A Transformation Tool for Functional Logic Program Development*

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. We present a tool to develop functional logic programs from their specifications. Specifications of functional logic languages, i.e., contracts in the form of pre- and postconditions, are written in the same language as the final programs. Thus, contracts serve either as initial prototypical implementations or as assertions to check the expected behavior of more efficient implementations. We describe a tool that supports this software development process. Our tool can either instrument ordinary programs with run-time assertions obtained from declarative contracts or can transform declarative contracts into prototypical implementations.

1 Introduction

Ideally, software development follows a rigorous specification. In practice, the situation is quite different. Specifications often are non existent, incomplete, informal and/or unintended because:

- Specifications cannot be easily mechanically processed. Hence their adequacy and consequences are not fully understood.
- Specifications are a laborious step of the development process which may not be profitable without a significant commitment.
- Specifications could support proofs of program correctness for more reliable programs, but these proofs are difficult and seldom successful.

Some methodologies and tools to improve this situation had some success in specific areas such as embedded systems. For instance, wide-spectrum languages, such as CIP-L [4], aim at developing an efficient program from a formal specification by means of well-defined transformation steps so that a program is correct by construction. However, this approach may prevent skilled programmers from developing superior software since the code must be obtained throughout a fixed set of transformation rules.

Specifications that are written in a declarative programming language are naturally executable. This greatly extends their uses and makes them more profitable because they can be exploited in different forms:

* This work was partially supported by the DAAD grant D/08/11852.

– Executable specifications serve as initial prototypical implementations.
– Executability allows experimentation which promotes understanding and increases the confidence that a specification captures the intent.
– When an implementation directly obtained from a specification is too inefficient, the specification can be executed to check, via run-time assertions, that a more efficient implementation behaves as intended.

The possibilities sketched above become practical through tools together with a language and conventions to formulate specifications and connect them to the code they specify.

As we will demonstrate in some examples, functional logic languages [3] are quite appropriate for this purpose. The logic side, in particular nondeterminism, allows high-level specifications of operations without concern for their efficient implementation. The functional side allows implementing efficient algorithms [16]. This combination produces a wide-spectrum language [4]. The tight integration of specification and code opens wide possibilities with simple tools. In particular, we present a transformation tool for functional logic programs that uses specifications in either of two ways. Specifications without a corresponding implementation serve as prototypical implementations. Specifications with a corresponding implementation serve as run-time assertions.

The language for our presentation is the multi-paradigm declarative language Curry [13]. We assume familiarity with the general concepts of functional logic programming [3, 10, 11]. In the next section we review some notions crucial for this paper.

2 Functional Logic Programming and Curry

The multi-paradigm language Curry [13] amalgamates important features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, constraints) in a single language. Curry has a Haskell-like syntax[3] [17] extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [1] which is a conservative extension of lazy functional programming and (concurrent) logic programming.

Expressions in Curry programs contain *operations* (defined functions), *constructors* (introduced in data type declarations), and *variables* (arguments of operations or free variables). The goal of a computation is to obtain a value of some expression, where a *value* is an expression that does not contain any operation. Note that in a functional logic language expressions might have more than one value due to nondeterministically defined operations. For instance, Curry contains a *choice* operation defined by:

```
x ? _ = x
_ ? y = y
```

[3] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. Thus, we say that a constraint c is *satisfied* if it can be evaluated to `success`. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values. Furthermore, if c_1 and c_2 are constraints, c_1 `&` c_2 denotes their concurrent conjunction (i.e., both argument constraints are concurrently evaluated) and c_1 `&>` c_2 denotes their sequential conjunction (i.e., c_2 is evaluated after the successful evaluation of c_1).

