The Generalized Intensional Transformation for Implementing Lazy Functional Languages[⋆]

Georgios Fourtounis[1], Nikolaos Papaspyrou[1], and Panos Rondogiannis[2]

[1] School of Electrical and Computer Engineering
National Technical University of Athens, Greece
[2] Department of Informatics and Telecommunications
University of Athens, Greece

Abstract. The intensional transformation is a promising technique for implementing lazy functional languages based on a demand-driven execution model. Despite its theoretical elegance and its simple and efficient execution model, the intensional transformation suffered, until now, from two main drawbacks: it could only be applied to programs that manipulate primitive data-types and it could only compile a simple (and rather restricted) class of higher-order functions. In this paper we remedy the above two deficiencies, obtaining a transformation algorithm that is applicable to mainstream lazy functional languages. The proposed transformation initially uses defunctionalization in order to eliminate higher-order functions from the source program. The original intensional transformation is then extended in order to apply to the target first-order language with user-defined data types that resulted from the defunctionalization. It is demonstrated that the proposed technique can be used to compile a relatively large subset of Haskell into portable C code whose performance is comparable to existing mainstream implementations.

Keywords: intensional transformation, dataflow programming, defunctionalization, compilation, lazy functional languages

1 Introduction

The *intensional transformation* [20, 16, 17] has been proposed as an alternative technique for implementing lazy functional languages based on a demand-driven execution model. The key idea behind the intensional approach is to transform a source functional program into a program consisting of nullary variable definitions enriched with intensional (i.e., context-switching) operators. The transformation was initially proposed as a technique for implementing first-order functional languages [20] and was also used in the implementation of the first-order

⋆ Work partially supported by the research project "ΘΑΛΗΣ−ΕΜΠ: Handling uncertainty in data intensive applications on a distributed computing environment (cloud computing)" (MIS 380153), funded by the European Social Fund and the Greek national funds through the Operational Program "Education and Lifelong Learning".

dataflow language Lucid [19]. Later on, the correctness of the transformation was formally established [16] and it was extended to apply to a simple class of higher-order programs [17], in which partially applied objects can only be top-level function names. For the class of programs that it can compile, the transformation has been demonstrated to be quite efficient [4].

Despite its theoretical elegance and its simple and efficient execution model, the intensional transformation continues to suffer from the two main drawbacks that were present since its inception:

- It can only be applied to programs with primitive data-types (such as integers, characters, boolean values, and so on). For example, the dataflow language Lucid never supported user-defined data-types [19, Sec. 7.1].
- It can only compile a simple (and rather restricted) class of higher-order functions. More specifically, the extension of the intensional transformation [17] can only compile programs that make a Pascal-like use of higher-order functions (i.e., programs that do not use function closures and therefore do not support currying in its full-generality).

In this paper we remedy the above two deficiencies, obtaining a transformation algorithm that is applicable to mainstream higher-order lazy functional languages. The proposed transformation initially uses defunctionalization [15] in order to eliminate higher-order functions from the source program (at the cost of introducing data constructors representing explicit closures in the target first-order program). In this way, the two problems above are trivially reduced to the first one. The first problem is then solved by demonstrating that the original intensional transformation can be appropriately extended to handle a language with user-defined data types (and pattern matching). This problem is solved in this paper, which extends an idea that was presented last year in an informal symposium [6]. It is also demonstrated that the proposed technique can be used to compile a relatively large subset of Haskell into portable C code whose performance is comparable to existing Haskell implementations, based on more traditional compilation techniques.

The rest of the paper is organized as follows: Section 2 provides background on the original intensional transformation and introduces the proposed generalized transformation at an intuitive level, whereas Section 3 presents a formalization thereof. Section 4 discusses the details of an implementation of the proposed technique. Section 5 provides a performance comparison with several well-known and efficient Haskell compilers. The paper concludes (Sections 6 and 7) with a discussion of related work and directions for future research.

2 From the Original to the Generalized Transformation

In this section we introduce the intensional transformation in an intuitive way. We start by outlining the original transformation (for an extensive discussion, see [20, 16]) and proceed by sketching our new approach with a simple example.

2.1 The Original Intensional Transformation

The input to the original intensional transformation [20, 16] is a first-order functional program that only uses base data-types (such as integers, Boolean values, and so on). We assume that all the variables in the program (i.e., function names and their formal parameters) are distinct; this can obviously be achieved by a straightforward preprocessing. The source program is then transformed into a zero-order *intensional* program that only contains nullary definitions. The name "intensional" reflects the fact that the resulting program additionally uses two context-switching operators, whose semantics will be shortly described. The transformation can be intuitively described as follows [16]:

1. Let `f` be a function defined in the source functional program. Number the textual occurrences of calls to `f` in the program, starting at 0 (including calls in the body of the definition of `f`).
2. Replace the i-th call of `f` in the program by $\texttt{call}_i\texttt{(f)}$. Remove the formal parameters from the definition of `f`, so that `f` is defined as an ordinary individual variable.
3. Introduce a new definition for each formal parameter of `f`. The right hand side of the definition is the operator `actuals` applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered.

