The Generalized Intensional Transformation for Implementing Lazy Functional Languages

Georgios Fourtounis Nikolaos Papaspyrou
National Technical University of Athens
{gfour, nickie}@softlab.ntua.gr

Panos Rondogiannis
University of Athens
prondo@di.uoa.gr

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: (a) it could only be applied to programs that manipulate primitive data-types, and (b) 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 resulting first-order program is then transformed into a program in a simple tuple-based language. Finally, the original intensional transformation is extended in order to be applicable to the tuple language. The correctness of the generalized transformation is formally established. 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* [14, 15, 19] 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 [19] and was also used in the implementation of the first-order dataflow language Lucid [18]. Later on, the correctness of the transformation was formally established [14] and it was extended to apply to a simple class of higher-order programs [15]. 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 that manipulate primitive data-types (such as integers, characters, boolean values, and so on). For example, the dataflow language Lucid never supported user-defined data-types [18, 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 [15] 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 [13] in order to eliminate higher-order functions from the source program (at the cost of introducing explicit closures in the target first-order program). The two problems above are therefore trivially reduced to a single one, namely that of extending the intensional transformation to a language with user-defined data types (and pattern matching). The next step in the proposed approach is the elimination of constructor calls by introducing tuples in the language, whose first member is the constructor's name and whose subsequent arguments are the constructor's arguments. The advantage of the tuple representation is that we can now eliminate local pattern-matching variables by introducing in the language a selection operator. In this way we get a first-order program on which we can easily apply the original intensional transformation (with minimal modifications). The correctness of the proposed transformation is formally established. Finally, and most importantly, 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 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 at an intuitive level the proposed generalized transformation. Section 3 presents the new transformation at a formal level and establishes its correctness. 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. The Generalized Transformation (intuitively)

In this section we introduce in an intuitive way the generalized intensional transformation. We start by outlining the original intensional transformation for first-order functional languages that support only base types (for a more extensive discussion, see [14, 19]).

2.1 The Original Intensional Transformation

The input to the original intensional transformation [14, 19] is a first-order functional program that only uses base data-types (such as integers, Boolean values, and so on). After a simple processing, the source program is 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 [14]:

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 `call`$_i$`(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 functional program:

```
result = f 3 + f 5
f x    = g (x-1)
g y    = y+2
```

The transformation produces the following target program:

```
result = call₀(f) + call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x-1)
```

The above intensional code can be easily evaluated with respect to an initially empty *context*, which is in fact a list of natural numbers. Intuitively, the list keeps track of the exact position in the recursion tree where the execution currently is. The operators `call`$_i$ and `actuals` are context-switching operators. Intuitively, `call`$_i$ augments a list w by prefixing it with i. On the other hand, `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, which will often be omitted to simplify presentation. The function *lookup*(v, P) returns the defining expression of a variable v in program P. The evaluation of the usual constructs of functional languages (such as if-then-else, arithmetic operations, and so on) 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). The execution of our example intensional program derived above, is given in Figure 2. Notice that we assume that our source programs have a distinguished variable `result` whose value we want to compute.

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 [13]. 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 program is initially transformed into a defunctionalized first-order program:

```
data Func  = Add Int | Sq

result     = inc (Add 1) 2 + inc Sq 3
inc f x    = apply f (x+1)
apply cl d = case cl of
               Add c → add c d
               Sq    → sq d
add a b    = a+b
sq z       = z*z
```

We now have a first-order program on which however we can not directly apply the intensional transformation. The problem is the existence of data structures and in particular the existence of `case` definitions. How can local pattern-matching variables (such as the variable `c` in the definition of `apply`) be treated by the intensional transformation? The problem is both a practical one — how do we perform the transformation? — but also a semantic one — what will be the status of a variable like `c` in the zero-order intensional program?

The key idea for solving this problem is to note that the value of the parameter `cl` can either be the (nullary) data constructor `Sq` or the unary data constructor `Add` applied to some parameter `c`. We can view the result of the application of a data constructor as a tuple whose first member is the name of the data constructor itself and the rest of the members (if any) are the parameters of the data constructor. Viewing `cl` as a tuple is quite convenient, since we can introduce a selection function `#` which returns members of the tuple. For example, `cl#0` returns the name of the data constructor; if the data constructor has parameters (like `Add` above) then `cl#1` returns the first such parameter, `cl#2` the second, and so on. In our case, if `cl#0` is equal to `Add` then we can ask for `cl#1` which corresponds to the local pattern matching variable `c` above. Therefore, the above program can be transformed to:

```
result     = inc ⟨Add, 1⟩ 2 + inc ⟨Sq⟩ 3
inc f x    = apply f (x+1)
apply cl d = case cl#0 of
               Add → add cl#1 d
               Sq  → sq d
add a b    = a+b
sq z       = z*z
```

Notice that `Add` and `Sq` above are now just constants of our language and can be understood as (different) labels or natural numbers. Notice also that this is not anymore a well-typed program in

$$\begin{array}{lcl}
\mathit{EVAL}_P(v,w) & = & \mathit{EVAL}_P(\mathit{lookup}(v,P),w)\\
\mathit{EVAL}_P(\texttt{call}_i(E),w) & = & \mathit{EVAL}_P(E,i:w)\\
\mathit{EVAL}_P(\texttt{actuals}(E_0,\ldots,E_{n-1}),i:w) & = & \mathit{EVAL}_P(E_i,w)\\
\mathit{EVAL}_P(c(E_0,\ldots,E_{n-1}),w) & = & c(\mathit{EVAL}_P(E_0,w),\ldots,\mathit{EVAL}_P(E_{n-1},w))
\end{array}$$

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

$$\begin{array}{ll}
 & \mathit{EVAL}(\texttt{result},[\,])\\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f) + call}_1\texttt{(f)},[\,])\\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)},[\,])+\mathit{EVAL}(\texttt{call}_1\texttt{(f)},[\,])\\
= & \mathit{EVAL}(\texttt{f},[0])+\mathit{EVAL}(\texttt{f},[1])\\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(g)},[0])+\mathit{EVAL}(\texttt{call}_0\texttt{(g)},[1])\\
= & \mathit{EVAL}(\texttt{g},[0,0])+\mathit{EVAL}(\texttt{g},[0,1])\\
= & \mathit{EVAL}(\texttt{y},[0,0])+\mathit{EVAL}(\texttt{2},[0,0])+\mathit{EVAL}(\texttt{y},[0,1])+\mathit{EVAL}(\texttt{2},[0,1])\\
= & \mathit{EVAL}(\texttt{actuals(x-1)},[0,0])+2+\mathit{EVAL}(\texttt{actuals(x-1)},[0,1])+2\\
= & \mathit{EVAL}(\texttt{x-1},[0])+2+\mathit{EVAL}(\texttt{x-1},[1])+2\\
= & \mathit{EVAL}(\texttt{x},[0])-\mathit{EVAL}(\texttt{1},[0])+2+\mathit{EVAL}(\texttt{x},[1])-\mathit{EVAL}(\texttt{1},[1])+2\\
= & \mathit{EVAL}(\texttt{actuals(3, 5)},[0])-1+2+\mathit{EVAL}(\texttt{actuals(3, 5)},[1])-1+2\\
= & \mathit{EVAL}(\texttt{3},[\,])-1+2+\mathit{EVAL}(\texttt{5},[\,])-1+2\\
= & 3-1+2+5-1+2\\
= & 10
\end{array}$$

Figure 2. Execution of the target intensional program.

