XSG: Fair Language with Built-in Equality

Yuri A. Klimov[1] and Anton Yu. Orlov[2] *

[1] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences
RU-125047 Moscow, Russia, yuklimov@keldysh.ru
[2] Program Systems Institute, Russian Academy of Sciences
RU-152140 Pereslavl-Zalessky, Russia, orlov@mccme.ru

Abstract. We describe the XSG programming language and define a formal semantics for it.

1 Introduction

XSG is a functional-logic untyped first-order language. Like a functional language it has functions which return results (not predicates only as classic logic languages). And like a logic language it allows implicit definition of variables' values.

XSG is developed as a model language for metacomputations. It is a successor of the TSG and NTSG languages used by S. M. Abramov and R. Glück for formal description of basic metacomputation tools such as driving, PPT and URA [1,2,3,4,5,6,7].

In XSG the concept of pattern matching is generalized by introducing equations. Both free and bound variables in equations can go both at the left and at the right sides. Also a variable can occur in an equation several times. Thus there is a notion of *equality* inherent in the language.

Every variable in XSG is a logic variable: it designates a set of possible values. The equations are global constraints on the variables. Thus there is an embedded nondeterminism in the language as the program result is an unordered set of possible answers.

Free variables may also occur in function arguments. In order to find values for such variables universal resolving algorithm (URA) [2,3,4,5,6,7] is used. URA guaranties to find *every* solution for an equation system with such implicitly defined variables in finite time (though, of course, URA itself does not always terminate). In that sense the language is *fair*: every solution will be found eventually.

2 Key Features of XSG

XSG has several particular features that can not be found in the majority of programming languages.

* Supported by Russian Foundation for Basic Research projects No. 06-01-00574-a, and No. 07-07-92100-GFEN_a, and No. 08-07-00280-a, and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

A function in XSG can have several arguments and several results (*nontrivial coarity*). As XSG is first-order untyped language, coarity is essential for avoiding dynamic checks of function results.

In the majority of existing programming languages equality is not built-in construction: for every user-defined data structure the user should provide an equality test. The user should be aware of the evaluation order while specifying equality so that it does terminate.

For example in Haskell language library function (`==`) is defined to work in the left to right order for tuples. Consequently, the following expression in Haskell does not terminate:

```
(undef, 'A') == (undef, 'B') where undef = undef
```

Some languages (see Curry [14]) provide built-in *strict equality* which is satisfied if both sides are reducible to a same ground data. The problem with strict equality is that it forces evaluation of the ground data even if it can be proven that there is not any possible one. Consider, for example, the following code in Curry:

```
undef = undef

f x = x =:= 0 & x =:= 1

Main1 = f x where x free
Main2 = f undef
```

Equational constraint (`=:=`) is evaluated as strict equality in Curry. The result of the evaluation of `Main1` function is an empty set of answers. That is, the system have proven there is no any possible values for `x`. But the evaluation of `Main2` function does not terminate.

XSG provides built-in equality which is not strict. Contrary to other programming languages, in XSG, condition $x = y$ is always immediately true if x is textually identical to y up to free variables renaming (note that x and y can contain variables bounded to function calls but not the calls themselves). Moreover, in XSG the order of evaluation guaranties that all reachable to the moment equations will be considered in finite time, so `Main2` from the Curry example above would terminate as well as `Main1`.

3 Formal Semantics of XSG

3.1 Syntax

Data domain for an XSG program is built by user-defined constructors. Each constructor has a fixed arity. Atoms are presented as nullary constructors.

XSG has a rather simple grammar (see figure 1). A program consists of a number of function definitions.

Each function has a fixed number of arguments and a fixed number of results.

Function definition contains a number of sentences. A sentence consists of left hand side and right hand side parts. Before **with** all the function results are constructed. After **with** there goes a set of conditions and terms. The order of terms and conditions is not important: they are considered as a whole.

A condition is an equality test of two expressions. As both expressions can contain free variables it is more general then the equality test and the pattern matching in traditional programming languages and corresponds to the mathematical notion of an equation. Note that function calls are presented in equations not directly but by liaison variables introduced in terms. A term is just a function call assigned to fresh liaison variables.

