The Intensional Transformation for Functional Languages with User-Defined Data Types

Georgios Fourtounis[1,*] gfour@softlab.ntua.gr
Nikolaos Papaspyrou[1] nickie@softlab.ntua.gr
Panos Rondogiannis[2] prondo@di.uoa.gr

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

Abstract

We extend the intensional transformation [6, 4] so as to apply to first-order lazy functional languages with user-defined data types. Since higher-order functional programs can be translated to first-order ones with the use of data types [3], the proposed approach can also be used to transform fully higher-order lazy functional programs.

1 Introduction

The intensional transformation [6, 4, 5] is a promising implementation technique for lazy functional languages, whose performance has been shown to compete with existing graph-reduction implementations [1]. The transformation was initially defined for a first-order functional language [6, 4], and was later extended to apply to a higher-order language with a restricted syntax [5]. In both of these cases, the source language supported only base data types (such as integers and reals). The problem of treating user-defined data types under the intensional transformation, has never been given a satisfactory answer so far.

In this paper we describe how we can extend the intensional transformation so as to apply to first-order lazy functional languages with user-defined data types. Since higher-order functional programs can be encoded as first-order ones with the use of data types [3], the proposed approach can be used to transform fully higher-order lazy functional programs. Due to space limitations, the interested reader should consult the article of Rondogiannis and Wadge [4] for the relevant background and the basic notions regarding the intensional transformation.

2 The Source Functional Language FL$^+$

The source language of our transformation is FL$^+$, a simple first-order, lazy functional language. FL$^+$ extends the language FL [5] by additionally supporting user-defined data types and pattern matching. The full syntax of FL$^+$ can be found in Figure 1, while Figure 2 shows a valid FL$^+$ program. We make certain simplifying assumptions regarding the syntax of our source programs. First, we assume that every program has a definition for the variable *result*; the output of the program will simply be the value of result. We also assume that the formal parameters of all functions defined in the program are different. Finally, the components of a pattern match clause (such as *e0* and *es0* in *Cons*(*e0*, *es0*)) always have the same names anywhere in the program.

The underlined variables of the syntax are those that are bound by patterns and refer to constructor components. It should be noted that the user does not have to explicitly state which variables of the input program are bound by patterns, as they can be statically detected.

*Partly supported by the EEA FM EL0086 NTUA Mobility and Scholarship Program.

dtype_name: datatype names
c_name: constructor names
$prog := (dt^*, def^+)$
$constr := c\_name\ (cc^*)$
$cc := dtype\_name\ var \mid \textbf{Int}\ var$
$dt := dtype\_name :\ constr^+$

$def := var(var^*) = expr$
$expr := num \mid var(expr^*) \mid \underline{var} \mid expr\ binop\ expr$
$\mid \textbf{if}\ expr\ \textbf{then}\ expr\ \textbf{else}\ expr \mid c\_name(expr^*)$
$\mid \textbf{match}\ expr\ \textbf{with}\ pmatch^+$
$binop := + \mid - \mid * \mid / \mid \leq \mid ==$
$pmatch := c\_name(var^*) \rightarrow expr$

Figure 1: The syntax of FL$^+$.

List : Nil, Cons (Int e0) (List es0)
result = head(f(300))
head(hl) = **match** hl **with** Cons(e0, es0) → <u>e0</u>
f(x) = **if** (x ≤ 0) **then** Cons (x+10, Nil) **else** Cons(x∗2, f(x−1))

Figure 2: A program in FL$^+$.

3 The Target Intensional Language NVIL$^+$

The target language of the proposed transformation is the zero-order intensional language NVIL$^+$ which extends the language NVIL of Rondogiannis and Wadge [4] in order to allow the handling of user-defined data types in the source language FL$^+$. The syntax of NVIL$^+$, is given in Figure 3. In the following, we start by giving an informal presentation of the semantics of NVIL$^+$ which is subsequently formalized by providing a corresponding interpreter for the language.

In order to evaluate an expression of the form (**match** e **with**...), the expression e is evaluated first until a call to a constructor is encountered; then there is enough information to select the appropriate branch of the matching clause. The effect of evaluating a constructor is memory allocation: the current context is saved in a heap in order to be later used when the constructor contents are requested inside the **match** branches. This memory allocation is made explicit in the semantics by the **thunk** operator of NVIL$^+$, which updates the heap with the current context.

