Toy(FD): Sketch of Operational Semantics

Antonio J. Fernández[1], Teresa Hortalá-González[2], and Fernando Sáenz-Pérez[2]

[1] Dept. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, Spain[⋆]
[2] Dept. Sistemas Informáticos y Programación, Univ. Complutense de Madrid
afdez@lcc.uma.es, {teresa,fernan}@sip.ucm.es

1 Introduction

In [2] we proposed the integration of finite domain (FD) constraints into the functional logic programming language TOY and, as result, presented the language TOY(FD) . We showed that TOY(FD) integrates the best features of existing functional and logic languages into FD constraint solving. This paper describes a sketch (due to space limitations) of the TOY(FD) operational semantics that consists of a novel combination of lazy evaluation and FD constraint solving.

2 Denotational Semantics

Types. We assume a countable set $\mathcal{TV}ar$ of *type variables* α, β, $\dots$ and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* $C \in TC^n$. Types $\tau \in Type$ have the syntax $\tau ::= \alpha \mid C\ \tau_1 \dots \tau_n \mid \tau \to \tau' \mid (\tau_1, \dots, \tau_n)$, $C\ \overline{\tau}_n$ abbreviates $C\ \tau_1 \dots \tau_n$, "$\to$" associates to the right, $\overline{\tau}_n \to \tau$ abbreviates $\tau_1 \to \cdots \to \tau_n \to \tau$ and $(\tau_1, \dots, \tau_n)$ denotes n-tuples. A type without any occurrence of "$\to$" is called a *datatype*. A *polymorphic signature* over TC is a triple $\Sigma = \langle TC,\ DC,\ FS\rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*. Each n-ary $c \in DC^n$ comes with a principal type declaration $c :: \overline{\tau}_n \to C\ \overline{\alpha}_k$, where $n, k \geq 0, \alpha_1, \dots, \alpha_k$ are pairwise different, τ_i are datatypes, and the set of type variables occurring in τ_i is included in $\{\alpha_1, \dots, \alpha_k\}$ for all $1 \leq i \leq n$. Every n-ary $f \in FS^n$ comes with a principal type declaration $f :: \overline{\tau}_n \to \tau$, where τ_i, τ are arbitrary types. In practice, each TOY(FD) program P has a signature which corresponds to the type declarations occurring in P. In the sequel, we always assume a given signature Σ, often not made explicit in the notation and write $\Sigma_\perp$ for the result of extending Σ with a new data constructor $\perp :: \alpha$, intended to represent an undefined value belonging to every type. As notational conventions, in the rest of the paper, we use $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$.

Patterns and Expressions. We assume a countable set $\mathcal{V}ar$ of (data) variables $X, Y, \dots$ disjoint from $\mathcal{TV}ar$ and Σ. *Partial expressions* have the syntax $e ::= \perp \mid X \mid h \mid e\ e' \mid (e_1,\ \dots,\ e_n)$ where $X \in \mathcal{V}ar$, $h \in DC \cup FS$ and

[⋆] This author has been partially supported by the projects TIC2001-2705-C03-02, and TIC2002-04498-C05-02 funded by both the Spanish Ministry of Science and Technology and FEDER.

e, e' and e_i (for $1 \leq i \leq n$) are partial expressions ($e \in \mathit{Exp}_\perp$). Expressions of the form $e\,e'$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument), while expressions $(e_1, \ldots, e_n)$ represent tuples with n components. An expression e is *non-primitive*, and we write *non–primitive*(e), iff it contains no function symbol. *Partial patterns* are built as $t ::= \perp \ \mid X \mid c\ t_1 \ldots t_l \mid f\ t_1 \ldots t_m$ where $X \in \mathcal{V}ar$, $c \in DC^k$, $0 \leq l \leq k$, $f \in FS^n$, $0 \leq m < n$ and t_i's are partial patterns ($t \in \mathit{Pat}_\perp \subset \mathit{Exp}_\perp$). Expressions and patterns without any occurrence of $\perp$ are called *total*. The sets of total expressions and patterns are denoted, respectively, by *Exp* and *Pat*.