To illustrate the algorithm, consider the following simple first-order program on the left. The transformation produces the target program on the right:

```
result = f 3 + f 5            result = call0(f) + call1(f)
f x    = g (x*x)              f      = call0(g)
g y    = y+2                  g      = y+2
                              x      = actuals(3, 5)
                              y      = actuals(x*x)
```

The above intensional code can be easily evaluated with respect to an initially empty *context*. Evaluation contexts are in fact lists of natural numbers which, intuitively, keep track of the exact position in the recursion tree where the execution currently is. The operators $\texttt{call}_i$ and `actuals` are context-switching operators: $\texttt{call}_i$ augments a list w by prefixing it with i, whereas `actuals` takes the head i of a list, and uses it to select its i-th argument. One can now easily define an *EVAL* function which evaluates the intensional program that results from the transformation, as shown in Figure 1. The function is parameterized by the program p in which all evaluation takes place; this will often be omitted to simplify presentation. The function $body(v, p)$ returns the defining expression of a variable v in program p. The evaluation of the usual constructs of functional languages (if-then-else, arithmetic operations, etc.) are all expressed by the rule for n-ary constants `c` (which, when $n = 0$ also covers the case of nullary constants, such as numbers, characters, and so on). Notice that the order of evaluation in this case depends on the meaning of the constant `c`: if `c` is

$$\begin{array}{lcl}
EVAL_p(v, w) & = & EVAL_p(body(v, p), w) \\
EVAL_p(\texttt{call}_i(e), w) & = & EVAL_p(e, i : w) \\
EVAL_p(\texttt{actuals}(e_0, \ldots, e_{n-1}), i : w) & = & EVAL_p(e_i, w) \\
EVAL_p(\texttt{c}(e_0, \ldots, e_{n-1}), w) & = & c(EVAL_p(e_0, w), \ldots, EVAL_p(e_{n-1}, w))
\end{array}$$

Fig. 1. The *EVAL* function for the intensional language.

$$\begin{array}{l}
\quad EVAL(\texttt{result}, [\,]) \\
= EVAL(\texttt{call}_0\texttt{(f) + call}_1\texttt{(f)}, [\,]) \\
= EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) \\
= EVAL(\texttt{g}, [0, 0]) + EVAL(\texttt{g}, [0, 1]) \\
= EVAL(\texttt{y}, [0, 0]) + EVAL(\texttt{2}, [0, 0]) + EVAL(\texttt{y}, [0, 1]) + EVAL(\texttt{2}, [0, 1]) \\
= EVAL(\texttt{actuals(x*x)}, [0, 0]) + 2 + EVAL(\texttt{actuals(x*x)}, [0, 1]) + 2 \\
= EVAL(\texttt{x*x}, [0]) + 2 + EVAL(\texttt{x*x}, [1]) + 2 \\
= EVAL(\texttt{x}, [0]) * EVAL(\texttt{x}, [0]) + 2 + EVAL(\texttt{x}, [1]) * EVAL(\texttt{x}, [1]) + 2 \\
= EVAL(\texttt{actuals(3, 5)}, [0]) * EVAL(\texttt{actuals(3, 5)}, [0]) + 2 + \\
\quad EVAL(\texttt{actuals(3, 5)}, [1]) * EVAL(\texttt{actuals(3, 5)}, [1]) + 2 \\
= EVAL(\texttt{3}, [\,]) * EVAL(\texttt{3}, [\,]) + 2 + EVAL(\texttt{5}, [\,]) * EVAL(\texttt{5}, [\,]) + 2 \\
= 9 + 2 + 25 + 2 \;=\; 38
\end{array}$$

Fig. 2. Execution of the target intensional program.

an arithmetic operator (e.g., "+") then the recursive calls to *EVAL* will have to be computed strictly; if on the other hand `c` corresponds to a non-strict operator (e.g., if-then-else), then evaluation is dictated by the meaning of this operator.

The execution of our example intensional program derived above is given in Figure 2. Notice that we assume that all source programs have a distinguished variable `result` whose value we want to compute.

The evaluation function just described roughly corresponds to call-by-name: notice how `x` is evaluated again and again under the same context. To obtain a call-by-need implementation, one can use an appropriate warehouse, in which triples of the form (*variable*, *context*, *value*) are stored — see [16, Sec. 12] for a more extensive discussion on the history and details of this issue. Every time the value of a variable under a given context is demanded, the warehouse is searched. If an entry is found, the corresponding value is returned; otherwise, the value of the variable under the current context is computed and placed in the warehouse for possible future reuse. A more efficient way of memoizing results, using *lazy activation records* (LARs), has been proposed in [4]; the idea of LARs is generalized and used in Section 4.

2.2 The New Intensional Transformation

As mentioned in the introductory section, the intensional transformation was never generalized to apply to a fully higher-order functional language nor to a language that supports user-defined data-structures. From an implementation point of view, higher-order functions and data-structures are closely connected, since, using Reynold's defunctionalization, one can reduce a higher-order program to a first-order one that is enriched with appropriate data-structures [15]. In other words, the two problems can be simultaneously solved if we generalize the intensional transformation to apply to first-order programs with user-defined data types. For example, consider the following second-order Haskell program:

```
result  = inc (add 1) 2 + inc sq 3
inc f x = f (x+1)
add a b = a+b
sq z    = z*z
```

