
# 프로그래밍언어논문지

한국정보과학회
프로그래밍언어연구회

제 20 권 제 2호 통권 제38호 2006년 11월 ISSN 1975-5961

## 차 례

■ 편집사 ····· 편집위원회

■ 연구논문

• Retargetable Compilation Technique for Optimal Placement of Scaling Shifts in a Fixed-point Processor ····· 박상현 · 안민욱 · 조두산 · 윤종희 · 백윤흥 1

• 온톨로지 기반의 시맨틱 어노테이터 구현 … 박재훈 · 유재규 · 전양승 · 정영식 · 한성국 17

• 다단계 프로그램에 대한 흐름을 고려한 분석 ····· 김덕환 · 이광근 23

• C 프로그램에서 사용되지 않는 자료를 찾는 정적 분석기의 개발 ····· 황의권 · 이광근 35

• 가벼운 모양분석을 통한 메모리 누수 탐지 ····· 이욱세 · 정승철 · 김태호 41

• XML 기반의 문서통합 시스템 설계 ····· 유재규 · 박재훈 · 전양승 · 정영식 · 한성국 51

■ 학회소식

• 연구회 규정 ····· 57

• 논문지 투고 및 심사규정 ····· 58

한국정보과학회 프로그래밍언어연구회

# 편•집•사

프로그래밍언어 연구회 회원 여러분께,

안녕하십니까?

프로그래밍언어논문지의 제 20권 제 2호를 발간하게 된 것을 기쁘게 생각합니다. 이번 논문지는 2006년 11월 18일에 대구 경북대학교에서 개최되는 2006 프로그래밍언어연구회 추계학술대회 발표 논문을 담고 있습니다. 연구 주제를 소개하면, 부동소수점-고정소수점 변환을 고려한 컴파일러 최적화, 온톨로지 기반의 시맨틱 어노테이터 구현, 다단계 프로그램에 대한 흐름을 고려한 분석, C 프로그램에서 사용되지 않는 자료를 찾는 정적 분석기, 모양분석을 통한 메모리 누수 탐지, XML 기반의 문서 통합 시스템이 있습니다. 본 논문지에 실리지는 않았지만 추계학술대회에는 포항공대 박성우 교수님의 튜토리얼 “A Functional Hardware Description Language”이 있을 예정입니다.

본 논문지에 연구 결과들을 기고해 주신 저자 여러분과 논문지 발간을 위해 수고해주신 프로그래밍언어연구회 운영위원 여러분께 감사를 드리며, 모든 회원들의 건승과 프로그래밍언어연구회의 무궁한 발전을 기원합니다.

2006년 11월 8일

프로그래밍언어연구회

편집위원회

20-2-R01

# Retargetable Compilation Technique for Optimal Placement of Scaling Shifts in a Fixed-point Processor

Sanghyun Park, Minwook Ahn, Doosan Cho, Jonghee Yoon, Yunheung Paek
Software Optimizations and Restructuring Group
School of Electrical Engineering and Computer Science
Seoul National University
(shpark; mwahn; dscho; jhyoon)@optimizer.snu.ac.kr ypaek@snu.ac.kr

**Abstract**

In the past decade, several tools have been developed to automate the floating-point to fixed-point conversion for DSP systems. In the conversion process, they first determine the integer/fractional word lengths for each fixed-point variable, and attempt to optimize the SQNR of the fixed-point code while precluding overflows. In this attempt, a number of scaling shifts need to be inserted into the code, and inevitably they alter the original code sequence. Recently, we have observed that a compiler can often be adversely affected by this alteration of the source code, and consequently fails to generate efficient machine code for its target processor. In this paper, we discuss how we circumvent this problem with a simple peephole optimization technique that safely migrates scaling shifts to other places within the code so that the compiler can have a higher chance to produce better code. We consider our technique to be safe in that it does not introduce new overflows, yet preserving (sometimes even improving) the original SQNR. We implemented this technique on our retargetable compiler, Soargen. The experimental results on a commercial fixed-point DSP processor exhibit that our technique is effective enough to achieve tangible improvement on code size and speed for a set of benchmarks.

## 1. Introduction

Fixed-point processors are generally cheaper than their floating-point counterparts. Thus, most high-volume, low-end DSP systems use fixed-point processors since the priority is low energy and cost. However, dynamic range and precision of a fixed-point processor are often strictly limited [10]. As a result, programming fixed-point processors is usually more painful since programmers must spend much time to maintain proper numeric accuracy and performance with the limited dynamic range and precision. So, the common practice is that programmers

first employ floating-point processors to verify their designs and algorithms, and later implement the verified algorithms on fixed-point processors by converting floating-point data types into equivalent fixed-point ones.

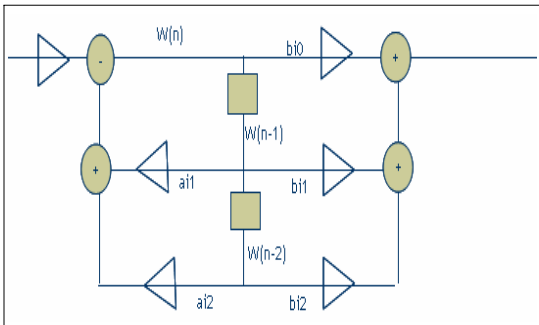
As a first step in this floating-point to fixed-point conversion (FFC) process, they must find the dynamic range and precision needs of each variable in the code. Based on their findings, they insert shift operations to scale variables in the code. The integral part of this conversion process is to decide adequate places where to insert these scaling shifts because this decision deeply affects the two key factors, the signal-to-quantization noise ratio (SQNR) and overflow, which determine the numeric accuracy of the resulting fixed-point code. Therefore, in the FFC process, programmers must perform rigorous static analysis or simulation to compute exact run-time value ranges of all the variables, which will be used to obtain the accurate dynamic ranges and precisions for the variables.

As can be expected, processing the whole conversion by hand would be quite a time-consuming and error-prone task. According to empirical studies [3], the manual process accounts for roughly a third of the total implementation time. To relieve programmers from this burdensome task, many researchers have developed various FFC tools such as Autoscaler and FRIDGE [4][6][8] which automate the FFC process efficiently. However, to the best of our knowledge, all these tools do not fully consider detrimental effects of newly added scaling shifts in the fixed-point code on compiler code generation. Our recent experience reveals that such lack of consideration often result in substantial degradation of the quality of the output machine code.

## 2. Motivation

To illustrate the need of our technique, consider the ordinary floating-point C code segment in Figure 1(b) which implements a popular DSP filter, called IIR, displayed in Figure 1(a). We used the Autoscaler tool [4] to convert this code into the fixed-point one in Figure 1(c) where we see that many scaling shifts have been inserted during the conversion. Figure 1 (d) shows the assembly code for the ZSP400 DSP processor [11] generated directly from the code of Figure 1 (c). As can be noticed from Figure 1 (a) and (b), the IIR filter originally contains several nice operation patterns which should be easily translated by the compiler into some DSP-specific instructions (e.g., multiply - accumulate and dot-product). However, the compiled output in Figure 1 (d) suggests that the compiler failed to utilize those instructions when it compiled the code of Figure 1 (c). Actually as demonstrated in Figure 1 (e), the compiler should be able to further reduce the code size if it could exploit the ZSP mac/nmac instructions. In this example, the main cause that hinders the efficient code generation is the scaling shifts inserted between the add and multiply operations in Figure 1 (c).

To explain this more clearly, consider Figure 1 (f) where the code of Figure 1 (c) is represented in a DAG, the common intermediate representation (IR) form adopted by many compilers. The IR in the figure has been automatically constructed from




(a) IIR Biquad N Sections Filter

```
w(n) = x(n)-ai1*w(n-1) - ai2*w(n-2)
y(n) = bi0*w(n)+ bi1*w(n-1) + bi2*w(n-2)
```

(b) Floating-point code for IIR Filter

```
w(n)=(x(n) - multf(ai1,w(n-1))>>1 -
multf(ai2,w(n-2))>>2)<<3
y(n)=(multf(bi0,w(n))>>1+multf(bi1,w(n-1))
>>2+ multf(bi2,w(n-2)))<<1
```

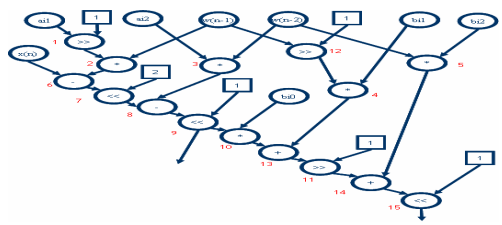
(c) Fixed-point code for IIR Filter

```
mul.a r4, r6
shra r0, 1;
sub r2, r0;
mul.a r5, r7;
shra r0,2;
sub r2, r0;
shla r2, 3;
mul.a r2,r8;
shra r0, 1;
mul.b r6, r9;
shra r2, 2;
add r0, r2;
shra r0, 1;
mac.a r7, r10;
shla r0, 1,
```

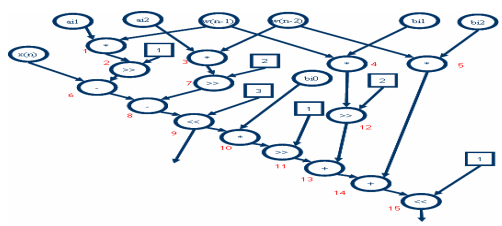
(d) Original code for ZSP400

```
shra r4, 1;
nmac.b r4, r6;
shrla r2, 2;
nmac.b r5, r7;
shrla r2, 1;
mul.a r2, r8;
shra r6, 1;
mac.a r6, r9;
shra r0, 2;
mac.a r7, r10;
shla r0, 1;
```

(e) Improved code for ZSP400



(f) Original IR



(g) Algerabraic Transformed IR

Figure 1

the fixed-point code by our compiler. From this IR, the compiler may recognize a multiply operation in node 5 immediately followed by an add operation in node 14, so it can translate them together to a mac in the assembly. To the contrary, in the case of the multiply-add pair in nodes 10 and 13, it is not straightforward for a compiler to generate a mac because the shift operation in node 11intervenes between the two operations. In consequence, a compiler would translate the pair into two separate multiply and add instructions along with a shift instruction in-between. As seen from Figure 1 (b), this shift operation was in fact not part of the original IIR filter code, but later inserted for scaling between the multiply and add operations during the FFC process. From our recent experience with several FFC techniques, we have learned that when they insert a scaling shift between two fixed-point operations, they normally ignore whether their compilers can translate the two operations later as

part of a single efficient machine instruction. Such ignorance often raises a critical performance issue on fixed-point DSP processors because these processors mostly aim to gain the performance via DSP-specific CISC instructions, each of which is typically a composite instruction that encodes multiple operations in a single word [10].

In this paper, we discuss how we complement existing FFC techniques through algebraic transformations to facilitate better code generation. And we present how the rules for algebraic transformation are easily described in Architecture Description Language (ADL), and how our retargetable compiler use them to generate efficient code. For this, we start our discussion with the description of a typical FFC process in Section 3. Then in Section 4, we describe our optimization technique that transforms IR (intermediate representation) using algebraic transformation. We introduce our retargetable compiler, Soargen[14], and how the rules for algebraic transformation are described in SoadDL, which is our ADL, in Section 5. Section 6 shows the experimental results and we conclude the paper in Section 7.

## 3. Floating-point to Fixed-point Conversion

Typically, a fixed-point data format D consists of three fields of bits: a sign bit, integer bits and fractional bits. The integer word length (IWL) and fractional word length(FWL) represent the number of integer bits and that of fractional bits, respectively [1]. The word length (WL) of D can be defined as 1+IWL+FWL. As an example consider a variable x in Figure 2, which stores a binary value 01010110 in an 8-bit data format with IWL = 3 and FWL = 4. It represents a positive binary number 101.011.So its value is interpreted as 5.375. Similarly, the value of y in the example is interpreted as 1.5625.

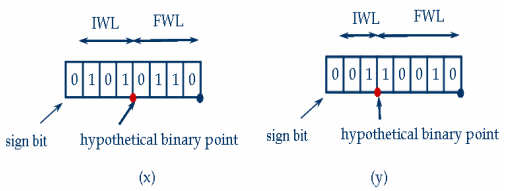



Figure 2. an example for fixed point data format

FFC techniques are elaborated to maximize the SQNR of the application while preventing new overflows from being introduced through the conversion process. The SQNR may be improved by minimizing quantization error, which is the numeric error occurred when a value requiring a data format with longer word length is stored to a shorter word. For a fixed-point format, the precision of the format is identical to its WL since the amount of quantization error is inversely proportional to the WL [10]. In theory, the longer WL the format has, the higher precision we have. But in practice, the WL is limited by hardware constraints. In fixed-point DSP processors, it is typically 16 bits for float-type formats and 32 bits for double-type ones. Thus, many FFC techniques employ a simple heuristic that assigns 16-bit integers for float-type variables and 32-bit integers for double-type variables.

Once the WL of a fixed-point data format is determined for each variable v, the IWL and FWL in the format are to be carefully selected to prevent overflows. This decision is contingent on the maximum value |vmax| that v can have at run time. Clearly, the IWL must be no less than $\lceil \log_2 |vmax| \rceil$ to avoid overflows. To find |vmax|, the

run-time value ranges of all variables in the code must be evaluated during the FFC process. For the evaluation, there have been two approaches [2]. In the first one, the value range of a variable is estimated from its statistical parameters obtained with a floating-point code simulation. One advantage of this approach is that it can obtain an accurate estimation of the range for specific signal input patterns. Another advantage is that it ensures a low probability of overflow for signal with the same input patterns. However, for different input patterns, the estimated results may be incorrect. The other approach [6][7] uses interval analysis [5] to estimate the value analytically. The estimated results are conservative so they are always safe and ensure no overflow. But for some cases, this approach may take too conservative stands to have the useful range information for effective FFC.