Haskell, as function `inc` is applied to two tuples of obviously different types.

We can now apply the intensional transformation in the standard way [14, 19] and obtain a zero-order intensional program:

```
result = call₀(inc)+call₁(inc)
inc    = call₀(apply)
f      = actuals(⟨Add, 1⟩, ⟨Sq⟩)
x      = actuals(2, 3)
apply  = case cl#0 of
             Add → call₀(add)
             Sq  → call₀(sq)
cl     = actuals(f)
d      = actuals(x+1)
add    = a+b
a      = actuals(cl#1)
b      = actuals(d)
sq     = z*z
z      = actuals(d)
```

The only additions that have to be made to the semantics of the intensional language for the execution of the above program are three extra cases: one for the selection operator, one for the tuples and one for the `case` expressions. The new *EVAL* is shown in Figure 3.

It can be checked (see Figure 4) that the above program can be easily evaluated using a demand-driven interpreter. Notice that for clarity we present separately the evaluation of the two subexpressions that comprise the right hand side of the `result` variable. Obviously, the final result of the evaluation of the program is the sum of the results of the two subcomputations.

3. The Generalized Transformation (formally)

As mentioned in the previous sections, we assume that our source language is a first-order language with user defined data-types, which in the following we call FOL (First-Order Language). In order to eliminate pattern-matching variables from FOL programs, the selection operator `#` is introduced, as outlined in the previous section. The programs that result belong to the language FOLT (First-Order Language with Tuples). Finally, FOLT programs are transformed into zero-order intensional ones that belong to the language NVIL (Nullary Variables Intensional Language). The syntax and semantics of the three languages, as well as the two transformations from FOL to FOLT and from FOLT to NVIL are described below. In the rest of this section, we assume some familiarity with the basic notions of denotational semantics, as well as the basic ideas regarding intensional languages (the reader is referred to the first few sections of [14]).

3.1 The Language FOL

The syntax of FOL is given by the following context-free grammar, where $f, v \in \mathit{Var}$ range over variables, $c \in \mathit{Con}$ ranges over constants, δ ranges over user defined data types, κ ranges over constructors, b ranges over basic types (such as `Int`, `Bool`, and so on), and $n, m \geq 0$.

$$\begin{array}{lll}
\mathit{Prog} & ::= & \mathit{Tdef}_0,\ \ldots,\ \mathit{Tdef}_{n-1},\ \mathit{Fdef}_0,\ \ldots,\ \mathit{Fdef}_m\\
\mathit{Tdef} & ::= & \texttt{data}\ \delta\ \texttt{=}\ \mathit{Cdef}_0\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{Cdef}_n\\
\mathit{Cdef} & ::= & \kappa\ \mathit{Typ}\ \ldots\ \mathit{Typ}_{n-1}\\
\mathit{Typ} & ::= & \delta \mid b\\
\mathit{FDef} & ::= & f(v_0,\ \ldots,\ v_{n-1}) = \mathit{Expr}\\
 & \mid & f(v_0,\ \ldots,\ v_{n-1}) = \mathit{CExpr}\\
\mathit{Expr} & ::= & c(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1})\\
 & \mid & f(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1})\\
 & \mid & \kappa(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1})\\
\mathit{CExpr} & ::= & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \{\ \mathit{Cls}_0;\ \ldots;\ \ \mathit{Cls}_n\ \}\\
\mathit{Cls} & ::= & \kappa(v_0,\ \ldots,\ v_{n-1}) \to \mathit{Expr}
\end{array}$$

We omit the parentheses around empty sequences of expressions and variables. As we have already mentioned, we assume that every program in our language contains a distinguished variable `result` whose value will represent the desired output of the

$$
\begin{array}{lcl}
\mathit{EVAL}_P(v, w) & = & \mathit{EVAL}_P(\mathit{lookup}(v, P), w) \\
\mathit{EVAL}_P(\mathtt{call}_i(E), w) & = & \mathit{EVAL}_P(E, i : w) \\
\mathit{EVAL}_P(\mathtt{actuals}(E_0, \ldots, E_{n-1}), i : w) & = & \mathit{EVAL}_P(E_i, w) \\
\mathit{EVAL}_P(c(E_0, \ldots, E_{n-1}), w) & = & c(\mathit{EVAL}_P(E_0, w), \ldots, \mathit{EVAL}_P(E_{n-1}, w)) \\
\mathit{EVAL}_P(\langle E_0, \ldots, E_{n-1}\rangle, w) & = & \langle \mathit{EVAL}_P(E_0, w), \ldots, \mathit{EVAL}_P(E_{n-1}, w)\rangle \\
\mathit{EVAL}_P(E\,\#\,i, w) & = & \mathit{EVAL}_P(E, w)\,\#\,i \\
\mathit{EVAL}_P(\mathtt{case}\ E\ \mathtt{of}\ i_0 \to E_0; \ldots; i_{n-1} \to E_{n-1}, w) & = & \mathit{EVAL}_P(E_k, w) \quad \text{if}\ \mathit{EVAL}_P(E, w) = i_k
\end{array}
$$

Figure 3. The *EVAL* function for the extended intensional language.

