Fusion Transformation on Functional Programs with Regular Patterns

Kazuhiko KAKEHI *[1,2] Zhenjiang HU *[1,3] Masato TAKEICHI *[1]

*[1] Graduate School of Information Science and Technology
The University of Tokyo

*[2] Institute for Software Production Technology
Waseda University

*[3] PRESTO 21, Japan Science and Technology Corporation

With the increasing popularity of XML for representing structured data, more and more researches have been devoted to better functional frameworks for supporting XML processing. Of that, XDuce employs regular expression types and this enables programmers to manipulate XML-based data flexibly. This paper investigates automatic fusion of functional programs with regular patterns, so that XML data can be manipulated more efficiently through eliminating intermediate data passed between two functions.

1 Introduction

With the increasing popularity of XML for representing structured data, more and more researches have been devoted to better functional frameworks for supporting XML processing [9, 4]. XDuce [4, 3] is a powerful system, which employs regular expression types and regular pattern matching, enabling programmers to manipulate XML-based data flexibly. Figure 1 gives a simple program with regular (expression) patterns.

This flexibility, however, brings difficulty for efficient implementation and optimization. Consider a composition of two functions `directmail` and `takeone` defined using regular pattern matching. This program is inefficient, because the intermediate data passed between `takeone` and `directmail` is unnecessary and can be eliminated. A more efficient after fusion of this composition is as follows.

```
fun f ( val e as Person* ) : Person4* =
match e with
  person <val n as Name,
          val e1 as Email,
          val e2 as Email*,
          val rest>, val next
     -> person <n, e>, f ( next )
| person <val n as Name,
          val m as (Cell|PHS)>, val next
     -> person <n, pmail <m>>, f ( next )
| () -> ()
```

When we turn eyes on function fusion, a powerful optimization mechanism is already in hand, say the Hylo system [5, 6]. The idea of function fusion was first proposed by Wadler [8] whose approach follows fold-unfold transformation [1] in essence. Later, another more practical approach using a combination of `foldr/build`, called shortcut fusion, was proposed by Gill [2]. The Hylo system, based on the acid rain theorem [7], is an extension of shortcut fusion, where functions with its input and output data structures are generalized from lists to arbitrary data structures.

It is, however, not straightforward to extend the existing fusion technique for functions with regular patterns. This is due to the complexity caused by flexible regular patterns and inherited subtyping mechanism (e.g., the range type of `takeone` is a subtype of the domain type of `directmail`).

This paper makes the first attempt to automatically fuse functions with regular patterns. We propose a type-directed translation from regular types to algebra types, regular patterns to simple patterns, regular expressions to simple expressions, and subtypes to type-embedded functions. This translation reduces fusion on functions with regular patterns into fusion on those without regular patterns,

```
type Person  = person  < Name, Email*, (Cell|PHS) >
type Person2 = person  < Name, (Email|Cell|PHS) >
type Person3 = person  < Name, (Address|()), (Email|Cell|PHS|Pager) >
type Person4 = person  < Name, (Address|Email|Pmail) >
type Name    = name    < String >    type Email   = email < String >
type Cell    = cell    < String >    type PHS     = phs   < String >
type Pager   = pager   < String >    type Pmail   = pmail < Cell|PHS|Pager >
type Address = address < String >

fun takeone ( val e as Person* ) : Person2* =
  match e with
      person < val n as Name, val e1 as Email, val e2 as Email*, val rest >, val next
         -> person < n, e >, takeone ( next )
    | person < val n as Name, val m as (Cell|PHS) >, val next
         -> person < n, m >, takeone ( next )
    | () -> ()

fun directmail ( val e as Person3* ) : Person4* =
  match e with
      person < val n as Name, val a as Address, val rest >, val next
         -> person < n, a > , directmail ( next )
    | person < val n as Name, val e as Email >, val next
         -> person < n, e >, directmail ( next )
    | person < val n as Name, val e as (Cell|PHS|Pager) >
         -> person < n, pmail < e > >, directmail ( next )
    | () -> ()

fun main ( val e as Person* ) : Person4* = directmail ( takeone ( e ) )
```

Figure 1: A running example

and thus we are able to use the existing fusion system. Our simple experiment with the Hylo fusion system shows that this approach is promising.

The organization of this paper is as follows. Section 2 introduces the source language. Section 3 explains our transformation strategy with formal transformation rules. The final Section 4 concludes this paper with mentioning future works.

