Math.comput.sci. 2 (2008), 123–141

1661-8270/010123-19, *published online* November 7, 2008
DOI 10.1007/s11786-008-0047-5

Mathematics in Computer Science

Efficient Intensional Implementation for Lazy Functional Languages

Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis

Abstract. The *intensional transformation* is a technique that can be used in order to eliminate higher-order functions from a functional program by introducing appropriate context-manipulation operators. The transformation can be applied to a significant class of higher-order programs and results in equivalent zero-order *intensional* programs that can be executed in a simple demand-driven way. Despite its simplicity, the transformation has never been seriously evaluated with respect to its efficiency and potential. Certain simple implementations of the technique have been performed, but questions regarding the merits of the method have remained inconclusive. In this paper we demonstrate that the transformation can be efficiently implemented by using what we call *lazy activation records*, namely activation records in which some entries are filled on-demand. An evaluation of our implementation demonstrates that the technique outperforms some of the most well-known functional programming systems, for the class of programs that can be transformed.

Mathematics Subject Classification (2000). 68N15, 68N18, 68N20.

Keywords. Implementation of functional languages, intensional logic, dataflow computation.

1. Introduction

The intensional transformation [8, 12, 13, 16] is a technique that systematically reduces the order of a source functional program by introducing appropriate context-manipulation operators. More specifically, given a functional program of order M, the technique in its first step reduces the program to a $(M-1)$-order *intensional* program; at the second step, the new program is reduced so as to become $(M-2)$-order, and so on. The final outcome is a program that contains only zero-order

This work has been partially supported by the University of Athens under the project "Kapodistrias" (grant no. 70/4/5827).

definitions but which also contains context-switching operators that work on M independent dimensions (all these will be further explained and illustrated in the coming sections). The resulting program can be evaluated in a simple demand-driven way, and this gives an interesting scheme for implementing lazy functional languages.

The transformation in its present form can be applied to functional languages that allow function names to be passed as parameters in function calls. At present it remains an open problem whether the technique can be generalized so as to be applicable to a more general class of higher-order functional programs, e.g. including partial application of functions (see the concluding section of [13] for a further discussion on this issue).

Despite the above restriction, the technique can be used to implement some useful and non-trivial functional programming languages. Experimental implementations of the transformation have been undertaken in the past. In [9] an experimental interpreter and a compiler are described, based on the intensional approach and appearing to be relatively efficient. However, both systems are based on a technique that extensively uses hashing at run-time, and this makes the performance of the system to be unpredictable in some cases. Moreover, this hashing-based approach requires a garbage-collection process that is not always very accurate, and also imposes a significant time overhead to the execution. Conclusively, the whole idea appeared to be simple and easy to implement, but not viable as a serious implementation framework. In [11] it is proposed that the runtime of the intensional technique can be adapted to work along the lines of traditional activation records. Of course, there is a significant difference, namely that each activation record should now contain information that would enable the context manipulation operators of the target program to work properly. This approach appeared to be much more promising than the hashing-based one. First-order programs appeared to run as efficiently as the best functional implementations available at that time; however, as the order of the programs increased, the performance of the technique seemed to degrade.

In this paper we refine and extend the technique of [11]. More specifically, we extend the activation records of [11] to contain additional information related to the higher-order functions of the initial program. The resulting implementation behaves in a much more uniform way than all the previous attempts. Moreover, our current system outperforms most of the existing compilers for lazy functional languages, for the class of programs that can be transformed.

2. The intensional transformation technique

The intensional transformation technique can be applied to higher-order functional programs with the following two restrictions:

1. There are no functions returning functions, i.e. the body of every function definition is zero-order.
2. Partial application of functions is not allowed, i.e. a function name can only be called with the right number of actual parameters (resulting in a zero-order value) or passed as a parameter to another function.

For example, the following (in Haskell syntax) is a legitimate program:

```
result   =  map inc [1,2,3]
map f l  =  if l==[] then [] else f (head l) : map f (tail l)
inc y    =  y+1
```

but the following is not an acceptable program, because of the partial application `add (s x)` in the definition of `f`:

```
result   =  f sq 4
f s x    =  if x<=1 then s x else f (add (s x)) (x-1)
sq y     =  y*y
add a b  =  a+b
```

Of course, there exist many programs of the second form that can be preprocessed so as to become programs of the first form, but this will not concern us any further here. In the rest of this paper, we only consider higher-order functional programs that obey the two aforementioned restrictions and, as partial application is not allowed, we use a Pascal-like notation with parentheses in function calls for clarity (e.g. `f(s,x)` instead of `f s x`). Moreover, for simplicity reasons, we assume that function parameters have unique names.

In the rest of this section we give an intuitive introduction to the intensional transformation technique. The main idea of the transformation is that an M-order functional program can first be transformed into an $(M-1)$-order intensional program, i.e. a program that uses functions of order at most $(M-1)$ and which is enriched with context manipulation operators. The same procedure can then be repeated for the new program, until we finally get a zero-order intensional program. As the transformation consists of a number of steps, we use a different set of operators for each step. For the first step we use the operators $\texttt{case}^{M-1}$, $\texttt{actuals}_i^{M-1}$ and $\texttt{call}_i^{M-1}$, where i ranges over the natural numbers. For the second step we use $\texttt{case}^{M-2}$, $\texttt{actuals}_i^{M-2}$ and $\texttt{call}_i^{M-2}$, and so on. The semantics of these operators will be described shortly.

The final program that results from the transformation is zero-order, i.e. it consists solely of nullary variable definitions. The value of this program can be computed by evaluating its top-level variable `result` with respect to a *context* (or *tag*). For an initial program of order M, a tag is an M-sequence (a sequence of length M) of lists, where each list corresponds to a different order of the program. The operators we mentioned above operate on these M-dimensional tags. Before we proceed with the details of the execution, we give an example of the transformation.

2.1. The transformation algorithm

Consider the following simple second-order program:

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{f(inc, 2)} \\
\texttt{f(g, y)} & = & \texttt{g(y) * apply(g, y)} \\
\texttt{apply(h, x)} & = & \texttt{h(x)} \\
\texttt{inc(a)} & = & \texttt{a+1}
\end{array}
$$

By an inspection of the program, it is easy to see that the functions `f` and `apply` are second-order. The intensional transformation in its first step eliminates all first-order parameters: the first argument of `f` and the first argument of `apply` disappear from the calling sites and reappear as additional definitions. Furthermore, as witnesses of the disappearing parameters, operators of the form $\texttt{call}_i^1$ appear in the calling sites, where the superscript 1 is the order of the parameters that disappeared at this step and the subscript i is simply a counter, one for each function of the source program.