$$
\begin{array}{cl}
 & \mathit{EVAL}(\mathtt{call}_0\mathtt{(inc)}, [\,]) \\
= & \mathit{EVAL}(\mathtt{inc}, [0]) \\
= & \mathit{EVAL}(\mathtt{call}_0\mathtt{(apply)}, [0]) \\
= & \mathit{EVAL}(\mathtt{apply}, [0, 0]) \\
= & \mathit{EVAL}(\mathtt{case\ cl\#0\ of\ \{\ Add} \to \mathtt{call}_0\mathtt{(add);\ Sq} \to \mathtt{call}_0\mathtt{(sq)\ \}}, [0, 0]) \\
 & \quad \mathit{EVAL}(\mathtt{cl\#0}, [0, 0]) \\
 & \quad \mathit{EVAL}(\mathtt{cl}, [0, 0])\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{actuals(f)}, [0, 0])\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{f}, [0]|\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{actuals(}\langle\mathtt{Add,\ 1}\rangle\mathtt{,\ }\langle\mathtt{Sq}\rangle\mathtt{)}, [0])\,\#\,0 \\
 & \quad \mathit{EVAL}(\langle\mathtt{Add,\ 1}\rangle,))[\,]\,\#\,0 \\
 & \quad \langle\mathtt{Add,\ 1}\rangle\,\#\,0 \\
 & \quad \mathtt{Add} \\
= & \mathit{EVAL}(\mathtt{call}_0\mathtt{(add)}, [0, 0]) \\
= & \mathit{EVAL}(\mathtt{add}, [0, 0, 0]) \\
= & \mathit{EVAL}(\mathtt{a+b}, [0, 0, 0]) \\
= & \mathit{EVAL}(\mathtt{a}, [0, 0, 0]) + \mathit{EVAL}(\mathtt{b}, [0, 0, 0]) \\
= & \mathit{EVAL}(\mathtt{actuals(cl\#1)}, [0, 0, 0]) + \mathit{EVAL}(\mathtt{actuals(d)}, [0, 0, 0]) \\
= & \mathit{EVAL}(\mathtt{cl\#1}, [0, 0]) + \mathit{EVAL}(\mathtt{d}, [0, 0]) \\
= & \mathit{EVAL}(\mathtt{cl}, [0, 0])\,\#\,1 + \mathit{EVAL}(\mathtt{d}, [0, 0]) \\
= & \mathit{EVAL}(\mathtt{actuals(f)}, [0, 0])\,\#\,1 + \mathit{EVAL}(\mathtt{actuals(x+1)}, [0, 0]) \\
= & \mathit{EVAL}(\mathtt{f}, [0])\,\#\,1 + \mathit{EVAL}(\mathtt{x+1}, [0]) \\
= & \mathit{EVAL}(\mathtt{actuals(}\langle\mathtt{Add,\ 1}\rangle\mathtt{,\ }\langle\mathtt{Sq}\rangle\mathtt{)}, [0])\,\#\,1 + \mathit{EVAL}(\mathtt{x}, [0]) + 1 \\
= & \mathit{EVAL}(\langle\mathtt{Add,\ 1}\rangle, [\,])\,\#\,1 + \mathit{EVAL}(\mathtt{actuals(2,\ 3)}, [0]) + 1 \\
= & 1 + 2 + 1 \\
= & 4
\end{array}
$$

$$
\begin{array}{cl}
 & \mathit{EVAL}(\mathtt{call}_1\mathtt{(inc)}, [\,]) \\
= & \mathit{EVAL}(\mathtt{inc}, [1]) \\
= & \mathit{EVAL}(\mathtt{call}_0\mathtt{(apply)}, [1]) \\
= & \mathit{EVAL}(\mathtt{apply}, [0, 1]) \\
= & \mathit{EVAL}(\mathtt{case\ cl\#0\ of\ \{\ Add} \to \mathtt{call}_0\mathtt{(add);\ Sq} \to \mathtt{call}_0\mathtt{(sq)\ \}}, [0, 1]) \\
 & \quad \mathit{EVAL}(\mathtt{cl\#0}, [0, 1]) \\
 & \quad \mathit{EVAL}(\mathtt{cl}, [0, 1])\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{actuals(f)}, [0, 1])\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{f}, [1])\,\#\,0 \\
 & \quad \mathit{EVAL}(\mathtt{actuals(}\langle\mathtt{Add,\ 1}\rangle\mathtt{,\ }\langle\mathtt{Sq}\rangle\mathtt{)}, [1])\,\#\,0 \\
 & \quad \mathit{EVAL}(\langle\mathtt{Sq}\rangle, [\,])\#0 \\
 & \quad \langle\mathtt{Sq}\rangle\,\#\,0 \\
 & \quad \mathtt{Sq} \\
= & \mathit{EVAL}(\mathtt{call}_0\mathtt{(sq)}, [0, 1]) \\
= & \mathit{EVAL}(\mathtt{sq}, [0, 0, 1]) \\
= & \mathit{EVAL}(\mathtt{z*z}, [0, 0, 1]) \\
= & \mathit{EVAL}(\mathtt{z}, [0, 0, 1]) * \mathit{EVAL}(\mathtt{z}, [0, 0, 1]) \\
= & \mathit{EVAL}(\mathtt{actuals(d)}, [0, 0, 1]) * \mathit{EVAL}(\mathtt{actuals(d)}, [0, 0, 1]) \\
= & \mathit{EVAL}(\mathtt{d}, [0, 1]) * \mathit{EVAL}(\mathtt{d}, [0, 1]) \\
= & \mathit{EVAL}(\mathtt{actuals(x+1)}, [0, 1])\,\#\,1 * \mathit{EVAL}(\mathtt{actuals(x+1)}, [0, 1]) \\
= & \mathit{EVAL}(\mathtt{x+1}, [1]) * \mathit{EVAL}(\mathtt{x+1}, [1]) \\
= & (\mathit{EVAL}(\mathtt{x}, [1]) + 1) * (\mathit{EVAL}(\mathtt{x}, [1]) + 1) \\
= & (\mathit{EVAL}(\mathtt{actuals(2,\ 3)}, [1]) + 1) * (\mathit{EVAL}(\mathtt{actuals(2,\ 3)}, [1]) + 1) \\
= & (3 + 1) * (3 + 1) \\
= & 16
\end{array}
$$

Figure 4. Execution of the target intensional program.

program. Moreover, we assume that the FOL programs under consideration are well-typed.

The semantics of FOL is the standard domain-theoretic semantics for a first-order functional language with data-structures (see for example [16]).

3.2 The Language FOLT

The language FOLT is similar to FOL, the main difference being that all user-defined data types have been replaced by a single one, namely the *tuple*. Moreover, there is a selection operator `#` that applies to tuple expressions. Finally, `case` expressions are somewhat simpler since they don't contain pattern matching variables. The syntax of FOLT is as follows, where $i \geq 0$:

$$\begin{array}{lcl}
\mathit{Prog} & ::= & \mathit{Def}_0,\ \ldots,\ \mathit{Def}_n \\
\mathit{Def} & ::= & f(v_0,\ \ldots, v_{n-1}) = \mathit{Expr} \\
 & \mid & f(v_0,\ \ldots, v_{n-1}) = \mathit{CExpr} \\
\mathit{Expr} & ::= & c(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1}) \\
 & \mid & f(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1}) \\
 & \mid & \langle \mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_n \rangle \\
 & \mid & \mathit{Expr}\ \texttt{\#}\ i \\
\mathit{CExpr} & ::= & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \{\ \mathit{Cls}_0;\ \ldots;\ \mathit{Cls}_n\ \} \\
\mathit{Cls} & ::= & c \to \mathit{Expr}
\end{array}$$

Notice how constructor names have been replaced by constants in the clauses of `case` expressions.