Using the value range of each variable v, we can identify |vmax| from which we determine IWLv and FWLv for v. For IWLv |vmax|, we can guarantee no overflow of v. Notice that particularly when IWLv = |vmax|, the precision of v is maximized. However, if we choose a naïve policy that uniformly assigns all variables their maximum values for their IWLs, we would have many different variables with many different IWLs. Previous studies indicate that this policy generally results in many scaling shifts inserted in the final code. Therefore, often in practice, a heuristic is additionally applied to reduce the number of scaling shifts by assigning the same IWL to the two variables with different maximum values, despite some precision loss.

Table 1 displays fixed-point arithmetic rules which direct when and where to insert scaling shifts inside the fixed-point code during the FFC process. In the table, Iv denotes the IWL of a fixed-point variable v. To briefly explain the rules, suppose that we have Ix>Iy for two variables x and y. According to the rules, for assignment x=y, we should perform y>>(Ix-Iy) to align the radix point of y to that of x before y is assigned to x. Likewise, if Ix<Iy, we should perform y << (Ix-Iy) before the assignment. In reality, floating-point arithmetic operations are either additive or multiplicative.

We use the notation to denote the additive operations and the notation the multiplicative operations. As shown in Table 1, an additive floating-point operation z=xy can be converted into either of three fixed-point operation patterns containing scaling shifts. Note hereby that two scaling shifts are always added in any situation.

| | floating point | fixed-point | | | IWL |
|---|---|---|---|---|---|
| | | Ix>Iy, Iz | Iy>Ix, Iz | Iz>Ix,Iy | |
| assign | x = y | x=y>>(Ix-Iy) | x=y<<(Iy-Ix) | | No change in Ix |
| add/sub | z = xy | z = (x(y>>(Ix-Iy)))<<(Ix-Iz) | z = ((x>>(Iy-Ix))y)<<(Iy-Iz) | z = (x>>(Iz-Ix))(y>>(Iz-Iy)) | Iz = max (Ix,Iy,Iz) |
| mult | z = xy | multf(x,y) | | | Iz = Ix +Iy+1 |

Table 1. fixed point arithmetic rules.

Fixed-point processors commonly provide dedicated functions for fixed-point multiplication as well as ordinary integer multiplication. This is because integer multiplication stores the lower half of the product while fixed-point multiplication needs to access the upper half [1]. For instance, the ZSP fixed-point processor supports two intrinsics: N_mul and N_extract. The first one performs 16-bit fixed-point multiplication and returns the result in 32 bits, and the second returns the upper half of the 32-bit result. In our work, we define a C function multf (see Table 1) for fixed-point multiplication on our target processor. The function can be implemented on the ZSP fixed-point processor as follows.

```
inline long multf(int a, long b){
long z;  // 32 bits
int x,y;  // 16 bits
x = a;
N_extract(y,b);
N_mul(z,x,y);
return z;
}
```

Notice from Table 1 that the IWL for the product of two variables is the sum of their IWLs with an extra 1-bit extension. Using the rules in the table, the code in Figure 1 (b) has been converted to the one in Figure 1 (c). Based on the range analysis, the IWLs for all variables in Figure 1 (b) are estimated as below.

```
Iw(n) = 2; Ix(n) = 5; Iai1 = 1; Iai2 = 0;
Ibi0 = 0; Ibi1 = -1; Ibi2 = 1; Iy(n) = 3.
```

After substituting a multf for each floating-point multiplication, we will have for the first line of Figure 1 (b),

w(n)=x(n)-multf(ai1,w(n-1))-multf(ai2,w(n-2))

Accoring to the rules, the IWLs of the two multfs would evaluate to 4 and 3, respectively. Using these IWLs, we insert scaling shifts into the code as guided by the fixed-point arithmetic rules, and consequently produce the fixed-point code on the first line of Figure 1 (c).

## 4. Algebraic Transformation

In this section we discuss how algebraic transformations can be applied to a give DAG IR so as to move the scaling shifts inserted as described in Section 3.

### 4.1 Rewriting Rules for Transformations

Algebraic transformations have been used in many domains such as compiler optimizations [12] and high-level synthesis [13]. Given an arbitrary DAG, finding its optimal transformation subject to certain conditions is a well-known intractable problem. So in practice, the problem is approximated by a series of local pattern matching problems where a predetermined set of rewriting rules are applied subsequently to varied subgraphs of the DAG in order to gradually form an (near-)optimal structure. A rewriting rule, pspt, consists of a pair of patterns ps and pt, which we call the source pattern and target pattern, respectively. When ps matches a subgraph of the subject DAG, the rule is applied by substituting pt for the subgraph in the DAG. Figure 3 lists three rewriting rules with the same source pattern p0 and its three functionally-equivalent target patterns (p1, p2, p3); that is, p0→p1, p0→p2 and p0→p3.

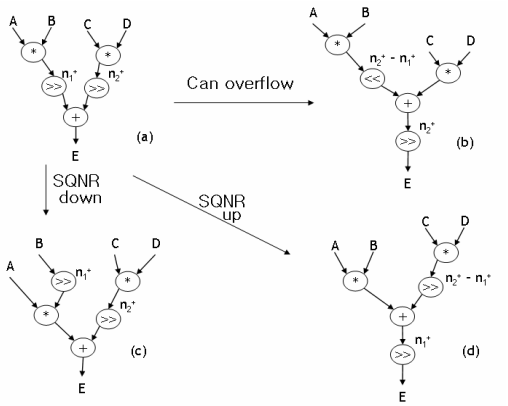



Figure 3. Rule based algebraic transformation

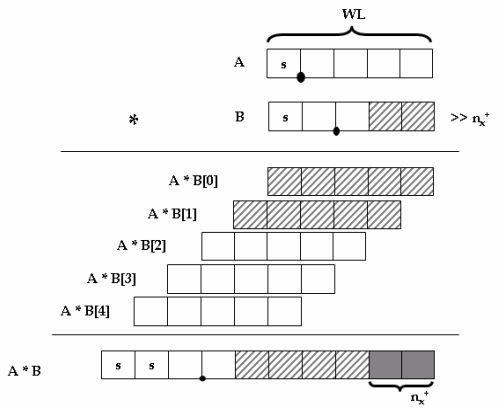



Figure 4 SQNR analysis

example where the operands of scaling shifts n1+ and n2+ are positive integers such that without loss of generality, n2+ > n1+.

Although different rules bring the same effect on code generation, they usually have different effects on the SQNR (or precision) and overflow within the output code. Therefore, when we define new rules, we must predict their exact effects and exclude any rules with bad effects. For instance, all three rules in Figure 3 basically lead the original DAG to the forms that a compiler finds an operation pattern for the mac instruction from. However, notice that unlike the other target patterns, the pattern p1 makes the code more vulnerable to fixed-point overflows than the source pattern p0. Thus, we will disregard the rule p0→p1 inour transformations.

As for p0→p2, we find that the SQNR is degraded if the rule is applied. To explain this, consider Figure 4 where we multiply two 5-bit integers A and B. The main difference between p0 and p2 is whether the right shift by nx+ bits is applied before or after the multiplication. According to Figure 4, when nx+ = 2, the result of (A*B)>>nx+ would contain error only at the least significant 2 bits. However, in the case of A*(B>>nx+), the error would contaminate the result up to the last 6 bits. As a rule of thumb, if we lose 1-bit information in a fixed-point format, the SQNR is degraded by 6 dB. Therefore, in this case, the SQNR will be degraded roughly by 36 dB. Unlike overflow, the SQNR is not a critical factor that determines whether a rule is included or not. In our work, this rule still will be considered for transformation unless the deterioration of the SQNR by 36 dB is beyond the allowable limits which have been predetermined by the programmer.

In case of p0→p3, we see that the original SQNR is improved since the product A*B is used immediately (without right shifts in-between) by the subsequent add operator, thereby preserving the data at the least significant n1+ bits. Also, there will be no overflow even if the product is directly given to the add operator without scaling down. This is because it has two sign bits at the most significant bits, as shown in Figure 4.

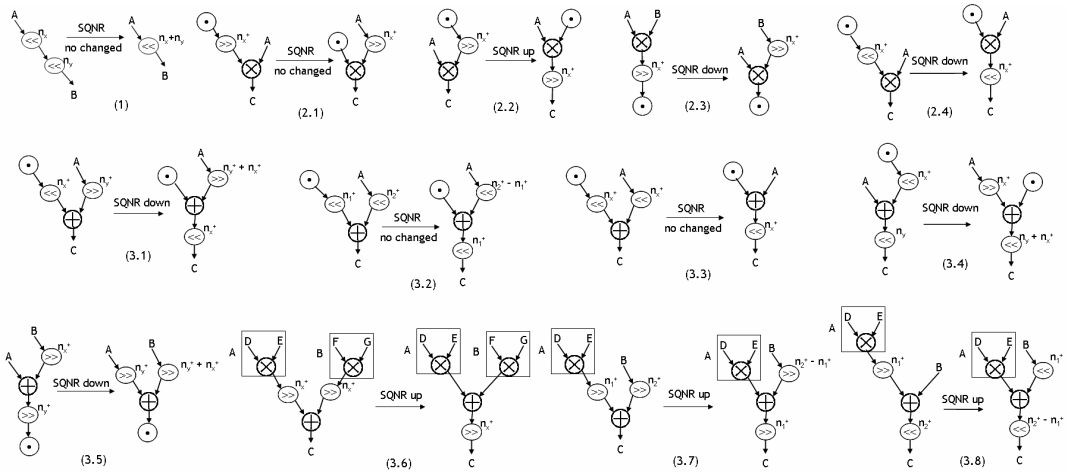



Figure 5. Rewriting Rules

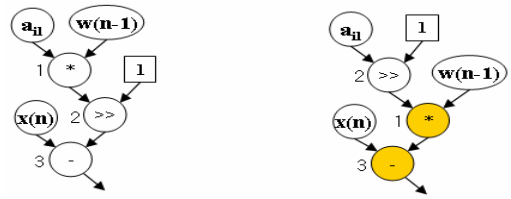
Figure 5 shows all the rules defined for our work, each of which contains scaling operations. These rules were built according to the arithmetic rules in Table 1. Given a subject DAG, the complexity of algebraic transformations grows rapidly as the number of rewriting rules increases [12]. The number of rules is the exponentially proportional to the size of patterns in each rule. Therefore, as can be seen from Figure 5, the pattern is restricted to encompass the operators at the distance of at most two from the scaling shift at the center. The rationale for this is that composite instructions are normally generated by the compiler from at most three operations on neighboring nodes in the IR.

As displayed in Figure 5, we divide the arithmetic operators in a pattern into three classes: additive , multiplicative and scaling shift operators. In the figure, the symbol ⊙ denotes an arbitrary arithmetic operator including and . We also divide the patterns in the figure roughly into three cases, depending on the relative positions of these operators. The first case is when two scaling shifts are adjacent, as shown in rule 1 of Figure 5 (1). Ordinary shifts for other than scaling cannot always be merged since they are usually used for masking their operands. But, we find that any adjacent scaling shifts can be safely merged without detrimental effects on the SQNR and overflow. So, in our transformations, an expression

B=(A<<nx)>>ny would be simplified to
B=A<<(nx-ny), according to the rule 1.

The second cases can be found from Figure 5 (2.1) to (2.4), where a scaling shift is adjacent to an operator, intervening between the operator and another one ⊙. If the processor has a composite instruction consisting of ⊙ and , we may want to move this scaling shift out of this place by the four rules2.1, 2.2, 2.3 and 2.4, thereby allowing the compiler to generate the composite instruction. Note that rules 2.1, 2.2 and 2.3 contain a right shift, and rule 2.4 contains a left shift. We can see that the two operators ⊙ and are neighboring

in the target patterns, facilitating code generation of a composite instruction [⊙,]. As an example, the patterns C=A(B>>nx) and C=(AB)>>nx are functionally equivalent. So the first pattern can be transformed to the second one by rule 2.2, or inversely by rule 2.3. If B is ⊙, then we will apply rule 2.2. If C is ⊙, we will apply rule 2.3. As explained above with Figure 4, rule 2.2 improves the SQNR while rule 2.3 does the opposite. Lastly, the remaining ten rules in Figure 5 correspond to the third case where a scaling shift is adjacent to an operator and intervenes between and ⊙. We can easily prove by well-known algebraic properties that all rules perform valid transformations between functionally-equivalent expression DAGs.




(a) Original IR (b) Transformed IR

Figure 6. DAG IRs for IIR Filter Code

| Priority | Rules | Changes in precision and computation |
|---|---|---|
| 1 | 3.6 | precision ↑, computation ↓ |
| 2 | 1 | computation ↓ |
| 3 | 2.2, 3.7, 3.8 | precision ↑ |
| 4 | 2.1, 3.2, 3.3 | no change |
| 5 | 2.3, 2.4, 3.1, 3.4, 3.5 | precision ↓ |

Table 2. Priorities of Rules in Figure 5

## 4.2 Priority-based Rule Application

In this subsection, we discuss how we apply the rules in Figure 5 to solve a local pattern matching problem in our transformations. We use a conventional DAG pattern matching algorithm for our problem [12]. To reduce the complexity of the pattern matching, we prioritize all the rules in the following sequence.

The priority is given according to the two metrics: precision and computation. The precision is evaluated by the values of SQNR, and the computation is by the number of nodes in the pattern. When two rules are simultaneously applicable, the one with the higher priority will be used to transform the subject DAG. For example, in Figure 6(a), nodes 1 and 3 can be combined and translated to a mac instruction if node 2 is removed from the two nodes via both rules 2.3 and 3.8. However, in this case, we prefer 3.8 since it has a higher priority two nodes via both rules 2.3 and 3.8. However, in this case, we prefer 3.8 since it has a higher priority over 2.3 as shown in Table 2.

