Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 355-365

TYPE CHECKING AND PROBLEM OF OVERLOADED ARGUMENTS

L.Szarapka, Z.Budimac and M.Ivanović
(Novi Sad, Yugoslavia)

Abstract. The aim of this paper is to reveal, from more sides than before, the benefits which type checking brings to the efficiency of functional programs. For the first time, several strategies for dealing with the problem of overloaded arguments (of which one, the so-called LL approach is an original one) have been analyzed. The influence of these different strategies to the execution efficiency has been established. Efficiency is analyzed on two abstract machine architectures.

1. Introduction and previous work

Functional programming is one of many programming paradigms in use by computer scientists, educators, and practitioners. Pure functional programs consist solely of (mathematical) expressions and functions. On the other side, (impure) functional programs make use of the "procedural" apparatus as well (locations, statements, assignments, ...). Functional programming emerged with McCarthy's LISP and Landin's ISWIM in the late fifties. Recently, functional programming became more popular with the invention and efficient implementation of: purely functional input/output operations, purely functional arrays, and powerful purely functional programming language HASKELL [8].

Type checking of functional programs is nowadays a standard feature of functional programming languages. Type checking of (functional) programs offers two advantages:

- many potential run-time errors can be discovered at compile-time, during type checking (for example addition of a logical value and an integer),

- language implementations are potentially more efficient, because correct types are assumed at run-time and no additional checks are needed.

Type checking of functional programs was for the first time implemented in the functional programming language ML [4], where Hindley's type inference system for λ calculus was reinvented by R. Milner (and since then traditionally named as "Hindley-Milner Type System"). Hindley-Milner type system can often find the most general type for every identifier occuring in a functional program, even if the identifier's type is not declared. In this way functions of a functional program are (if possible) polymorphic.

Definition 1.1. *A function (operator) is* polymorphic *if it does not depend on the types of its arguments (operands). This means that the body (i.e. action) of the function (operator) is the same regardless of the argument types.*

Note that what is called "polymorphic" in the functional programming community is called "generic" in the object-oriented community.

However, in cases when operands of overloaded (arithmetic) operators are lifted up to the level of function arguments, Hindley-Milner type system cannot infer the most general type without additional help of the programmer. There are currently several ways to deal with this problem (further: "problem of overloaded arguments") in functional programming languages.

Definition 1.2. *A function (operator) is* overloaded *(or* "ad hoc polymorphic"*) if it performs different actions depending on the argument (operand) types.*

Note that what is called "overloaded" in the functional programming community is called "polymorphic" in the object-oriented community.

The literature on type checking of functional programs is extensive. However, there are not many papers dealing with the influence of type checking to the execution efficiency of (type-)checked programs. Furthermore, there are no papers dealing with problem of overloaded arguments and its influence to the execution efficiency. The aim of this paper is to reveal from more sides than before the benefits what type checking brings to the efficiency of functional programs. For the first time, several strategies for dealing with the problem of overloaded arguments (of which one, the so-called LL approach is an original one) have been analyzed. The influence of these different strategies to the execution efficiency has been established. Efficiency is analyzed on two abstract machine architectures.

The rest of the paper is organized as follows. Second section gives a very short and informal overview of a type checking algorithm and introduces the language LL which was used for all analyses. The third section explains the problem of overloaded arguments in more details and possible ways to override it. The fourth section presents isolated benchmark tests for every of

three solutions given in the third section. The fifth and sixth sections show performance of (abstract) SECD and SK reduction machine with respect to type checking and solution of the problem of overloaded arguments. The last section concludes the paper.

2. An overview of type checking algorithm and LL

The type checking algorithm for a functional program infers its most general type. If inference is successful, a program is correctly typed, otherwise it is not. Informally, a type checking algorithm can be explained on the following example:

Example 2.1. *The inferred type for the identifier* `len` *defined as*

```
len(x) := if x = [] then 0 else 1+len(tl(x))
```

is $[T] \rightarrow Num$, *i.e. a function which maps a list of any type into a number (where* T *represents a "type variable" and stands for any type).*

The function in the above example returns a number, because both branches of the `if`-expression are numbers: either the number `0`, or the result of the addition. An argument `x` of `len` have to be a list, because it can be equal to the empty list `[]` or must have a "tail" (in `tl(x)`.) There are no constraints concerning the type of the list elements, so they can be of any type.

For a more formal overview of the type checking (i.e. type inference) algorithm we refer to (for example) [3, 5]. To the rest of this section we introduce the language LL that was used for all analyses described in this paper.

2.1. The language

LL is intermediate functional programming language [1]. It is a LISP-like language suitable for representation of many higher-level functional programming languages and for implementation on many concrete and abstract machine architectures. By implementing a type checking algorithm on an intermediate language, the type checking became available to every language which is implemented by translation to the intermediate language.