The source program is initially defunctionalized as shown below:

```
result     = inc (fadd 1) 2 + inc fsq 3
inc f x    = apply f (x+1)
add a b    = a+b
sq z       = z*z

data Func  = Fadd Int | Fsq
fadd c     = Fadd c
fsq        = Fsq

apply cl d = case cl of
                 Fadd c → add c d
                 Fsq    → sq d
```

The above is a standard defunctionalization with two small tricks. First, we have introduced functions `fadd` and `fsq` which have replaced all occurrences of the constructors `Fadd` and `Fsq`. Second, in the `case` pattern corresponding to `Fadd`, we have used the same variable `c` that appears in the definition of `fadd`. These two conventions (to be discussed more generally in Section 3) ensure that we can apply the intensional transformation and obtain an equivalent zero-order intensional program, exactly as we did before:

```
result = call0(inc) + call1(inc)
inc    = call0(apply)
add    = a+b
sq     = z*z

fadd   = Fadd
fsq    = Fsq

apply  = case cl of
           Fadd → call0(add)
           Fsq  → call0(sq)
```

```
f       = actuals(call_0(fadd), fsq)
x       = actuals(2, 3)
a       = actuals(c)
b       = actuals(d)
z       = actuals(d)
c       = actuals(1)
cl      = actuals(f)
d       = actuals(x+1)
```

The above program can be executed following the same basic principles as the one presented in the previous subsection, using a demand-driven interpreter in the form of a function $EVAL_p(e, w)$ that will be defined formally in Section 3.

3 A Formal Account of The Generalized Transformation

In this section we present the generalized transformation in a more formal way. Since defunctionalization is a well-known and broadly used technique, in the following we will not discuss it any further. Instead, from now on we will assume that our source language is a lazy first-order functional language with user defined data-types (i.e., a language whose syntax matches the syntax of the programs that are produced by defunctionalization). We will call this language FOFL (First-Order Functional Language).

The syntax of FOFL is defined by the following context-free grammar, where f and v range over variables, c ranges over constants, κ ranges over constructors, and $n, m \geq 0$. When $n = 0$, we will omit the empty parentheses.

$$
\begin{array}{rcll}
p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\
d & ::= & f(v_0,\ \ldots,\ v_{n-1}) = e & \textit{definition} \\
e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f(e_0,\ \ldots,\ e_{n-1}) \mid \kappa(e_0,\ \ldots,\ e_{n-1}) & \textit{expression} \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ \texttt{;}\ \ldots\ \texttt{;}\ b_n\ \} \mid \texttt{\#}^m(v) & \\
b & ::= & \kappa(v_0,\ \ldots,\ v_{n-1}) \rightarrow e & \textit{case clause}
\end{array}
$$

As outlined in the previous section, we assume that FOFL programs are in a *normalized form*. We assume that the formal parameters of all functions are distinct. This can be achieved by simple renaming. Furthermore, for each constructor κ with n arguments, there will be a function defined as:

$$f_\kappa(v_0, \ldots, v_{n-1}) = \kappa(v_0, \ldots, v_{n-1})$$

and all occurrences of κ in the program will be replaced by occurrences of f_κ. We also assume that patterns corresponding to κ in all `case` expressions will use the same variables $v_0, \ldots, v_{n-1}$ that appear in the definition of f_κ. Unfortunately, this cannot be achieved by simple renaming, as there may be nested `case` expressions. For this reason, we introduce a special form of expressions $\texttt{\#}^m(v)$ that will resolve such scoping issues.

Roughly speaking, $\mathtt{\#}^m(v)$ corresponds to the variable v that is bound in a pattern of the m-th enclosing `case` expression. For example, function `apply` in the example of the previous section will be written as:

$$
\begin{array}{lcl}
\mathit{apply}(cl, d) & = & \mathtt{case}\ cl\ \mathtt{of}\ \{ \\
 & & \quad \mathit{Add}(c) \to \mathit{add}(\mathtt{\#}^0(c), d); \\
 & & \quad \mathit{Sq} \to \mathit{sq}(d) \\
 & & \}
\end{array}
$$

where $\mathtt{\#}^0(c)$ corresponds to the variable c bound by the pattern $\mathit{Add}(c)$ of the `case` expression. An example with nested `case` follows, where the expression on the left (in Haskell syntax, calculating the sum of the first two elements of a list) can be normalized as shown on the right:

```
case l of
  Nil → 0
  Cons x xs →
    case xs of
      Nil → x
      Cons y ys → x+y
```

$$
\begin{array}{l}
\mathtt{case}\ l\ \mathtt{of}\ \{ \\
\quad \mathit{Nil} \to 0; \\
\quad \mathit{Cons}(h, t) \to \\
\quad\quad \mathtt{case}\ \mathtt{\#}^0(t)\ \mathtt{of}\ \{ \\
\quad\quad\quad \mathit{Nil} \to \mathtt{\#}^1(h); \\
\quad\quad\quad \mathit{Cons}(h, t) \to +(\mathtt{\#}^1(h), \mathtt{\#}^0(h)) \\
\quad\quad \} \\
\}
\end{array}
$$

Notice here that the same set of variables (h, t) is used in both patterns for *Cons* and that `x` and `y`, which both correspond to h, are distinguished by the value of m (the nesting depth of `case` expressions).