Our pattern matching is priority-based peephole optimization. This means that a rule is applied only when its target pattern is found to be useful for the code generation on our fixed-point processor. The usefulness is determined by either machine-independent or machine-independent properties. Each rule is iteratively applied to the subject DAG until no more rules are applicable.

# 5. ADL-based Compilation Framework

In this section, we first discuss the overall structure of our retargetable compiler, and then describe our ADL with examples to demonstrate how a given ISA is described in this language and the description is used to target the compiler at the ISA. Finally, we

show that the rules for algebraic transformation are easily described in our ADL so that our compiler can recognize the rules and use it to select and generate instructions.

## 5.1 Overview of the compiler

Figure 7 shows our compiler infrastructure where retargetability can be achieved by enabling the users to describe their target architectures in our ADL. The ADL characterizes an architecture by specifying its structural and behavioral information. Although it is still being extended at present, structural information in the current implementation only describes register and memory architecture. Behavioral information contains a set of machine instructions and addressing modes.

As can be seen in Figure 7, the compiler is implemented with several C++ modules such as

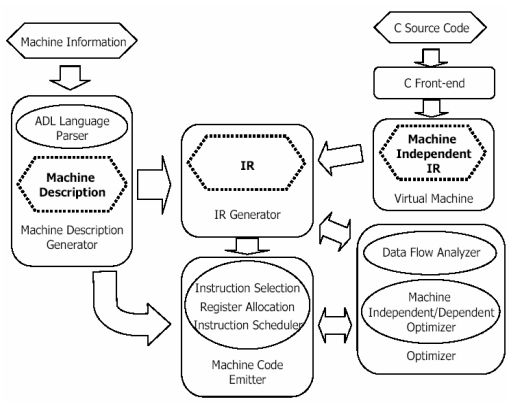



Figure 7. Our ADL-based Compiler Infrastructure

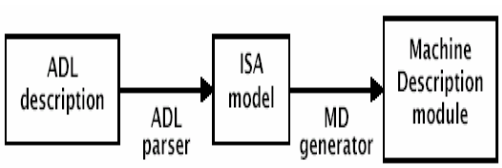



Figure 8. ADL Translation Process for the Code Generation

MachineDescription (MD), IRGen, VirtualMachine (VM), CodeGen, GlobalRegAllocator and Optimizer. The MD module contains a collection of C++ routines that carry all the machine specific information necessary for the compiler. As shown in Figure 8, from the ADL description for an architecture, an MD module is automatically generated and given as input to the CodeGen module which uses the module as a set of machine instruction templates in the phases of instruction selection, register allocation and instruction scheduling.

The ISA model is a group of C++ data structures all constituting an ISA template that will be used to build MD routines. It consists of two major components, resource and operation. The resource component represents storage elements such as registers and memory. The operation component abstracts the ISA of the target machine. Each address mode and instruction description is converted into an instruction template in operation of the ISA model. Among the attribute of an instruction template, the action template represents register transfer level behavior directly. The action template is a list of tree-shaped register transfers and the tree shaped templates can be directly used in the CodeGen module for instruction selection.

The MD generator builds a machine description module from an ISA model. Because our compiler needs various parameters for efficient code generation, the MD generator analyzes the ISA model and extracts necessary information such as register classes or register transfer graphs. The VM module provides a generic interface between the C front-end and our compiler. The virtual machine is an imaginary

machine with virtual assembly as its ISA. The virtual assembly is a simultaneous composition of a list of register transfer expressions (RTEs), each of which corresponds to a single instruction of the form (set lvalue rvalue) where the rvalue expression is evaluated and stored to the lvalue location. Operators in the expression are unary or binary depending on their types, and operands can be either symbolic registers or memory locations. The following shows an example of virtual assembly code:

```
L8:
  (set (SI: r2) (SI: 12(fp)))
  (set (SI: r3) (SI: 0(r2)))
  (set (SI: r4) (SI: 4(fp)))
  (set (SI: r2) (SI: 0(r4)))
  (set (SI: r2) (mult:SI (SI: r3) (SI: r2)))
  (set (SI: r3) (SI: 16(fp)))
  (set (SI: r3) (ss_plus:SI (SI: r3) (SI: r2)))
  (set (SI: 16(fp)) (SI: r3))
```

The virtual assembly is very intuitive. For instance, the first line means ‘load from memory located at fp + 12 to register r2’. All possible machine independent optimizations are performed by the front-end on virtual assembly. When the user compiles application code, virtual assembly code is first produced, and then converted to a graph-structural intermediate representation (IR) through common sub-expression elimination and control flow analysis. Our IR has a hierarchical graph structure. Each basic block node is a forest containing trees or DAGs, each of which represents a set of interdependent RTEs. Several basic block nodes in the same function form a control flow graph (CFG) to represent a function node. Finally, all function

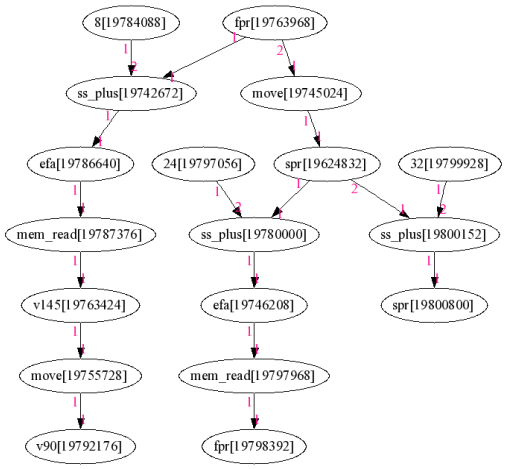



Figure 9. Visualization of the IR for the Kernel Code of convolution

nodes in a program forms a call graph representing the whole program. To visualize the entire hierarchical structure, we developed a visualization tool, called GraphViz. As demonstrated in Figure 9, this tool has been greatly helpful for us to analyze source code and debug our compiler modules when they are targeted to a new processor. The IR glues all compiler modules through a uniform interface. For instance, data flow analysis techniques such as reaching definition and live range analysis in the Optimizer module are performed on the code in the IR.

A DAG containing RTEs at the lowest level of the IR hierarchy is called an Expression DAG (EDAG). The EDAG represents data dependency between operators and values that the operators produces and consumes. In this sense, the nodes in an EDAG can be largely classified into two types: operator and value nodes. The value nodes are further broken down into four - that is, symbolic variable, memory location, effective address and constant. When the application code is transformed to

an IR, the CodeGen and GlobalRegAllocator modules sequentially take the IR and generate the target code using the routines in the MD module.

## 5.2 ADL for Machine Description

As stated earlier, the main purpose of an ADL is to provide a formal method to describe a target processor as is necessary to verify the completeness and correctness of the ISA. To attain this purpose, our ADL has been rigorously built on the formal definition of ISADesc.

***Definition 1. ISADesc=<IS,AM,ST,RIA,RAS>, where***
***IS : a hierarchical structured set for the instructions of target architecture***
***AM : a set of addressing modes of target architecture***
***ST : a set of the storages of target architecture***
***RIA $\subseteq$ IS X $AM^n$, where n > 0***
***RAS $\subseteq$ AM X $ST^n$, where n > 0.***

The primitive section defines primitive operations and types. Each primitive operation stands for an atomic behavior of the target machine. For instance, primitives ss_minus and div stand for subtraction and division operations in hardware. Type information is also represented in the primitive section. For instance, the prefix ss_here represents a signed single precision type. The storage section gives abstract resource structure of the target machine such as memory and registers. Each storage element has two fields: number and mode. Figure 10 shows an example of primitive and storage descriptions.

```
primitive {
   type { qi 8; hi 16; si 32; byte 8; halfword 16; word 32; }
   operation {
      ss_plus; ss_minus; mult; div; ... mem_write; jump;
    }
}
storage {
   memory qi dmem {
        latency 1 @[0x0000..0x9999];
   }
   register si reg[16];
   programCounter r15 PC;
   stackPointer r13 SPR;
}
```

Figure 10. Primitive Operations and Storage declared in our ADL

The address mode and instruction sections describe the machine instructions and addressing modes in the target ISA. In ISADesc, both instructions and addressing modes have three fields: name, action and syntax. They describe abstract behavioral level actions of the target processor. They specify the instruction semantics explicitly and hide the hardware details. As an example, Figure 11 presents an add-shift-left instruction described in our ADL.

To effectively support a top-down design methodology, each description in these sections are hierarchically defined; that is, each description can include several lower level descriptions. Its hierarchical property makes it easy to manage the ISA, and allows us to independently describe instructions, addressing modes and storage, thereby maximizing reusability of the architecture description. At the bottom of this hierarchical structure lies primitives and storage elements defined in the primitive and storage sections. They play role of basic building blocks for the action field in addressing mode and instruction descriptions. This is an example description of displacement addressing mode defined by the register storage type and the primitive ss_plus.

***action { efa = ss_plus(Rd, imm5); }***

The address mode section defines how to access hardware resource like memory or registers. When we need a memory reference for a load/store instruction, the address mode can be used to represent the effective address for the access. Instructions are described in a similar way. The following shows a multiply and accumulate instruction.

***action { rd = mult(ss_plus(rs, rm)); }***

When an instruction is described, its operands are usually of the register or memory type directly designated by the storage section. However, the user defines more complex addressing modes and uses them as the operands in an instruction description. For instance, in Figure 11, addressing mode dataAddrMode1 is defined in the address mode section and used as an operand in the description for instruction addsl1.

```
addressModeSet   dataAddrMode1 : dataAddressMode {
     regAddrMode1;     imm4AddrMode;
}
addressModeSet   dataAddrMode2 : dataAddressMode {
     regAddrMode2;     imm8AddrMode;
}
addressMode   regAddrMode1 : dataAddrMode1 {
     reg r3;     action {  r3;  }     syntax {  r3;  }   cost(0);
}
addressMode   imm4AddrMode : dataAddrMode1 {
     integer(4) int4;     action  {  int4; }      syntax  {  int4; }
cost(0);
}
addressMode   regAddrMode2 : dataAddrMode2 {
     reg r3;     action {  r3;  }     syntax {  r3;  }   cost(0);
}
addressMode   imm8AddrMode : dataAddrMode2 {
     integer(8) int8;      action  {   int8; }     syntax  {  int8; }
cost(0)
}
instruction   addsl1: MultipleOps {
        reg  r1;
        dataAddrMode1 opn1;
        dataAddrMode2 opn2;
        action {   r1 = ashift(ss_plus(r1, opn1), opn2));   }
        syntax {   " addsl "::r1::", "::opn1::", "::opn2;   }
        cost(1);
}
instruction   addsl2 : MultipleOps {
        reg  r1, r2, r3;
        integer(8)  imm8;
        action {   r1 = ashift(ss_plus(r2, r3), imm8));   }
        syntax {   " addsl "::r1::", "::r2::", "::r3::", "::imm8;   }
        cost(1);
}
```

Figure 11. Our ADL Description of Two add-shift Instructions

## 5.3 Specifying Rule in ADL

The rewriting rules described in Section 4.1 can be specified in our ADL, enabling our compiler to perform the transformation before selecting instructions. Thanks to its retargetability, we can make rules for any target architecture. In this section, we describe how we can describe the rules in ADL, and how the compiler exploits this information during instruction selection phase. We retargeted our compiler to ZSP400 processor, and described one simple pattern for MAC instruction in ADL. Figure 12 shows the rewriting rule to utilizeMAC instruction. And Figure 13 shows the instruction set description of our ADL.

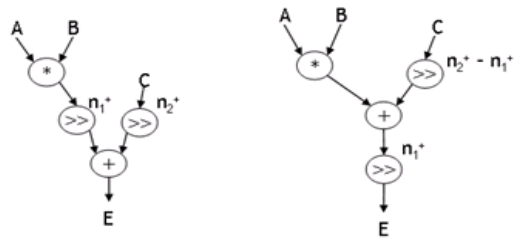



Figure 12. Rewriting Rule for MAC Instruction in ZSP400

```
instruction move shift up
: find MAC with moving shift
{
   uinteger(4) n1;
   uinteger(4) n2;
   GPR A, B;
   r0 C;

   action {
      C = ss plus(shift(\
                mult(A,B),n1, shift(C,n2));
   }

   syntax {
      "shla"::"  "::C::","::\
                   "("::n2::"-"::n1::")"
      "mac.a"::"  "::A::","::B;
      "shla"::"  "::C::","::n1;
   }
}
```

Figure 13. Example of ADL description for the rewriting rules.

## 6. Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of the proposed technique, which is implemented for Soargen compiler which is a retargetable compiler being developing in our group. The experimental input is a set of floating point code from DSPstone. In order to isolate the impacts on performance and code size purely from our techniques, two sets of executables for the ZSP400 processor are produced for the benchmark codes; ORGIN: floating point to fixed point conversion with original Autoscaler and TRANS: floating point to fixed point conversion with Autoscaler included the algebraic transformation. With these two sets of executables, we measured (1) cycle counts with simulator and (2) code size with utility tool. The performance improvements and code size reduction due to proposed technique are measured in percentage, using the formula:

((ORGIN-TRANS)/ORGIN)*100

Figure 14 reports the performance improvements, which is based on the proposed technique. The graph shows that there is up to 21.5% and average 12.7% performance improvement by using our technique.