Free, liaison, and argument variables can repeat in one or several equations as well as in function arguments in terms.

A particular expression is a result of a function if it can be obtained from the left hand side of some sentence by applying a substitution which turns all the equations into identities. That is, all sentences are considered independently.

In each term the number of liaison variables is equal to the number of results of the corresponding function. The number of arguments in a call is equal to the number of arguments of the corresponding function.

See section 5 for examples of simple XSG programs.

Grammar

$$
\begin{array}{llll}
p \in \text{Program} & ::= \; q^{+} & k \in \text{Condition} & ::= \; (\textbf{eq?}\ e\ e) \\
q \in \text{Definition} & ::= \; (\textbf{define}\ f\ \bar{x}\ s^{*}) & t \in \text{Term} & ::= \; (\bar{x}\ := \ (\textbf{call}\ f\ \bar{e})) \\
s \in \text{Sentence} & ::= \; (\bar{e}\ \textbf{with}\ k^{*}\ t^{*}) & e \in \text{Expression} & ::= \; (\textbf{cons}\ c\ \bar{e}) \mid x \\
f \in \text{Function name} & & x \in \text{Variable} & \\
c \in \text{Constructor name} & & &
\end{array}
$$

a^{*} – set of items of type a $\qquad$ $\bar{a}$ – ordered sequence of items of type a

a^{+} – nonempty set of items of type a

Fig. 1. Abstract syntax of XSG

3.2 Natural Semantics

Natural semantics of XSG is presented in figure 2.

First two rules are usual for logical programming languages such as Prolog.

The first rule says that each sentence result can be obtained by applying a substitution to the left hand side of the sentence. The substitution assigns extended values to some free variables. The extended values can contain indefinite constructors, see “Indefinite Call” rule. The substitution must be correct: after applying it to the right hand side of the sentence all the equations must become true.

The second rule just says that one can obtain a result of a function call from any sentence from the function definition.

The third rule is the one that differentiate XSG from other logical programming languages. In essence it introduces a possibility for laziness in equality. It allows one to proceed without computing the actual value of the called function. The results of the function call are assumed to be some unique indefinite data — new indefinite constructors. Each indefinite constructor is equal to itself only.

Sentence

$$\frac{\exists\theta \quad \forall k \in k^* \quad k = (\mathbf{eq?}\ e_1\ e_2) \quad e_1/\theta = e_2/\theta \qquad \forall t \in t^* \quad t = (\bar{x}\ := \ (\mathbf{call}\ f\ \bar{e}_{arg})) \quad \vdash_\Gamma (\mathbf{call}\ f\ \bar{e}_{arg}/\theta) \Rightarrow \bar{x}/\theta}{\vdash_\Gamma (\bar{e}\ \mathbf{with}\ k^*\ t^*) \Rightarrow \bar{e}/\theta}$$

Call

$$\frac{\Gamma(f) = (\mathbf{define}\ f\ \bar{x}_{par}\ s^*) \qquad \exists s \in s^* \quad \vdash_\Gamma s/[\bar{x}_{par} \mapsto \bar{e}_{arg}] \Rightarrow \bar{e}_{res}}{\vdash_\Gamma (\mathbf{call}\ f\ \bar{e}_{arg}) \Rightarrow \bar{e}_{res}}$$

Indefinite Call

$$\frac{\bar{u} - \text{new indefinite constructors}}{\vdash_\Gamma (\mathbf{call}\ f\ \bar{e}_{arg}) \Rightarrow \bar{u}}$$

Fig. 2. Natural semantics of XSG-programs

3.3 Trace Semantics

Now let us consider the semantics of XSG from the interpreter point of view (see figure 3).

Conditions are simplified (step-by-step) by means of the *most general unification* algorithm (MGU). For a system of equations MGU returns a substitution for some variables or fails if the system is inconsistent. MGU also changes the system of equations by removing identities, so we denote the resulting system as k^*_{new}.