3.1 The Generalized NVIL

FOFL programs are transformed into zero-order intensional ones in the language NVIL (Nullary Variables Intensional Language). For more background on such languages, the interested reader can consult the first sections of [16]. The only difference of NVIL from the corresponding language defined in [16] is that the former supports user-defined data types. The syntax of NVIL is given by the following context-free grammar. Notice that the syntax of the intensional operators (`call` and `actuals`) is slightly different from the one informally introduced in Section 2 and that $\mathtt{\#}^m(v)$ has been replaced by the more general $\mathtt{\#}^m(e)$.

$$
\begin{array}{lcll}
p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\
d & ::= & f = e & \textit{definition} \\
e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f \mid \kappa \mid \mathtt{case}\ e\ \mathtt{of}\ \{\ b_0\ ;\ldots\ ;\ b_n\ \} & \textit{expression} \\
 & \mid & \mathtt{\#}^m(e) \mid \mathtt{call}_\ell(e) \mid \mathtt{actuals}(\langle e_\ell \rangle_{\ell \in I}) & \\
b & ::= & \kappa \to e & \textit{case clause}
\end{array}
$$

In Section 2, operator `call` was labeled by a natural number i and operator `actuals` received a sequence of expressions, indexed by i. Here, we slightly

$$
\begin{array}{l}
\mathit{EVAL}_p(c(e_0,\ \ldots,\ e_{n-1}), w) \ \ = \ \ c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w)) \\
\mathit{EVAL}_p(f, w) \ \ = \ \ \mathit{EVAL}_p(\mathit{body}(f, p), w) \\
\mathit{EVAL}_p(\kappa, w) \ \ = \ \ \langle \kappa, w \rangle \\
\mathit{EVAL}_p(\texttt{case}\ e\ \texttt{of}\ \{\kappa_0 \to e_0;\ \ldots;\ \kappa_n \to e_n\}, \langle \ell, w, \mu \rangle) \ \ = \ \ \mathit{EVAL}_p(e_i, \langle \ell, w, w' : \mu \rangle) \\
\qquad \text{if}\ \ \mathit{EVAL}_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle \\
\mathit{EVAL}_p(\texttt{\#}^m(e), \langle \ell, w, \mu \rangle) \ \ = \ \ \mathit{EVAL}_p(e, \mu_m) \\
\mathit{EVAL}_p(\texttt{call}_\ell(e), w) \ \ = \ \ \mathit{EVAL}_p(e, \langle \ell, w, \bullet \rangle) \\
\mathit{EVAL}_p(\texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) \ \ = \ \ \mathit{EVAL}_p(e_\ell, w)
\end{array}
$$

Fig. 3. Semantics of NVIL.

change this and take the index to be any element ℓ from an appropriate set *Labels*. Therefore, `call` is labeled by ℓ and `actuals` receives a sequence of expressions e_ℓ indexed by labels ranging over a subset $I \subseteq \mathit{Labels}$. We represent this sequence as $\langle e_\ell \rangle_{\ell \in I}$. This convention does not affect the semantics of NVIL but will be useful in the definition of the intensional transformation and its proof of correctness (not discussed in this paper).

The semantics of NVIL is given in Figure 3. As discussed in Section 2, it is defined in the form of an evaluation function $\mathit{EVAL}_p(e, w)$, where p is the program, e is the expression to be evaluated, and w is the intensional context. In contrast to the simple structure of contexts (lists of labels) used in [16], the introduction of user-defined data types requires a more complex kind of contexts, similar to lists with backpointers (b-lists) defined by Yaghi [20].

Contexts are defined by the following grammar. The new element is μ, which is a list of contexts corresponding to nested `case` expressions.

$$
\begin{array}{lcl}
w & ::= & \bullet \mid \langle \ell, w, \mu \rangle \\
\mu & ::= & \bullet \mid w : \mu
\end{array}
$$

The result of function $\mathit{EVAL}_p(e, w)$ is either a ground value, which is returned by the meaning of some operator c (e.g., an integer number), or a pair of the form $\langle \kappa, w \rangle$, which corresponds to a value of a user-defined data type. In the latter case, κ is the constructor that was used to build this value and w is the context that must be used to evaluate the constructor's arguments. This semantics is captured in the equation for $\mathit{EVAL}_p(\kappa, w)$; remember that such expressions can only occur in the bodies of functions f_κ that have been introduced for all constructors κ.

The semantics of `call` and `actuals` operate on the context in the same way as informally introduced in Section 2; `call` adds a new label to the context and `actuals` selects the expression to evaluate based on the current label, which it removes from the context. The most interesting parts of the semantics are the equations for `case` and for `#`^{m}. In the former, the expression to be analyzed is evaluated and is found to be of the form $\langle \kappa_i, w' \rangle$ for some constructor κ_i that is mentioned in one of the clauses of `case`. (This is guaranteed if the program is well typed and `case` clauses are exhaustive, but we do not discuss typing issues in this paper.) Evaluation proceeds with the body e_i of that clause but the context w' is prepended to the list μ of contexts corresponding to nested `case` expressions. If later, in the evaluation of e_i, an expression of the form $\texttt{\#}^m(e)$ is found, the context μ_m found in the m-th position of the list μ is used for evaluating e, instead of the current context.