Figure 15 demonstrates that we can reduce the code size by helping the compiler to select

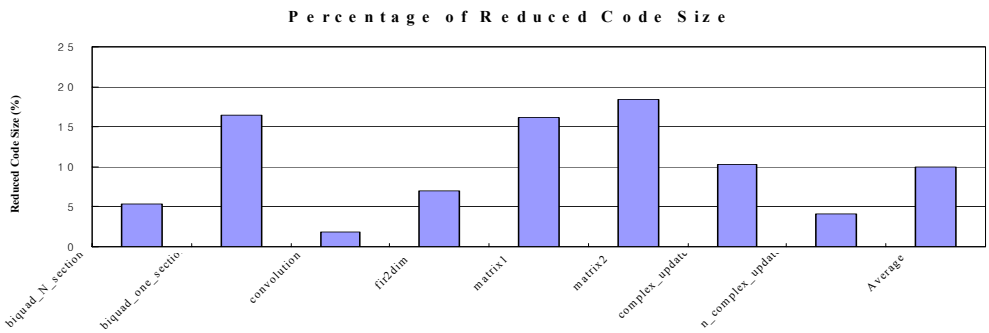



Figure 14. Reduced Execution Time


Figure 15. Reduced Code Size

DSP-specific instructions. The graph show that there is up to 16.7% and average 10% code size reduction. by using our technique.

## 7. Conclusion

For DSP systems, there have been many techniques to convert the floating-point to fixed-point. However, existing techniques do not consider the side effect of scaling shifts on code generation. Such ignorance often raises a critical performance issue on fixed-point DSP processors because these processors mostly aim to gain the performance via DSP-specific CISC instructions. In this paper, we propose retargetable compilation framework for a rule-based algebraic transformation to alleviate the side effect of scaling shifts. As a special case, we applied our transformation technique for ZSP400 processor using our ADL and compiler. We observed substantial improvement on code size and execution time.

### Acknowledgement

This work was partially funded by the MIC (Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), KRF contract D00191, MIC under Grant A1100-0501-0004 and IT R&D Project, the Korea Ministry of Science and Technology(MoST) under Grant M103BY010004-05B2501-00411, Nano IP/SoC promotion group of Seoul R&BD Program in 2006.

## References

[1] Ki-Il Kum, Jiyang Kang, Wonyong Sung, "autoscaler for C: An optimizing floating-point to Integer C Program Converter For fixed-Point Digital Signal Processors". IEEE Transactions on Circuits & Systems II -Analog and Digital Signal Processing, 47:840 – 848,September 2000

[2] Daniel Menard, Daniel Chillet, Francois Charot, Olivier Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation" CASES 2002.

[3] T. Gr¨otker, E. Multhaup, and O.Mauss. Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis. In ICSPAT'96, Boston, October 1996.

[4] S. Kim, K. Kum, and S. Wonyong. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. IEEE Transactions on Circuits and Systems II, 45(11), November 1998.

[5] R. Kearfott. Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin 2, 2(1): 95-112, 1996.

[6] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A Fixed-Point Design And Simulation Environment. In Design, Automation and Test in Europe, 1998.

[7] M. Willems, V. Bursgens, H. Keding, and H. Meyr. System Level Fixed-Point Design Based On An Interpolative Approach. In Design Automation Conference, 1997.

[8] C. Shi and R. Brodersen, Automated Fixed-point Data-type Optimization Tool for Signal Processing and Communication Systems. In Design Automation Conference, 2000.

[9] T. Parks and C. Burrus. Digital Filter Design. Jhon Willey and Sons Inc, 1987.

[10] P. Lapsely, J. Bier, A. Shoham and E. Lee, DSP Processor Fundamentals: Architectures and Features, IEEE Press 1997.
[11] ZSP 400 Digital Signal Processor Technical Manual, http://www.zsp.com.
[12] S. Muchinick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997.
[13] A. Chandrakasan, et. al, Optimizing Power Using Transformations. IEEE Transactions on CAD, Vol. 14, No. 1, 12–31, 1995
[14] M. Ahn, J. Cho, and Y. Paek, Using a H/W ADL-based Compiler for Fixed-point Audio Codec Optimization thru Application Specific Instructions. 정보처리학회논문지 제 13-A권 제4호, 2006.

박 상 현
2004년 서울대학교 전기공학부 (학사)
2004년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 저전력 설계.

안 민 욱
2003년 서울대학교 전기공학부 (학사)
2003년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 컴퓨터 시스템 설계

조 두 산
2001년 한국외국어대학교 전자정보공학부(학사)
2003년 고려대학교 전기공학과 (석사)
2003년~현재 서울대학교 전기컴퓨터공학부 박사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 컴퓨터 시스템 설계

윤 종 희
2005년 KAIST 전기및전자공학과(학사)
2005년~ 현재 서울대학교 전기컴퓨터공학부 석사과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 재구성 가능 프로세서

백 윤 흥
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과학(박사)
1997년~1999년 NJIT 조교수
1999년~2003년 KAIST 전자전산학과 부교수
2003년~현재 서울대학교 전기컴퓨터공학부 부교수
관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC

20-2-R02

# 온톨로지 기반의 시맨틱 어노테이터 구현
## (Implementation of Semantic Annotator Based on Ontology)

박 재 훈, 유 재 규, 전 양 승, 정 영 식, 한 성 국
원광대학교 컴퓨터공학과
(pjh98; jkyoo82; globaljeon; ysjeong; skhan)@wku.ac.k

요 약

이미 축적된 방대한 컨텐츠에 의미 기반 메타데이터 구축을 수작업으로 하는 것은 거의 불가능하다. 온톨로지나 시맨틱 웹 개념이 없는 일반 사용자가 자신의 컨텐츠에 효과적인 의미 정보를 부착을 하기란 매우 어렵다. 실질적인 KM, EDM, Semantic Portal, Semantic Search Engine 구현과 활성화를 위해서 필수적인 기술인 시맨틱 어노테이션은 자연언어 처리와 텍스트 마이닝 기술에 기반한다. 사람이름, 기업명, 주소 등의 개체명 인식과 사건, 원인, 결과, 상황 등의 정보 추출 기술에 기반하고 Semantic Annotation Tool은 다양한 온톨로지에 대한 적응력을 필요하며, 방대한 언어자원이 필수적이다. 이에 본 논문에서는 수많은 정보들에 태그를 붙여 컴퓨터가 의미처리를 할 수 있는 지식기반의 시맨틱 어노테이터를 구현한다.

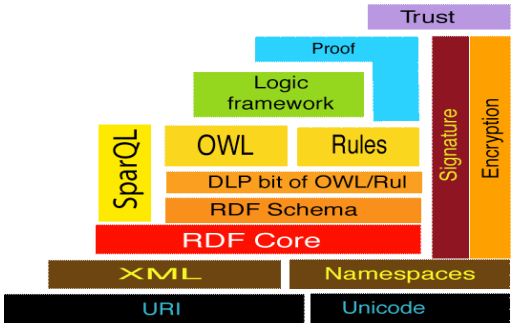
## 1. 서 론

### 1. 연구 배경

인터넷의 발전으로 기하급수적으로 늘어나고 있는 정보의 양은 사용자들에게 많은 지식과 다양한 서비스를 제공하고 있는 반면에 정보홍수라는 새로운 문제점을 초래하고 있다. 다양한 시도와 접근의 검색엔진이 개발되어 이러한 문제점을 해결하려고 시도하고 있지만 대부분의 검색엔진이 웹 문서의 내용보다는 단어나 구문 등의 키워드를 이용한 단편적인 방법으로 관련성을 검색하므로 사용자의 질의와는 관계없는 많은 문서를 결과로 가져올 수 있다. 이로 인해 사용자는 불필요한 정보를 걸러내느라 시간을 낭비하게 된다. 이런 문제점이 발생하는 가장 주된 원인은 현재의 웹이 사람을 위한 것이고 이를 위해 사람이 보고 잘 이해할 수 있도록 하기 위한 브라우저의 디스플레이 또는 레이아웃 기술에 초점을 맞추고 있다는 것이다. HTML 언어의 특징이 바로 이러한 디스플레이 용이라는 사실만 봐도 그러하다. HTML 을 이용하여 문서의 내용과 의미를 나타내는 시맨틱 정보를 표현하기가 어려우며, 따라서 사람이 아닌 프로그램 또는 소프트웨어 에이전트가 자동으로 문서로부터 의미를 추출하기가 어렵다. 시맨틱 웹은 메타데이터의 개념을 통하여 웹 문서에 시맨틱 정보를 덧붙이고 이를 이용하여 소프트웨어 에이전트가 이 의미 정보를 자동으로 추출할 수 있는 패러다임을 조성하는 것이다.

### 1.1.2. 문제 제기와 해결책

Tim Berners-Lee는 시맨틱 웹이 기존의 웹과 완전히 구별되는 새로운 웹의 개념이 아니라 현재 웹을 확장하여 웹에 올라오는 정보에 잘 정의된 의미를 부여하고 이를 통해 컴퓨터와 사람이 협동적으로 작업을 수행할 수 있도록 하는 패러다임이라고 그 역할을 정의하였다. 대표적인 월드와이드웹 표준화 단체인 W3C(World Wide Web Consortium)에서는 시맨틱 웹을 RDF나 다른 표준 기반으로 웹에 있는 데이터를 추상적으로 표현하는 것이라 정의하였다[1].




(그림 1) 시맨틱 웹의 계층구조

시맨틱 웹의 궁극적인 목적은 웹에 있는 정보를 컴퓨터가 좀 더 이해할 수 있도록 도와주는 표준과 기술을 개발하여 시맨틱 검색, 데이터 통합, 네비게이션, 태스크의 자동화 등을 지원하는 것이다. 시맨틱 웹을 실현하기 위한 다양한 접근방법이 제시되었다. 하지만 HTML을 기반으로 한 현재의 웹을 개선하는 기본 취지에서 보면 시맨틱 웹을 달성하기 위해 웹 프로토콜과 같은 하위 레벨의 개념을 정의하고 이 하위레벨을 이용하여 다음 레벨의 개념을 정의하는 계층구조를 설정하는 것이 일반적인 연구 방향이다.

온톨로지에 대한 정의는 여러 가지가 있지만 Gruber는 온톨로지를 “공유된 개념화(shared conceptualization)에 대한 정형화되고 명시적인 명세(formal and explicit specification)”라고 정의하였다. 온톨로지는 간단히 표현하면 단어와 관계들로 구성된 사전으로 어느 특정 도메인에 관련된 단어들을 계층적 구조로 표현하고 추가적으로 이를 확장할 수 있는 추론 규칙을 포함한다. 온톨로지의 역할 중 하나는 서로 다른 데이터베이스가 같은 개념에 대해서 서로 다른 단어나 식별자를 사용할 경우 이를 해결해주는데 있다. 온톨로지는 웹 기반의 지식처리나 응용 프로그램 사이의 지식 공유, 재사용을 가능하게 하는 아주 중요한 요소로 자리 잡고 있다[4].

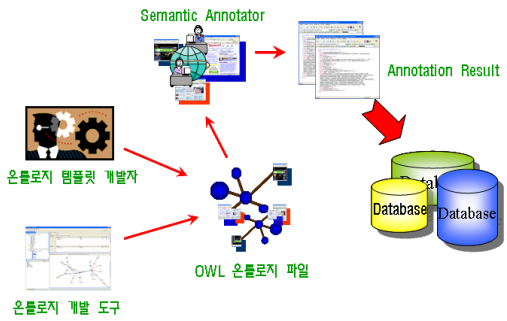
시맨틱 웹의 응용은 에이전트 기반의 웹 서비스 제공과 어노테이션 등과 같은 유용한 응용 프로그램의 개발로 요약된다. 어노테이션은 시맨틱 웹을 가장 쉽게 응용할 수 있는 매커니즘이다. 어노테이션은 이미 존재하는 웹 페이지에 대해 추가적인 설명을 덧붙여서 다시 웹에 공개하는 것으로 주로 정보 검색의 정확도를 높이는데 크게 기여할 수 있다.

## 2. 시맨틱 어노테이션

### 2.1. 어노테이션 방식

수많은 정보들에 태그를 붙여 컴퓨터가 의미처리를 할 수 있도록 지식베이스를 구축하는 것이 어노테이터의 목적이다. 이런 지식베이스는 사용자가 원하는 정확한 정보를 컴퓨터가 찾아줄 수 있다. 대표적인 어노테이션 방식은 다음과 같다.

첫째, 직접 온톨로지 문서에 어노테이션하여 저장하는 방식이다. 이 방식은 방대한 온톨로지 문서로 검색 시 불필요한 시간이 소모된다. 계속적인 파싱으로 인해 서버의 과부하를 발생하게 된다. 둘째, 다른 하나의




(그림 2) 어노테이션 시스템 구성도

XML 문서를 만들어 어노테이션 한 결과만을 저장하는 방식이다. 보다 가벼운 문서들로 인해 빠른 검색과 수정 및 관리가 용이하다. 또한 온톨로지 템플릿을 통해 많이 사용하는 주제의 틀을 만들어 배포한 뒤 사용자들에게 자신이 만들 온톨로지에 맞는 틀에 인스턴스를 추가하므로 편리하다. 셋째, 웹 페이지의 HTML 파일 자체에 어노테이션 한 결과를 입혀 쓰는 방식이다. 웹 페이지의 소스를 그대로 가져와서 따로 저장한다. HTML의 무질서한 태그, 의미를 내포하지 않은 보이기 위한 태그들로 인한 혼란을 야기한다.

### 2.2. 개선된 어노테이션 방식

본 논문에서 구현한 어노테이터의 어노테이션 방식은 기존의 방식 중 첫째와 셋째를 혼합한 방식이다. 어노테이션은 온톨로지 기반이고 웹 페이지의 텍스트를 블록 지정해 온톨로지에 드래그하는 방식이다. 온톨로지를 웹에서 표현하기 위한 OWL(Ontology Web Language) 파일을 파싱하여 클래스를 추출한 후 계층구조인 트리 형태로 출력한다.