Another way to proceed with a sentence is to fulfil a function call. That is done by substituting the results from one of the called function sentences for liaisons. Note that a term to be reduced as well as a sentence from that term's function can be chosen arbitrarily. This is nondeterministic step and an interpreter should try all possible choices.

The "Main" rule says that a given function call can produce a particular result if there exists such a sequence of MGU- and Call-steps that leads to it.

4 Discussion

We have shown big-step and small-step semantics for the language. In order to present the possibility of comparing expressions without actually evaluating them to a ground data we have introduced indefinite constructors.

MGU

$$\frac{mgu(k^*) = (k^*_{new}, \theta)}{\vdash_\Gamma (\bar{e} \textbf{ with } k^*\ t^*) \to (\bar{e}/\theta \textbf{ with } k^*_{new}\ t^*/\theta)}$$

Call

$$\frac{\begin{array}{c}\text{for some } t \in t^* \quad t = (\bar{x} \ := \ (\textbf{call } f\ \bar{e}_{arg})) \quad \Gamma(f) = (\textbf{define } f\ \bar{x}_{par}\ s^*) \\ \text{for some } s \in s^* \quad s/[\bar{x}_{par} \mapsto \bar{e}_{arg}] = (\bar{e}_{res} \textbf{ with } k^*_1\ t^*_1) \quad \theta = [\bar{x} \mapsto \bar{e}_{res}]\end{array}}{\vdash_\Gamma (\bar{e} \textbf{ with } k^*\ t^*) \to (\bar{e}/\theta \textbf{ with } k^*/\theta\ k^*_1\ (t^* \backslash t)/\theta\ t^*_1)}$$

Main

$$\frac{\begin{array}{c}\vdash_\Gamma (\bar{x} \textbf{ with } (\bar{x} \ := \ (\textbf{call } f\ \bar{e}_{arg}))) \to^* (\bar{e}_{res} \textbf{ with } t^*) \\ \bar{e}_{res} \text{ does not contain variables from } t^*\end{array}}{\vdash_\Gamma (\textbf{call } f\ \bar{e}_{arg}) \rightsquigarrow \bar{e}_{res}}$$

Fig. 3. Trace semantics of XSG-programs

Indefinite constructors obviously can not be presented in a program answer as they are abandoned function calls. Apart from that the result for a given program evaluation by either of the presented semantics is the same. So we can formulate the following theorem.

Theorem 1. *$\vdash_\Gamma$ $(\textbf{call } f\ \bar{e}_{arg}) \Rightarrow \bar{e}_{res}$ and $\bar{e}_{res}$ does not contain indefinite constructors iff $\vdash_\Gamma$ $(\textbf{call } f\ \bar{e}_{arg}) \rightsquigarrow \bar{e}_{res}$.*

Now we have fixed the language semantics, so we can build a *perfect process tree* (PPT) for a given program [11]. The amazing fact is that the trace semantics for an XSG program coincides with the trace semantics for its perfect process tree.

In other words, PPT can be considered as the language interpreter. This proves that there exists an interpreter for the XSG language with the following remarkable property.

Theorem 2 (Fairness). *Any result for any function call that can be obtained by applying any evaluation strategy will be eventually computed by the interpreter.*

5 Examples

Due to the embedded URA it is possible to specify a function by its inverse in XSG.

For example, if we have defined addition, then subtraction definition is trivial. See figure 4 for addition and subtraction for unary numbers. The definition of *Sub* can be read as following: $x_1 - x_2$ is such number x_3 that $x_2 + x_3$ is equal to x_1. As can be seen it is precisely the algebraic definition of subtraction.