Heap allocation is only forced by pattern matching, as this is where the constructor call will be invoked. This makes pattern matching the only point in a program where thunks are created and lazy data structures are processed. The domain of NVIL$^+$ contains two classes of values: *normal values*, and lazily created *heap structures* that may be forced to evaluate their contents. This means that a program can have two kinds of results: either a value, or a suspended constructor (which could lead to an infinite data structure). In the second case, the user must have a way to force its evaluation. This mechanism is external to the technique described here and should be provided by the implementation, either as a top-level interactive loop, a pretty printer, or any other suitable mechanism.

The heap is modeled as a *context-indexed dictionary of contexts*. These contexts link the program point that a constructor subexpression is needed, to the point when the constructor

$def := var = expr \mid var = \textbf{case}(expr^+)$
$expr := num \mid var \mid \underline{var} \mid expr\ binop\ expr \mid \textbf{if}\ expr\ \textbf{then}\ expr\ \textbf{else}\ expr \mid \textbf{thunk}\ c\_name$
$\mid \textbf{call}_n \cdot expr \mid \textbf{actuals}_n \cdot expr \mid \textbf{match}\ expr\ \textbf{with}\ pmatch^+$
$pmatch := c\_name \rightarrow expr$

Figure 3: The syntax of NVIL$^+$.

Semantic domains:
Var, $\underline{Var}$, C_name
$Ctxt \equiv List[Nat]$
$Heap \equiv Ctxt \to Ctxt$
$Value := FValue\ Int \mid Thunk\ C\_name\ Heap$

Helper functions:
$lookup :: (Var + \underline{Var}) \to Ctxt \to Prog \to Expr$
$val :: Value \to Int$
$cstr :: Value \to Constructor$
$h :: Value \to Heap$

Evaluation function:
$eval :: (Expr, Ctxt, Heap, Prog, Ctxt) \to Value$
$eval(num, ctxt, heap, p, sc) = FValue\ num$
$eval(var, ctxt, heap, p, sc) = eval(lookup(var, ctxt, p), ctxt, heap, p, sc)$
$eval(\underline{var}, ctxt, heap, p, sc) = eval(lookup(\underline{var}, ctxt', p), ctxt', heap, p, sc)$
where $ctxt' = heap(ctxt)$
$eval(e_1 + e_2, ctxt, heap, p, sc) = FValue(val(v_1) + val(v_2))$
where $v_1 = eval(e_1, ctxt, heap, p, sc)$ and $v_2 = eval(e_2, ctxt, heap, p, sc)$
(similar rules for other strict binary operators like $-, *, /, \leq, ==$)
$eval(\textbf{thunk}\ c, ctxt, heap, p, sc) = Thunk\ c\ (heap \cup [sc \mapsto ctxt])$
$eval(\textbf{call}_j \cdot e, ctxt, heap, p, sc) = eval(e, (j : ctxt), heap, p, sc)$
$eval(\textbf{actuals}_j \cdot e, (j : ctxt), heap, p, sc) = eval(e, ctxt, heap, p, sc)$
$eval(\textbf{match}\ e\ \textbf{with}\ patterns, ctxt, heap, p, sc) = eval(pat, ctxt, h(res), p, sc)$
where $res = eval(e, ctxt, heap, p, ctxt)$ and $pat = patterns[cstr(res)]$
$eval(\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2, ctxt, heap, p, sc) = eval(e', ctxt, heap, p, sc)$,
where $e' = e_1$ if $val(eval(e, ctxt, heap, p, sc)) = 1$ and $e' = e_2$ otherwise

Figure 4: An interpreter for NVIL$^+$.

was initially reached. To reference the constructor components, we introduce bound ("*heap*") variables that may only exist inside a pattern branch that corresponds to a constructor. The variables of a program are thus of two kinds: normal variables and bound variables. The first are evaluated as normal, but the second are evaluated under a new context, which is looked up on the heap, according to the current context.

For example, when a *Cons* constructor is reached, the context needed for its subexpressions is saved in the dictionary, using the originating context *sc* of the **match** expression as the key. During the rest of the execution, when the value of *e0* or *es0* is needed, it will be demanded in the context that amounts to the start of the specific branch that was followed. Using this context, the correct thunk will be retrieved and the evaluation will proceed for its subexpressions.