온톨로지 개발도구를 이용하거나 온톨로지 템플릿 개발자는 유효한 온톨로지 파일을 제작하고 이를 시맨틱 어노테이터에서 사용한다. 시맨틱 어노테이터 프로그램에서 URL을 입력해 웹 사이트를 열고 원하는 단어나 문장을 드래그해서 온톨로지 클래스 트리에 드롭하면 자동으로 어노테이션 결과가 기록되는 방식이다. 웹 서핑을 하면서 드래그 앤 드롭으로 어노테이션 할 수 있으므로 사용자들에게는 보다 정확하고 명료한 컨텐츠를 제공할 수 있다.

## 3. 시맨틱 어노테이션 구현

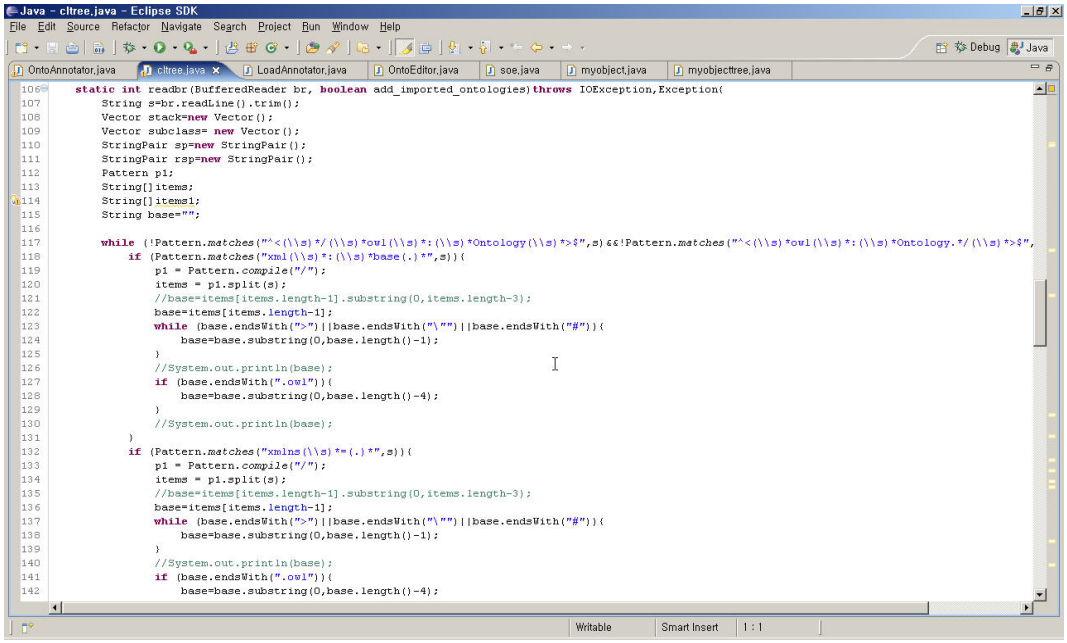
### 3.1. 온톨로지 파싱

온톨로지 기반의 어노테이션을 위해 온톨로지는 파싱되어 클래스를 추출하는 선행 작업이 필요하다.

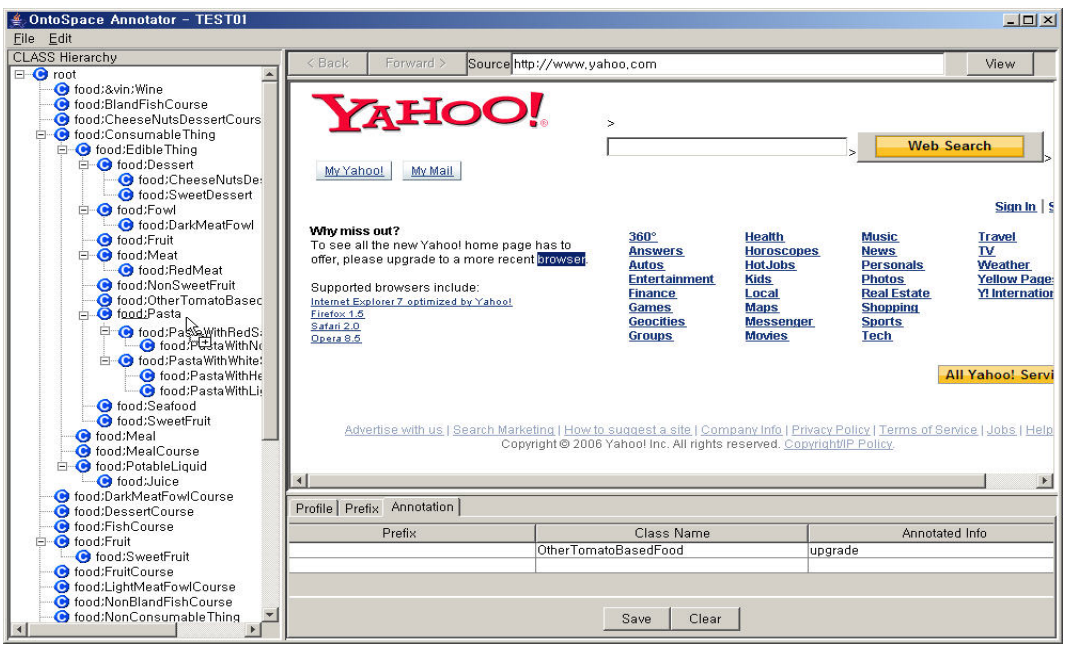
그림 3은 로컬 드라이브의 온톨로지 파일을 선택하면 온톨로지 파일 내용을 버퍼에 저장하고 읽어가며 클래스만 추출해 트리로 구성한다. 온톨로지는 계층구조로 이루어져 있기 때문에 온톨로지의 클래스 계층구조를 표현하는데 트리가 적합하기 때문이다. 온톨로지 파일을 파싱할 때 무결성을 보장하기 위해 온톨로지 파일은 유효성 체크가 되어 있는 온톨로지를 사용해야 한다. 본 논문에서는 food 온톨로지를 샘플로 사용해 클래스 트리 출력하고 이를 기반으로 어노테이션 결과를 저장한다. 어노테이션의 결과는 데이터베이스 형태로 저장된다.

그림 4는 시맨틱 어노테이션 프로그램을 이용해 드래그 앤 드롭 방식으로 어노테이션 하는 화면이다. 왼쪽의 트리는 food 온톨로지 파일을 클래스만 추출한 트리이다. 오른쪽 웹 브라우저에서 원하는 단어나 문장을 드래그한 후 왼쪽 트리에 드롭하면 온톨로지의 클래스명과 드래그 한 단어가 하단의 탭에 보여 진다. 하단 탭은 프로젝트의 정보와 어노테이션 결과를 보여준다.

어노테이션 결과는 사용자가 직접 정보를 입력하는 방식이 아닌 자동 입력이다. 온톨




(그림 3) 온톨로지 파일 파싱




(그림 4) 어노테이션 (드래그 & 드롭)

로는 의미를 가진 체계적인 구조이기 때문에 자동화 어노테이션 결과는 명확하다. 추후 검색 시스템에 활용할 경우 별도의 인덱싱 작업 필요 없이 효율적인 검색이 가능하다.

## 4. 결 론

본 논문에서 제시한 시맨틱 어노테이터는 온톨로지 파일을 기반으로 한다. 어노테이션을 하려면 프로그램을 시작한 후 매번 파일을 로딩하고 파싱해서 클래스 트리를 구성해야하는 시간적인 소모가 있다. 온톨로지 파일을 파싱해 클래스 트리를 구성할 때 데이터베이스에서 클래스 정보를 읽어와 트리를 구성한다면 시간을 절약하고 번거로움을 극복할 수 있다. 이를 위해 온톨로지 파일을 관계형 데이터베이스로 변환하는 연구가 필요하다. 온톨로지 파일의 문법적인 구성을 분석해 관계형 스키마를 추출하면 온톨로지 파일의 파싱을 통해 데이터베이스로 저장이 가능하다. 그리고 하나씩 어노테이션 하던 방식 외에 대량의 어노테이션 또는 자동 어노테이션 방식을 연구해 도입하면 전문지식이 없는 일반 사용자도 쉽게 어노테이션 할 수 있다. 어노테이션의 대상은 단어나 문장과 같은 텍스트 형식의 정보에만 국한되면 안 된다. 현재 웹에는 이미지 정보, 표 정보 등의 일정한 형식을 갖춘 정보들도 존재한다. 단순 텍스트 정보뿐만 아니라 이미지나 표와 같은 형식의 정보도 어노테이션 할 수 있는 연구가 필요하다. 본 연구는 아직 초기 단계이기에 많이 부족하지만 보완점을 연구하고 개선한다면 추후 다른 연구에 더 나은 결과를 이끌어 낼 수 있는 초석이 되는 연구였다.

## Acknowledgement

이 논문은 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(원광대학교, 헬스케어기술개발사업단).

## 참 고 문 헌

[1] Berners-Lee, T., Hendler, J. and Lassila, O., *The Semantic Web,* Scientific American, 2001.

[2] Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdmann, M. and Horrocks, I., "The Semantic Web: the roles of XML and RDF", *IEEE Internet Computing*, Vol. 4, No. 5, pp.63-73, 2000.

[3] Lassila, O., "Web metadata: a matter of semantics", *IEEE Internet Computing*, Vol. 2, No. 4, pp.30-37, 1998.

[4] Gruber, T., "A translation approach to portable ontologies", *Knowledge Acquisition*, Vol. 5, No. 2, pp.199-220, 1993.

[5] McGuiness, D., Fikes, R., Hendler, J. and Stein, L., "DAML+OIL: an ontology language for the Semantic Web", *IEEE Intelligent Systems*, Vol. 17, No. 5, pp. 72-80, 2002.

[6] McIlraith, S., Son, T. and Honglei, Z., "Semantic Web services", *IEEE Intelligent Systems*, Vol. 16, No. 2, pp.46-53, 2001.

[7] Euzenat, J., "Eight questions about Semantic Web annotations", *IEEE Intelligent Systems*, Vol. 17, No. 2, pp.55-62, 2002.

[8] Heflin, J., Hendler, J. and Luke, S., "SHOE: a knowledge representation language for Internet applications", tech. report CS-TR-4078. Dept. of Computer Science, Univ. of Maryland at College Park, 1999.

[9] Swartz, A.,"MusicBrainz: a semantic Web service", *IEEE Intelligent Systems*, Vol. 17, No. 1, pp.76-77, 2002.

**박 재 훈**

2004 원광대학교 컴퓨터공학과 (공학사)

2005~현재 원광대학교 컴퓨터공학과 (석사과정)

관심분야: 웹서비스, 온톨로지 공학

**유 재 규**

2002~현재 원광대학교 전기전자 및 정보공학부

관심분야: 시맨틱 웹서비스, 온톨로지 공학

**전 양 승**

2001 원광대학 컴퓨터공학과 (공학사)

2006 원광대학교 컴퓨터공학과 (석사과정)

2006~현재 원광대학교 컴퓨터공학과(박사과정)

관심분야: 시맨틱 웹서비스, 온톨로지 공학, 지능형 e-Business

**정 영 식**

1993 고려대학교 전산학(박사)

1993~현재 원광대학교 컴퓨터공학부 교수

1997 미시간 주립대학교 전산학과 객원교수

2004 웨인 주립대학교 컴퓨터공학과 객원교수

관심분야: 그리드컴퓨팅, LBS, 분산병렬처리

**한 성 국**

1979 인하대학교 전자공학과 (공학박사)

1984~현재 원광대학교 컴퓨터공학부 교수

1989 University of Pennsylvania 방문교수

2003~2004 University of Innsbruck와 DERI 연구교수

2004~현재 대한전자공학회 컴퓨터소사이어티 감사

2005~현재 한국정보과학회 호남·제주지부장

관심분야: 시맨틱 웹서비스, 온톨로지 공학, 웹서비스, 의료정보, e-Learning

20-2-R03

# 다단계 프로그램에 대한 흐름을 고려한 분석[1]

*Flow-sensitive Analysis of Multi-Staged Programs*

김덕환 · 이광근
서울대학교 전기·컴퓨터공학부 프로그래밍 연구실
{dk; kwang}@ropas.snu.ac.kr

요 약

고차 함수를 지원하는 다단계 프로그래밍 언어에 대한 흐름을 고려한 집합 제약식 분석을 제안한다. 다단계 프로그램이란 프로그램 텍스트를 값처럼 다루는 프로그램이다. 최근에 다양한 언어 특징을 포함한 다단계 프로그램을 위한 let-다형 타입 체계와 그 추론 시스템이 제시되었다. 그러나, 타입 체계는 일반적으로 프로그램에서 실제로 실행되지 않는 부분에 대해서도 보수적으로 타입 검사를 수행하는 단점을 지니고 있다. 이 논문에서는 프로그램의 실행의 흐름을 고려하는 단순 타입 체계보다 정교한 프로그램 분석을 고안한다.

## 1 서론

다단계 프로그램(multi-staged program)이란 프로그램 텍스트를 전형적인(first-class) 값처럼 다루는 프로그램을 뜻한다. 즉, 실행 중에 프로그램의 원시 소스에 드러나지 않는 새로운 프로그램 텍스트를 조합하고 실행할 수 있다. 다단계 프로그램의 전형적인 예로는 부분 계산(partial evaluation) [2], 실시간 코드 생성(run-time code generation), 함수 펼치기(function inlining), 매크로 확장(macro expansion), 콰지-쿼트(quasi-quote) 시스템 [1]이 있다.

최근에 다단계 프로그램을 위한 let-다형 타입 체계(let-polymorphic type system)와 그 추론 시스템이 고안되었다 [3]. 그 타입 체계는 자유 변수를 포함한 코드 템플릿(open code template), 코드 템플릿에 대한 명령형 연산(imperative operation), 의도적으로 변수가 환경에 영향을 받도록 하는 치환(variable-capturing substitution), 환경에 영향을 받지 않도록 하는 치환(capture-avoiding substitution), 값을 코드로 변환(lifting)하는 연산을 모두 지원한다.