The most important language constructs of LL are the following (in abstract syntax):

- `tuple` n $E_1 \ldots E_n$ is a data constructor which constructs an n-tuple with elements $E_1, \ldots E_n$.

- `list` $E_1 \ldots E_n$ is a data constructor which constructs a list of n elements. Note that the number of elements in a tuple is known in advance, while in the list is not. If an LL program is to be type-checked, then the types of elements in a tuple can be different, while the elements of a list must be of the same type.
- `lambda` V E has the value of a (new) function which will evaluate the expression E, when called with actual argument in place of V.
- $E_1.E_2$ is an application of an expression E_1 to an expression E_2 (i.e. a call of function E_1 with argument E_2.)
- `let` $V_1 = E_1 \; ; \ldots ; \; V_n = E_n$ `in` E has the value of E where $V_i, i = 1, \ldots, n$ are identifiers which are in E replaced with corresponding E_i. The scope of every V_i is corresponding E_i.
- `letrec` $V_1 = E_1 \; ; \ldots ; \; V_n = E_n$ `in` E has the value of E where $V_i, i = 1, \ldots, n$ are identifiers which are in E replaced with corresponding E_i. The scope of every V_i is every $E_i, i = 1, \ldots, n$ and E.
- `if` E `then` E_1 `else` E_2 has the value of E_1 if E has the logical truth value `TRUE`. Otherwise the conditional expression has the value E_2.

Besides these basic building blocks, LL contains about 50 built-in functions.

Example 2.2. *The call of* `len` *and its definition (from example 2.1) would be in LL written down as (in abstract syntax):*

```
letrec
  len = lambda x (if (= x (list)) then 0 else (+ 1 (len (tl x))))
      in
  len (list 1 2 3 4 5 6 7 8 9 0)
```

The type system (i.e. rules for infering a correct type) for LL is appropriate extension of the type system described in [3].

3. Problem of overloaded arguments

The Hindley-Milner type system sometimes cannot infer the most general type.

Example 3.1. *Consider the following definition of a function:* `f(x, y) = x+y`, *where* `+` *is overloaded operator and can accept both integers and real numbers as its operands (as well as their combination.) The possible types for*

function `f` *are thus* $(Int \times Int) \rightarrow Int$, $(Int \times Real) \rightarrow Real$, $(Real \times Int) \rightarrow$ $\rightarrow Real$, and $(Real \times Real) \rightarrow Real$.

In these cases a programmer must supply appropriate declarations to enable the type checking algorithm to proceed.

There are two general ways of dealing with this problem:

- ignore it (like in ML), or
- "forget" the difference between integer and real numbers. This can be done:
  - conceptually (or statically, at compile-time), by building a mechanism into the compiler which will accept any number but will later separate them, and generate appropriate code according to types (like in HASKELL, through a mechanism of *type classes*), or
  - truly, (or dynamically, at run-time), by having only one data type (for example *Num*) and generating the code which will always first check the number type(s) and after that apply appropriate action. If so, then it is the case that numbers are represented internally either as integers or real numbers and coercion between them is allowed only:
    * from integer to real numbers (like in MIRANDA [11] (trademark of Research Software Ltd.), or
    * from integer to real numbers, and vice versa, like in LL [1].

It can be noted from the above observations that there is no simple solution to the problem of overloaded arguments - there is always someone in the chain from programming to program execution which has to "pay the price" for a solution to the problem. To summarize,

- from a programmer's point of view, an approach taken in HASKELL, MIRANDA and LL is better than the approach in ML, because the programmer need not worry about number types,
- from a compiler's point of view, the ML's, MIRANDA's and LL's approach is better than HASKELL's approach, because it need not be extended to support compile-time analysis of number types,
- from an "executor's" point of view the ML's and HASKELL's approach is better than MIRANDA's and LL's, because there is no need to check internally whether the number is a real one or an integer.

(In the preceding items we informally introduced a relation "is better" with a meaning "to do less").

Since the main concern of this paper is the influence of type checking to an *execution efficiency* of polymorphic functional programs, the cost of type checking at *compile-time* will be in the rest of the paper neglected. It must be said, however, that extensions to a (HASKELL) compiler to support type

classes are relatively complex and time consuming (see for example [10], [1] p. 69.)

4. Three approaches to number representation

We observe three possible solutions to the problem of overloaded arguments which can affect the *execution* efficiency of polymorphic functional programs:

1. HASKELL's (or ML's) approach, where real numbers and integers are separate types and have distinct internal representation, with forbidden coercions,
2. MIRANDA's approach, where there is only one numeric type (i.e. **Num**,) distinct internal representation, and coercions are allowed from integer to real numbers,
3. LL approach, which is similar to MIRANDA approach, but coercions from real numbers to integers are also allowed (always when the "internal representation" cannot "see" the difference between integer and real representation of the same number).