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{call}_0^1\texttt{(f)(2)} \\
\texttt{f(y)} & = & \texttt{g(y) * call}_0^1\texttt{(apply)(y)} \\
\texttt{apply(x)} & = & \texttt{h(x)} \\
\texttt{inc(a)} & = & \texttt{a+1} \\
\texttt{g} & = & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(inc))} \\
\texttt{h} & = & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(g))}
\end{array}
$$

In the above program the definitions of `g` and `h` contain the actual parameters that were removed from the calling sites, each prefixed by an intensional operator of the form $\texttt{actuals}_i^1$. Both definitions are equations between function expressions. We can change this by introducing new parameters, `z` and `w`:

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{call}_0^1\texttt{(f)(2)} \\
\texttt{f(y)} & = & \texttt{g(y) * call}_0^1\texttt{(apply)(y)} \\
\texttt{apply(x)} & = & \texttt{h(x)} \\
\texttt{inc(a)} & = & \texttt{a+1} \\
\texttt{g(z)} & = & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(inc)(z))} \\
\texttt{h(w)} & = & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(g)(w))}
\end{array}
$$

After this step of the transformation, all functions are now first-order (they have only zero-order parameters). A non-standard aspect of this new program is the existence of certain function calls of the form $\mathbf{q(f)(E}_0, \ldots \mathbf{E}_{n-1}\mathbf{)}$ where $\mathbf{q}$ is an *intensional* operator, e.g. $\texttt{call}_0^1\texttt{(f)(2)}$, or $\texttt{actuals}_0^1\texttt{(inc)(z)}$. Such calls will receive a special treatment in the next step of the transformation.

We can now perform the second and final step of the transformation that will remove all zero-order parameters and result in a zero-order intensional program. We proceed as before, the main difference being that we use a new dimension and corresponding new operators. Notice also below the use of the "$\cdot$" syntactic composition operator, which is introduced for notational convenience. In particular,

an expression of the form $\mathbf{q}_1 \cdot \mathbf{q}_2\mathtt{(f)}$ will be considered equivalent to $\mathbf{q}_1\mathtt{(}\mathbf{q}_2\mathtt{(f))}$, i.e. the composition operator binds stronger than function application.

$$
\begin{array}{lcl}
\mathtt{result} & = & \mathtt{call}_0^1 \cdot \mathtt{call}_0^0\mathtt{(f)} \\
\mathtt{f} & = & \mathtt{call}_0^0\mathtt{(g)} \ \mathtt{*} \ \mathtt{call}_0^1 \cdot \mathtt{call}_0^0\mathtt{(apply)} \\
\mathtt{apply} & = & \mathtt{call}_0^0\mathtt{(h)} \\
\mathtt{inc} & = & \mathtt{a+1} \\
\mathtt{g} & = & \mathtt{case}^1\mathtt{(actuals}_0^1 \cdot \mathtt{call}_0^0\mathtt{(inc))} \\
\mathtt{h} & = & \mathtt{case}^1\mathtt{(actuals}_0^1 \cdot \mathtt{call}_1^0\mathtt{(g))} \\
\mathtt{y} & = & \mathtt{case}^0\mathtt{(actuals}_0^0 \cdot \mathtt{actuals}_0^1\mathtt{(2))} \\
\mathtt{x} & = & \mathtt{case}^0\mathtt{(actuals}_0^0\mathtt{(y))} \\
\mathtt{a} & = & \mathtt{case}^0\mathtt{(actuals}_0^0 \cdot \mathtt{call}_0^1\mathtt{(z))} \\
\mathtt{z} & = & \mathtt{case}^0\mathtt{(actuals}_0^0\mathtt{(y),\ actuals}_1^0 \cdot \mathtt{call}_0^1\mathtt{(w))} \\
\mathtt{w} & = & \mathtt{case}^0\mathtt{(actuals}_0^0 \cdot \mathtt{actuals}_0^1\mathtt{(x))}
\end{array}
$$

The transformation is similar to the one that took place in the first step, the main difference being the treatment of calls of the form $\mathbf{q}\mathtt{(f)(}\mathbf{E}_0, \ldots \mathbf{E}_{n-1}\mathtt{)}$. Consider as an example the (generalized) function call $\mathtt{call}_0^1\mathtt{(f)(2)}$ and notice the new expression $\mathtt{actuals}_0^0 \cdot \mathtt{actuals}_0^1\mathtt{(2)}$ that appears in the final program corresponding to the actual parameter 2. The new aspect here is the appearance of the operator $\mathtt{actuals}_0^1$, which we will call the *inverse* of the operator $\mathtt{call}_0^1$ that existed in the initial call. In general, the inverse of $\mathtt{call}_i^m$ is $\mathtt{actuals}_i^m$ and vice-versa.

A thorough description of the transformation algorithm is given in [13]. Informally, it consists of repeating the following steps until the program becomes zero-order. For each function $\mathbf{f}$ of the current highest order m:

1. Number the textual occurrences of calls to $\mathbf{f}$ in the program, starting at 0.
2. Remove from the i-th call to $\mathbf{f}$ all the actual parameters of order $m-1$. Prefix the call to $\mathbf{f}$ with $\mathtt{call}_i^{m-1}$.
3. Remove from the definition of $\mathbf{f}$ the formal parameters of order $m-1$.
4. For every formal parameter $\mathbf{x}$ of $\mathbf{f}$ that was eliminated, introduce a $\mathtt{case}^{m-1}$ definition with as many alternatives as there are calls to $\mathbf{f}$ in the program. More specifically, the i-th argument of $\mathtt{case}^{m-1}$ corresponds to the i-th call to $\mathbf{f}$ in the program, and is an expression starting with $\mathtt{actuals}_i^{m-1}$. Moreover, if the particular call to $\mathbf{f}$ has form $\mathbf{Q}\mathtt{(f)(}\mathbf{E}_0, \ldots \mathbf{E}_{n-1}\mathtt{)}$, where $\mathbf{Q}$ is the syntactic composition of a number of intensional operators, the inverse of $\mathbf{Q}$ must be taken into consideration when creating the subexpressions of $\mathtt{case}^{m-1}$.

2.2. Evaluating the transformed program

Given a source functional program of order M, the execution model for the final zero-order program that results from the transformation requires tags to be M-sequences of lists of natural numbers, where each list corresponds to one step in the transformation. We will use the notation $\langle w_0, \ldots, w_{M-1} \rangle$ to denote a tag. The evaluation of a program starts with the empty tag: an M-sequence of empty lists.