With nondeterministic programming, it is sometimes useful to examine the set of all the values of some expression. A "set-of-values" operation applied to an arbitrary argument might depend on the degree of evaluation of the argument, which is difficult to grasp in a non-strict language. Hence, *set functions* [2] have been proposed to encapsulate nondeterministic computations in non-strict functional logic languages. For each defined function f, $f_{\mathcal{S}}$ denotes the corresponding set function. In order to be independent of the evaluation order, $f_{\mathcal{S}}$ encapsulates only the nondeterminism caused by evaluating f except for the nondeterminism caused by evaluating the arguments to which f is applied. For instance, consider the operation `decOrInc` defined by:

```
decOrInc x = (x-1) ? (x+1)
```

Then "$\texttt{decOrInc}_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set $\{2, 4\}$, i.e., the nondeterminism caused by `decOrInc` is encapsulated into a set. However, "$\texttt{decOrInc}_{\mathcal{S}}$ `(2?5)`" evaluates to two different sets $\{1, 3\}$ and $\{4, 6\}$ due to its nondeterministic argument, i.e., the nondeterminism caused by the argument is not encapsulated. In this paper we use set functions to check the (non) satisfiability of constraints by testing the emptiness of the result sets with the predefined operation `isEmpty`.

3 Contracts and Assertions

A specification consists of a pre- and postcondition, also called a *contract*, for an operation. Contracts have been introduced in the context of imperative and object-oriented programming languages [15] to improve the quality of software. The *precondition* is an obligation for the arguments of an operation application. The *postcondition* is an obligation for both the arguments of an operation application and the result of the operation application to those arguments. Intuitively, the application or call to each operation must satisfy its precondition, and, if the precondition is satisfied and the operation returns a result, this result must satisfy the postcondition. When a contract is checked at run-time, the pre- and postcondition are called *assertions*.

In general, contracts can specify arbitrary properties of operations. For instance, a type restriction on arguments and results can be considered as a contract which is checked at compile time in statically typed languages. However, contracts can also be used to specify the desired functionality of an operation in a precise manner. When a

pair of pre- and postcondition specifies all and only the intended arguments and the intended results of an operation, respectively, the contract is called a *specification*.

Since functional logic languages encompass logic programming principles, they are equipped to generate values satisfying the constraints of a given program. We can exploit this property to execute a specification by generating values satisfying the postcondition of an operation. Obviously, this requires that the postcondition is expressed as a functional logic program. This is not a serious restriction due to the expressiveness of functional logic languages, in particular, nondeterminism and existentially quantified variables. Thus, our tool is based on source-to-source transformations of functional logic programs.

We formalize *contracts* as follows. Let f be an operation of type $\tau_1 \to \cdots \to \tau_n \to \tau$ ($n \geq 0$). A *precondition* for f is a constraint f^{pre} of type $\tau_1 \to \cdots \to \tau_n \to$ `Success`. A *postcondition* for f is a constraint f^{post} of type $\tau_1 \to \cdots \to \tau_n \to \tau \to$ `Success`. An example follows.

Suppose that the problem is to develop an operation, `sort`, to sort a list of integers. The type of `sort` is:

```
sort :: [Int] → [Int]
```

The contract is shown below. Our tool assumes that the pre- and postcondition of an operation f are called f`'pre` and f`'post`, respectively.

```
sort'pre :: [Int] → Success
sort'pre _ = success

sort'post :: [Int] → [Int] → Success
sort'post x y = (y =:= permute x) & sorted y
```

The precondition is trivially satisfied. The postcondition states that the result of `sort` is a permutation in ascending order of its input. These concepts are easily formalized:

```
permute []     = []
permute (x:xs) = ndinsert x (permute xs)
  where
    ndinsert x ys     = x : ys
    ndinsert x (y:ys) = y : ndinsert x ys

sorted []       = success
sorted [_]      = success
sorted (x:y:ys) = (x<=y) =:= True & sorted (y:ys)
```