Functions and FD constraints. Each function $f \in FS^n$ is defined by a set of conditional rules of the form $f\ t_1 \ldots t_n\ = r \ \Leftarrow \psi_1, \ldots, \psi_k$, where $(t_1 \ldots t_n)$ form a tuple of linear (i.e., with no repeated variable) *patterns*, r is an *expression* and ψ_j can be either a *joinability statement* of the form $e\ == \ e'$, or a *disequality statement* of the form $e\,/= \ e'$, with $e, e' \in \mathit{Exp}$, or a Boolean function. Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to r if all the conditions ψ_i are satisfied $(1 \leq i \leq k)$. FD constraints are defined as functions and their complete definitions were shown in [2] and are available in [3]. In this paper, $FS_{FD} \subset FS^n$ denotes the set of FD constraints that return a Boolean value.

Substitutions. A *substitution* is a mapping $\theta : \mathcal{V}ar \rightarrow \mathit{Pat}$ with a unique extension $\hat{\theta} : \mathit{Exp} \rightarrow \mathit{Exp}$, which is also denoted as θ. Let *Subst* denote the set of all substitutions and let the set of all the *partial substitutions* $\theta : \mathcal{V}ar \ \rightarrow \ \mathit{Pat}_\perp$ denote $\mathit{Subst}_\perp$, and defined analogously. We define the *domain* $\mathit{dom}(\theta)$ as the set of all variables X s.t. $\theta(X) \neq X$. By convention, we write $e\theta$ instead of $\theta(e)$, and $\theta\sigma$ for the composition of θ and σ, such that $e(\theta\sigma) = (e\theta)\sigma$ for any e.

Finite Domains. A *finite domain* (FD) is a mapping $\delta : \mathcal{V}ar \rightarrow \wp(\mathit{Integer})$, where $\wp(\mathit{Integer})$ denotes the powerset of integers. The set of all FDs is denoted as $\mathcal{FD}$. Also δ is *inconsistent* (resp.*consistent*), and write *inconsistent*(δ) (resp. *consistent*(δ)), if there exists (resp. does not exist) X such that $\delta(X) = \emptyset$.

Programs. A program defines a set of functions where each $f \in FS^n$ has an associated principal type $\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau$ (with τ not containing $\rightarrow$). As usual in functional programming, types are inferred and, optionally, can be declared in the program.

3 Operational Semantics

This section presents part (due to space limitations) of the operational semantics of TOY(FD) that deals with higher order (HO) programming by translating HO expressions into first order, and consists of a novel combination of lazy narrowing and constraint solving. $\psi_1, \ldots, \psi_n$ is a goal whose variables have an existential reading. Solving a goal means obtaining conditions (a mixture of substitutions and finite domains) over their variables to ensure the satisfiability of the initial goal.

Notational conventions. Let $e, e' \in \mathit{Exp}$; by $[e]_\mu$, $e[\mu \leftarrow e']$ and $e[\forall\mu \in I.\mu \leftarrow X_\mu]$, we respectively mean the sub-expression of e at position μ, the expression resulting from replacing $[e]_\mu$ in e by e', and the expression resulting from replacing, for each $\mu \in I$ with $I \in \wp(\mathit{Integer})$, $[e]_\mu$ in e by a fresh variable X_μ. If $e \equiv f\ e_1 \ldots e_n$ and $f \in FS^n$, $\mathit{NonPri}_e \equiv \{j \mid \mathit{non\text{–}primitive}(e_j) \wedge 1 \leq j \leq n\}$ is the set identifying the positions of all the non-primitive arguments in e.

Let $\mathcal{P}$ be a TOY(FD) program with a signature $\Sigma = \langle TC, DC, FS \rangle$. There is a natural notion of model of rules and programs, for which it can be proved that every semantically non-ambiguous TOY(FD) program $\mathcal{P}$ has a least model $\mathcal{I}_\mathcal{P}$ [4]. Then, a *solution wrt. $\mathcal{P}$ for a goal ψ* is a substitution σ such that σ satisfies ψ in $\mathcal{I}_\mathcal{P}$ ($\sigma \models_{\mathcal{I}_\mathcal{P}} \psi$). We also say that σ satisfies $\exists \overline{U} \psi$ if there is σ' which satisfies ψ and coincides with σ over $\mathit{dom}(\sigma) - U$.