Note that MIRANDA and LL approaches to internal number representations and coercions are not only interesting as possible solutions to the problem of overloaded arguments. These two approaches also enhance the data abstraction of functional programming languages by hiding the differences between internal number representations from a programmer (LL being "more declarative" than MIRANDA) [2].

We implemented all three approaches in LL. Every of three approaches is in two versions - with and without type checking at run-time. Performances of all six implementations of addition and multiplication are displayed in the following tables. Data is obtained by measuring the time needed for 30,000 operations of four possible operand types (on a 12 MHz 286 machine.)

Performances of integer addition are given as procentual delays with respect to the fastest implementation of 0.1 ms (denoted as 0%.) Similarly, performances of floating point addition, integer multiplication and floating point multiplication are given with respect to the most efficient implementation of corresponding operations (0.38 ms, 0.11 ms, and 0.42 ms, respectively). Integer operations are separated from floating point operations with a line in the tables. Note that a floating point operation is used whenever at least one operand is a real value. Results of measurements with assumed correct

types are denoted as 'TC'. As 'No TC' are denoted results where types are not checked previously (and are checked at run-time).

Add	HASKELL		MIRANDA		LL	
	TC	No TC	TC	No TC	TC	No TC
Int × Int	0%	27.03%	35.83%	35.83%	35.83%	35.83%
Real × Int	-	-	5.77%	5.77%	5.77%	5.77%
Int × Real	-	-	11.72%	11.72%	11.72%	11.72%
Real × Real	0%	7.28%	14.56%	21.49%	144.32%	151.15%

Mul	HASKELL		MIRANDA		LL	
	TC	No TC	TC	No TC	TC	No TC
Int × Int	0%	21.49%	48.96%	48.96%	48.96%	48.96%
Real × Int	-	-	3.01%	3.01%	3.01%	3.01%
Int × Real	-	-	8.71%	8.71%	8.71%	8.71%
Real × Real	0%	6.49%	13.54%	19.56%	129.61%	136.58%

Performances of subtraction and division are comparable with the given ones for addition and multiplication, respectively.

There are several obvious conclusions drawn from the displayed data: typed implementations are (somewhat unexpectedly) only sometimes more efficient than untyped ones, because type checking of operands cannot be completely avoided. LL approach is the least efficient one, while HASKELL approach is the most efficient of all three. MIRANDA's approach is more efficient than LL's only in the cases when both operands are real values (when MIRANDA does not examine the possibility of coercion into an integer). These results obtained by *isolated* measurements can be similar only theoretically with performances of "real" programs. However, exact relationship between these results and performances of programs cannot be easily established. It depends on at least two conditions:

- the number of numeric operations in a program, and
- the number of operations with real numbers which give an integer as a result (because integer operations on coerced values in LL are executed much more efficiently than floating ones on uncoerced values in MIRANDA).

In the following two sections, we examine the influence of type checking and the three approaches to number representation, to an execution efficiency of "real" (numeric and non-numeric) functional programs.

5. Efficiency of SECD machine

SECD machine was the first abstract machine invented for implementation of functional programming languages [9, 6]. It implements a "strict" semantics of functional programming languages, i.e. an "eager" evaluation. SECD machine is still a good basis for many other (and more modern) abstract machines (G machine, for example) and excellent test bed for "eager" implementations of functional programming languages.

The execution efficiency of 10 benchmark programs was measured on the SECD simulator. Benchmark programs were divided into three groups: programs involving integer arithmetics (Fibonacci numbers on two different ways, Takeuchi function and integer matrix operations), programs involving real arithmetic (computing of π, real matrix operation and Takeuchi function with real numbers), and programs involving symbolic computation and no numerical operations (queens on a chess board and two manipulations with "lazy" lists).

The results are displayed in the following two tables. The first table presents a speedup of type checked programs with respect to programs that are not typed checked (hence type checking has to be done at run-time) for all three described approaches to number representations. The second table presents the speedup of (checked and unchecked) MIRANDA's and HASKELL's approach to number representation with respect to (checked and unchecked) LL's approach.

Speedup: type checked vs. unchecked programs			
Programming language	Integer Arithmetics	Real Arithmetics	Symbolic Computation
LL	41.45%	32.14%	42.57%
MIRANDA	40.93%	34.15%	41.92%
HASKELL	40.93%	34.46%	41.92%

Speedup: with respect to LL (checked or unchecked)			
Programming language	Integer Arithmetics	Real Arithmetics	Symbolic Computation
MIRANDA (checked)	0.01%	0.08%	0%
MIRANDA (unchecked)	0.01%	0.11%	0%
HASKELL (checked)	0.01%	0.08%	0%
HASKELL (unchecked)	0.01%	0.12%	0%

As expected, if type checking is done at compile-time (and thus avoided at run-time), execution efficiency is significantly improved. Note that the results

of symbolic computation show an improvement which is as much as possible independent of the chosen approach to number representation (and solution of a problem of overloaded arguments). The smaller speedup of programs with integer operations in the case of MIRANDA's and HASKELL's approaches can be explained by the following law: "the faster the execution, the smaller gain by optimization". Indeed, as the second table shows, LL's approach is generally slower with integer operations than MIRANDA's and HASKELL's approach.