Assume that this code is in a module called `Sort.curry`. DSDCurry, the tool described in Sect. 4, transforms it into a new module called `SortC.curry` in which pre- and postconditions provide a first implementation of `sort`. This is triggered by the fact that `sort` has not (yet) been coded. Hence, we are already able to sort a list.

```
> dsdcurry -r Sort
...
SortC> sort [5,1,2,6,3]
Result: [1,2,3,5,6]
```

Now, for more efficient sorting, we code `sort` using the well-known quicksort algorithm. We extend module `Sort.curry` with the following definition:

```
sort :: [Int]  → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++
              sort (filter (>x) xs)
```

DSDCurry applied to the extended module behaves differently. The pre- and postcondition provide run-time assertions to check the behavior of the now implemented `sort` operation. The first test goes without a hitch.

```
> dsdcurry -r Sort
...
SortC> sort [5,1,2,6,3]
Result: [1,2,3,5,6]
```

However, a further test shows a non-intended behavior:

```
SortC> sort [5,1,2,6,5,3]
ERROR: Postcondition of operation 'sort' failed for:
[5,1,2,6,5,3]  → [1,2,3,5,6]
```

After looking at (and maybe debugging) our implementation, we spot the error: we forgot the elements equal to the pivot. Thus, we correct the last rule defining `sort`:

```
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++
              sort (filter (>=x) xs)
```

and transform and execute the resulting program again:

```
SortC> sort [5,1,2,6,5,3]
Result: [1,2,3,5,5,6]
```

Other approaches that define assertions for functional (logic) programs (e.g., [7, 8, 12]) use Boolean functions as assertions. Constraints, instead, fit more naturally into functional logic programming. In particular, they are intended to generate values whereas Boolean functions are used to check properties of given values.

As our example shows, a contract provides either an implementation of a yet to be coded operation, or an assertion for an already coded operation. The assertion checks at run-time that an operation meets its obligations. The first case is implemented in a functional logic language by applying the postcondition to an uninstantiated variable, as the following snippet shows (although the precondition is trivially satisfied and could be omitted, we keep it around to show the general scheme):

```
sort xs | sort'pre xs &> sort'post xs ys = ys
  where ys free
```

The result argument of the postcondition is declared as a free variable, `ys`, so that the evaluation of the postcondition instantiates it with values that satisfy the postcondition. The soundness and completeness of the functional logic computation ensure that all and only the intended results of `sort` are computed.

When the specification-based prototype is not efficient enough for the problem (in our sorting example because the number of permutations of a long list is too large), the program is developed as usual by the programmer. In this case, a contract is used to check whether the implementation produces the expected output. This requires to answer the following questions:

1. Which result values should be taken into account?
2. How should the assertion check be operationally treated?

For the first question, remember that operations of a functional logic language might have more than one result value. For instance, the operation `zero` (the name, somewhat misleading, is suggested by the postcondition) defined by:

```
zero = 0 ? 1
```

has `0` or `1` as results. The postcondition:

```
zero'post x = x =:= 0
```

states that the result of *any* application of `zero` should be equal to `0`. Thus, it is not sufficient that some result of `zero` satisfies the postcondition—any computed result must satisfy it. On the other hand, the postcondition might accept more values than actually computed because postconditions may state weak requirements rather than precise specifications. For instance, a postcondition of the factorial function may state that the result should be a positive number without intending that all positive numbers should be produced by the factorial function.

Altogether, we check the pre- and postconditions for a unary operation f as follows (the generalization to multiple arguments is straightforward):

1. If f is called with some argument value x, the constraint $f^{pre}\ x$ must be satisfiable.
2. If $f^{pre}\ x$ is satisfiable and $f\ x$ returns some value y, the constraint $f^{post}\ x\ y$ must be satisfiable.

Thus, assertions will be implemented by exploring the computation spaces of $f^{pre}\ x$ and $f^{post}\ x\ y$, for each call to f during run time. Whenever one of these computation spaces is finite but does not contain any solution, an assertion violation is reported (and the program terminates). As we will see, set functions are quite handy to implement this check.