The semantics of FOLT is similar to that of FOL, the only difference being the existence of tuples in the language instead of data types. Let us denote by B the semantic domain of basic values of FOLT. For example, when:

$$B = \{\perp\} + \mathbb{N} + \{\mathit{true}, \mathit{false}\}$$

our base domain consists of the natural numbers, the Boolean values and the bottom element for handling non-termination.

The denotations of the zero-order variables of FOLT (such as nullary function definitions or formal parameters of functions) are elements of the following domain:

$$D = B + \sum_{n \in \omega} D^n$$

Intuitively, D consists of the denotations of the base types as well as the denotations of tuples.

The denotation of an n-ary function in FOLT is a member of $D^n \to D$. Notice that this is isomorphic to D, when $n = 0$, and we will use the two interchangeably.

As usual, environments assign denotations to function variables according to their arity:

$$\mathit{Env} = \prod_{n \in \omega} (\mathit{Var}_n \to D^n \to D)$$

where Var_n is the subset of Var whose members are variables of arity $n \geq 0$.

Finally, the denotations of constant symbols are assigned by the constant interpretation function $\mathcal{C}$:

$$\mathcal{C} : \prod_{n \in \omega} (\mathit{Con}_n \to D^n \to D)$$

where Con_n is the subset of Con whose members are constants of arity $n \geq 0$.

The semantics of FOLT expressions with respect to $u \in \mathit{Env}$ is defined in Figure 5 as a function of type:

$$[\![\cdot]\!] : \mathit{Expr} \to \mathit{Env} \to D$$

The meaning of a FOLT program P is expressed by the least environment u that satisfies all the definitions in P; it can be constructed as the least upper bound of a sequence of approximations $u_0, u_1, \ldots$ (see for example [16] for details).

3.3 The Language NVIL

The language NVIL is a zero-order intensional language. For more background on such languages, the interested reader can consult the first sections of [14]. The of this NVIL from the corresponding language defined in [14] is that the former supports tuples and the selection operator `#`. The syntax of NVIL is given by the following context-free grammar, where *Labels* is a set of labels, ℓ is a variable ranging over this set and $I \subseteq \mathit{Labels}$. Notice that the syntax of the $\mathit{actuals}$ operator is slightly different than the one informally introduced in Section 2. This syntax will be explained shortly and will be more convenient in the proof that will be given in Section 3.5.

$$\begin{array}{lcl}
\mathit{Prog} & ::= & \mathit{Def}_0,\ \ldots,\ \mathit{Def}_n \\
\mathit{Def} & ::= & f = \mathit{Expr} \\
 & \mid & f = \mathit{CExpr} \\
\mathit{Expr} & ::= & c(\mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_{n-1}) \\
 & \mid & f \\
 & \mid & \texttt{call}_\ell(f) \\
 & \mid & \langle \mathit{Expr}_0,\ \ldots,\ \mathit{Expr}_n \rangle \\
 & \mid & E\ \texttt{\#}\ i \\
 & \mid & \texttt{actuals}(\langle \mathit{Expr}_\ell \rangle_{\ell \in I}) \\
\mathit{CExpr} & ::= & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \{\ \mathit{Cls}_0;\ \ldots;\ \mathit{Cls}_n\ \} \\
\mathit{Cls} & ::= & c \to \mathit{Expr}
\end{array}$$

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 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 $\langle E_i \rangle_{i \in I}$. This convention will help us formally define and prove the correctness of the intensional transformation in an easier way.

The semantics of NVIL is defined with respect to a set of contexts. In Section 2, contexts were lists of natural numbers (see the third argument of *EVAL* in Figures 3 and 4). Since we have now replaced numeric labels with elements $\ell \in \mathit{Labels}$, the set of contexts now becomes the set of finite lists of such labels ℓ:

$$\mathit{Ctxt} = \prod_{n \in \omega} \mathit{Labels}^n$$

The semantic domains B and D are the same as in the case of FOLT. The domain of denotations of zero-order intensional variables is now

$$I = \mathit{Ctxt} \to D$$

and intensional environments assign elements of I to nullary variables:

$$\mathit{IEnv} = \mathit{Var}_0 \to I$$

The semantics of NVIL with respect to $\widehat{u} \in \mathit{IEnv}$ is defined in Figure 6 as a function of type:

$$[\![\cdot]\!] : \mathit{Expr} \to \mathit{IEnv} \to I$$

Notice that this function is in fact the denotational analogue of the *EVAL* function of Figure 3. The meaning of an NVIL program P is the least environment $\widehat{u}$ that satisfies the definitions of P.

$$\begin{array}{lcl}
[\![c(E_0,\ldots,E_{n-1})]\!](u) & = & \mathcal{C}(c)([\![E_0]\!](u),\ldots,[\![E_{n-1}]\!](u)) \\
[\![f(E_0,\ldots,E_{n-1})]\!](u) & = & u(f)([\![E_0]\!](u),\ldots,[\![E_{n-1}]\!](u)) \\
[\![\langle E_0,\ldots,E_{n-1}\rangle]\!](u) & = & \langle [\![E_0]\!](u),\ldots,[\![E_{n-1}]\!](u)\rangle \\
[\![E\,\#\,i]\!](u) & = & d_i \quad \text{if } [\![E]\!](u) = \langle d_0,\ldots,d_{n-1}\rangle \quad \text{and} \quad 0 \le i < n \\
[\![\texttt{case}\ E\ \texttt{of}\ \{c_0 \to E_0;\ \ldots;\ c_n \to E_n\}]\!](u) & = & [\![E_i]\!](u) \quad \text{if } c_i = [\![E]\!](u)
\end{array}$$

Figure 5. The denotational semantics of FOLT.

$$\begin{array}{lcl}
[\![c(E_0,\ldots,E_{n-1})]\!](u)(w) & = & \mathcal{C}(c)([\![E_0]\!](u)(w),\ldots,[\![E_{n-1}]\!](u)(w)) \\
[\![f]\!](u)(w) & = & u(f)(w) \\
[\![\texttt{call}_\ell(f)]\!](u)(w) & = & u(f)(\ell : w) \\
[\![\langle E_0,\ldots,E_{n-1}\rangle]\!](u)(w) & = & \langle [\![E_0]\!](u)(w),\ldots,[\![E_{n-1}]\!](u)(w)\rangle \\
[\![E\,\#\,i]\!](u)(w) & = & d_i \quad \text{if } [\![E]\!](u)(w) = \langle d_0,\ldots,d_{n-1}\rangle \quad \text{and} \quad 0 \le i < n \\
[\![\texttt{actuals}(\langle E_j\rangle_{j\in I})]\!](u)(\ell : w) & = & [\![E_\ell]\!](u)(w) \\
[\![\texttt{case}\ E\ \texttt{of}\ \{c_0 \to E_0;\ \ldots;\ c_n \to E_n\}]\!](u)(w) & = & [\![E_i]\!](u)(w) \quad \text{if } c_i = [\![E]\!](u)(w)
\end{array}$$

Figure 6. The denotational semantics of NVIL.