As expected, there is no difference between LL's and HASKELL's approach in symbolic computations. An interesting result, however, is that programs with real number operations are only slightly more efficient than corresponding LL equivalents!

6. Efficiency of SK reduction machine

SK reduction machine was a first abstract machine to implement nonstrict semantics of functional programming languages, i.e. "lazy" evaluation [7]. SK is nowadays in widespread use and is a basis for similar machines.

The execution efficiency of the same 10 benchmark programs was measured on the SK machine simulator. The results are presented in tables analogous to the tables in previous section.

Speedup: type checked vs. unchecked programs			
Programming language	Integer Arithmetics	Real Arithmetics	Symbolic Computation
LL	24.85%	20.12%	24.95%
MIRANDA	24.46%	22.46%	25.01%
HASKELL	24.47%	33.03%	25.01%

Speedup: with respect to LL (CHECKED OR UNCHECKED)			
Programming language	Integer Arithmetics	Real Arithmetics	Symbolic Computation
MIRANDA (checked)	0%	0.89%	0%
MIRANDA (unchecked)	0%	1.12%	0%
HASKELL (checked)	0%	0.89%	0%
HASKELL (unchecked)	0%	1.12%	0%

The speedup of the SK reduction machine is smaller to the performance

of the SECD machine. Generally, it is another "proof" of the law from previous chapter. More specifically, it can be explained by more rigid internal representation of a functional program in SK machine than in SECD machine (most of the time, an element to be fetched is surely a list and therefore need not be type checked).

6. Conclusion

As shown (and expected), the gains of type checking at compile time are worth implementing the algorithm as a part of a compiler on both architectures: SECD machine (for "eager" evaluation) and SK machine (for "lazy" evaluation).

The main conclusion of the investigation presented in this paper is, however, that MIRANDA's and LL's approaches to solving the problem of overloaded arguments, do not worsen performances significantly even in the case of heavy numeric operations. In most cases it can be a successful replacement for more sophisticated (and more complicated) HASKELL's concept and implementation of type classes.

References

[1] **Budimac Z.,** *A contribution to the theory of functional programming languages and to an implementation of their processors*, PhD Thesis, University of Novi Sad, Faculty of Science, Novi Sad, 1994.

[2] **Budimac Z.,** Abstracting number representation, *Proc. of 39. Conf. of ETRAN, Zlatibor, Yugoslavia, 1995*, 235-237.

[3] **Damas L. and Milner R.,** Principal type schemes for functional programs, *Proc. of IX. ACM Symposium on Principles of Programming Languages, Albuquerque, USA, 1982*, 207-212.

[4] **Gordon M., Milner R., Morris L., Newey M. and Wadsworth C.,** A metalanguage for interactive proof in LCF, *Proc. of V. Annual ACM Symposium on Principles of Programming Languages, 1970,* 119-130.

[5] **Hancock P.,** Polymorphic type checking, *Peyton Jones, S., The implementation of functional programming languages*, Prentice Hall, London, 1987, 139-162.

[6] **Henderson P.,** *Functional programming - Application and implementation*, Prentice Hall, New York, 1980.

[7] **Turner D.A.,** A new implementation technique for applicative languages, *Software - Practice and Experience,* **9** (1979), 31-49.

[8] **Hudak P., Peyton-Jones S. and Wadler P. (eds.),** Report on the programming language HASKELL - a non-strict purely functional language Version 1.2., *SIGPLAN Notices*, Num. 5 (1992), R.1-R.164.

[9] **Landin P.J.,** The mechanical evaluation of expressions, *Computer Journal,* **6** (4) (1964), 308-320.

[10] **Peterson J. and Jones M.,** Implementing type classes, *Proc. of ACM Conf. on Programming Languages and Design, Albuquerque, USA, 1993,* 227-236.

[11] **Turner D.A.,** *MIRANDA System Manual, MIRANDA Version 2*, Research Software Ltd., 1989.

L. Szarapka

University of Novi Sad
Faculty of Civil Engineering
Dept. of Mathematics and Informatics
Kozaračka 2/a
21000 Novi Sad, Yugoslavia
ilehel@unsim.ns.ac.yu

Z. Budimac
M. Ivanović
University of Novi Sad
Faculty of Science
Institute of Mathematics
Trg D. Obradovića 4
21000 Novi Sad, Yugoslavia
[zjb,mira]@unsim.ns.ac.yu