A subtlety of laziness is that a function call might have arguments and/or produce results that are not fully evaluated. An approach is to evaluate the arguments of pre- and postconditions as much as needed to check an assertion. In this case, the pre- and postconditions are treated as *strict assertions*, i.e., they are evaluated when the corresponding operation is evaluated. This has the advantage that assertions are checked as soon as they could detect a violation. However, this evaluation might influence the operational behavior of a program. For instance, consider an operation "`insert n xs`" that inserts an integer `n` into a supposedly ascending list of integers `xs`. The following (weak) contract states that the input and output lists should be ordered:

```
insert'pre  _ xs = sorted xs
insert'post _ _ zs = sorted zs
```

Now consider the expression e = `head (insert 3 [1,2..])`. The standard (lazy) strategy evaluates e to `1`. If we check the assertions of `insert` in a strict manner, the program will not terminate due to the evaluation of the infinite list `[1,2..]`. Moreover, assertion violations might be reported by strict assertions, whereas the standard lazy evaluation of the program does not violate these assertions (see [12] for an example).

To prevent assertions from influencing the behavior of a program, Chitil et al. [7] proposed *lazy assertions* that do not enforce argument evaluation but are checked when the argument expression has been evaluated by the application program so far that the assertion can be evaluated without further evaluation of its argument. Thus, as long as every assertion is satisfied, program executions with or without lazy assertion checking deliver the same results.

A disadvantage of lazy assertions is that if the assertion arguments are not sufficiently evaluated the assertion itself is not evaluated and consequently a violation is not detected. For instance, "`head (insert 3 [5,1])`" returns `3` without any assertion violation if the contract is lazily checked, although the programmer could assume that the result is the minimal element of the inputs. Thus, it is debatable whether full assertion checking should be avoided in order to keep the behavior of programs [8, 12]. Lazy assertions do not modify the behavior, but a lazily computed result cannot be trusted as long as some assertion has not been checked. As a compromise between these conflicting goals, [12] proposed *enforceable assertions* (also called "faithful assertions" in [12]). These assertions behave like lazy assertions, but they can also be checked upon an explicit request of the programmer, e.g., at the end of a program run or at key intermediate execution points, e.g., before some irrevocable action (deleting a file, launching a rocket, etc) takes place.

Since there seems to be no silver bullet for assertion checking in lazy languages, our tool supports strict, lazy, and enforceable assertions so that the programmer can select the most appropriate method for a particular application.

4 The Tool

In this section, we discuss a transformation tool, DSDCurry[4], for the software development approach sketched earlier. Basically, DSDCurry takes as input a Curry module M containing contracts for some operations and produces a new Curry module MC that extends M in the following ways. MC implements operations that are undefined, but have a contract, in M. MC implements assertion checking for operations that in M both are implemented and have a contract. Trivial preconditions can be omitted, i.e., if the postcondition f`'post` is defined, but the precondition f`'pre` is missing, the trivial precondition:

```
f'pre _ = success
```

is assumed (for the sake of simplicity, we consider only unary operations in this section).

[4] The tool together with more examples is available at:
`http://www.informatik.uni-kiel.de/~pakcs/dsdcurry/`.

If a contract for an operation f is given, but the operation f is not defined or defined by the rule "f `= unknown`", which is necessary if f is referenced in the definition of other operations in M, DSDCurry inserts the following definition of f in MC:

```
f x | checkPre "f" (f'pre_S x) &> f'post x y = y
 where y free
```

DSDCurry uses the postcondition to generate values satisfying the specification and the precondition to check violations of arguments. To detect and report a failure, our tool computes the set of all the values of `(f'pre x)` using the corresponding set function $f'\mathtt{pre}_{\mathcal{S}}$. This set is passed together with the name of the operation to the generic precondition checker, `checkPre`, which is defined by:

```
checkPre fname valset =
  if isEmpty valset
  then error
         ("Precondition of operation '"++fname++"' failed!")
  else success
```