그러나, 타입 체계는 일반적으로 실제로 실행되지 않는 프로그램 부분에 대해서도 보수적으로 타입 검사를 수행하는 단점을 지니고 있다. 예를 들어, [3]의 타입 체계는 $\lambda x.(x\,x)$와 $\mathtt{box}_t\,(c\,c)$와 같은 프로그램을 받아들이지 못한다. 여기서, 함수 적용(function application)인 $(x\,x)$와 $(c\,c)$은 프로그램 실행 중에 실제 계산이 일어나지 않는 부분이다.

이 논문에서는 다단계 프로그램에 대한 흐름을 고려한 보다 정교한 분석을 제안한다. 안전할(sound) 뿐만 아니라, 단순 타입 체계보다 더 많은 올바른 프로그램을 받아들이는 것을 목표로 한다. 구체적으로, 위에서 말한 $\lambda x.(x\,x)$와 $\mathtt{box}_t\,(c\,c)$와 같은 프로그램도 올바른 프로그램으로 판단한다.

[1]이 연구는 정보통신부 선도기반기술개발사업과 교육인적자원부 두뇌한국21사업의 지원을 받았음을 밝힙니다.

$$
\begin{array}{rcl}
l & \in & \mathit{Label} \\
\\
e & \in & \mathit{Expr} \\
e & := & l : e' \\
e' & := & c \mid x \mid \lambda x.e \mid e\ e \mid \mathtt{box}_t\ e \mid \mathtt{unbox}_{k(>0)}\ e \mid \mathtt{eval}\ e
\end{array}
$$

[그림 1] 핵심 문법

분석은 집합 제약식(constraint-based) 분석의 형태로 표현한다. 단계가 없는(non-staged) 프로그램에 대한 Jens Palsbergs와 Michael I. Schwartzbach의 안전성 분석(safety analysis) [4]과 유사한 형태로 볼 수 있다.

# 2 언어

분석 대상 언어는 고차 함수(higher-order function)를 지원하는 값전달 호출(call-by-value) 방식의 전형적인 형태에 실행 중에 새로운 코드를 생성하고 실행할 수 있도록 하는 연산을 추가한 언어이다. 코드 템플릿 속에 포함된 자유 변수(free variable)가 그 코드 템플릿이 사용되는 환경에 따라 값이 달라지는 것을 허용한다.

## 2.1 핵심 문법

그림 1에 대상 언어의 핵심 문법(abstract syntax)을 나타내었다. 프로그램의 각 부분을 쉽게 지칭할 수 있도록, 프로그램의 모든 하부 수식에 표지(label)를 매달아 놓는다. 특별히 $\mathtt{box}_t$ 연산자들은 프로그램 어느 부분에서도 쉽게 지칭할 수 있도록 표지($t$)를 매달아 놓는다. 이러한 것들은 논문의 가독성을 위한 것들로 프로그램의 의미 구조에는 영향을 미치지 않는다. 이후로 프로그램의 특정 부분을 지칭할 때, 해당 식으로 직접 표현하거나 표지를 사용해서 간접적으로 표현할 것이다.

## 2.2 의미 구조

그림 2는 대상 언어의 과정을 드러내는 의미 구조(operational semantics)를 보여준다. $\sigma \vdash e \xrightarrow{n} v$는 환경 $\sigma$을 사용하여 수식 $e$를 $n$ 단계(stage)에서 계산하면 그 값이 $v$가 된다는 것을 의미한다.

다단계 프로그램에서는 값들을 단계 별로 분류할 수 있으며, 각각 $v^0$, $v^1$, $\cdots$로 표현한다.

$$
\begin{array}{rcl}
v & \in & \mathit{Val} = \mathit{Val}^0 + \mathit{Val}^1 + \cdots \\
v^n & \in & \mathit{Val}^n (n \geq 0) \\
v^0 & ::= & c \mid \langle \lambda x.e, \sigma \rangle \mid \mathtt{box}_t\ v^1 \\
v^n & ::= & c \mid x \mid \lambda x.v^n \mid v^n\ v^n \mid \mathtt{box}_t\ v^{n+1} \mid \mathtt{unbox}_k\ v^{n-k} \mid \mathtt{eval}\ v^n \quad (\text{단}, 0 < k < n)
\end{array}
$$

$\mathit{Var}$는 프로그램에 나타나는 모든 변수들의 유한 집합이다. 환경(environment)의 집합 $\mathit{Env}$는 변수들의 집합 $\mathit{Var}$에서 값의 집합 $\mathit{Val}$으로 가는 유한한 함수들의 집합이다.

$$\sigma \in \mathit{Env} = \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$$

$$[\texttt{ECON}]\ \frac{}{\sigma \vdash c \xrightarrow{n} c}$$

$$[\texttt{EVAR}]\ \frac{\sigma(x) = v}{\sigma \vdash x \xrightarrow{0} v} \qquad [\texttt{EVAR}']\ \frac{}{\sigma \vdash x \xrightarrow{n+1} x}$$

$$[\texttt{EABS}]\ \frac{}{\sigma \vdash \lambda x.e \xrightarrow{0} \langle \lambda x.e, \sigma \rangle} \qquad [\texttt{EABS}']\ \frac{\sigma \vdash e \xrightarrow{n+1} v}{\sigma \vdash \lambda x.e \xrightarrow{n+1} \lambda x.v}$$

$$[\texttt{EAPP}]\ \frac{\sigma \vdash e_1 \xrightarrow{0} \langle \lambda x.e', \sigma' \rangle \qquad \sigma \vdash e_2 \xrightarrow{0} v_2 \qquad \sigma'\{x \mapsto v_2\} \vdash e' \xrightarrow{0} v}{\sigma \vdash e_1\ e_2 \xrightarrow{0} v}$$

$$[\texttt{EAPP}']\ \frac{\sigma \vdash e_1 \xrightarrow{n+1} v_1 \qquad \sigma \vdash e_2 \xrightarrow{n+1} v_2}{\sigma \vdash e_1\ e_2 \xrightarrow{n+1} v_1\ v_2}$$

$$[\texttt{EBOX}]\ \frac{\sigma \vdash e \xrightarrow{n+1} v}{\sigma \vdash \texttt{box}_t\ e \xrightarrow{n} \texttt{box}_t\ v}$$

$$[\texttt{EUNBOX}]\ \frac{\sigma \vdash e \xrightarrow{0} \texttt{box}_t\ v}{\sigma \vdash \texttt{unbox}_{n+1}\ e \xrightarrow{n+1} v} \qquad [\texttt{EUNBOX}']\ \frac{\sigma \vdash e \xrightarrow{n+1} v \qquad k > 0}{\sigma \vdash \texttt{unbox}_k\ e \xrightarrow{n+1+k} \texttt{unbox}_k\ v}$$

$$[\texttt{EEVAL}]\ \frac{\sigma \vdash e \xrightarrow{0} \texttt{box}_t\ v' \qquad \varnothing \vdash v' \xrightarrow{0} v}{\sigma \vdash \texttt{eval}\ e \xrightarrow{0} v} \qquad [\texttt{EEVAL}']\ \frac{\sigma \vdash e \xrightarrow{n+1} v}{\sigma \vdash \texttt{eval}\ e \xrightarrow{n+1} \texttt{eval}\ v}$$

[그림 2] 과정을 드러내는 의미 구조

대상 언어는 전통적인 람다 계산법(lambda calculus)에 실행 중에 코드 템플릿을 조합하고 계산할 수 있도록 하는 연산자들을 추가한 언어이다. 단계 0에서의 계산들 [ECON], [EVAR], [EABS], [EAPP]는 값전달 호출 방식의 람다 계산법의 그것과 동일하다.

[EBOX]는 코드 템플릿을 생성하는 규칙이다. 닫힌(closed) 코드 템플릿 뿐만 아니라 코드 템플릿에 자유 변수가 포함된 열린(open) 형태의 코드 템플릿도 허용한다. 코드 템플릿 안의 하부 수식은 현재 단계보다 한 단계 높은 단계에서 계산한다. 현재의 단계가 0보다 크다면, 코드 템플릿 안을 계산 중인 상태를 의미하는데, 그런 단계에서는 $\beta$-줄이기($\beta$-reduction)가 일어나지 않고, [EUNBOX]에 의한 코드 치환 작업만 발생할 수 있다. [EUNBOX]에 의해 $\texttt{unbox}_k\ e$는 단계 $n$에서의 현재 코드 템플릿에 단계 $n-k$에서의 코드 템플릿을 삽입하는 역할을 한다. [EEVAL]은 하부 수식을 계산한 결과인 코드 템플릿 $\texttt{box}_t\ v'$을 프로그램 $v'$으로 변환한 다음, 다시 $v'$을 실행한다. 이 때, $v'$은 자유 변수가 없는 닫힌 코드 템플릿이어야 한다. 단계 0에서 자유롭게 $\alpha$-변환을 수행할 수 있도록 하기 위함이다. 일반적으로, 코드 템플릿 안에서는 $\alpha$-변환을 수행할 수 없다. 코드를 조합할 때 삽입되는 곳의 환경에 코드 템플릿 속에 포함된 자유 변수가 바인딩되기 때문에, 자유 변수의 이름을 그대로 보존해야 한다. 코드 템플릿 속에서 $\alpha$-변환을 수행하면 전체 프로그램의 의미가 달라질 수 있다. 반면, [EEVAL]에서 오직 닫힌 코드 템플릿만 허용하도록 제한하기 때문에, 코드 템플릿 속에 있던 자유 변수가 단계 0에서 환경에 바인딩되는 일은 일어나지 않는다. 따라서, 단계 0에서는 자유롭게 $\alpha$-변환을 수행할 수 있다.

$$[\texttt{TCON}]\quad \frac{}{\Gamma_0 \cdots \Gamma_n \vdash c : \iota}$$

$$[\texttt{TVAR}]\quad \frac{\Gamma_n(x) = \tau}{\Gamma_0 \cdots \Gamma_n \vdash x : \tau}$$

$$[\texttt{TABS}]\quad \frac{\Gamma_0 \cdots \Gamma_n + x : \tau_1 \vdash e : \tau_2}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$[\texttt{TAPP}]\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_0 \cdots \Gamma_n \vdash e_2 : \tau_1}{\Gamma_0 \cdots \Gamma_n \vdash e_1\ e_2 : \tau_2}$$

$$[\texttt{TBOX}]\quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : \tau}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}_t\ e : \Box(\Gamma \triangleright \tau)}$$

$$[\texttt{TUNBOX}]\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\Gamma_{n+k} \triangleright \tau)}{\Gamma_0 \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k\ e : \tau}$$

$$[\texttt{TEVAL}]\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \triangleright \tau)}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{eval}\ e : \tau}$$

[그림 3] 단순 타입 체계

# 3 타입 체계

그림 3은 대상 언어에 대한 단순 타입 체계(simple type system)를 보여주고 있다. [3]의 단순 타입 체계를 우리 언어에 맞게 정리한 것이다. $\Gamma_0 \cdots \Gamma_n \vdash e : \tau$는 타입 환경들 $\Gamma_0 \cdots \Gamma_n$ 하에서 단계 $n$ 상의 수식 $e$는 타입 $\tau$를 가짐을 의미한다.

세 가지 종류의 타입이 존재한다. 기저 타입($\iota$), 함수 타입($\tau \to \tau$), 코드 템플릿 타입($\Box(\Gamma \triangleright \tau)$)이 그것들이다. 타입 환경 $\Gamma$는 변수에서 타입으로 가는 유한 함수이다.

$$\begin{aligned} \tau &\in \mathit{Type} \\ \tau &::= \iota \mid \tau \to \tau \mid \Box(\Gamma \triangleright \tau) \end{aligned}$$

$$\Gamma \in \mathit{TypeEnv} = \mathit{Var} \xrightarrow{\text{fin}} \mathit{Type}$$

[TBOX]는 닫힌 코드 템플릿 뿐만 아니라 자유 변수를 포함한 열린 코드 템플릿도 타입을 가질 수 있도록 한다. $\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : \tau$가 자유 변수를 포함하고 있는 수식 $e$가 타입을 가질 수 있게 하기 때문이다. 이 때, 코드 템플릿 타입 $\Box(\Gamma \triangleright \tau)$에서 조건 $\Gamma$가 $\tau$ 타입의 코드 템플릿 속에 포함된 자유 변수들의 타입을 지정한다. [TUNBOX]는 현재 단계의 타입 환경이 하부 수식에서 만들어진 코드 템플릿이 요구하는 조건을 충족하는 지를 검사한다. [TEVAL]은 하부 수식을 계산해서 얻어진 코드 템플릿이 자유 변수를 포함하지 않은 닫힌 코드 템플릿인지 검사한다.