3.2 The Intensional Transformation from FOFL to NVIL

We start by defining the set $\mathit{labels}(f, p)$, i.e., the set of labels of calls to f in program p. These labels will form the indices of `call` operators. More specifically, the label of a function call $f(e_0, \ldots, e_{n-1})$ is simply the sequence of its arguments $\langle e_0, \ldots, e_{n-1} \rangle$. In other words, the transformed form of the call $f(e_0, \ldots, e_{n-1})$ will be $\texttt{call}_\ell$ where $\ell = \langle e_0, \ldots, e_{n-1} \rangle$. This assumption is slightly different from the one presented in Section 2.1 but it helps us in two ways. First, using this assumption, two identical function calls in the program receive exactly the same label. Second, since a label ℓ is a sequence of the actual parameters of a function call, we can write ℓ_m in order to specify the m-th actual parameter of this call. This helps us simplify notation. Recapitulating:

$$\mathit{labels}(f, p) \;=\; \{\langle e_0, \ldots, e_{n-1} \rangle \mid f(e_0, \ldots, e_{n-1}) \text{ in } p\}$$

We can now define the overall transformation from FOFL to NVIL, as shown in Figure 4. Given a program p, the function $\mathit{Trans}(p)$ removes the formal parameters from all definitions and adds one extra definition for every formal parameter of every function in the program. The creation of these extra definitions is performed by the function *actdefs*. More specifically, given a function f with formal parameters $v_0, \ldots, v_{n-1}$, the function $\mathit{actdefs}(f, p)$ creates one `actuals` definition for each v_j; this definition contains a sequence of all the (processed) actual parameters of f in p that correspond to the j-th position. Finally, we have the functions $\mathcal{E}$ and $\mathcal{B}$, which process expressions and `case` clauses. The main role of these two functions is to replace function calls with corresponding occurrences of the operator `call`.

4 The Implementation

In this section we describe an implementation of the generalized intensional transformation. The key idea of the implementation is that for every definition in the target intensional program, a corresponding piece of C code is generated, parameterized by the current context. In fact, the C code implements a more efficient version of the *EVAL* function in Figure 3. The runtime system uses a stack and a heap. However, in contrast to the standard implementation of user-defined data types that are represented as heap objects, the only entities that are stored in the stack and the heap are *Lazy Activation Records* (LARs), which we adapt here from our previous work [4]. A LAR is created when an expression of the form $\texttt{call}_\ell(f)$ is encountered during the execution of the program. LARs are similar to traditional activation records where, among other things, function parameters are stored. Some of the fields in a LAR are not filled at the time

$$\begin{array}{lcl}
\mathcal{E}(c(e_0,\ldots,e_{n-1})) & = & c(\mathcal{E}(e_0),\ldots,\mathcal{E}(e_{n-1})) \\
\mathcal{E}(f) & = & f \\
\mathcal{E}(f(e_0,\ldots,e_n)) & = & \mathtt{call}_\ell(f) \quad \text{where} \;\; \ell = \langle e_0,\ldots,e_n \rangle \\
\mathcal{E}(\kappa(e_0,\ldots,e_{n-1})) & = & \kappa \\
\mathcal{E}(\mathtt{case}\; e \;\mathtt{of}\; \{b_0\,;\ldots;b_n\}) & = & \mathtt{case}\; \mathcal{E}(e) \;\mathtt{of}\; \{\mathcal{B}(b_0);\;\ldots;\;\mathcal{B}(b_n)\} \\
\mathcal{E}(\mathtt{\#}^m(e)) & = & \mathtt{\#}^m(\mathcal{E}(e)) \\
\mathcal{B}(\kappa(v_0,\;\ldots,\;v_{n-1}) \to e) & = & \kappa \to \mathcal{E}(e)
\end{array}$$

$$\mathit{actdefs}(f,p) \quad = \quad \bigcup_{j=0}^{n-1} \quad \{v_j = \mathtt{actuals}(\langle \mathcal{E}(l_j)\rangle_{l \in I})\}$$

where $v_0,\ldots,v_{n-1}$ are the formal parameters of f and $I = \mathit{labels}(f,p)$

$$\mathit{Trans}(p) \quad = \quad \bigcup_{f(v_0,\ldots,v_{n-1})=e \text{ in } p} \quad \{f = \mathcal{E}(e)\} \;\cup\; \mathit{actdefs}(f,p)$$

Fig. 4. The transformation algorithm from FOFL to NVIL.

of the function call, when the LAR is constructed, but only when their value is actually demanded by the implementation. Notice that when the value of a formal parameter under a given context is demanded *again* during execution, then the existing value for this formal parameter can be retrieved from the LAR. In other words, the LARs implement a call-by-need semantics, as discussed at the end of Subsection 2.1.

A LAR corresponds directly to a context of the form $w = \langle \ell, w', \mu \rangle$ in the definition of function *EVAL* in Figure 3. More specifically, it contains the fields:

- *prev*: a pointer to the parent LAR, i.e., the LAR of the function that invoked this one. It corresponds directly to w' above.
- $\mathit{arg}_0,\ldots,\mathit{arg}_{n-1}$: each arg_i points to the code corresponding to the i-th formal parameter of the function call that generated this LAR. This is an encoding of ℓ, in the formal semantics of NVIL, and can be directly used to evaluate the function's arguments.
- $\mathit{val}_0,\ldots,\mathit{val}_{n-1}$: each val_i memoizes the value of the corresponding arg_i. It is initially empty and will be filled on demand: if at some point the code stored in arg_i is executed and computes a value, this value will be stored in val_i for future use. This implements a call-by-need semantics.
- *nested*: this field corresponds directly to μ. It is in fact an array which memoizes the values of expressions used in nested `case` constructs. In particular, when an expression of the form $\mathtt{\#}^m(e)$ is later encountered, $\mathit{nested}[m]$ points to the LAR that must be used to evaluate e.