The postcondition checker, `checkPost`, is similarly defined. It is used when the contract acts as an assertion, i.e., if the module M contains a contract as well as an implementation of f, DSDCurry replaces this implementation with:

```
f x | checkPre "f" (f'pre_S x) &> checkPost "f" (f'post_S x y)
    = y
  where y = f' x
        f' ...
```

in which "f' `...`" contains the original definition of f with every occurrence of f replaced by f'. Thus, the original interface of any function is preserved by DSDCurry. The use of set functions is quite useful here to distinguish the nondeterminism originating from the evaluation of the operation (i.e., the different values of `y`) from the nondeterminism originating from the evaluation of the postcondition: only the latter should be encapsulated (by the set function $f'\mathtt{post}_{\mathcal{S}}$) to detect a possible assertion violation.

The transformation scheme presented above supports only strict assertions: if an operation f is applied to some argument, the pre- and postcondition are checked before any result is returned. To implement lazy (and also enforceable) assertions, we use a variant of the implementation of lazy and enforceable assertions in Curry proposed in [12] which we sketch in the following. The desired kind of assertions is selected in DSDCurry by a flag.

To implement lazy assertions, we need two operations for each type τ:

```
wait   :: τ → τ
ddunif :: τ → τ → τ
```

The evaluation of "`wait` x" suspends as long as the value of x is unknown (i.e., an unbound variable) and returns the value when it is known. "`ddunif` x e" is a demand-driven unification of x (which is usually a free variable) and the expression e and returns the unified value. "Demand-driven" means that the unification is performed to the degree requested by other operations that refer to the result of "`ddunif` x e". These

operations can be mechanically generated for each concrete data type by a case distinction on data constructors. For lack of space, we refer to [12] for the details.

These two type-specific operations can be encapsulated into a single type, `Assert`, defined as:

```
data Assert a = Assert (a → a) (a → a → a)
```

DSDCurry automatically generates the necessary definitions for all types used in the contracts, i.e., if τ is a type used in some contract, there is an expression $\texttt{a}\tau$ of type $\texttt{Assert}\,\tau$ encapsulating the lazy assertion operations for this type.

Based on these definitions, DSDCurry translates an implementation of operation f of type $\tau_0 \to \tau_1$ into the following definition, if the contract should be lazily checked:

```
f x = withLazyContract aτ1 aτ2 "f" f'pre_S f'post_S f' x
  where f' ...
```

Again, f' is the renamed original definition of f, and the new implementation of f calls a generic operation `withLazyContract` that decorates the evaluation with a lazy checking of pre- and postconditions as follows:

```
withLazyContract (Assert wta unifa) (Assert wtb unifb)
                 fname presetfun postsetfun fun arg =
  spawnConstraint
     (checkPre  fname (presetfun  (wta x))
      & checkPost fname (postsetfun (wta x) (wtb fx)))
     (unifb fx (fun (unifa x arg)))
 where x,fx free
```

The operation "`spawnConstraint` c e", introduced in [5] for observation debugging, evaluates the constraint c concurrently with the expression e, i.e., the declarative semantics is identical to "c `&>` e", but the suspension of c does not hamper the evaluation of e. Thus, `withLazyContract` spawns two concurrent constraints to check the pre- and postcondition. However, they are not eagerly checked, but the checking is suspended (by the "wait" operations `wta` and `wtb`) until the required values are known. This is done by the demand-driven unifications (`unifa`, `unifb`) that instantiate their first argument whenever the evaluation of the second argument is demanded. Hence, this implementation ensures that the assertion checking does not change the evaluation order of the original program. Since spawned constraints are evaluated with high priority, a violated contract is reported immediately if the argument values are available.