The operators $\texttt{call}_i^m$ and $\texttt{actuals}_i^m$ can now be thought of as operations on tags. The semantics of $\texttt{call}_i^m$ can be described as follows: given a tag, m is used in order to select the corresponding list from the tag. The list is then prefixed with i and returned to the tag. On the other hand, $\texttt{actuals}_i^m$ takes from the tag the list corresponding to m, verifies that the head of the list is equal to i and returns the tail of the list to the tag. The $\texttt{case}^m$ construct that appears on newly introduced definitions selects the alternative to be evaluated by inspecting the head of the list that corresponds to m. More formally, the semantic equations for the intensional operators that appear in the transformed (zero-order) programs are:

$$\begin{array}{lcl}
\texttt{call}_i^m(a)\ \langle w_0, \ldots w_m, \ldots w_{M-1}\rangle & = & a\ \langle w_0, \ldots i : w_m, \ldots w_{M-1}\rangle \\
\texttt{actuals}_i^m(a)\ \langle w_0, \ldots i : w_m, \ldots w_{M-1}\rangle & = & a\ \langle w_0, \ldots w_m, \ldots w_{M-1}\rangle \\
\texttt{case}^m(a_0, \ldots a_{n-1})\ \langle w_0, \ldots i : w_m, \ldots w_{M-1}\rangle & = & a_i\ \langle w_0, \ldots i : w_m, \ldots w_{M-1}\rangle\,.
\end{array}$$

Notice that in the case of the $\texttt{actuals}_i^m$ operator, the semantic equation does not specify what happens if the check made by the operator fails. The result in this case is undefined. However, the test performed by $\texttt{actuals}_i^m$ never fails in the case of programs generated by the transformation.

The final zero-order programs that result from the transformation can be executed using an *EVAL* function, which takes an expression and a tag and returns the result of evaluating this expression under this tag. One can think of *EVAL* as a simple interpreter that works by following the semantic equations of the intensional operators given above. Moreover, *EVAL* also performs a simple form of substitution: every time it needs to evaluate a nullary variable of the program under a specific context, it simply replaces the variable with its defining expression and continues the evaluation. Execution of a program starts by demanding the value of `result` under the empty tag. A complete denotational semantics for the zero-order intensional language that can serve as a basis for *EVAL* is given in [13]. We now demonstrate the execution of the previous program:

$$\begin{array}{ll}
 & \mathit{EVAL}(\texttt{result},\ \langle[],[]\rangle) \\
= & \mathit{EVAL}(\texttt{call}_0^1 \cdot \texttt{call}_0^0\texttt{(f)},\ \langle[],[]\rangle) \\
= & \mathit{EVAL}(\texttt{f},\ \langle[0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{call}_0^0\texttt{(g)} * \texttt{call}_0^1 \cdot \texttt{call}_0^0\texttt{(apply)},\ \langle[0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{call}_0^0\texttt{(g)},\ \langle[0],[0]\rangle) * \mathit{EVAL}(\texttt{call}_0^1 \cdot \texttt{call}_0^0\texttt{(apply)},\ \langle[0],[0]\rangle)
\end{array}$$

Now, the overall result can be computed by evaluating independently the two expressions and then multiplying the two results. The evaluation of the first expression proceeds as follows:

$$\begin{array}{ll}
 & \mathit{EVAL}(\texttt{call}_0^0\texttt{(g)},\ \langle[0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{g},\ \langle[0,0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{case}^1\texttt{(actuals}_0^1 \cdot \texttt{call}_0^0\texttt{(inc))},\ \langle[0,0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{actuals}_0^1 \cdot \texttt{call}_0^0\texttt{(inc)},\ \langle[0,0],[0]\rangle) \\
= & \mathit{EVAL}(\texttt{inc},\ \langle[0,0,0],[]\rangle)
\end{array}$$

$$
\begin{aligned}
&= EVAL(\texttt{a+1},\ \langle[0,0,0],[]\rangle)\\
&= EVAL(\texttt{a},\ \langle[0,0,0],[]\rangle)+1\\
&= EVAL(\texttt{case}^0\texttt{(actuals}_0^0\cdot\texttt{call}_0^1\texttt{(z))},\ \langle[0,0,0],[]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\cdot\texttt{call}_0^1\texttt{(z)},\ \langle[0,0,0],[]\rangle)+1\\
&= EVAL(\texttt{z},\ \langle[0,0],[0]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\texttt{(y)},\ \langle[0,0],[0]\rangle)+1\\
&= EVAL(\texttt{y},\ \langle[0],[0]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\cdot\texttt{actuals}_0^1\texttt{(2)},\ \langle[0],[0]\rangle)+1\\
&= EVAL(\texttt{2},\ \langle[],[]\rangle)+1\\
&= 2+1\\
&= 3
\end{aligned}
$$

The second expression can be evaluated as follows:

$$
\begin{aligned}
&EVAL(\texttt{call}_0^1\cdot\texttt{call}_0^0\texttt{(apply)},\ \langle[0],[0]\rangle)\\
&= EVAL(\texttt{apply},\ \langle[0,0],[0,0]\rangle)\\
&= EVAL(\texttt{call}_0^0\texttt{(h)},\ \langle[0,0],[0,0]\rangle)\\
&= EVAL(\texttt{h},\ \langle[0,0,0],[0,0]\rangle)\\
&= EVAL(\texttt{case}^1\texttt{(actuals}_0^1\cdot\texttt{call}_1^0\texttt{(g))},\ \langle[0,0,0],[0,0]\rangle)\\
&= EVAL(\texttt{actuals}_0^1\cdot\texttt{call}_1^0\texttt{(g)},\ \langle[0,0,0],[0,0]\rangle)\\
&= EVAL(\texttt{g},\ \langle[1,0,0,0],[0]\rangle)\\
&= EVAL(\texttt{case}^1\texttt{(actuals}_0^1\cdot\texttt{call}_0^0\texttt{(inc))},\ \langle[1,0,0,0],[0]\rangle)\\
&= EVAL(\texttt{actuals}_0^1\cdot\texttt{call}_0^0\texttt{(inc)},\ \langle[1,0,0,0],[0]\rangle)\\
&= EVAL(\texttt{inc},\ \langle[0,1,0,0,0],[]\rangle)\\
&= EVAL(\texttt{a+1},\ \langle[0,1,0,0,0],[]\rangle)\\
&= EVAL(\texttt{a},\ \langle[0,1,0,0,0],[]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\cdot\texttt{call}_0^1\texttt{(z)},\ \langle[0,1,0,0,0],[]\rangle)+1\\
&= EVAL(\texttt{z},\ \langle[1,0,0,0],[0]\rangle)+1\\
&= EVAL(\texttt{actuals}_1^0\cdot\texttt{call}_0^1\texttt{(w)},\ \langle[1,0,0,0],[0]\rangle)+1\\
&= EVAL(\texttt{w},\ \langle[0,0,0],[0,0]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\texttt{(x)},\ \langle[0,0,0],[0,0]\rangle)+1\\
&= EVAL(\texttt{x},\ \langle[0,0],[0,0]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\cdot\texttt{actuals}_0^1\texttt{(y)},\ \langle[0,0],[0,0]\rangle)+1\\
&= EVAL(\texttt{y},\ \langle[0],[0]\rangle)+1\\
&= EVAL(\texttt{actuals}_0^0\cdot\texttt{actuals}_0^1\texttt{(2)},\ \langle[0],[0]\rangle)+1\\
&= EVAL(\texttt{2},\ \langle[],[]\rangle)+1\\
&= 2+1\\
&= 3
\end{aligned}
$$