With all this in mind, the compilation of the NVIL program to C code faithfully follows the rules of EVAL_p given in Figure 3.

The main difference between our approach and the standard implementation of non-strict functional languages is the absence of *closures*. In the traditional implementation of call-by-need, the field arg_i would contain a closure consisting of: (i) a pointer to the code that will compute the i-th parameter, and (ii) an environment, providing the values of the captured variables that this code needs to use. On the other hand, in our implementation, arg_i is just a code pointer. The environment has been eliminated, as the intensional transformation has encoded it in the context (i.e., a pointer to a LAR) that will be passed to arg_i. All variables correspond to top-level, zero-order definitions and it is the context that guides evaluation and produces the correct values of these variables.

The implementation includes certain rather simple optimizations which focus on allocating LARs on the stack whenever this is possible:

– Functions returning ground values (e.g., integers or booleans) or data types with only nullary constructors allocate their LARs on the stack and deallocate them on return.
– Functions that may return data types built by non-nullary constructors allocate their LARs on the heap.

Using this scheme, programs that do not make extensive use of user-defined data types can benefit from stack allocation. Further optimizations are possible, such as tail call elimination, but have not yet been implemented. Usage analysis can also be handy for further optimizations. If, for example, it is known that the value of some arg_i is only used once, then it need not be stored in val_i.

Stack-allocated LARs are discarded immediately when the active function call terminates. On the other hand, a garbage collector is required to discard heap-allocated LARs. We have currently implemented a simple semi-space copying garbage collector but we intend to investigate this further and expect that much better performance can be achieved with a garbage collector more suitable for the nature and usage of LARs; this is one of the primary goals for our future research. The root set for garbage collection is calculated by traversing stack-allocated LARs and the active context.

5 Performance Evaluation

In order to evaluate the performance of our implementation, we benchmarked it against four other well-known Haskell compilers:[3]

– The Glasgow Haskell Compiler (GHC): the definitive compiler for Haskell.
– The Utrecht Haskell Compiler (UHC): implemented using attribute grammars and supporting most features of Haskell 98 and Haskell 2010.
– The NHC98: a small and portable compiler for Haskell 98.
– The JHC: an experimental and fast compiler for Haskell, implemented in order to test various optimizations for the language.

[3] The code of our implementation and the benchmark programs that we used are available from `http://www.softlab.ntua.gr/~gfour/dftoic/`.

Program	GIC	GIC-llvm	GHC7	GHC6	NHC	UHC	JHC
ack	2.47	1.25	0.62	0.48	6.18	40.03	0.05
church	3.55	2.09	0.61	0.55	11.58	68.37	0.17
collatz	0.69	0.41	1.07	2.66	84.28	46.90	0.16
digits_of_e1	2.30	2.09	0.77	1.74	60.71	75.29	–[1]
fast-reverse	3.03	1.95	1.74	1.82	1.35	9.41	–[2]
fib	1.35	1.12	0.50	0.51	10.43	55.55	0.17
naive-reverse	3.02	2.87	0.49	0.42	0.79	3.56	0.75
ntak	8.62	5.87	2.91	3.65	154.74	91.95	7.18
primes	2.55	1.58	2.19	2.30	172.45	173.81	0.73
queens-num	0.33	0.23	0.31	0.33	21.16	12.43	0.14
queens	3.92	3.24	0.44	0.48	27.17	123.98	0.82
quick-sort	3.18	2.77	1.92	1.90	1.51	5.42	8.58
tree-sort	2.19	1.97	0.39	0.33	0.91	6.58	0.72
GMR[3]	1.38	1.00	0.51	0.57	7.28	18.49	0.33

[1] jhc compilation error, [2] jhc runtime error.
[3] Geometric mean of the ratios, compared to GIC-llvm.

Fig. 5. Runtime comparison for 13 benchmarks. Execution times are in seconds.

The comparison is based on a set of 13 benchmark programs, most of which are standard benchmarks for lazy functional languages, e.g. coming from the NoFib benchmark suite [11]. Some of the programs perform purely numerical computations (such as the programs `ack`, `fib`, `primes` and `queens-num`), pure list processing (such as `naive-reverse` and `fast-reverse`), numerical computations combined with list-processing and/or higher-order functions (such as `church`, `ntak`, `collatz`, `digits_of_e1`, `quick-sort`), and other user-defined data types (such as `queens` and `tree-sort`).

The benchmarks were performed on a machine with four quad-core Intel Xeon E7340 2.40GHz processors and 16 GB memory, running Debian 6.0.5. The versions of the compilers tested were GHC 7.4.1 and GHC 6.12.1, UHC/EHC 1.1.4, NHC98 1.22, and JHC 0.8.0. Our own compiler is shown in the benchmarks table as `GIC` (the Generalized Intensional Compiler). All benchmarks were executed five times and the median (elapsed) execution time was recorded. For all compilers the effects of garbage collection were minimized by setting a large size for the heap — in practice all programs either did no garbage collection at all or only a few. Finally, we disabled strictness analysis from all compilers that supported such an option, so as to focus on the performance of genuine lazy implementations. As this results in a significant slowdown for compilers like GHC, we will have to repeat the experiment when a competitive strictness analysis has been implemented for our compiler.