**정리 1 (안전성)** *$\varnothing \vdash e : \tau$이고 $\varnothing \vdash e \xrightarrow{0} v$이면, $\varnothing \vdash v : \tau$이다.*

$$
\begin{array}{rcl}
sv & \in & SetVar \\
sv & := & \mathcal{X}_l^{\mathcal{Y}_t} \mid \mathcal{Y}_t^x \mid \mathcal{Z}_l \\
\\
sc & \in & SetCstr \\
sc & := & \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \iota \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{Y}_{t'}^x \to \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \Box \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \\
 & & \mathcal{X}_l^{\mathcal{Y}_t} \supseteq apply(\mathcal{X}_{l'}^{\mathcal{Y}_t}, \mathcal{X}_{l''}^{\mathcal{Y}_t}) \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq unbox_k\ \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq eval\ \mathcal{X}_{l'}^{\mathcal{Y}_t} \mid \\
 & & \mathcal{Z}_l \supseteq true \mid \\
 & & \mathcal{Y}_t^x \subseteq \emptyset \mid \mathcal{Z}_l \Rightarrow \mathcal{Y}_t^x \subseteq \emptyset \mid \mathcal{Y}_t^x \not\subseteq \emptyset \mid \mathcal{Z}_l \Rightarrow \mathcal{Y}_t^x \not\subseteq \emptyset \mid \\
 & & \mathcal{X}_l^{\mathcal{Y}_t} \subseteq \text{LAMBDA} \mid \mathcal{Z}_l \Rightarrow \mathcal{X}_{l'}^{\mathcal{Y}_t} \subseteq \text{LAMBDA} \mid \mathcal{X}_l^{\mathcal{Y}_t} \subseteq \text{BOX} \mid \mathcal{Z}_l \Rightarrow \mathcal{X}_{l'}^{\mathcal{Y}_t} \subseteq \text{BOX} \mid \\
 & & \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{Y}_{t'}^x \mid \mathcal{Y}_t^x \supseteq \mathcal{X}_l^{\mathcal{Y}_{t'}} \mid \mathcal{Y}_t^x \supseteq \mathcal{Y}_{t'}^x \mid \mathcal{Z}_l \supseteq \mathcal{Z}_{l'}
\end{array}
$$

[그림 4] 집합 제약식

# 4 집합 제약식 분석

집합 제약식 분석은 프로그램을 훑어 집합 제약식들을 도출하고, 도출된 집합 제약식을 푸는 과정으로 이루어진다.

## 4.1 집합 제약식

그림 4에 분석에 사용하는 집합 제약식을 나타내었다.

집합 변수 $\mathcal{X}_l^{\mathcal{Y}_t}$, $\mathcal{Y}_t^x$는 각각 타입 체계의 타입 $\tau$과 타입 환경 $\Gamma$에 대응하는 정보를 모으는 변수들이다. $\mathcal{X}_l^{\mathcal{Y}_t}$은 타입 환경 $\mathcal{Y}_t^x, \mathcal{Y}_t^y, \cdots$ 하에서 실행되는 하부 수식 $l : e$이 가지는 타입 정보를 모으고, $\mathcal{Y}_t^x$은 프로그램 내의 $\texttt{box}_t$ 내부의 변수 $x$에 바인딩되는 값들을 모은다. 이와 달리, $\mathcal{Z}_l$는 타입 체계에는 대응되는 개념이 존재하지 않는 것으로 하부 수식 $l : e$에 대해 타입 검사를 할 필요가 있는 지를 보수적으로 판단하는 변수이다.

다양한 형태의 집합 제약식이 존재하지만, 근본적으로는 $X \supseteq Y$, $X \subseteq Y$, $X \not\subseteq Y$, $X \Rightarrow \cdots$ 형태로 분류할 수 있다. $X \supseteq Y$ 형태의 제약식은 $X$가 의미하는 집합이 $Y$가 의미하는 집합을 포함한다는 것을 의미한다. 직관적으로, 실행 중에 $Y$의 정보가 $X$로 흘러들어 가는 것을 묘사한다. $X \subseteq Y$와 $X \not\subseteq Y$ 형태의 제약식은 $X$가 의미하는 집합이 $Y$가 의미하는 집합의 부분집합이어야 한다 혹은 그렇지 않아야 한다는 것을 나타내는 것들이다. 해당 수식이 어떤 타입을 가져야 하는 지를 제한한다. $X \Rightarrow \cdots$는 화살표 오른쪽에 있는 제약식이 $X$가 참일 경우에만 적용된다는 뜻으로, 화살표 오른쪽에 있는 제약식의 적용을 미루기 위한 용도로 사용된다. 각 집합 제약식의 구체적 의미는 그림 5에서 정의한다.

## 4.2 집합 제약식 도출

우선, 단계 0에서는 자유롭게 $\alpha$-변환할 수 있으므로 단계 0에서 바운딩되는 변수들은 적절히 $\alpha$-변환하여 모두 다르다고 가정한다. 이런 변환 없이 분석하는 것도 가능하나 분석의 정확도가 떨어질 수 있다.

그림 6은 프로그램으로부터 집합 제약식을 도출하는 방법을 정의한다. 여기서, $\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \vdash l : e \triangleright \mathcal{C}$는 수식 $l : e$를 단계 $n$에서 계산하게 되면 제약식 $\mathcal{C}$을 도출한다는 뜻이다. $l_0$는 전체 프로그램의 표지(label)를 의미한다.

단계 0에서 반드시 실행이 되는 수식이나 단계 $n+1$에서의 단계 0까지 내려가는 $\texttt{unbox}_{n+1}\ e$에 대해서는 타입을 검사하는 $X \subseteq Y$ 형태의 제약식을 도출한다. 특정 단계

$$
\begin{array}{rcl}
h_t & \in & Herb_T \\
h_t & := & \iota \mid x \to h_t \mid \Box\, h_t \\
h_b & \in & Herb_B \\
h_b & := & true
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma & \in & SetVar \to 2^{Herb_T} \\
\Delta & \in & SetVar \to 2^{Herb_B}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\cdot]\!] & \in & SetExpr \to (SetVar \to 2^{Herb_T}) \to (SetVar \to 2^{Herb_B}) \to 2^{Herb} \\
[\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta & = & \Sigma(\mathcal{X}_l^{\mathcal{Y}_t}) \\
[\![\mathcal{Y}_t^x]\!]\,\Sigma\,\Delta & = & \Sigma(\mathcal{Y}_t^x) \\
[\![\mathcal{Z}_l]\!]\,\Sigma\,\Delta & = & \Delta(\mathcal{Z}_l) \\
[\![\iota]\!]\,\Sigma\,\Delta & = & \{\iota\} \\
[\![\mathcal{Y}_t^x \to \mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta & = & \{x \to h_t \mid h_t \in [\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta\} \\
[\![\Box\, \mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta & = & \{\Box\, h_t \mid h_t \in [\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta\} \\
[\![apply(\mathcal{X}_l^{\mathcal{Y}_t}, \mathcal{X}_{l'}^{\mathcal{Y}_t})]\!]\,\Sigma\,\Delta & = & \{h_t \mid x \to h_t \in [\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta\} \\
[\![unbox_k\, \mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta & = & \{h_t \mid \Box\, h_t \in [\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta\} \\
[\![eval\, \mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta & = & \{h_t \mid \Box\, h_t \in [\![\mathcal{X}_l^{\mathcal{Y}_t}]\!]\,\Sigma\,\Delta\} \\
[\![true]\!]\,\Sigma\,\Delta & = & \{true\} \\
[\![\emptyset]\!]\,\Sigma\,\Delta & = & \emptyset \\
[\![\text{LAMBDA}]\!]\,\Sigma\,\Delta & = & \{x \to h_t \mid x \to h_t \in Herb_T\} \\
[\![\text{BOX}]\!]\,\Sigma\,\Delta & = & \{\Box\, h_t \mid \Box\, h_t \in Herb_T\}
\end{array}
$$

[그림 5] 집합 제약식의 의미

에서 실행될 경우 타입 오류가 발생할 수 있지만 실제로 그 단계에서 실행될 지를 알 지 못하는 수식에 대해서는 $X \subseteq Y$ 대신 타입 검사를 일단 보류하는 $Z \Rightarrow X \subseteq Y$ 형태의 제약식을 도출한다. 집합 제약식을 푸는 과정에서 해당 수식이 그 단계에서 실행될 가능성이 있다고 판단되면(즉, 조건 $Z$가 참이 되면) $X \subseteq Y$ 제약식을 추가함으로써 보류되었던 타입 검사를 수행한다.

각 제약식 도출 규칙을 개별적으로 살펴보자.

$$\frac{}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \vdash l : c \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \iota\}}$$

상수 $c$는 현재 단계 $n$에 무관하게 $\iota$의 타입을 가진다.

$$\frac{}{\mathcal{Y}_{t_0} \vdash l : x \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathcal{Y}_{t_0}^x, \mathcal{Y}_{t_0}^x \not\subseteq \emptyset, \mathcal{Z}_l \supseteq true\}}$$

단계 0에서 변수 $x$는 함수 적용을 통해 타입 환경($\mathcal{Y}_{t_0}^x$)에 바인딩된 타입을 가진다. 단계 0에서는 변수 $x$가 자유 변수일 수 없으며($\mathcal{Y}_{t_0}^x \not\subseteq \emptyset$), 단계 0에서 일어나는 계산이므로 타입을 검사한다($\mathcal{Z}_l \supseteq true$).

$$\frac{}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash l : x \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathcal{Y}_{t_{n+1}}^x, \mathcal{Z}_l \Rightarrow \mathcal{Y}_{t_{n+1}}^x \not\subseteq \emptyset\}}$$

단계가 0보다 큰 경우에도 여전히 $\mathcal{Y}_{t_{n+1}}^x$에서 타입 정보를 가져온다. 그러나, 이 수식에 대해 타입 검사를 할 필요가 있다고 판단될 때만($\mathcal{Z}_l$이 참일 때만), 타입 검사를 수행하도록 한다($\mathcal{Z}_l \Rightarrow \mathcal{Y}_{l_{n+1}}^x \not\subseteq \emptyset$).

$$\frac{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \vdash l : \lambda x.e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \mathcal{Y}_{t_n}^x \to \mathcal{X}_e^{\mathcal{Y}_{t_n}}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

$$\frac{}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_n}\vdash l : c \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \iota\}}$$

$$\frac{}{\mathcal{Y}_{t_0}\vdash l : x \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathcal{Y}_{t_0}^x, \mathcal{Y}_{t_0}^x \not\subseteq \emptyset, \mathcal{Z}_l \supseteq \mathit{true}\}}$$

$$\frac{}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash l : x \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathcal{Y}_{t_{n+1}}^x, \mathcal{Z}_l \Rightarrow \mathcal{Y}_{t_{n+1}}^x \not\subseteq \emptyset\}}$$

$$\frac{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_n}\vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_n}\vdash l : \lambda x.e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \mathcal{Y}_{t_n}^x \rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_n}}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

$$\frac{\mathcal{Y}_{t_0}\vdash e_1 \triangleright \mathcal{C}_1 \qquad \mathcal{Y}_{t_0}\vdash e_2 \triangleright \mathcal{C}_2}{\mathcal{Y}_{t_0}\vdash l : e_1\ e_2 \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_0}}), \mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}} \subseteq \mathrm{LAMBDA}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash e_1 \triangleright \mathcal{C}_1 \qquad \mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash e_2 \triangleright \mathcal{C}_2}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash l : e_1\ e_2 \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_{n+1}}}), \mathcal{Z}_l \Rightarrow \mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}} \subseteq \mathrm{LAMBDA}, \mathcal{Z}_{e_1} \supseteq \mathcal{Z}_l, \mathcal{Z}_{e_2} \supseteq \mathcal{Z}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_n}\mathcal{Y}_t\vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_n}\vdash l : \mathtt{box}_t\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \Box\,\mathcal{X}_e^{\mathcal{Y}_t}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

$$\frac{\mathcal{Y}_{t_0}\vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash l : \mathtt{unbox}_{n+1}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{unbox}_{n+1}\,\mathcal{X}_e^{\mathcal{Y}_{t_0}}, \mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \mathrm{BOX}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}}$$

$$\frac{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash e \triangleright \mathcal{C} \qquad k > 0}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1+k}}\vdash l : \mathtt{unbox}_k\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1+k}}} \supseteq \mathit{unbox}_k\,\mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}, \mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \mathrm{BOX}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

$$\frac{\mathcal{Y}_{t_0}\vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0}\vdash l : \mathtt{eval}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{eval}\,\mathcal{X}_e^{\mathcal{Y}_{t_0}}, \mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \mathrm{BOX}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}}$$

$$\frac{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0}\cdots\mathcal{Y}_{t_{n+1}}\vdash l : \mathtt{eval}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{eval}\,\mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}, \mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \mathrm{BOX}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

[그림 6] 집합 제약식 도출

람다 추상화(lambda abstraction)는 단계에 상관없이 인자를 받아서 몸체(body)를 계산해서 얻어지는 타입을 가진다($\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \mathcal{Y}_{t_n}^x \rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_n}}$). 이 수식에 대해 타입 검사를 할 필요가 있다면, 하부 수식에 대해서도 타입을 검사할 필요가 있다($\mathcal{Z}_e \supseteq \mathcal{Z}_l$).

$$\frac{\mathcal{Y}_{t_0}\vdash e_1 \triangleright \mathcal{C}_1 \qquad \mathcal{Y}_{t_0}\vdash e_2 \triangleright \mathcal{C}_2}{\mathcal{Y}_{t_0}\vdash l : e_1\ e_2 \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_0}}), \mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}} \subseteq \mathrm{LAMBDA}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

단계 0에서의 함수 적용의 타입은 먼저 $e_1$의 타입을 알아야 알 수 있으므로, 제약식 도출 과정에서는 함수 적용이다는 사실만을 기록하고 실제 타입 정보는 제약식을 푸는 과정에서 얻는다($\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_0}})$). 단계 0에서의 함수 적용이므로 $e_1$의 타입은 반드시 함

수 타입이어야 한다($\mathcal{X}_{e_1}^{\mathcal{Y}_{t_0}} \subseteq \text{LAMBDA}$, $\mathcal{Z}_l \supseteq \mathit{true}$).

$$\frac{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash e_1 \triangleright \mathcal{C}_1 \qquad \mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash e_2 \triangleright \mathcal{C}_2}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash l : e_1\ e_2 \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_{n+1}}}), \mathcal{Z}_l \Rightarrow \mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{LAMBDA}, \mathcal{Z}_{e_1} \supseteq \mathcal{Z}_l, \mathcal{Z}_{e_2} \supseteq \mathcal{Z}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

