings of the fifth Constraint Handling Rules Workshop CHR'08*. 2008.

[Frü02] Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic constraint handling rules. In *WFLP 2002, 11th International Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, vol. 76, pp. 115–130. 2002. doi:DOI:10.1016/S1571-0661(04)80789-8.

[Hae07] Rémy Haemmerlé, François Fages, and Sylvain Soliman. Closures and modules within linear logic concurrent constraint programming. In V. Arvind and Sanjiva Prasad (eds.), *Proceedings of FSTTCS 2007, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science*, vol. 4855, pp. 544–556. Springer-Verlag, 2007. doi:10.1007/978-3-540-77050-3_45.

[Kon07] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *Proceedings of PPDP'07, International Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland*, pp. 25–36. ACM Press, 2007.

[Mar09] Julien Martin, Thierry Martinez, and François Fages. On the specification of search tree heuristics by pattern-matching in a rule-based modelling language. In *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation, associated to CP'09*, pp. 73–86. 2009.

[Mar10] Thierry Martinez. Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In *Proceedings of PPDP'10, International Conference on Principles and Practice of Declarative Programming, Edinburgh, UK (to appear)*. 2010.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.