In the following, by $\mid \psi \mid$, *the shell of* ψ, we denote the result of replacing in ψ all the outermost sub-expressions of the form $f\ e_1 \ldots e_n$ by $\perp$. Following the schema in [1], we say a goal ψ is *semantically finished* wrt. σ if σ is a solution of $\mid \psi \mid$ wrt. $\mathcal{P}$ and by simplicity we also write $\sigma \models_{\mathcal{I}_\mathcal{P}} \psi$. The words *semantically finished* are used to express ψ may still contain non-primitive sub-expressions but their values are irrelevant to the fact that σ is a solution of the goal.

We consider configurations $\langle e, \sigma, \delta \rangle_{C_a}$ where $e \in \mathit{Exp}$, $\sigma \in \mathit{Subst}$, $\delta \in \mathcal{FD}$ and C_a is a set of primitive FD constraints (i.e., with no function symbol in the arguments). The initial state to solve a goal ψ is $\langle \psi, \epsilon, \delta \rangle_\emptyset$ where ϵ denotes the empty substitution and $\delta(X) = \mathit{Integer}$ for any integer variable X in $\mathcal{V}ar$. Next table shows some important rules of the TOY(FD) operational semantics.

NON-SATISFACTION
$$\frac{\mathit{inconsistent}(\delta) \vee \sigma \not\models_{\mathcal{I}_\mathcal{P}} e}{\langle e, \sigma, \delta \rangle_{C_a} \mapsto \text{ termination with failure}}$$

SOLUTION
$$\frac{\mathit{consistent}(\delta) \wedge \sigma \models_{\mathcal{I}_\mathcal{P}} e}{\langle e, \sigma, \delta \rangle_{C_a} \mapsto \text{ termination with solution } \sigma}$$

ONE-STEP NARROWING
$$\frac{\begin{array}{c}[e]_\mu \equiv f\ e_1 \ldots e_n,\ f \in FS^n - FS^n_{FD},\ \sigma' \equiv \{X_1 \mapsto e_1, \ldots, X_n \mapsto e_n\},\\ f\ X_1 \ldots X_n = r \Leftarrow \psi \text{ is a variant rule for } f \text{ in } FS \text{ with fresh variables } \overline{X} \cup \overline{Y}\end{array}}{\langle e, \sigma, \delta \rangle_{C_a} \mapsto \langle e[\mu \leftarrow r\sigma'] \wedge \psi\sigma', \sigma, \delta \rangle_{C_a}}$$

FD CONSTRAINT SOLVING
$$\frac{\begin{array}{c}[e]_\mu = (g\ e_1 \ldots e_n),\ g \in FS_{FD},\ C_{FD} \equiv [e]_\mu[\forall j \in \mathit{NonPri}_{[e]_\mu}.j \leftarrow X_j]\\ {C_a}' = C_a \cup C_{FD},\ C_{FD} \rightsquigarrow_\delta^{{C_a}'} \delta'\end{array}}{\langle e, \sigma, \delta \rangle_{C_a} \mapsto \langle e[\mu \leftarrow \mathit{true}] \bigwedge \{X_j == e_j \mid j \in \mathit{NonPri}_{[e]_\mu}\}, \sigma, \delta' \rangle_{{C_a}'}}$$

The *non-satisfaction* and *solution* rules check for termination returning a failure or a solution, respectively. The *lazy computation mechanism* is based mainly in the rule *one-step narrowing* that basically rewrites a goal by taking into account the demanded positions [4].

We note that, due to space limitations, we do not provide correctness proof and also that the semantics described here is a simplification of the operational semantics of TOY(FD) (observe for example that it generates reductions that are actually not performed, because of variable sharing, and also that we do not show the rule that considers pattern matching in the function arguments).

TOY(FD) also integrates a solving mechanism for FD constraints that is mainly based in the rule *FD constraint solving* in which it is assumed the existence of a mechanism $C_{FD} \rightsquigarrow_{\delta}^{C_a} \delta'$ to define the resolution of a FD constraint C_{FD} under the initial conditions imposed by both the finite domain δ and the constraints in C_a. The resolution gives place to a new (possibly inconsistent) finite domain δ' that replaces the original δ in the transition process among configurations. Observe that only primitive constraints are sent to the FD constraint solver. This is because non-primitive constraints are first translated to primitive ones by replacing the non-primitive arguments by new fresh variables before executing constraint solving and by registering new bindings in forms of equality constraints between the non-primitive arguments and the new variables. This last step is reflected in the addition of the sub-goal $X_j == e_j$, with X_j as fresh variable, corresponding to each non-primitive argument e_j in the original constraint $[e]_\mu$. Note also that this allows for HO computations possibly to be done on the arguments $e_1, \ldots, e_n$.