3.4 The Intensional Transformation for FOLT

As we have discussed in the previous sections, the source FOL programs are initially transformed to FOLT programs using a simple preprocessing. Since FOL and FOLT are both standard functional languages, the transformation from the former to the latter and the corresponding correctness proof are easy to establish. For this reason, we mainly focus here on the formal definition of the transformation from FOLT programs to NVIL programs. This will allow us to derive the correctness proof of the proposed technique, in Section 3.5.

We start by defining the set $labels(f, P)$, 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 its sequence of 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:

$$labels(f, P) \;=\; \{\langle E_0,\ldots,E_{n-1}\rangle \mid f(E_0,\ldots,E_{n-1}) \text{ in } P\}$$

We can now define the overall transformation from FOLT to NVIL, as shown in Figure 7. Given a program P, the function $Trans(P)$ 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 $actdefs(f, P)$ 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. Notice that since a label ℓ is in fact a sequence of the actual parameters, by writing ℓ_j we select the the j-th such parameter. Finally, we have the function $\mathcal{E}$ which processes expressions of the program. The main role of $\mathcal{E}$ is to replace function calls with corresponding occurrences of the operator `call`.

3.5 Correctness Proof

In this section we demonstrate the correctness of the extended intensional transformation. In the following we assume some familiarity with the basic definitions and techniques used in the proof of the original intensional transformation (see [14]).

In order to establish the correctness of the generalized intensional transformation, it suffices to show that given a FOLT program P, the transformed program $Trans(P)$ is semantically equivalent to P. Since our programs have a distinguished variable `result` which holds the output of the program, it suffices to show that the denotation of `result` is the same in both programs. Actually, let $u, \widehat{u}$ be the least environments satisfying the definitions of P and $\widehat{u}$ respectively. Then, it suffices to show that for every $w \in Ctxt$ it is $u(\texttt{result}) = \widehat{u}(\texttt{result})(w)$. In order to establish this fact, we show the following more general theorem.

Theorem 1. Let P be a FOLT program, let u be the least environment satisfying the definitions in P and let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(P)$. Consider a definition in P whose formal parameters are $v_0,\ldots,v_k$. Then, for every subexpression S that appears in the body of the aforementioned definition and for every context w, we have:

$$[\![\mathcal{E}(S)]\!](\widehat{u})(w) = [\![S]\!](u \oplus \rho)$$

where ρ is an environment such that $\rho(v_j) = \widehat{u}(v_j)(w)$ and $u \oplus \rho$ is an environment u' such that for every $v \in Var$, if $v \in \mathrm{dom}(\rho)$ then $u'(v) = \rho(v)$, else $u'(v) = u(v)$.

Proof. The proof consists of two parts. In the first part we demonstrate that $[\![\mathcal{E}(S)]\!](\widehat{u})(w) \sqsubseteq [\![S]\!](u \oplus \rho)$ and in the second part that $[\![\mathcal{E}(S)]\!](\widehat{u})(w) \sqsupseteq [\![S]\!](u \oplus \rho)$. We give the full details for the former statement; the proof for the latter statement can be performed symmetrically.

In order to establish the former statement we perform a double induction: an outer induction on the approximations $\widehat{u}_0, \widehat{u}_1, \ldots$ of $\widehat{u}$ and an inner structural induction on S.

The base case for the outer induction is:

$$[\![\mathcal{E}(S)]\!](\widehat{u}_0)(w) \sqsubseteq [\![S]\!](u \oplus \rho_0)$$

where ρ_0 is an environment such that $\rho_0(v_j) = \widehat{u}_0(v_j)(w)$. By a structural induction on S, the above can easily be shown, as $\widehat{u}_0$ maps all variables to $\perp$.

$$
\begin{array}{lcl}
\mathcal{E}(v\,\texttt{\#}\,i) & = & v\,\texttt{\#}\,i \\
\mathcal{E}(c(E_0,\ldots,E_{n-1})) & = & c(\mathcal{E}(E_0),\ldots,\mathcal{E}(E_{n-1})) \\
\mathcal{E}(f(E_0,\ldots,E_n)) & = & \texttt{call}_\ell(f) \quad \text{where } \ell = \langle E_0,\ldots,E_n\rangle \\
\mathcal{E}(\langle E_0,\ldots,E_{n-1}\rangle) & = & \langle \mathcal{E}(E_0),\ldots,\mathcal{E}(E_{n-1}\rangle \\
\mathcal{E}(\texttt{case}\ E\ \texttt{of}\ \{c_0 \to E_0;\ \ldots;\ c_n \to E_n\}) & = & \texttt{case}\ \mathcal{E}(E)\ \texttt{of}\ \{c_0 \to \mathcal{E}(E_0);\ \ldots;\ c_n \to \mathcal{E}(E_n)\}
\end{array}
$$

$$
\mathit{actdefs}(f,P) \;=\; \bigcup_{j=0}^{n-1} \{v_j = \texttt{actuals}(\langle \mathcal{E}(l_j)\rangle_{l\in I})\} \quad \text{where } v_0,\ldots,v_{n-1} \text{ are the formals of } f \text{ and } I = \mathit{labels}(f,P)
$$

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

Figure 7. The transformation algorithm from FOLT to NVIL.

Assume that the statement holds for k, namely that for every S:

$$[\![\mathcal{E}(S)]\!](\widehat{u}_k)(w) \sqsubseteq [\![S]\!](u \oplus \rho_k)$$

where ρ_k is an environment such that $\rho_k(v_j) = \widehat{u}_k(v_j)w$. We prove the result holds for $k+1$, i.e., we show that:

$$[\![\mathcal{E}(S)]\!](\widehat{u}_{k+1})(w) \sqsubseteq [\![S]\!](u \oplus \rho_{k+1})$$

We perform an inner structural induction on S.

Structural Induction Basis: The base cases are for $S = c$ (a nullary constant) and $S = v$ (a nullary variable). Consider first the former case. The left hand side is equal to $[\![\mathcal{E}(S)]\!](\widehat{u}_k)(w) = \mathcal{C}(c)$, while the right hand side gives also $\mathcal{C}(c)$. The result in the latter case follows immediately from the relationship between ρ_{k+1} and $\widehat{u}_{k+1}$.