2 A Simplified XDuce Language

While the target program will be written in Haskell where the Hylo mechanism exists, we set to use a simplified XDuce language as its source programs.

Figure 2 gives the formal definition of the source language. We here omit the semantics of this language. The difference from XDuce is to introduce * in the source language to represent repetition explicitly. Instead, we do not allow recursion in types. For types and patterns, occurrences lined up by `|` or by `,` are assumed to appear in a flattened manner; an sequence $(O_1, O_2), O_3$ should be written as O_1, O_2, O_3 . This clarifies how entries appear in types or patterns, but it is not possible to be imposed on terms due to existence of function calls in terms. The empty symbol `()` appears in two contexts. One is in unions (`|`), which implies that some elements appear optionally (like `?` in regular expressions, or like `Maybe` in Haskell); the other is in sequences (`,`) . It will receive different treatments during transformation.

We yet set some more restrictions for realizing simple transformations; they are natural ones and do not essentially narrow the expressive power of the source language. These restrictions will be mentioned in the following sections.

3 Transformation

Our idea is to make use of the existing Hylo fusion system (for Haskell) to remove unnecessary inter-

Term

$$\begin{array}{rcl} e & ::= & \texttt{()} \\ & | & x \\ & | & \texttt{l<}\ e\ \texttt{>} \\ & | & e\texttt{,}e \\ & | & f\texttt{(}e\texttt{)} \\ & | & \texttt{match}\ e\ \texttt{with}\ P_i \rightarrow e_i \end{array}$$

Function definition

$$\texttt{fun}\ f\ \texttt{(val}\ x\ \texttt{as}\ S\texttt{)}\ \texttt{:}\ T\ \texttt{=}\ e$$

Type

$$\begin{array}{rcl} T & ::= & \texttt{()} \\ & | & N \\ & | & \texttt{l<}\ T\ \texttt{>} \\ & | & T\texttt{,}T\texttt{,}\ldots \\ & | & T\texttt{|}T\texttt{|}\ldots \\ & | & T\texttt{*} \end{array}$$

Type definition

$$\texttt{type}\ t\ \texttt{=}\ T$$

Pattern

$$\begin{array}{rcl} P & ::= & \texttt{()} \\ & | & x \\ & | & x\ \texttt{as}\ P \\ & | & N \\ & | & \texttt{l<}\ P\ \texttt{>} \\ & | & P\texttt{,}P\texttt{,}\ldots \\ & | & P\texttt{|}P\texttt{|}\ldots \\ & | & P\texttt{*} \end{array}$$

Figure 2: The source language

mediate data structures in XDuce programs. The problem is rephrased as: how to transform XDuce programs into such Haskell programs that are suitable for later fusion.

Figure 5 shows an example of the Haskell obtained from the XDuce program in Figure 1. We shall briefly explain how to eliminate regular expression types and patterns, and how to deal with subtyping in the following paragraphs. Then we explain that the transformation is successful by giving our experimental result.

Eliminating regular types Consider to map the following type definition

```
type Person =
     person <Name, Email*, (Cell|PHS)>
```

to an abstract data type without regular expressions. We may treat the tag `person` as the constructor name (translated into `Person`), and the contents bracketed with `< >` as components of the constructor. `Email*`, a repetition (`*`) of type `Email`, is suitably represented by a list of `Emails`. Type names, like `Name` are left as they are. Union `(Cell|PHS)`, on the other hand, is considered to construct a new type, and we give a type name and constructors for `Cell` and `PHS`. The resulting type definition in Haskell will be as follows.

```
data Person = Person Name [Email] CP
data CP     = CP1 Cell
            | CP2 PHS
```

Type

$$\begin{array}{rcl} [\![\texttt{()}]\!] & = & \epsilon \\ [\![N]\!] & = & N \\ [\![\texttt{l<}T\texttt{>}]\!] & = & \underline{\texttt{l}}[\![T]\!] \\ [\![T_1\texttt{,}\ldots\texttt{,}T_n]\!] & = & \texttt{(}[\![T_1]\!]\texttt{,}\ldots\texttt{,}[\![T_n]\!]\texttt{)} \\ [\![T_1\texttt{|}\ldots\texttt{|}T_n]\!] & = & N' \text{ where} \\ & & \texttt{data}\ N'\ \texttt{=}\ \underline{\texttt{c}_1}\ [\![T_1]\!] \\ & & \qquad\quad \texttt{|}\ \vdots \\ & & \qquad\quad \texttt{|}\ \underline{\texttt{c}_\texttt{n}}\ [\![T_n]\!] \\ [\![T\texttt{*}]\!] & = & \texttt{[}[\![T]\!]\texttt{]} \end{array}$$

$$[\![\ \texttt{type}\ N\ \texttt{=}\ T\ ]\!] \Rightarrow \texttt{data}\ N\ \texttt{=}\ [\![T]\!]$$