Upon termination and finding a solution, the final state is $\langle \phi, \sigma, \delta \rangle$ with δ consistent and σ satisfying ϕ in $\mathcal{I}_{\mathcal{P}}$ .Termination and correctness of constraint solving is responsibility of the constraint solving mechanism $\rightsquigarrow_{\delta}^{C_a}$.

4 An Example: Imposing Infinite Lists of Constraints

TOY(FD) provides lazy evaluation (i.e., *call-by-need*) that means that the arguments (to functions) are evaluated to the required extent in contrast to *eager or strict evaluation* in which arguments are evaluated before the call (i.e., *call-by-value*). This aspect of TOY(FD) increases the possibilities of constraint solving by, for example, using infinite list of constraints. Consider the (well-known) magic series problem [6] and the following TOY (FD) functions[3]:

```
generateFD :: int -> [int]
generateFD N = [X | generateFD N] <== domain [X] 0 (N-1)

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain []  A  B  [] = true
constrain [X|Xs]  L  I  [I|S2] = true <== count I  L (#=)  X,
                                          constrain Xs  L  I+1  S2
lazymagic :: int -> [int]
lazymagic N = L <== take N (generateFD N) == L, constrain L  L  0  Cs,
              sum L (#=) N, scalar_product Cs  L (#=) N, labeling [ff] L

magicfrom :: int -> [[int]]
magicfrom N = [lazymagic N | magicfrom (N+1)]
```

The function `lazymagic/1` uses the predefined FD constraints `count/4` (via `constrain/4`), `sum/3, scalar_product/4, #=/2` and `labeling/2` and the primitive function `take:: int -> [A] -> [A]` defined such that `take N L` returns

[3] Lists follows the syntax of Prolog lists and Variables start with uppercase, whereas the remaining symbols start with lowercase.

the list with the first `N` elements of `L`. `generateFD/1` imposes an infinite list of membership constraints (i.e., `domain/3`) by generating an infinite list of variables ranging in the interval [0,N-1] for some N. The N-magic serial is calculated by lazy evaluation by solving the goal `lazymagic N`, for some natural N. However, observe that an eager evaluation would not terminate as it tries to evaluate first the second argument in `take N (generateFD N) == L` yielding to an infinite list. Also, `magicfrom/1` generates an infinite list of N-magic series from a number N, and, again by lazy evaluation, it is possible to return answers; for example, the goal `take 3 (magicfrom 7) == L` returns in L a 3-element list containing, respectively, the solution to the problems of 7, 8, and 9-magic series.

5 Conclusions

We have presented a sketch of the operational semantics of TOY(FD), a functional logic programming language with support for FD constraint solving. This semantics consists of a novel combination of laziness and constraint solving in such a way that both remain independent; the advantage is that termination and correctness of lazy evaluation is left to the functional logic language that acts as host language whereas the same properties for constraint solving are responsability of a FD constraint solver connected to the host language. The system TOY(FD) is available in [3].

Note that we focus on the integration of finite domains into a functional-logic language, a proposal quite different from the language Oz [5], which combines FD constraints and functions.

References

1. P. Arenas, A. Gil, and F. López-Fraguas. Combining lazy narrowing with disequality constraints. In *6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94)*, number 844 in LNCS, pages 385–399, Madrid, Spain, 1994. Springer-Verlag.
2. A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In P. Wadler and V. Dahl, editors, *Practical Aspects of Declarative Languages (PADL'2003)*, number 2562 in LNCS, pages 320–338, New Orleans, Louisiana, USA, 2003. Springer-Verlag.
3. A. J. Fernández, T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD): User manual, latest version. `http://www.lcc.uma.es/~afdez/cflpfd/index.html`, 2002.
4. F. López-Fraguas. *Programación funcional y lógica con restricciones.* PhD thesis, Universidad Complutense de Madrid, Departamento de Informática y Automática, Septiembre 1994.
5. G. Smolka. The Oz programming model. In J. Van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 324–343, Berlin, 1995. Springer-Verlag.
6. P. Van Hentenryck. *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA, 1989.