The final value of `result` is therefore $3 * 3 = 9$.

3. The new execution model

The execution model of Section 2.2, based on the denotational semantics of the zero-order intensional language [13], is quite simple to understand and implement. However, the efficiency of a naïve implementation is doomed to be poor. Values

that have been computed are often demanded again; an efficient implementation must memorize such values and not recompute them.

For example, consider the execution of the program in Section 2.2. It can be argued that the evaluation process takes several unecessary steps. Some obvious recomputations occur, e.g. the variable `y` is demanded twice under the tag $\langle[0],[0]\rangle$ and computed twice, evidently yielding the same value. A more subtle case is the variable `g`, a second-order formal parameter in the source program, which is demanded under the tags $\langle[0,0],[0]\rangle$ and $\langle[1,0,0,0],[0]\rangle$. Its evaluation inspects the second list in the tag, which is the same in both cases, and after some steps leads to the evaluation of `inc`. One may argue that an efficient implementation should memorize the fact that the evaluation of `g` under a tag that has [0] as a second list leads to the evaluation of `inc` and therefore avoid the intermediate steps. Both recomputation patterns tend to become very frequent in larger and more complex programs.

In this section, we propose an evaluation model that can handle these issues. In this model, tags hold intermediate information (e.g. already computed values) which can be used to avoid recomputations. We first define a variation of the zero-order intensional language. Although modifications are required in the transformation proposed in [13] to produce final programs in this new target language, these modifications are minor. Subsequently, we define a denotational semantics for the new zero-order intensional language, which amounts to a new evaluation model.

3.1. The modified zero-order intensional language

The zero-order intensional language that we use as the target of the transformation is a variant of the *NVIL* language used in [13], which is the one that we used in Section 2. The new syntax, however, stores additional information that will be used to improve the performance of the execution model. A program in the modified zero-order intensional language can be translated to *NVIL* in a straightforward way. However, the inverse cannot be achieved without additional information that can only be obtained during the transformation.

Definition 1 (Syntax). The syntax of the zero-order intensional language is defined recursively as follows, where x is a variable identifier, c is a constant and $m, n \in \mathbb{N}$.

p	$::=$	$d_0 \ \ldots \ d_{n-1}$	*program*
d	$::=$	$x = b$	*definition*
b	$::=$	$\sigma\alpha(e)$	*body of definition*
α	$::=$	$\epsilon \mid \texttt{actuals}^m \cdot \alpha$	*sequence of "actuals" operators*
σ	$::=$	$\epsilon \mid \texttt{save}_n^m$	*memorization operator*
e	$::=$	$c(e_0, \ldots e_{n-1}) \mid \gamma(z)$	*expression*
γ	$::=$	$\epsilon \mid \texttt{call}^m_{x_0,\ldots x_{n-1}} \cdot \gamma$	*sequence of "call" operators*
z	$::=$	$x \mid \texttt{arg}_n^m$	*variable*

Notice that a variable z can be either a variable identifier or a function argument, which is denoted by $\texttt{arg}_n^m$. Variable identifiers are used for top-level variables, i.e. variables that have definitions in the source program. An argument of the form $\texttt{arg}_n^m$ stands for the current function's n-th formal parameter of order m, as it was in the source higher-order program. The body of an intensional definition contains a *memorization operator*, which is either empty or of the form $\texttt{save}_n^m$. This operator specifies whether the evaluation of the body must be followed by memorization of the result and where to; its purpose will become clear later. As an abuse of notation in favour of simplicity, the parentheses will be omitted after an empty sequence of `call` or `actuals` operators. Also, special constants (e.g. integers, arithmetic operators, etc.) will be written as is ordinary in programming languages.

The most striking differences between the modified zero-order intensional language and *NVIL* are the $\texttt{arg}_n^m$ arguments, the lack of `case` construct, the lack of subscript for `actuals` operators and the presence of a finite sequence of variable identifiers instead of an integer counter as the subscript for `call` operators. To illustrate how all these work, let us consider again the simple second-order program that we used as an example in Section 2.1.

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{f(inc,2)} \\
\texttt{f(g,y)} & = & \texttt{g(y) * apply(g,y)} \\
\texttt{apply(h,x)} & = & \texttt{h(x)} \\
\texttt{inc(a)} & = & \texttt{a+1}
\end{array}
$$

For purposes of comparison, we copy here the equivalent *NVIL* program from Section 2.1, as produced by the intensional transformation [13].

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{call}_0^1 \cdot \texttt{call}_0^0\texttt{(f)} \\
\texttt{f} & = & \texttt{call}_0^0\texttt{(g) * call}_0^1 \cdot \texttt{call}_0^0\texttt{(apply)} \\
\texttt{apply} & = & \texttt{call}_0^0\texttt{(h)} \\
\texttt{inc} & = & \texttt{a+1} \\
\texttt{g} & = & \texttt{case}^1\texttt{(actuals}_0^1 \cdot \texttt{call}_0^0\texttt{(inc))} \\
\texttt{h} & = & \texttt{case}^1\texttt{(actuals}_0^1 \cdot \texttt{call}_1^0\texttt{(g))} \\
\texttt{y} & = & \texttt{case}^0\texttt{(actuals}_0^0 \cdot \texttt{actuals}_0^1\texttt{(2))} \\
\texttt{x} & = & \texttt{case}^0\texttt{(actuals}_0^0\texttt{(y))} \\
\texttt{a} & = & \texttt{case}^0\texttt{(actuals}_0^0 \cdot \texttt{call}_0^1\texttt{(z))} \\
\texttt{z} & = & \texttt{case}^0\texttt{(actuals}_0^0\texttt{(y), actuals}_1^0 \cdot \texttt{call}_0^1\texttt{(w))} \\
\texttt{w} & = & \texttt{case}^0\texttt{(actuals}_0^0 \cdot \texttt{actuals}_0^1\texttt{(x))}
\end{array}
$$