The performance results are depicted in Figure 5. In this table, `GIC-llvm` is the generalized intensional compiler whose C output is compiled using `llvm-gcc`, the front-end of `gcc` to the LLVM compiler. We used GCC 4.4.5 and LLVM 2.6. The benchmarks appear to suggest the following conclusions:

– Compiling the target C code of the generalized intensional compiler with `llvm-gcc` is quite more efficient than with standard `gcc`. Very similar results were also obtained using `clang`. In the following, when we refer to the intensional compiler, we mean `GIC-llvm`.
– The intensional implementation is on the average 2-3 times slower than the fully optimized implementations `GHC6` and `GHC7`. Notably, for `collatz`, `primes`, and `queens-num`, the intensional system performs better than `GHC6` and `GHC7`. Since the intensional compiler does not currently support any sophisticated optimizations, we believe that there is room for much improvement in our implementation.
– In certain programs (e.g., `ack` and `church`) `GHC6` performs better that `GHC7`. This has been reported (ticket #5888 in the GHC bug tracking system); it is related to a GHC optimization for unboxing integer values which seems to have deteriorated in GHC 7. It is expected to be fixed in release 7.6.1.

In general, we feel that the performance results are quite promising for the intensional approach, especially if we take into consideration that it is a far less mature compiler and that its implementation mainly aimed at simplicity and not performance, at this point.

6 Related Work

The work described in this paper, has its roots in the area of *dataflow programming*, which flourished more than three decades ago. It is also connected to the area of *intensional* and *multidimensional programming* [2] which was later developed as an extension of dataflow programming. The proposed technique has its origins in the key ideas that have been developed in order to implement dataflow and intensional languages.

Implementation Techniques for Dataflow Languages. In the dataflow model of computation, data are processed while they are flowing through a network of interconnected nodes (or *dataflow network*). A dataflow network is a system of *processing units* (or *nodes*) which are connected with *communication channels* (or *arcs*). Nodes can have multiple input and output arcs. The most advanced form of dataflow is the so-called *tagged token dataflow* in which the data-items are labeled with *tags* (or *contexts*). A node can fire if it receives in its input arcs data-items that have the same tags. The tagged-token approach obviates the need of data-items to arrive in a strictly pipelined way.

The majority of languages that were used to program dataflow computers were functional in flavor. Therefore, there existed an obvious need to compile recursive functions in a way compatible with the tagged-token model. Many such implementations were developed (e.g., see [9, 1]). The key idea of such implementations was to use tags to distinguish data items that belong to different function invocations. This tag-based implementation of recursive functions was known in the dataflow circles as *coloring*. Under the coloring scheme, higher-order functions were implemented by introducing special *apply* nodes in the dataflow graph that used a closure representation for function dispatch [18, 12].

The similarity of coloring with the approach proposed in this paper should be apparent by now. Tags correspond to the contexts in our technique. In particular, a context in our technique is used in order to uniquely identify a particular function call in the recursion tree of a program. One can say that the proposed approach transfers the key ideas of dataflow implementations to mainstream lazy functional languages. The novel aspects of our approach are the extension of the coloring technique to a language with user-defined data-types and its efficient implementation on stock hardware.

Intensional Languages and their Implementation. The development of dataflow languages was continued during the nineties with the invention of an extension of dataflow programming, namely *intensional programming* [2]. The first intensional/dataflow language was Lucid [19] whose implementation was based on the original intensional transformation which was formalized through the use of *intensional logic* in A. Yaghi's Ph.D. dissertation [20]. The correctness of the intensional transformation was established in [16]. The novel aspect of the current approach with respect to the original intensional transformation is the support of user-defined data-types and pattern matching.

A recent extension of Lucid is the language TransLucid [13]. The problem of implementing higher-order functions in the context of TransLucid has been considered and the solution that has been proposed is through an explicit representation for closures using extra dimensions (which amount to multiple contexts). To our knowledge, the technique for implementing TransLucid has not been applied to more mainstream functional languages.

Finally, we should note that (to our knowledge) all implementations of intensional languages rely on a runtime structure known as the *warehouse*. The warehouse is a hash-table in which intermediate results are stored in order to be reused when demanded again. Despite the fact that our technique shares the same underlying demand-driven execution model with the intensional languages (since they all rely on the original intensional transformation), our runtime structures and implementation decisions are completely different.

Implementations of Functional Languages. In general, the intensional approach to implementing functional languages appears to differ in philosophy with respect to the graph-reduction-based implementations. The work that appears to be closest to our approach is Boquist's GRIN compiler [3], which is also based on a defunctionalized representation. While GRIN uses a variety of "tags" to characterize different constructs of a lazy language (constructors, function applications, and partial applications), we use a uniform representation for these three types of constructs. GRIN was based on a strict first-order language, in contrast to our source language, FOFL, which is non-strict. Moreover, GRIN directly compiled its language for graph reduction using custom optimizations such as a unique interprocedural register allocation algorithm; we transform it to a zero-order intensional language and compile the intensional representation into C code, using a runtime that is based on lazy activation records.