The above discussion is formalized by the interpreter for NVIL$^+$ given in Figure 4. The set Var contains the normal variables var of the intensional transformation, while the set $\underline{Var}$ contains the heap variables $\underline{var}$ that depend on the heap to be computed. The set C_name contains the constructor names c_name. Evaluation of the program P starts in the empty context $ctxt_0$ and the empty dictionary $heap_0$ with an empty saved context. The following auxiliary functions are used:

- *lookup* returns a (normal or heap) variable definition from the NVIL$^+$ program structure
- *val* returns the value of a result (if it is a normal value)
- *cstr* returns the constructor of a result (if it is a thunk)
- h returns the stored heap of a result (if it is a thunk)

Given a constructor c, the notation $patterns[c]$ selects the correct branch of the *patterns* cases of a **match** expression. The intensional operators **call** and **actuals** have their standard semantics,

Original source program in FL^+

⇓ (preprocessing, adding functions for constructors)

List : Nil, Cons (Int e0) (List es0)
result = head(f(300))
head(hl) = **match** hl **with** Cons(e0, es0) → <u>e0</u>
f(x) = **if** (x ≤ 0) **then** cons(x+10, nil) **else** cons(x∗2, f(x−1))
cons(e0, es0) = **thunk** Cons
nil = **thunk** Nil

⇓ (intensional transformation, to NVIL^+)

constructors : *Nil*, *Cons*
result = $\mathbf{call}_0 \cdot$ *head*
head = **match** *hl* **with** *Cons* → <u>*e0*</u>
hl = **case**[$\mathbf{actuals}_0 \cdot \mathbf{call}_0 \cdot f$]
f = **if** $(x \leq 0)$ **then** $\mathbf{call}_0 \cdot$ *cons* **else** $\mathbf{call}_1 \cdot$ *cons*
x = **case**[$\mathbf{actuals}_0 \cdot 300, \mathbf{actuals}_1 \cdot (x - 1)$]
e0 = **case**[$\mathbf{actuals}_0 \cdot (x + 10), \mathbf{actuals}_1 \cdot (x * 2)$]
es0 = **case**[$\mathbf{actuals}_0 \cdot$ *nil*, $\mathbf{actuals}_1 \cdot \mathbf{call}_1 \cdot f$]
cons = **thunk** *Cons*
nil = **thunk** *Nil*

Figure 5: The transformation process for the program of Figure 2.

i.e. they push and pop values in the context. Another standard intensional construct is **case**, which uses the current context's head to select an expression to return. Variables look up their definition in the program and continue evaluating that (under a new looked up context in the case of the heap variables). A **thunk** expression will store context information in the dictionary and a **match** will evaluate its expression until it reaches a thunk, at which point it will select the correct branch to evaluate with the new dictionary that now contains the thunk.

4 The transformation

The first step in the proposed transformation is to create a function definition for each constructor in the source program. The body of each such definition is simply a **thunk**. For example, if the source FL^+ program contains the constructor *Cons*, we add the function definition $cons(e0, es0) =$ **thunk** *Cons* and replace calls to *Cons* with calls to *cons*. We then apply the standard intensional transformation (e.g., as described by Yaghi [6]), treating the newly added definitions in the same way as ordinary function definitions. Notice that in pattern matching clauses we simply remove the bound arguments from pattern constructors (without introducing any intensional operators). The example given in Figure 5 illustrates the above ideas. One can easily verify that by executing the resulting intensional program, the correct output is obtained.

5 Discussion

The transformation described in this paper can be used in order to intensionalize a fully higher-order functional language: higher-order programs are initially defunctionalized using the well-known technique introduced by Reynolds [3]. The program that results from the defunctionalization is an FL^+ one and can therefore be transformed in the way that we have described in the previous sections. In conclusion, given a higher-order functional program, we can obtain a zero-order $NVIL^+$ one in the way just described. The target program can then be executed following an extended demand-driven evaluation strategy; the details of such an implementation will be reported in a forthcoming paper.

The problem of intensionalizing functional languages with user-defined data types has not (to our knowledge) been considered before in the literature. The problem of intensionalizing higher-order functional languages has been considered before, but not in such a generality. For example, Rondogiannis and Wadge [5] only treat a restricted class of higher-order programs. Another interesting proposal for this problem is the one proposed by Plaice and Mancilla [2] in which extra dimensions are used in order to represent closures.

An interesting direction for future work would be the implementation of a fully higher-order functional language based on the above ideas, and the comparative evaluation of such an implementation against existing graph-reduction based implementations of functional languages.

References

[1] Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008.

[2] John Plaice and Blanca Mancilla. The practical uses of TransLucid. In *Proceedings of the 1st International Workshop on Context-aware Software Technology and Applications (CASTA'09)*, pages 13–16, New York, NY, USA, 2009. ACM.

[3] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM Annual Conference*, volume 2, pages 717–740, New York, NY, USA, 1972. ACM. Reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[4] Panos Rondogiannis and William W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

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

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