The modified transformation produces the following program in the modified zero-order intensional language.

$$
\begin{array}{lcl}
\texttt{result} & = & \texttt{call}_{\texttt{g}_0}^1 \cdot \texttt{call}_{\texttt{y}_0}^0\texttt{(f)} \\
\texttt{f} & = & \texttt{call}_{\texttt{z}_0}^0\texttt{(arg}_0^1\texttt{)} \texttt{ * call}_{\texttt{h}_0}^1 \cdot \texttt{call}_{\texttt{x}_0}^0\texttt{(apply)} \\
\texttt{apply} & = & \texttt{call}_{\texttt{w}_0}^0\texttt{(arg}_0^1\texttt{)} \\
\texttt{inc} & = & \texttt{arg}_0^0 \texttt{ + 1}
\end{array}
$$

$$
\begin{aligned}
\mathtt{g}_0 &= \mathtt{save}_0^1\ \mathtt{actuals}^1 \cdot \mathtt{call}_{\mathtt{a}_0}^0(\mathtt{inc}) \\
\mathtt{h}_0 &= \mathtt{save}_0^1\ \mathtt{actuals}^1 \cdot \mathtt{call}_{\mathtt{z}_1}^0(\mathtt{arg}_0^1) \\
\mathtt{y}_0 &= \mathtt{save}_0^0\ \mathtt{actuals}^0 \cdot \mathtt{actuals}^1(2) \\
\mathtt{x}_0 &= \mathtt{save}_0^0\ \mathtt{actuals}^0(\mathtt{arg}_0^0) \\
\mathtt{a}_0 &= \mathtt{save}_0^0\ \mathtt{actuals}^0 \cdot \mathtt{call}_{\mathtt{g}_0}^1(\mathtt{arg}_0^0) \\
\mathtt{z}_0 &= \mathtt{save}_0^0\ \mathtt{actuals}^0(\mathtt{arg}_0^0) \\
\mathtt{z}_1 &= \mathtt{save}_0^0\ \mathtt{actuals}^0 \cdot \mathtt{call}_{\mathtt{h}_0}^1(\mathtt{arg}_0^0) \\
\mathtt{w}_0 &= \mathtt{save}_0^0\ \mathtt{actuals}^0 \cdot \mathtt{actuals}^1(\mathtt{arg}_0^0)
\end{aligned}
$$

By comparing the two programs, we first notice that variable `g` has been substituted with $\mathtt{arg}_0^1$. This happens because `g` was `f`'s first formal parameter of order 1 (remember that subscripted indices start with 0). Similarly, `x` has been substituted with $\mathtt{arg}_0^0$ as it was `apply`'s first formal parameter of order 0. We also notice that the `case` definitions have been removed. Each alternative has been introduced as a new definition, e.g. $\mathtt{z}_0$ and $\mathtt{z}_1$ instead of a single `z`. At the calling sites, the subscript of the `call` operator now contains a list of the formal parameters that were removed, each tagged with the original subscript, e.g. $\mathtt{call}_{\mathtt{g}_0}^1$ when the formal parameter `g` (of order 1) was removed from the first call to `f`. All elements of a list that is a subscript of `call` must appear in the left-hand side of newly introduced definitions. The subscripts of the `actuals` operators have been dropped and memorization operators have been added to all newly introduced definitions. For example, in the definition for $\mathtt{h}_0$ the memorization operator is $\mathtt{save}_0^1$ because `h` was `apply`'s first formal parameter of order 1.

3.2. Semantics of execution

The modified zero-order intensional language can be given a denotational semantics in a similar way as in [13]. In this paper we will define an *EVAL* function (or, more precisely, a family of *EVAL* functions) similar to the one described in Section 2.2.

We first introduce a new notion of *tags* which support memorization. A tag contains essentially all information that is available during execution, including memorized computation results. This information changes during the evaluation process. It is possible that the evaluation of expression e_1 affects the way in which another expression e_2 is evaluated, even if the two are unrelated, because of memorization. Therefore, tags must be treated as *states* in the new semantics, not as *environments* as in [13]. If **Expr** is the syntactic domain of expressions, **W** is the semantic domain of tags and **Val** the semantic domain of zero-order values (integer numbers, booleans, etc.), the signature of *EVAL* must be:

$$EVAL \quad : \quad \mathbf{Expr} \times \mathbf{W} \to \mathbf{Val} \times \mathbf{W}$$

Similar functions are needed for the syntactic domains of variables and bodies of definitions. As an abuse of notation, we use the name *EVAL* for them too.

A tag is an M-sequence of *stacks*. Each stack contains *records*, which are finite sequences of *arguments*. An argument can be either of the form $value(v)$, where v is an element of **Val**, or of the form $name(x)$, where x is a variable identifier.

Intuitively, a record contains a sequence of the actual parameters that correspond to a function call; all parameters of order 0 are put together in a record that is placed in the first stack of the tag, and so on for parameters of higher order. When a record is first built, all its arguments are of the form $name(x)$. Their values are not yet known. When a zero-order argument is later computed, yielding the value v, it is replaced by $value(v)$ so as not to be recomputed in the future. On the other hand, when a higher-order argument is later computed, it may be replaced by another argument of the form $name(x)$ to avoid redundant evaluation steps.

A revised definition for $EVAL$ must redefine the semantics of intensional operators. Before going so far, we start with the easy cases. Assuming that there is a definition $x = b$ in the program, evaluating a variable identifier leads to evaluating the body of the definition.

$$EVAL(x,t) \ = EVAL(b,t)$$

Also, assuming that a semantic function $[\![c]\!]$ exists for each constant c of arity n, the semantics of constants is straightforward. Evaluation of a constant's operands is performed in left-to-right order.

$$\begin{array}{l} EVAL(c(e_0,\ldots e_{n-1}),t) \ = \\ \quad \textbf{let} \quad (v_0,t_0) = EVAL(e_0,t) \\ \qquad\quad (v_1,t_1) = EVAL(e_1,t_0) \\ \qquad\quad \ldots \\ \qquad\quad (v_{n-1},t_{n-1}) = EVAL(e_{n-1},t_{n-2}) \\ \quad \textbf{in} \quad ([\![c]\!](v_0,\ldots v_{n-1}),t_{n-1}) \end{array}$$