The implementation of enforceable contracts is similar to that of lazy contracts except for the following modification. In addition to spawning the constraints for lazy assertion checking as in `withLazyContract`, the constraints for strictly checking pre- and postconditions are registered for evaluation at some later time. These constraints are suspended until a global "control variable" becomes instantiated. Thus, the evaluation of these constraints can be enforced by instantiating the control variable. In order to avoid the double evaluation of enforced constraints that have already been lazily evaluated, one can connect both constraints with an individual flag (i.e., a free variable) that is set when the lazy assertion has been evaluated so that this flag can be checked before the corresponding constraint is enforced.

This implementation scheme uses only standard Curry features and libraries. Thus, DSDCurry was implemented as a source-level program transformation without changing the run-time system of the underlying Curry implementation.

5 Conclusions and Related Work

We have presented a tool towards the development of reliable declarative programs. Our tool uses contracts for two purposes: rapid prototyping, in which the particular features of functional logic programming allows us to compute the result of an undefined operation from its postcondition, and assertion checking, in which pre- and postconditions are evaluated at run-time to ensure that the operation is called as expected and produces expected results.

In contrast to a wide-spectrum language like CIP-L [4] that supports the development of correct programs by applying a stepwise transformation process to specifications, we support more flexibility. As a consequence, we can not ensure that the developed programs satisfy the specification. This property is only checked in each concrete program execution. Assertion checking has been proposed for many programming languages and paradigms. The use of assertions in languages with an eager evaluation strategy, like imperative, logic, or strict functional languages, is easier than in our case. For instance, [18] proposes an assertion language for (constraint) logic programming that is combined in [14] with a static verification framework. [9] considered a strict language with side effects and proposed the evaluation of assertions in parallel to the application program to exploit the power of multi-core computers.

As already discussed in Section 3, the treatment of assertions in non-strict languages is more subtle. Since eager assertion checking might influence the outcome of an execution, lazy assertions are proposed in [7] as a meaning preserving alternative. Since lazy assertions might not report some violations, Chitil and Huch [6] improved the situation by introducing "prompt" assertions that deliver more results but are still meaning preserving. Degen et al. [8] discussed the different approaches and put this into the slogan "faithfulness is better than laziness." Therefore, [12] proposed a further option: enforceable assertions are lazy but can also be eagerly checked when faith is required. Since it is not obvious which alternative is the most appropriate in a non-strict language, our tool allows the programmer to select the evaluation mode of assertions, namely strict, lazy, or enforceable.

Future work will investigate support for proving the correctness of an operation w.r.t. its contract. Proofs are very challenging for realistic programs, but in contrast to assertions they incur no run-time cost, and guarantee the behavior of a program statically rather than for a particular execution.

References

1. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
2. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

3. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
4. F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, H. Wössner, and H. Wössner, H.Wössner. Towards a wide spectrum language to support program specification and program development. *ACM SIGPLAN Notices*, 13(12):15–24, 1978.
5. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.
6. O. Chitil and F. Huch. Monadic, prompt lazy assertions in Haskell. In *Proc. APLAS 2007*, pages 38–53. Springer LNCS 4807, 2007.
7. O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL 2003)*, pages 1–19. Springer LNCS 3145, 2004.
8. M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*, pages 370; 2946–59. Springer LNI 154, 2009.
9. C. Dimoulas, R. Pucella, and M. Felleisen. Future contracts. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 195–206. ACM Press, 2009.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
12. M. Hanus. Lazy and faithful assertions for functional logic programs. In *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, pages 50–64. Universidad Politécnica de Madrid, 2010.
13. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
14. E. Mera, P. López-García, and M. Hermenegildo. Integrating software testing and run-time checking in an assertion verification framework. In *25th International Conference on Logic Programming (ICLP 2009)*, pages 281–295. Springer LNCS 5649, 2009.
15. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.
16. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
17. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
18. G. Puebla, F. Bueno, and M. Hermenegildo. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–62. Springer LNCS 1870, 2000.