Figure 3: Transformation rules for type

It is worth noting that, while XDuce allows the same tag name appearing different type definitions, Haskell does not allow the same data constructor being used in different type definitions. When continuing transformation of `Person2` in Figure 1, we notice that the constructor `Person` is already occupied. Another constructor name, like `Person2`, is used instead.

```
data Person2 = Person2 Name ECP
data ECP     = ECP1 Email
             | ECP2 Cell
             | ECP2 PHS
```

The detailed transformation rules are summarized in Figure 3.

Pattern

$$
\begin{array}{rcll}
\llbracket \mathtt{l<}P\mathtt{>} \rrbracket(\mathcal{T}) & = & (\underline{\mathtt{l}}\llbracket P \rrbracket(\mathcal{T}')) & \text{if } \mathcal{T} = \mathtt{l<}\mathcal{T}'\mathtt{>} \\
 & = & (\underline{\mathtt{c}}\llbracket \mathtt{l\ <}P\mathtt{>} \rrbracket(\mathcal{T}')) & \text{if } \mathcal{T} \sqsupset \mathcal{T}' \text{ where } \mathcal{T}' \text{ finally contains the tag } \mathtt{l} \\
\llbracket \mathtt{()} \rrbracket(\mathcal{T}) & = & \mathtt{[]} & \text{if } \mathcal{T} = \mathcal{T}'\mathtt{*} \\
 & = & (\underline{\mathtt{c}}\llbracket \mathtt{()} \rrbracket(\mathcal{T}')) & \text{if } \mathcal{T} \sqsupset \mathcal{T}' \text{ where } \mathcal{T}' \text{ finally contains } \mathtt{()} \\
\llbracket \mathtt{val\ v\ as\ } P \rrbracket(\mathcal{T}) & = & \mathtt{v} & \text{if the type of } P \text{ equals to } \mathcal{T} \\
 & = & \multicolumn{2}{l}{\textit{create a new pattern for each } P'_i \textit{ in } P = (P'_1|P'_2|\ldots) \textit{ using } \mathtt{v@}\llbracket P'_i \rrbracket(\mathcal{T})} \\
 & & & \text{if the type of } P \text{ is smaller than } \mathcal{T} \\
\llbracket N \rrbracket(\mathcal{T}) & = & \_ & \text{if } N \text{ equals to } \mathcal{T} \\
 & = & (\underline{\mathtt{c}}\llbracket N' \rrbracket(\mathcal{T}')) & \text{if } \mathcal{T} \sqsupset \mathcal{T}' \text{ where } \mathcal{T}' \text{ finally contains the type } N \\
\multicolumn{4}{l}{\llbracket P_1\mathtt{,}\ldots\mathtt{,}P_n \rrbracket(\mathcal{T}_1\mathtt{,}\ldots\mathtt{,}\mathcal{T}_m)} \\
 & = & \multicolumn{2}{l}{\mathtt{(}\llbracket P_1 \rrbracket(\mathcal{T}'_1)\mathtt{,}\ldots\mathtt{,}\llbracket P_m \rrbracket(\mathcal{T}'_m)\mathtt{)} \textit{ using longest match}} \\
\llbracket P_1\mathtt{,}\ldots\mathtt{,}P_n \rrbracket(\mathcal{T}\mathtt{*}) & = & \multicolumn{2}{l}{\mathtt{(}\llbracket P_1 \rrbracket(\mathcal{T})\mathtt{:}\ldots\mathtt{:}\llbracket P_{n-1} \rrbracket(\mathcal{T})\mathtt{:}\llbracket P_n \rrbracket(\mathcal{T}\mathtt{*})\mathtt{)}}
\end{array}
$$

Figure 4: Transformation rules for pattern

Eliminating regular patterns For simplicity we only deal with one pattern matching in a function. Patterns have almost the same definition as types, except for bindings to variables. The second line of pattern definition in Figure 2 means that x matches any type. Type inference with the type information given to a function definition determines x's type.