Another interesting example is shown in figure 5. Similarly to subtraction in figure 4 we define list splitting as an inverse for concatenation. Note that *Split* is different from *Sub* in two ways: 1) it returns two expressions — two parts of the

Definitions for Add and Sub

$$
\begin{array}{l}
(\textbf{define}\ \mathit{Add}\ x_1\ x_2 \\
\quad (\ x_2 \qquad\qquad\quad \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ O\ )) \qquad\qquad\qquad\qquad\qquad\qquad\quad) \\
\quad (\ (\textbf{cons}\ I\ x_3)\ \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ I\ x_1'))\ (x_3\ := \ (\textbf{call}\ \mathit{Add}\ x_1'\ x_2))\ ) \\
) \\
\\
(\textbf{define}\ \mathit{Sub}\ x_1\ x_2 \\
\quad (\ x_3\ \textbf{with}\ (\textbf{eq?}\ x_1\ x_1')\ (x_1'\ := \ (\textbf{call}\ \mathit{Add}\ x_2\ x_3))\ ) \\
)
\end{array}
$$

Fig. 4. XSG-functions for unary addition and subtraction

given list; 2) there is a lot of ways to split the list in two parts, so the function is nondeterministic.

Function *Perm* uses nondeterminism of the function *Split* to (nondeterministically) compute all permutations of the numbers from zero to its argument. It returns each permutation as a list of that unary numbers. Its definition can be read as following: 1) if the argument (x_1) is zero, then return the only possible permutation as a list of length one; 2) else find all permutations for $x_1 - 1$, split each in two parts (in all possible ways), and insert x_1 between the parts.

6 Conclusion and Future Work

We have defined formal semantics for the XSG language and have shown that it has some interesting properties which differentiate it from other programming languages.

The main obstacle for practical programming in XSG is the absence of negative restrictions. A programmer can specify positive tests (equality) only, and fails in those tests are not propagated anywhere but silently discarded. Programming without “else” is not very convenient for a lot of tasks, so adding negative restrictions to the language would be a major achievement.

XSG is developed as a model language for metacomputations simultaneously with the development of metacomputation tools for it. Another stage of development would be a supercompiler for XSG. Here arises the (hopefully, solvable) problem of splitting a configuration without changing the semantics of a program. The matter is identity equation in the original configuration can require (possibly, infinite) computation in the split one. Further issues for the supercompilation are raised by the rational XSG data (infinite periodic trees) which are not discussed in the present paper.

XSG interpreter is implemented in Haskell. All sources for the system and sample XSG programs are freely available from the web [33].

Authors would like to thank S. M. Abramov and A. S. Mishchenko who participated a lot in the development of the language.

Definitions for Concat, Split, and Perm

$$
\begin{array}{l}
(\textbf{define}\ \mathit{Concat}\ x_1\, x_2 \\
\quad (\ x_2 \qquad\qquad \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ \mathit{Nil}\ ))\qquad) \\
\quad (\ (\textbf{cons}\ \mathit{Cons}\ x_1'\, x_3)\ \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ \mathit{Cons}\ x_1'\, x_1'')) \\
\qquad\qquad\qquad\qquad (x_3\ := \ (\textbf{call}\ \mathit{Concat}\ x_1''\, x_2))\ ) \\
)
\end{array}
$$

$$
\begin{array}{l}
(\textbf{define}\ \mathit{Split}\ x_1 \\
\quad (\ x_2\, x_3\ \textbf{with}\ (\textbf{eq?}\ x_1\ x_1')\,(x_1'\ := \ (\textbf{call}\ \mathit{Concat}\ x_2\, x_3))\ ) \\
)
\end{array}
$$

$$
\begin{array}{l}
(\textbf{define}\ \mathit{Perm}\ x_1 \\
\quad (\ x_2\ \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ O\ )) \\
\qquad\qquad (\textbf{eq?}\ x_2\ (\textbf{cons}\ \mathit{Cons}\ (\textbf{cons}\ O\ )\,(\textbf{cons}\ \mathit{Nil}\ )))\ ) \\
\quad (\ x_5\ \textbf{with}\ (\textbf{eq?}\ x_1\ (\textbf{cons}\ I\ x_1')) \\
\qquad\qquad (x_2\ := \ (\textbf{call}\ \mathit{Perm}\ x_1')) \\
\qquad\qquad (x_3\, x_4\ := \ (\textbf{call}\ \mathit{Split}\ x_2)) \\
\qquad\qquad (x_5\ := \ (\textbf{call}\ \mathit{Concat}\ x_3\,(\textbf{cons}\ \mathit{Cons}\ x_1\, x_4)))\ ) \\
)
\end{array}
$$