Function arguments are the next easiest. To evaluate $\texttt{arg}_n^m$, we take the record that is on top of the m-th stack of the tag. Assuming that this is a sequence of $K \geq n$ arguments (the transformation guarantees that), we inspect the n-th argument in that sequence. If it is a value, it is immediately reused. Otherwise, if it is a variable identifier, we evaluate it as previously.

$$\begin{array}{l} EVAL(\texttt{arg}_n^m, \langle w_0,\ldots w_m,\ldots w_{M-1}\rangle) \ = \\ \quad \textbf{let} \quad \langle a_0,\ldots a_n,\ldots a_{K-1}\rangle = top(w_m) \\ \quad \textbf{in} \quad \textbf{case } a_n \textbf{ of} \\ \qquad\qquad value(v) \quad \rightarrow \quad (v, \langle w_0,\ldots w_m,\ldots w_{M-1}\rangle) \\ \qquad\qquad name(x) \quad \rightarrow \quad EVAL(x, \langle w_0,\ldots w_m,\ldots w_{M-1}\rangle) \end{array}$$

In the semantics of a $\texttt{call}^m$ operator, we create a new record that contains the arguments that appear as its subscript. We put this record on top of the m-th stack of the tag and then evaluate the operand of the $\texttt{call}^m$ operator under the new tag. When its evaluation is complete, we remove the top record from the m-th stack of the tag.

$$\begin{array}{l} EVAL(\texttt{call}^m_{x_0,\ldots x_{n-1}}(e), \langle w_0,\ldots w_m,\ldots w_{M-1}\rangle) \ = \\ \quad \textbf{let} \quad r = \langle name(x_0),\ldots name(x_{n-1})\rangle \\ \qquad\quad (v, \langle w'_0,\ldots w'_m,\ldots w'_{M-1}\rangle) \\ \qquad\quad = EVAL(e, \langle w_0,\ldots push(r,w_m),\ldots w_{M-1}\rangle) \\ \quad \textbf{in} \quad (v, \langle w'_0,\ldots pop(w'_m),\ldots w'_{M-1}\rangle) \end{array}$$

The semantics of $\texttt{actuals}^m$ operators that appear in the body of definitions is, in some sense, the inverse of that for $\texttt{call}^m$ operators. We remove the top record from the m-th stack of the tag, but we store it so as to be able to put it back later. We evaluate the operand of the `actuals` operator (the remaining of a body of definition, possibly containing more `actuals`) under the new tag. When its evaluation is complete, we put back the stored record on top of the m-th stack of the tag.

$$
\begin{array}{l}
EVAL(\texttt{actuals}^m(b), \langle w_0, \ldots w_m, \ldots w_{M-1}\rangle) \;=\\
\quad \textbf{let} \quad r = top(w_m)\\
\qquad\quad (v, \langle w'_0, \ldots w'_m, \ldots w'_{M-1}\rangle) = EVAL(b, \langle w_0, \ldots pop(w_m), \ldots w_{M-1}\rangle)\\
\quad \textbf{in} \quad (v, \langle w'_0, \ldots push(r, w'_m), \ldots w'_{M-1}\rangle)
\end{array}
$$

The most complicated part of the semantics is memorization. First, notice that there is always an $\texttt{actuals}^m$ operator following a $\texttt{save}^m_n$ memorization operator. The evaluation of a body of the form $\texttt{save}^m_n(\texttt{actuals}^m(b))$ is similar to that of $\texttt{actuals}^m(b)$. The difference is that the record that is put back at the end on top of the m-th stack of the tag is not (necessarily) the same as the one that was removed in the beginning; the two may differ in the position $\texttt{arg}^m_n$, which is updated as a result of evaluating the body. If $m = 0$, i.e. the argument to be updated is zero-order, then the value that was computed replaces the previous contents of $\texttt{arg}^m_n$. Otherwise, if the argument is higher-order, the updated argument depends on the syntactic form of b. If b ends with a variable identifier, then this replaces the previous contents of $\texttt{arg}^m_n$. On the other hand, if it ends with a (possibly different) function argument, this argument in the tag that resulted from the evaluation of e replaces the previous contents of $\texttt{arg}^m_n$.

$$
\begin{array}{l}
EVAL(\texttt{save}^m_n(\texttt{actuals}^m(b)), \langle w_0, \ldots w_m, \ldots w_{M-1}\rangle) \;=\\
\quad \textbf{let} \quad \langle a_0, \ldots a_n, \ldots a_{K-1}\rangle = top(w_m)\\
\qquad\quad (v, \langle w'_0, \ldots w'_m, \ldots w'_{M-1}\rangle) = EVAL(b, \langle w_0, \ldots pop(w_m), \ldots w_{M-1}\rangle)\\
\qquad\quad a'_n = \;\; \textbf{if } m = 0 \textbf{ then}\\
\qquad\qquad\qquad\quad value(v)\\
\qquad\qquad\qquad \textbf{else if } b \text{ is of the form } \alpha(\gamma(x)) \textbf{ then}\\
\qquad\qquad\qquad\quad name(x)\\
\qquad\qquad\qquad \textbf{else if } b \text{ is of the form } \alpha(\gamma(\texttt{arg}^{m'}_{n'})) \textbf{ then}\\
\qquad\qquad\qquad\qquad \textbf{let} \quad \langle c_0, \ldots c_{n'}, \ldots c_{L-1}\rangle = top(w'_{m'})\\
\qquad\qquad\qquad\qquad \textbf{in} \quad c_{n'}\\
\qquad\quad r = \langle a_0, \ldots a'_n, \ldots a_{K-1}\rangle\\
\quad \textbf{in} \quad (v, \langle w'_0, \ldots push(r, w'_m), \ldots w'_{M-1}\rangle)
\end{array}
$$

4. Implementation on stock hardware

The success of an execution model depends primarily on how efficiently it can be implemented. In this section we outline a quite efficient implementation of the model that was presented in Section 3 on stock hardware. Our implementation presents similarities to the traditional use of activation records, which are used

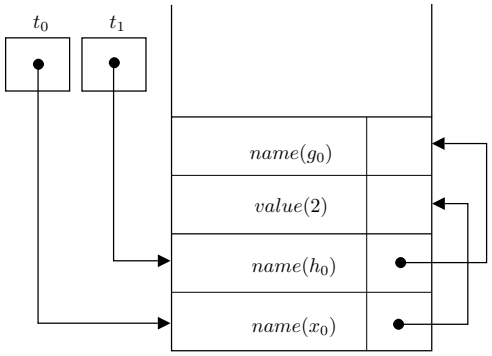

FIGURE 1. An example of lazy activation records.

to hold a function's actual parameters and are organized as a stack in the computer's memory. In our approach, which is an extension of the scheme proposed in [11], there are two main differences: (i) activation records are *lazy*, i.e. the actual parameters are filled in upon demand; and (ii) the construction and destruction of activation records is controlled by the presence of intensional operators in the zero-order program.