Structural Induction Step: We distinguish cases for S:

Case $S = c(E_0,\ldots,E_{n-1})$. Then, for every $w \in \mathit{Ctxt}$, we have:

$$
\begin{array}{cl}
 & [\![\mathcal{E}(S)]\!](\widehat{u}_{k+1})(w) \\
= & [\![\mathcal{E}(c(E_0,\ldots,E_{n-1}))]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Assumption for } S) \\
= & [\![c(\mathcal{E}(E_0),\ldots,\mathcal{E}(E_{n-1}))]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Definition of } \mathcal{E}) \\
= & \mathcal{C}(c)([\![\mathcal{E}(E_0)]\!](\widehat{u}_{k+1})(w),\ldots,[\![\mathcal{E}(E_{n-1})]\!](\widehat{u}_{k+1})(w)) \\
 & \text{(Semantics of NVIL)} \\
\sqsubseteq & \mathcal{C}(c)([\![E_0]\!](u\oplus\rho_{k+1}),\ldots,[\![E_{n-1}]\!](u\oplus\rho_{k+1})) \\
 & \text{(Structural induction hypothesis and monotonicity of } [\![\cdot]\!]) \\
= & [\![c(E_0,\ldots,E_{n-1})]\!](u\oplus\rho_{k+1}) \\
 & \text{(Semantics of FOLT)} \\
= & [\![S]\!](u\oplus\rho_{k+1}) \\
 & \text{(Assumption for } S)
\end{array}
$$

Case $S = f(E_0,\ldots,E_{n-1})$. Assume that f is defined in P as $f(x_0,\ldots,x_{n-1}) = B_f$. Then, for every $w \in \mathit{Ctxt}$, we have:

$$
\begin{array}{cl}
 & [\![\mathcal{E}(S)]\!](\widehat{u}_{k+1})(w) \\
= & [\![\mathcal{E}(f(E_0,\ldots,E_{n-1}))]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Assumption for } S) \\
= & [\![\texttt{call}_\ell(f)]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Definition of } \mathcal{E}) \\
= & [\![f]\!](\widehat{u}_{k+1})(\ell : w) \\
 & \text{(Semantics of } \texttt{call}_\ell) \\
= & [\![\mathcal{E}(B_f)]\!](\widehat{u}_k)(\ell : w) \\
 & \text{(Because } f = \mathcal{E}(B_f) \text{ in } \mathit{Trans}(P)) \\
\sqsubseteq & [\![B_f]\!](u\oplus\rho'),\ \text{where } \rho'(x_j) = \widehat{u}_k(x_j)(\ell : w) \\
 & \text{(Outer induction hypothesis)} \\
\sqsubseteq & [\![f(E_0,\ldots,E_{n-1})]\!](u\oplus\rho_{k+1}) \\
 & \text{(See explanation below)} \\
= & [\![S]\!](u\oplus\rho_{k+1}) \\
 & \text{(Assumption for } S)
\end{array}
$$

Notice that by the definition of Trans there exist in P definitions $x_j = \texttt{actuals}(\langle \mathcal{E}(l_j)\rangle_{l\in I})$. In the above proof, the label is $\ell = \langle E_0,\ldots,E_{n-1}\rangle$. Therefore, it holds:

$$
\begin{array}{rcl}
\rho'(x_j) & = & \widehat{u}_k(x_j)(\ell : w) \\
 & \sqsubseteq & [\![\texttt{actuals}(\langle \mathcal{E}(l_j)\rangle_{l\in I})]\!](\widehat{u}_k)(\ell : w) \\
 & = & [\![\mathcal{E}(E_j)]\!](\widehat{u}_k)(w) \\
 & \sqsubseteq & [\![E_j]\!](u\oplus\rho_{k+1})
\end{array}
$$

where the last step is justified by the induction hypothesis and the fact that $\rho_k \sqsubseteq \rho_{k+1}$.

Case $S = \langle E_0,\ldots,E_{n-1}\rangle$. We have:

$$
\begin{array}{cl}
 & [\![\mathcal{E}(S)]\!](\widehat{u}_{k+1})(w) \\
= & [\![\mathcal{E}(\langle E_0,\ldots,E_{n-1}\rangle)]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Assumption for } S) \\
= & [\![\langle \mathcal{E}(E_0),\ldots,\mathcal{E}(E_{n-1})\rangle]\!](\widehat{u}_{k+1})(w) \\
 & \text{(Definition of } \mathcal{E}) \\
= & \langle [\![\mathcal{E}(E_0)]\!](\widehat{u}_{k+1})(w), [\![\mathcal{E}(E_{n-1})]\!](\widehat{u}_{k+1})(w)\rangle \\
 & \text{(Semantics of NVIL)} \\
\sqsubseteq & \langle [\![E_0]\!](u\oplus\rho_{k+1}),\ldots,[\![E_{n-1}]\!](u\oplus\rho_{k+1})\rangle \\
 & \text{(Structural induction hypothesis)} \\
= & [\![\langle E_0,\ldots,E_{n-1}\rangle]\!](u\oplus\rho_{k+1}) \\
 & \text{(Semantics of FOLT)} \\
= & [\![S]\!](u\oplus\rho_{k+1}) \\
 & \text{(Assumption for } S)
\end{array}
$$

Case $S = v\,\texttt{\#}\,i$. Straightforward.

Case $S = \texttt{case}\ E\ \texttt{of}\ \{c_0 \to E_0;\ \ldots;\ c_n \to E_n\}$. Easy using the semantics of `case` and the induction hypothesis. □

The above result leads to the following theorem:

Theorem 2. Let P be a FOLT program, let u be the least environment satisfying the definitions in P and let $\widehat{u}$ be the least environment satisfying the definitions in the translated program NVIL program $\mathit{Trans}(P)$. Then, for every $w \in \mathit{Ctxt}$, we have $u(\texttt{result}) = \widehat{u}(\texttt{result})(w)$.

Proof. A direct consequence of Theorem 1 when applied to the body of the definition of the `result` variable in P. □

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. In fact, the C code implements a more efficient version of the *EVAL* function of Figure 3. In the rest of this section we describe the details of the actual implementation.

4.1 Implementing *EVAL*

Every definition in the target intensional program corresponds to a piece of C code. 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). A LAR is similar to a traditional activation record where, among other things, a function's parameters are stored. Some of the fields in a LAR are not filled at the time of the function call, when the LAR is constructed, but only when their value is actually demanded by the implementation. This is more or less the standard way in which non-strict languages implement call-by-need.

A LAR contains the following fields:

- $index$: the index ℓ of the $\texttt{call}_\ell(f)$ expression that generated this LAR.
- $prev$: a pointer to the parent LAR, i.e., the LAR of the function that invoked this one. Notice that the pair $(index, prev)$ corresponds to the context w of the *EVAL* function ($index$ is the head of w and $prev$ is a pointer to the tail).
- $arg_0, \ldots, arg_{n-1}$: each arg_i points to the code corresponding to the i-th formal parameter of the function call that generated this LAR.
- $val_0, \ldots, val_{n-1}$: each val_i memoizes the value of the i-th formal parameter of the function call that generated this LAR. 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.
- $nested$: this field is required in order to efficiently compile `case` expressions; its use will be further explained below.