The generalized intensional transformation has some conceptual similarities with environment-based abstract machines, like the work of Friedman and

Wise [7], Henderson and Morris [8], and Krivine [10], or the environment-based STG machines of De La Encina and Peña [5]. One important distinction of the intensional approach with respect to the above, is that our technique is based on a first-order source language. However, one could say that the contexts of our technique play in some sense the role of the environment, since they guide the execution mechanism to perform the correct substitution in the body of a function. We feel that a further investigation of the connections between the two approaches is quite worthwhile.

7 Conclusions and Future Work

We have introduced the *generalized intensional transformation*, an extension of the original intensional transformation that can be used to implement lazy functional languages with user-defined data types. We have demonstrated the usefulness of the proposed technique by implementing such a compiler for a subset of Haskell and by comparing its performance with existing Haskell implementations. There are certain aspects of the technique that appear to require a more extensive investigation:

- Our implementation currently compiles only a fragment of Haskell. It is our intention to extend the implementation to cover the full language. One possibility would be to make our implementation a back-end to GHC, since the GHC core language is (roughly speaking) a higher-order version of our FOFL language. This would allow us to take advantage of all the optimizations and language extensions of GHC.
- Our technique is heavily based on defunctionalization. It is a well-known fact that defunctionalization is a whole-program transformation and therefore one cannot do separate compilation. This is one aspect of our approach which we intend to further investigate. The discussion given in the concluding section of [14] might be a good start on lifting this shortcoming of defunctionalization.
- At present, the compiler only supports a minimal set of optimizations and the runtime system was implemented having simplicity as the driving criterion rather than efficiency. We are currently investigating optimizations at the intensional level and we plan to fine-tune the runtime in order to achieve a better performance. We also intend to investigate the possibility of using LLVM (instead of C) as the compiler's target language.
- We have implemented a simple-minded garbage collection scheme for LARs, which is currently non-portable and not mature enough to be discussed in this paper. We expect the implementation of an efficient garbage collector to be one of the major efforts of our future research, in conjunction with a possible re-implementation of the runtime system.

We feel that the simplicity of the technique and the promising performance results suggest that the intensional approach is worth further consideration as an alternative technique for implementing lazy functional languages.

References

1. Arvind, Nikhil, R.S.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Transactions on Computers 39, 300–318 (March 1990)
2. Ashcroft, E.A., Faustini, A.A., Jagannathan, R., Wadge, W.W.: Multidimensional Programming. Oxford University Press (1995)
3. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In: Kluge, W. (ed.) Implementation of Functional Languages, Lecture Notes in Computer Science, vol. 1268, pp. 58–84. Springer, Berlin/Heidelberg (1997)
4. Charalambidis, A., Grivas, A., Papaspyrou, N.S., Rondogiannis, P.: Efficient intensional implementation for lazy functional languages. Mathematics in Computer Science 2(1), 123–141 (2008)
5. De La Encina, A., Peña, R.: From natural semantics to C: A formal derivation of two STG machines. Journal of Functional Programming 19(01), 47–94 (2009)
6. Fourtounis, G., Papaspyrou, N., Rondogiannis, P.: The intensional transformation for functional languages with user-defined data types. In: Proceedings of the 8th Panhellenic Logic Symposium. pp. 38–42 (2011)
7. Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. In: Proceedings of the International Colloquium on Automata, Languages and Programming. pp. 257–284 (1976)
8. Henderson, P., Morris, Jr., J.H.: A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages. pp. 95–103. ACM, New York, NY, USA (1976)
9. Kirkham, C., Gurd, J., Watson, I.: The Manchester prototype dataflow computer. Communications of the ACM pp. 34–52 (1985)
10. Krivine, J.L.: Un interpréteur du lambda-calcul, `http://www.pps.univ-paris-diderot.fr/~krivine/articles/interprt.pdf`
11. Partain, W.: The nofib benchmark suite of Haskell programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming. pp. 195–202 (1993)
12. Pingali, K.: Lazy evaluation and the logic variable. In: Proceedings of the 2nd International Conference on Supercomputing. pp. 560–572. ACM, New York, NY, USA (1988)
13. Plaice, J., Mancilla, B.: The practical uses of TransLucid. In: Proceedings of the 1st International Workshop on Context-aware Software Technology and Applications. pp. 13–16. ACM, New York, NY, USA (2009)
14. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation 19, 125–162 (2006)
15. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the 25th ACM National Conference. pp. 717–740. ACM (1972)
16. Rondogiannis, P., Wadge, W.W.: First-order functional languages and intensional logic. Journal of Functional Programming 7(1), 73–101 (1997)
17. Rondogiannis, P., Wadge, W.W.: Higher-order functional languages and intensional logic. Journal of Functional Programming 9(5), 527–564 (1999)
18. Traub, K.R.: A compiler for the MIT tagged-token dataflow architecture. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1986)
19. Wadge, W., Aschroft, E.A.: Lucid, the Dataflow Programming Language. Academic Press (1985)
20. Yaghi, A.A.: The Intensional Implementation Technique for Functional Languages. Ph.D. thesis, Department of Computer Science, University of Warwick, Coventry, UK (1984)