The most crucial and at the same time tricky issue in the implementation of the execution model is the representation of tags, records and arguments and the implementation of intensional operators. We represent stacks of records by linked lists, i.e. we add to each record a pointer to the record that is immediately below on the stack. We represent a tag by a set of M pointers $\{t_0, \ldots t_{M-1}\}$ that point to the first elements of the lists representing the stacks. For the representation of a record's arguments, we use one bit to distinguish between a $value(v)$ and a $name(x)$; the representation of v or x follows. A variable identifier x can be represented by a pointer to the code that implements the corresponding definition.

There is an obvious similarity between the semantics of `call` and `actuals` operators and traditional stack-based activation records: a `call` can be viewed as putting a record on the stack, evaluating the "function body" and then removing the record from the stack. On the other hand, `actuals` ignores the record that is on top of the stack and switches the context to the one immediately below, in order to evaluate its body; then it switches the context back to what it was. This remark allows us to group the records of successive `call`s, in the sense of the execution model of Section 3, in *lazy activation records* that we place on a stack in the computer's memory. The form of some lazy activation records that appear during execution of the example in Section 2.2 is shown in Figure 1.

As execution begins, the first lazy activation record that is constructed is the one for `f`, with `g` and `y` as arguments (the top one in the figure). After some time, `y` is evaluated giving the value 2, which replaces the argument on the stack. When `apply` is called, with arguments `h` and `x`, a new lazy activation record is

placed on the stack (the stack grows downward in the figure). Notice at this point of execution the pointers $\langle t_0, t_1 \rangle$ in the current tag, pointing to two separate linked lists, one for each dimension.

5. Performance comparison and related work

In this section we briefly summarize other implementation techniques for higher-order functional programming languages and compare our implementation with some other known implementations in terms of performance.

The traditional implementation technique for lazy functional languages is based on graph reduction [7]. Great effort has been made over the years for efficient implementations [5] and optimizations. We consider the well-known compilers for Haskell (ghc and hbc) and Clean as representatives of the several existing implementations of graph reduction. They are mature compilers and implement more features than we discuss in this paper. Despite the restrictions of our technique, reduction-based compilers provide a good reference point for evaluating the efficiency and viability of the intensional approach.

We are also interested in comparing the new execution model with a hashing-based implementation of the intensional approach [9]. The same idea is used for implementing intensional languages, e.g. Lucid [15]. We use as a reference point a more recent implementation of the intensional approach that uses a garbage-collecting warehouse for memorizing intermediate values [4].

The experimental compiler that we developed[1] is based on the execution model and implementation technique described in Sections 3 and 4 [3]. It compiles source programs, written in a higher-order functional language, to low-level C code. At its present state, our compiler does not try to optimize function bodies (a number of such optimizations are performed by the C compiler) and does not eliminate tail-recursive calls, which occur very often in the final C program.

We used the following programs as benchmarks for comparing the performance of the aforementioned implementations. The order of each program is shown in parentheses.

nofib (1):. A standard benchmark, computing a variation of Fibonacci numbers in exponential time, for $n = 30$.

queens (1):. A program solving the N queens' problem for $N = 9$, using large integer numbers to encode chessboard configurations.

primes (1):. A program computing prime numbers up to 7500.

ack (1):. A program computing Ackermann's function for $m = 3$ and $n = 9$.

tak (2):. A program computing a second-order variation of Takeuchi's function for $x = 24$, $y = 16$ and $z = 8$.

ntak (3):. A program computing a third-order variation of Takeuchi's function for $x = 24$, $y = 16$ and $z = 8$.

[1] Our compiler and the sources of the benchmark programs can be obtained from: `ftp://ftp.softlab.ntua.gr/pub/users/nickie/software/lar.tar.gz`

Table 1. Execution times for benchmark programs.

	Intensional		Reduction-based		
	Stack-based	Hashing-based	GHC	HBC	Clean
		First order			
nofib	0.080	2.650	0.550	1.363	0.005
queens	0.050	2.170	0.320	0.837	0.005
primes	0.030	0.820	0.435	1.232	0.010
ack	1.270	40.580	2.453	6.889	0.270
		Higher order			
tak (2)	0.310	43.610	0.372	0.664	0.090
ntak (3)	0.750	126.410	1.452	4.854	0.240
integrate (3)	0.300	too long!	0.765	1.373	0.050
church (4)	0.010	0.040	0.001	0.001	0.001

integrate (3):. A program that computes several integrals using various methods of integration.

church (4):. A program that performs some integer arithmetic with Church numerals.

Table 1 shows the execution times obtained by executing the benchmark programs. For a fair comparison, optimizations such as strictness analysis were disabled in compilers that perform them. Moreover, when there was an option to produce C instead of native code, it was preferred.

First of all, it is clear that our implementation is a significant improvement over the hashing-based implementation, as it outperforms it by several orders of magnitude in most cases. For most benchmarks, our implementation is as fast as the reduction-based systems. In some cases, usually in first-order programs, our implementation appears to outperform Haskell's compilers. Moreover, the behaviour of our technique in programs with a high number of function calls (e.g. nofib, ack, tak, ntak) is competitive; it can be significantly improved by implementing tail recursion elimination. On the other hand, in most cases, Clean's compiler produces faster code than all the others. A possible explanation is that it produces native code instead of C and makes some tail-call transformation to obtain more efficient code. Garbage collection and the handling of heap-allocated closures is one of the reasons that add a time overhead to reduction-based implementations. Our technique operates using a stack; that is, no garbage collecting is needed as long as no heap is used. Garbage collection may become necessary when complex data structures are supported in the source language.

In general, the performance of our implementation is competitive to that of current efficient reduction-based implementations. Furthermore, no performance degradation occurs when the order of program increases, in contrast to [11].

6. Conclusion and future work

The intensional transformation has long been proposed as an alternative implementation technique for lazy functional languages. Its main idea is to transform a higher-order source functional program into an equivalent zero-order intensional program. The purpose of this paper was to assess the efficiency and viability of this technique. We have proposed an efficient model for executing the transformed intensional zero-order programs and compared a prototype implementation of this model with other techniques, reduction-based or intensional. The results indicate that the intensional technique can be as efficient as most modern implementation techniques for lazy functional languages, for the class of programs that can be transformed.