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 pointer to a thunk, i.e., 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 the environment has been eliminated by the intensional transformation and arg_i is just the code pointer. All variables correspond to top-level, zero-order definitions and it is the the context contained in the pair $(index, prev)$ that guides the computation of these variables and produces the correct value, when the code pointed to by arg_i is executed.

In order to simplify things more, our implementation treats occurrences of tuples in a FOLT program as a form of trivial function calls. For this reason a simple preprocessing step is performed which transforms a tuple $\langle E_0, \ldots, E_{n-1}\rangle$ to a call of the form $\texttt{tuple}_n(E_0, \ldots, E_{n-1})$ and introduces definitions of the form

$$\texttt{tuple}(v_0, \ldots, v_{n-1}) = \langle v_0, \ldots, v_{n-1}\rangle$$

This is similar to constructor wrapper functions in GHC. It is trivial to verify that this transformation preserves the semantics of the FOLT program. In this way, we eliminate the need to explicitly represent tuples at runtime: tuples are simply contexts and their elements are stored in LARs.

We now describe how each individual construct of the target intensional program is compiled:

- $\texttt{call}_\ell(f)$: The code produced in this case initially allocates an activation record in the heap or in the stack (depending on the result of escape analysis for function f, see Section 4.2). The $index$ field is set to ℓ and the $prev$ field is set to point to the LAR of the function that executed the call. The arg_i fields are set to point to the code corresponding to the formals of f and the val_i fields are initially empty. The $nested$ field is not used in this case.
- $\texttt{actuals}(E_0, \ldots, E_{n-1})$: The $index$ field of the present LAR is used to select the appropriate expression among the E_i. Evaluation of this expression is performed in the context pointed to by the $prev$ field of the present LAR.
- $\langle v_0, \ldots, v_{n-1}\rangle$: This expression now appears only once, in the body of a newly introduced $\texttt{tuple}_n$ function. It evaluates to a pointer to the current LAR.
- $v \,\texttt{\#}\, i$: The code produced in this case starts the evaluation of v under the current context, which should return a pointer to a LAR t that corresponds to a tuple. The val_i field of this LAR is examined. If it is nonempty, the corresponding value is returned. Otherwise, arg_i is evaluated under the context of t and the result is stored in val_i.
- $\texttt{case}\ v \,\texttt{\#}\, 0\ \texttt{of}\ \{c_0 \to E_0;\ \ldots;\ c_n \to E_n\}$: Let p be the current activation record when the evaluation of the `case` expression starts. The code produced starts by evaluating $v \,\texttt{\#}\, 0$, as described above, and the value returned is used to select the appropriate branch. Notice, however, that the evaluation of v has returned a pointer to an activation record t that corresponds to a tuple; this pointer is stored in the $nested$ field of p. The idea here is that other expressions of the form $v \,\texttt{\#}\, i$ that appear in the body of the `case` will be evaluated directly in the context of t. This saves us from evaluating the tuple v again and again, every time we encounter an expression of the form $v \,\texttt{\#}\, i$.
- $c(E_0, \ldots, E_{n-1})$: Constant c represents any of the usual constants or operators of a functional language (such as arithmetic constants and operators, if-then-else, and so on). In each case the produced code depends on the particular constant and its generation is pretty standard.

4.2 Optimizations

The implementation includes certain simple optimizations which focus on two issues: (a) allocating LARs on the stack whenever this is possible, and (b) making the evaluation of certain expressions less expensive. Further optimizations are possible, such as tail call elimination, but have not yet been implemented.

Escape Analysis. Constructing a lazy activation record when calling a function poses the following question: should it be allocated on the stack or on the heap? We use the following simple escape analysis scheme:

- Functions that return a ground value (such as an integer or a boolean) or a data type containing only nullary constructors allocate their LARs on the stack, popping them on return.
- Functions that may return constructors with parameters push their LARs on the heap.

Using this scheme, programs that do not make an extensive use of user-defined data types but which mainly use functions that manipulate and return numerical data, can benefit from the speed ensured by stack allocation.

Usage Analysis. Let x be the i-th formal parameter of a function f and assume that x appears only once in the body of f. Assume that a LAR is allocated for f and that the value of x under the current context has been computed. Then, there is no need to store the computed value of x in the field val_i of the LAR, since this value will not be demanded again (as x only appears once in the body of f).

A slightly more involved usage analysis also applies to the arguments of tuples. Assume that a LAR has been allocated due to the evaluation of a tuple. Assume also that we have just evaluated the i-th argument of tuple. Then, there is no need to store the result in val_i if this particular argument is accessed at most once in `case` branches that examine this tuple.

4.3 Garbage Collection

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 we expect that we will come up with a much better implementation of a garbage collector more suitable for the nature of LARs and the way they are used; this is one of the primary goals for our future research. The root set 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:[1]

- The Glasgow Haskell Compiler (GHC) which is the definitive compiler for Haskell.
- The Utrecht Haskell Compiler (UHC) which is implemented using attribute grammars and supports most features of Haskell 98 and Haskell 2010.
- The nhc98, which is a small, portable and standards compliant compiler for Haskell 98.
- The jhc, an experimental and fast compiler for Haskell, which has been implemented in order to test various optimizations for the language.

The comparison is based on a set of 13 benchmark programs, most of them standard benchmarks for lazy functional languages. 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 (tried to) disable strictness analysis from all compilers, so as to focus on the performance of genuine lazy implementations.

The performance results are depicted in Figure 8. The columns of this table correspond to the following:

- `GIC`: The generalized intensional compiler whose C output is compiled with `gcc`.
- `GIC-llvm`: The generalized intensional compiler whose C output is compiled using `llvm-gcc`, the front-end of `gcc` to the LLVM compiler.
- `GHC6`: The Glasgow Haskell Compiler, version 6.12.1, with full optimizations on.
- `GHC6-no-rewrite`: The Glasgow Haskell Compiler, version 6.12.1, invoked with flags `-fno-enable-rewrite-rules` and `-fno-spec-constr`. All other optimizations are enabled.
- `GHC7`: The Glasgow Haskell Compiler, version 7.4.1, with full optimizations on.
- `GHC7-no-rewrite`: The Glasgow Haskell Compiler, version 7.4.1, invoked with flags `-fno-enable-rewrite-rules` and `-fno-spec-constr`. All other optimizations are enabled.
- `NHC`: The nhc98 Haskell compiler, version 1.22, with full optimizations on.
- `UHC`: The uhc/ehc Haskell compiler, version 1.1.4, with full optimizations on.
- `JHC`: The jhc Haskell compiler, version 0.8.0, with full optimizations on.