Fig. 5. XSG-functions for list concatenation, splitting, and permutations generation

References

1. S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).
2. S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 1837, 187–212. Springer-Verlag, 2000.
3. S. M. Abramov, R. Glück. Inverse Computation and the Universal Resolving Algorithm. *Wuhan University Journal of Natural Sciences*, 6(1-2):31–45, 2001.
4. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
5. S. M. Abramov, R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Mogensen, D. Schmidt, I. H. Sudborough (eds.) *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, 269–295. Springer-Verlag, 2002.
6. S. M. Abramov, R. Glück, Yu. A. Klimov An improved universal resolving algorithm for inverse computation of non-flat languages. In A. K. Ailamazyana (ed.), *Matematika, informatika: teoriya i praktika. Sbornik trudov, posvyashennyi 10-letiyu Universiteta goroda Pereslavlya*, 11–23. Pereslavl'-Zalesskii: Izdatel'stvo "Universitet goroda Pereslavlya", 2003.
7. S. M. Abramov, R. Glück, Yu. A. Klimov An universal resolving algorithm for inverse computation of lazy languages. In I. Virbitskaite, A. Voronkov (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 4378, 27–40. Springer-Verlag, 2007.

8. E. Albert, G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
9. A. P. Ershov. On the essence of compilation. In E. Neuhold (ed.), *Formal Description of Programming Concepts*, 391–420. North-Holland, 1978.
10. R. Bird, O. d. Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
11. R. Glück, A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, A. Rauzy (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
12. R. Glück, M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, 165–181. Springer-Verlag, 1994.
13. M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. M. Hanus. Curry: an integrated functional logic language (version 0.8). Report, University of Kiel, 2003.
15. J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
16. B. Hoffmann, D. Plump. Implementing term rewriting by jungle evaluation. *Informatique Theorique et Applications/Theoretical Informatics and Applications*, 25(5):445–472, 1991.
17. N. D. Jones. The essence of program transformation by partial evaluation and driving. In N. D. Jones, M. Hagiya, M. Sato (eds.), *Logic, Language and Computation*, LNCS 792, 206–224. Springer-Verlag, 1994.
18. S. Katsumata, A. Ohori. Proof-directed de-compilation of Java bytecode. In D. Sands (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2028, 352–366. Springer-Verlag, 2001.
19. R. Kowalski. Predicate logic as programming language. In J. L. Rosenfeld (ed.), *Information Processing 74*, 569–574. North-Holland, 1974.
20. J. W. Lloyd. *Foundations of Logic Programming. Second, extended edition.* Springer-Verlag, 1987.
21. J. McCarthy. Recursive functions of symbolic expressions. *Communications of the ACM*, 3(4):184–195, 1960.
22. T. Æ. Mogensen, A. Bondorf. Logimix: a self-applicable partial evaluator for Prolog. In K.-K. Lau, T. Clement (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing. Springer-Verlag, 1993.
23. J. J. Moreno-Navarro, M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language Babel. *Journal of Logic Programming*, 12(3):191–223, 1992.
24. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.
25. A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.
26. J. P. Secher. Perfect Supercompilation. M. Sc. thesis, Department of Computer Science, University of Copenhagen, 1998.
27. J. P. Secher. Driving in the jungle. In O. Danvy, A. Filinsky (eds.), *Programs as Data Objects. Proceedings*, LNCS 2053, 198–217. Springer-Verlag, 2001.

28. J. P. Secher, M. H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, A. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 113–127. Springer-Verlag, 2000.
29. J. P. Secher, M. H. Sørensen. From checking to inference via driving and DAG grammars. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 41–51. ACM Press, 2002.
30. M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
31. V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker, J. van Leeuwen (eds.), *Automata, Languages and Programming*, LNCS 85, 645–657. Springer-Verlag, 1980.
32. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
33. XSG web resources. http://botik.ru/~xsg/