Although our results are very promising, this technique is still far from being able to implement a fully featured lazy functional language. Apart from the known shortcomings of the intensional transformation, imposing restrictions on the source programs [13], a long list of features remain to be supported by this technique. One of the most important missing features is arbitrary data structures. Furthermore, optimizations of our implementation, such as strictness analysis and tail recursion elimination, must be investigated as future work.

References

[1] E. Aschroft and W. W. Wadge. Lucid – A Formal System for Writing and Proving Programs. *SIAM Journal on Computing*, 5(3):336–354, 1976.

[2] E. Aschroft and W. W. Wadge. Lucid, a nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[3] A. Charalambidis. *An Efficient Technique for Implementing Lazy Functional Programming Languages*. Diploma Dissertation. Department of Informatics and Telecommunications, University of Athens, June 2005.

[4] A. Grivas. *Implementation of Functional Languages using the Branching Dimensions Transformation*. Diploma Dissertation. School of Electrical and Computer Engineering, National Technical University of Athens, October 2004.

[5] T. Johnsson. Efficient Compilation of Lazy Evaluation. in *Proceedings of ACM Compiler Construction*, Montreal, Canada, June 1984, ACM SIGPLAN Notices, 19(6), pp. 58–69.

[6] C. Ostrum. The Luthid 1.0 manual. Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[7] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[8] P. Rondogiannis. *Higher-Order Functional Languages and Intensional Logic*. PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994.

[9] P. Rondogiannis and W. W. Wadge. A Dataflow Implementation Technique for Lazy Typed Functional Languages. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 23–42, 1993.

[10] P. Rondogiannis and W. W. Wadge. Compiling Higher-Order Functions for Tagged-Dataflow. In *Proceedings of the IFIP International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278. North-Holland, August 1994.

[11] P. Rondogiannis and W. W. Wadge. Higher-Order Dataflow and its Implementation on Stock Hardware. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, 1994.

[12] P. Rondogiannis and W. W. Wadge. First-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

[13] P. Rondogiannis and W. W. Wadge. Higher-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 9(5):527–564, September 1999.

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

[15] W. W. Wadge. Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, 1991.

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

Appendix A. History of the transformation and Bill Wadge's involvement

The history of the intensional implementation technique starts more than 25 years ago. At that time, Ed Ashcroft and Bill Wadge had already designed the Lucid programming language, a first-order functional/dataflow language that is based on streams [1, 2]. It soon became obvious that in order to implement Lucid, an entirely new implementation philosophy should be followed. In other words, the traditional techniques that were applicable to mainstream functional languages appeared to be inadequate and slow when applied to Lucid. This state of affairs is described in detail in the "Lucid Book" [14, pages 92–98].

One problem with the implementation of Lucid was how one could treat in a uniform way the time parameter of the language as well as the filters (i.e., functions) that the user could define. If a program is zero-order (i.e., no filters at all) then there exists an elegant way to compute its output: if we want the value of the program at time t, we simply *demand* the value of the output variable of this program at time t; this action generates further demands for other variables at other (possibly different) time points, and so on. Obviously, this procedure creates a tree of demands. At some point these demands can be satisfied (i.e., we reach at some constant of the program) and the results can be returned until we get back to the root of the tree. This simple idea is the basis of *demand-driven dataflow*

and has since been used in all Lucid implementations. The technique is usually termed *eduction*, since the value of the output is educed using the definitions of the program. Notice that the philosophy of eduction does not require the program to change during execution (as for example is the case with graph reduction implementations).

The key question that now arises is: how can we generalize eduction in order to treat user-defined functions? The solution of this problem appears to have been given by Calvin Ostrum in 1981, who at that time was an undergraduate student at the University of Waterloo! Ostrum gave an implementation of Lucid that supported filters and which appeared to be purely eductive [6]. In order to understand Ostrum's implementation, one could only consult the source code itself. Trying to formalize the ideas behind this implementation, Bill Wadge decided to supervise a Ph.D. thesis on this topic! This turned out to be Ali Yaghi's Ph.D. dissertation (1984), the first ground-breaking publication on the topic of eduction. The essence of Yaghi's thesis is that the target language produced by Ostrum's implementation, can be understood as a zero-order *intensional* one: when one compiles a Lucid program that contains filters, then the target program is a zero-order Lucid one which contains an extra (branching) temporal dimension. In other words, first-order functions can be eliminated from a Lucid program at the expense of appropriately introducing an additional dimension (together with corresponding operators) to the program. The results of Yaghi's thesis allowed the implementation by Tony Faustini of a stable interpreter for Lucid. Yaghi's Ph.D. dissertation left open two interesting research problems:

- How can one demonstrate the correctness of the intensional transformation?
- Is it possible to generalize the intensional transformation so as to apply to higher-order functional languages?

Positive answers to the above two questions would render the transformation more widely applicable. For example, it would be possible to apply the intensional approach to more mainstream functional languages (such as Haskell).

The second of these questions was undertaken by Bill Wadge [15]. In this paper Bill suggested that the intensional transformation could be extended to higher-order programs by performing a gradual reduction of the order of the source functional program. This paper also suggested that at every stage of the transformation, a new dimension has to be introduced to the target language. This idea was ingenious, as it suggested that different orders of a (higher-order) function correspond to different dimensions of a (zero-order) intensional language. At that time, one of the authors of the present article (Panos Rondogiannis), was starting his Ph.D. studies under the supervision of Bill Wadge. In Rondogiannis' dissertation [8], the first problem left open by Yaghi's dissertation, was resolved; moreover, the transformation proposed by Bill [15], was formalized and proven correct. It goes without saying that Bill Wadge played a vital role in the supervision of Rondogiannis's work (by making deep remarks, asking for a thorough justification of every step in the proof, and insisting on consistently using various different fonts in order for the thesis to be accurate).

Of course, the history of the transformation has by no means ended. The major remaining open problem is the extension of the technique to languages that allow arbitrary partial function applications (at present, the only partial objects that the source functional language is allowed to use are function names). If this issue is resolved, then the intensional approach could definitely be characterized as an important alternative to the reduction-based implementations of functional languages.

Angelos Charalambidis and Panos Rondogiannis
Department of Informatics and Telecommunications
University of Athens
Panepistimiopolis
15784 Athens
Greece
e-mail: a.charalambidis@di.uoa.gr
prondo@di.uoa.gr

Athanasios Grivas and Nikolaos S. Papaspyrou
School of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli
15780 Zografou
Greece
e-mail: agrivas@softlab.ntua.gr
nickie@softlab.ntua.gr

Received: November 30, 2007.
Accepted: April 11, 2008.