In this matching we assume to incorporate the technique developed in XDuce, which realizes calculation on regular expression types, like inclusion, intersection and difference. During inference, we assume that patterns are reformatted to suit the type information, like adding `()`, or preparing individual variables to bind each type when a variable binds to a sequence of types.

Figure 4 shows transformation rules. Here, we explain the last two patterns. The first one shows that, according to the longest matching policy, plural pattern entries may be bound to a type with `*`. When plural variables are assigned to that type, the next one specifies that only the last variable is bound to T`*`, and the rest to T.

On terms, with type embedding Terms and function definitions are transformed with applying type embedding. Type embedding in our case is done by replacing constructors of the consumer function's domain type by those of the (larger) producer function's range type, and the producer's result will have the type of the consumer's domain type. Auxiliary functions added after `where` serves this purpose as Figure 5 shows. `f1` in `takeone` adds an entry of optional type `(Address|())`, and `f2` puts or replaces additional constructors to an original input of type `Email` or `(Cell|PHS)` .

An Experimental Result

The Hylo system successfully fuses the obtained code in Figure 5, not only on list constructors but also on other user-defined constructor types. Execution of the fused program requires 20% less heaps on average for several test inputs.

4 Conclusion

We have presented a simple idea to translate source programs written in a modified XDuce language into Haskell. While there are several restrictions, they can be removed with some preprocessing make fusion of functions with regular expression types be easily realized. An experiment indicates the fruitfulness of our proposal.

Our current transformation ignores preciseness of matching on union types, like the pattern `val b as B*, val a as A, val rest` to the type

```
data Person  = Person  Name [Email] CP

data Person3 = Person3 Name AE ECPP

data Person4 = Person4 Name AEP

data Name    = Name String

data Address = Address String

data Email   = Email String

data Cell    = Cell String

data PHS     = PHS String

data Pager   = Pager String

data CP      = CP1 Cell
             | CP2 PHS

data AE      = AE1 Address
             | AE2

data ECPP    = ECPP1 Email
             | ECPP2 Cell
             | ECPP3 PHS
             | ECPP4 Pager

data AEP     = AEP1 Address
             | AEP2 Email
             | AEP3 Pmail
```

```
data Pmail   = Pmail CPP

data CPP     = CPP1 Cell
             | CPP2 PHS
             | CPP3 Pager

takeone :: [Person] -> [Person3]
takeone ( ( Person n (e1:e2) _ ) : next )
       = ( Person3 n f1 (f2 e1) ) : takeone next
         where f1        = AE2
               f2 e      = ECPP1 e
takeone ( ( Person n  [] m ) : next )
       = ( Person3 n f1 (f2 m ) ) : takeone next
         where f1        = AE2
               f2 (CP1 c) = ECPP2 c
               f2 (CP2 p) = ECPP3 p
takeone [] = []

directmail :: [Person3] -> [Person4]
directmail ( ( Person3 n (AE1 a) _ ) : next )
       = ( Person4 n (AEP1 a) ) : directmail next
directmail ( ( Person3 n (AE2)   p ) : next )
       = ( Person4 n (f p) ) : directmail next
         where f (ECPP1 e) = AEP2 e
               f (ECPP2 c) = AEP3 ( Pmail ( CPP1 c ))
               f (ECPP3 p) = AEP3 ( Pmail ( CPP2 p ))
               f (ECPP4 p) = AEP3 ( Pmail ( CPP3 p ))
directmail [] = []


main:: [Person] -> [Person4]
main = ( directmail . takeone ) e
```

Figure 5: Transformation result

`(A|B)*, A`. Increasing preciseness remains as future works.

References

[1] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[2] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[3] Haruo Hosoya and Banjamin Pierce. Regular expression pattern mathcing for xml. to appear in Journal of Functional Programming, 2002.

[4] Haruo Hosoya, Jerome Vouilon, and Banjamin Pierce. Regular expression types for xml. In *Proceedings of 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 11–22. ACM Press, 2000.

[5] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[6] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[7] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[8] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

[9] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *ACM SIGPLAN International Conference on Functional Programming*, pages 148–159, Paris, 1999. ACM Press.