The benchmarks appear to suggest the following conclusions:

- Compiling the target C code of the generalized intensional compiler with `llvm-gcc` appears to be quite more efficient than with standard `gcc`. In the following remarks, when we refer to the intensional compiler, we will mean the `GIC-llvm` one.
- 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.
- The intensional implementation runs generally faster than the `GHC6-no-rewrite` and the `GHC7-no-rewrite`. The intensional approach performs much better in the programs `collatz`, `primes`, and `queens-num`; however, it has a poorer performance in `naive-reverse` and `tree-sort`, a fact which deserves further investigation. The flags `-fno-spec-constr` and `-fno-enable-rewrite-rules` disable two quite aggressive optimizations of ghc, the latter seemingly related to deforestation. It is conceivable that such optimizations can potentially lead to analogous performance benefits for the intensional transformation.
- In certain programs (e.g., `ack` and `church`) `GHC6` performs better that `GHC7`! In almost all programs `GHC6-no-rewrite` performs much better than `GHC7-no-rewrite`! This has been reported (ticket #5888 in the GHC bug tracking system) and seems to be 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 most cases, the intensional implementation is slower than `JHC`, which is indeed a very fast system. It is possible that certain techniques that have been used in the development of jhc can be used to enhance the performance of the intensional approach, a fact that we would like to further investigate.
- In 9 out of the 13 benchmarks the intensional approach is faster than `NHC`, a mature and fully featured Haskell compiler.
- In general, the intensional implementation is much faster than `UHC`, a compiler which however does not target performance but is used for research on Haskell compilation.

[1] 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	GHC7-no-rewrite	GHC6	GHC6-no-rewrite	NHC	UHC	JHC
ack	2.47	1.25	0.62	2.87	0.48	0.85	6.18	40.03	0.05
church	3.55	2.09	0.61	5.77	0.55	0.88	11.58	68.37	0.17
collatz	0.69	0.41	1.07	3.43	2.66	2.87	84.28	46.90	0.16
digits_of_e1	2.30	2.09	0.77	6.19	1.74	1.62	60.71	75.29	–[1]
fast-reverse	3.03	1.95	1.74	1.83	1.82	1.80	1.35	9.41	–[2]
fib	1.35	1.12	0.50	4.42	0.51	0.73	10.43	55.55	0.17
naive-reverse	3.02	2.87	0.49	0.49	0.42	0.42	0.79	3.56	0.75
ntak	8.62	5.87	2.91	8.80	3.65	3.64	154.74	91.95	7.18
primes	2.55	1.58	2.19	18.32	2.30	2.40	172.45	173.81	0.73
queens-num	0.33	0.23	0.31	1.02	0.33	0.35	21.16	12.43	0.14
queens	3.92	3.24	0.44	5.44	0.48	0.77	27.17	123.98	0.82
quick-sort	3.18	2.77	1.92	2.08	1.90	1.90	1.51	5.42	8.58
tree-sort	2.19	1.97	0.39	0.48	0.33	0.38	0.91	6.58	0.72

[1] jhc compilation error, [2] jhc runtime error.

Figure 8. Runtime comparison for 13 benchmarks. Execution times are in seconds.

In general, we feel that the performance results are quite promising for the intensional approach, especially if we take into consideration the fact that our 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. There have been developed two main models of dataflow, the *pipelined* and the *tagged token*. In pipeline dataflow data arrive in nodes in a FIFO manner. Therefore, a node can not execute unless all the corresponding data-items arrive in the correct order. On the other hand, in tagged-token dataflow, the data-items are labeled with *tags* (or *contexts*) which provide a conceptual ordering of data items. A node can fire if it receives in its input arcs data-items that have the same tags. The tagged-token approach is more advanced since it obviates the need of data-items to arrive in a strictly pipelined way.

The majority of languages that were being 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 (see for example [1, 8]). The key idea of such implementations was to use tags in order 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 to do function dispatch [10, 17].

The similarity of coloring with the approach proposed in this paper, should be apparent by now. Tags correspond to the contexts (i.e., lists) in our technique. In particular, a list 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, the formalization and proof of correctness of the corresponding transformation and finally the 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 [18]; the implementation of Lucid was based on the original intensional transformation which was formalized through the use of *intensional logic* in A. Yaghi's Ph.D. dissertation [19]. The correctness of the original intensional transformation was established in [14]. 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 [11]. 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 tuple representation treating all constructs of these three types in a uniform way. GRIN was based on a strict first-order language, in contrast to our source language, FOL, 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 [6], Henderson and Morris [7], and Krivine [9], 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 correctness of the proposed technique, described the runtime system of an actual implementation and presented a performance comparison with existing implementations of Haskell. There are certain aspects of the technique that appear to require a more extensive investigation:

- The present implementation currently compiles only a fragment of Haskell. It is our intention to extend the implementation to cover the full language. The main missing parts are: (i) polymorphism and polymorphic data types, (ii) type classes, and (iii) various non essential omissions, e.g. `where` clauses and a greater variety of primitive data types. We do not expect any of these to be a significant problem. Especially type classes and polymorphic records can be added using the polymorphic defunctionalization of Pottier and Gauthier for System F [12]; in this case, we should add support for guarded existential types.
- 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 functional languages.

References

[1] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990. ISSN 0018-9340. doi: 10.1109/12.48862.

[2] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional Programming*. Oxford University Press, 1995.

[3] U. Boquist and T. Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In W. Kluge, editor, *Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 58–84. Springer Berlin / Heidelberg, 1997. doi: 10.1007/3-540-63237-9_19.

[4] A. Charalambidis, A. Grivas, N. S. Papaspyrou, and P. Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008. doi: 10.1007/s11786-008-0047-5.

[5] A. De La Encina and R. Peña. From natural semantics to C: A formal derivation of two STG machines. *Journal of Functional Programming*, 19(01):47–94, 2009. doi: 10.1017/S0956796808006746.

[6] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In *ICALP*, pages 257–284, 1976.

[7] P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811543.

[8] C. Kirkham, J. Gurd, and I. Watson. The manchester prototype dataflow computer. *CACM*, pages 34–52, 1985.

[9] J.-L. Krivine. Un interpréteur du lambda-calcul. URL `http://www.pps.univ-paris-diderot.fr/~krivine/articles/interprt.pdf`.

[10] K. Pingali. Lazy evaluation and the logic variable. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 560–572, New York, NY, USA, 1988. ACM. ISBN 0-89791-272-1. doi: 10.1145/55364.55419.

[11] J. Plaice and B. Mancilla. The practical uses of translucid. In *Proceedings of the first international workshop on Context-aware software technology and applications*, CASTA '09, pages 13–16, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-707-3. doi: 10.1145/1595768.1595774.

[12] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, 2006. ISSN 1388-3690. doi: 10.1007/s10990-006-8611-7.

[13] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[14] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *J. Funct. Program.*, 7:73–101, January 1997. ISSN 0956-7968. doi: 10.1017/S0956796897002633.

[15] P. Rondogiannis and W. W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, 1999.

[16] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[17] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical report, Cambridge, MA, USA, 1986.

[18] W. Wadge and E. A. Aschroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[19] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.