현재 단계가 0보다 큰 경우에도 함수 적용에 관한 제약식을 도출하지만($\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{apply}(\mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}}, \mathcal{X}_{e_2}^{\mathcal{Y}_{t_{n+1}}})$), 타입을 제한하는 제약식은 해당 식이 타입을 검사할 필요가 있을 때만 수행되도록 한다($\mathcal{Z}_l \Rightarrow \mathcal{X}_{e_1}^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{LAMBDA}$). 해당 수식에 대해 타입 검사를 수행해야 한다면, 하부 수식들에 대해서도 타입 검사를 수행해야 한다($\mathcal{Z}_{e_1} \supseteq \mathcal{Z}_l$, $\mathcal{Z}_{e_2} \supseteq \mathcal{Z}_l$).

$$\frac{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \mathcal{Y}_t \vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_n} \vdash l : \texttt{box}_t\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \Box\, \mathcal{X}_e^{\mathcal{Y}_t}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

수식 $\texttt{box}_t\ e$의 타입은 하부 수식의 타입을 코드 템플릿으로 만든 타입이다($\mathcal{X}_l^{\mathcal{Y}_{t_n}} \supseteq \Box\, \mathcal{X}_e^{\mathcal{Y}_t}$). 해당 수식에 대해 타입을 검사할 필요가 있다면 하부 수식에 대해서도 타입을 검사할 필요가 있다는 보수적인(conservative) 입장을 취한다($\mathcal{Z}_e \supseteq \mathcal{Z}_l$)

$$\frac{\mathcal{Y}_{t_0} \vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash l : \texttt{unbox}_{n+1}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{unbox}_{n+1} \mathcal{X}_e^{\mathcal{Y}_{t_0}}, \mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \text{BOX}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}}$$

$\texttt{unbox}_{n+1}\ e$는 함수 적용과 마찬가지로 하부 수식의 타입 정보를 알아야만 타입을 결정할 수 있다($\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{unbox}_{n+1} \mathcal{X}_e^{\mathcal{Y}_{t_0}}$). 그러나, 단계 0까지 내려가므로 반드시 하부 수식의 타입은 코드 템플릿의 타입이어야 한다($\mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \text{BOX}$, $\mathcal{Z}_l \supseteq \mathit{true}$).

$$\frac{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash e \triangleright \mathcal{C} \qquad k > 0}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1+k}} \vdash l : \texttt{unbox}_k\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1+k}}} \supseteq \mathit{unbox}_k\, \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}, \mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{BOX}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

단계 0까지 내려가지 못하는 $\texttt{unbox}_k\ e$도 하부 수식의 타입에 따라 타입이 결정되지만($\mathcal{X}_l^{\mathcal{Y}_{t_{n+1+k}}} \supseteq \mathit{unbox}_k\, \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}$), 해당 수식에 대해 타입 검사를 반드시 수행해야 하는 것은 아니다($\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{BOX}$). 그러나, 일단 타입 검사를 수행한다면 하부 수식에 대해서도 타입 검사를 수행해야 한다($\mathcal{Z}_e \supseteq \mathcal{Z}_l$).

$$\frac{\mathcal{Y}_{t_0} \vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0} \vdash l : \texttt{eval}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{eval}\, \mathcal{X}_e^{\mathcal{Y}_{t_0}}, \mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \text{BOX}, \mathcal{Z}_l \supseteq \mathit{true}\} \cup \mathcal{C}}$$

단계 0에서 $\texttt{eval}\ e$을 수행한 결과값의 타입 정보는 하부 수식의 값의 타입에 따라 달라진다($\mathcal{X}_l^{\mathcal{Y}_{t_0}} \supseteq \mathit{eval}\, \mathcal{X}_e^{\mathcal{Y}_{t_0}}$). 이 때, 하부 수식은 반드시 코드 템플릿의 타입을 가져야 한다($\mathcal{X}_e^{\mathcal{Y}_{t_0}} \subseteq \text{BOX}, \mathcal{Z}_l \supseteq \mathit{true}$).

$$\frac{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash e \triangleright \mathcal{C}}{\mathcal{Y}_{t_0} \cdots \mathcal{Y}_{t_{n+1}} \vdash l : \texttt{eval}\ e \triangleright \{\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{eval}\, \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}, \mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{BOX}, \mathcal{Z}_e \supseteq \mathcal{Z}_l\} \cup \mathcal{C}}$$

단계 0보다 높은 단계에서 수행한 $\texttt{eval}\ e$도 하부 수식의 타입에 따라 그 타입이 결정된다($\mathcal{X}_l^{\mathcal{Y}_{t_{n+1}}} \supseteq \mathit{eval}\, \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}}$). 해당 수식이 결코 단계 0에서 수행되지 않을 수도 있으므로, 필요하다고 판단될 때만 타입을 검사하는 형태의 제약식을 도출한다($\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_{t_{n+1}}} \subseteq \text{BOX}$).

$$\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq apply(\mathcal{X}_{l'}^{\mathcal{Y}_t}, \mathcal{X}_{l''}^{\mathcal{Y}_t}) \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \mathcal{Y}_{t'}^{x} \to \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}}}{\{\mathcal{Y}_{t'}^{x} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_t}, \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}}, \mathcal{Z}_{l'''} \supseteq \mathcal{Z}_l\}}$$

$$\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq unbox_k\, \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t''}}}{\{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l'''}^{\mathcal{Y}_{t''}}, \mathcal{Z}_{l''} \supseteq \mathcal{Z}_l\}} \qquad \frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq unbox_k\, \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t''}} \qquad x \in FV^0(l'')}{\{\mathcal{Y}_{t''}^{x} \supseteq \mathcal{Y}_t^{x}\}}$$

$$\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq eval\, \mathcal{X}_{l'}^{\mathcal{Y}_t} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \Box\, \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}}}{\{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_{t'}}, \mathcal{Z}_{l''} \supseteq \mathcal{Z}_l\}} \qquad \frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq eval\, \mathcal{X}_{l'}^{\mathcal{Y}_t} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \Box\, \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}} \qquad x \in FV^0(l'')}{\{\mathcal{Z}_l \Rightarrow \mathcal{Y}_{t'}^{x} \subseteq \emptyset, \mathcal{Z}_l \Rightarrow \mathcal{Y}_{t'}^{x} \not\subseteq \emptyset\}}$$

$$\frac{\mathcal{Z}_l \Rightarrow \mathcal{Y}_t^{x} \subseteq \emptyset \qquad \mathcal{Z}_l \supseteq true}{\mathcal{Y}_t^{x} \subseteq \emptyset} \qquad \frac{\mathcal{Z}_l \Rightarrow \mathcal{Y}_t^{x} \not\subseteq \emptyset \qquad \mathcal{Z}_l \supseteq true}{\mathcal{Y}_t^{x} \not\subseteq \emptyset}$$

$$\frac{\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{LAMBDA} \qquad \mathcal{Z}_l \supseteq true}{\mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{LAMBDA}} \qquad \frac{\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{BOX} \qquad \mathcal{Z}_l \supseteq true}{\mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{BOX}}$$

$$\frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq \iota}{sv_1 \supseteq \iota} \qquad \frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq sv_3 \to sv_4}{sv_1 \supseteq sv_3 \to sv_4} \qquad \frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq \Box\, sv_3}{sv_1 \supseteq \Box\, sv_3}$$

$$\frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq true}{sv_1 \supseteq true}$$

[그림 7] 집합 제약식 풀기

$$\begin{array}{rcll}
FV^n(c) & = & \emptyset & \\
FV^n(x) & = & \{x\} & \text{if } n = 0 \\
 & = & \emptyset & \text{otherwise} \\
FV^n(\lambda x.e) & = & FV^n(e) - \{x\} & \text{if } n = 0 \\
 & = & FV^n(e) & \text{otherwise} \\
FV^n(e_1\, e_2) & = & FV^n(e_1) \cup FV^n(e_2) & \\
FV^n(\texttt{box}_t\, e) & = & FV^{n+1}(e) & \\
FV^n(\texttt{unbox}_k\, e) & = & FV^{n-k}(e) & \text{if } n - k \geq 0 \\
 & = & \emptyset & \text{otherwise} \\
FV^n(\texttt{eval}\, e) & = & FV^n(e) &
\end{array}$$

[그림 8] 자유 변수

## 4.3 집합 제약식 풀기

집합 제약식을 푸는 방법은 그림 7과 같다. 기본적으로 기존의 제약식에 위의 제약식들이 존재하면 아래의 새로운 제약식을 추가하는 형태이다. 제약식 집합의 최대 크기가 분석할 프로그램에 의해 한정되므로 푸는 과정은 유한한 시간 내에 끝난다.

해는 도출한 제약식들을 끝까지 풀어낸 각 집합 변수의 제약식들 중에서 최소 알갱이로 풀려진 제약식(atomic constraint)로 구성된다. 그림 9에서 최소 알갱이로 풀려진 제약식을 정의한다.

제약식을 푸는 규칙을 좀더 자세히 살펴보자.

$$\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq apply(\mathcal{X}_{l'}^{\mathcal{Y}_t}, \mathcal{X}_{l''}^{\mathcal{Y}_t}) \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \mathcal{Y}_{t'}^{x} \to \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}}}{\{\mathcal{Y}_{t'}^{x} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_t}, \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l'''}^{\mathcal{Y}_{t'}}, \mathcal{Z}_{l'''} \supseteq \mathcal{Z}_l\}}$$

$$
\begin{array}{rcl}
ac & \in & AtomSetCstr \\
ac & := & \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \iota \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{Y}_{t'}^{x} \to \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \mathcal{X}_l^{\mathcal{Y}_t} \supseteq \Box\, \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \mid \\
& & \mathcal{Y}_t^{x} \supseteq \iota \mid \mathcal{Y}_t^{x} \supseteq \mathcal{Y}_{t'}^{y} \to \mathcal{X}_l^{\mathcal{Y}_{t'}} \mid \mathcal{Y}_t^{x} \supseteq \Box\, \mathcal{X}_l^{\mathcal{Y}_{t'}} \mid \\
& & \mathcal{Z}_l \supseteq true \mid \\
& & \mathcal{Y}_t^{x} \subseteq \emptyset \mid \mathcal{Y}_t^{x} \not\subseteq \emptyset \mid \mathcal{X}_l^{\mathcal{Y}_t} \subseteq \text{LAMBDA} \mid \mathcal{X}_l^{\mathcal{Y}_t} \subseteq \text{BOX}
\end{array}
$$

[그림 9] 최소 알갱이로 풀려진 집합 제약식

함수 적용 $l : (l' : e')(l'' : e'')$에서 $(l' : e')$의 타입 정보가 구체적으로 얻어지면($\mathcal{X}_{l''}^{\mathcal{Y}_{t'}} \supseteq \mathcal{Y}_{t''}^{x} \to \mathcal{X}_{l'''}^{\mathcal{Y}_{t''}}$), 타입 환경에서 $x$에 바인딩되는 타입 정보를 알 수 있다($\mathcal{Y}_{t''}^{x} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_{t'}}$). 함수 적용을 실행한 결과의 타입 정보는 함수 몸체의 타입 정보로부터 유도할 수 있다($\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l'''}^{\mathcal{Y}_{t''}}$). 그리고, 함수 적용에 대해서 타입 정보를 검사할 필요가 있다면 그곳으로 흘러들어오는 함수들에 대해서도 타입 정보를 검사할 필요가 있다($\mathcal{Z}_{l'''} \supseteq \mathcal{Z}_l$).

$$
\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq unbox_k\, \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t''}}}{\{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_{t''}}, \mathcal{Z}_{l''} \supseteq \mathcal{Z}_l\}}
$$

$$
\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq unbox_k\, \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_{t'}} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t''}} \qquad x \in FV^0(l'')}{\{\mathcal{Y}_{t''}^{x} \supseteq \mathcal{Y}_t^{x}\}}
$$

수식 $\texttt{unbox}_k\ e$의 하부 수식 $e$의 타입 정보가 코드 템플릿 타입을 포함하면, 전체 수식의 타입 정보도 그에 대응하는 타입을 포함한다($\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_{t''}}$). 함수 적용과 유사하게, 해당 수식이 타입 검사를 할 필요가 있다면 흘러들어오는 코드 템플릿에 대해서도 타입 검사를 수행한다($\mathcal{Z}_{l''} \supseteq \mathcal{Z}_l$). 만약 흘러들어온 코드 템플릿 내에 자유 변수가 포함되어 있다면 타입 환경의 정보가 흘러가는데($\mathcal{Y}_{t''}^{x} \supseteq \mathcal{Y}_t^{x}$), 이는 타입 체계의 규칙 [TUNBOX]에서 현재 타입 환경($\Gamma_{n+k}$)이 하부 수식의 코드 템플릿의 타입 속에 포함된 조건 환경과 동일해야 한다는 규칙을 반영한다.

$$
\frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq eval\, \mathcal{X}_{l'}^{\mathcal{Y}_t} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t'}}}{\{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq \mathcal{X}_{l''}^{\mathcal{Y}_{t'}}, \mathcal{Z}_{l''} \supseteq \mathcal{Z}_l\}} \qquad \frac{\mathcal{X}_l^{\mathcal{Y}_t} \supseteq eval\, \mathcal{X}_{l'}^{\mathcal{Y}_t} \qquad \mathcal{X}_{l'}^{\mathcal{Y}_t} \supseteq \Box\, \mathcal{X}_{l''}^{\mathcal{Y}_{t'}} \qquad x \in FV^0(l'')}{\{\mathcal{Z}_l \Rightarrow \mathcal{Y}_{t'}^{x} \subseteq \emptyset, \mathcal{Z}_l \Rightarrow \mathcal{Y}_{t'}^{x} \not\subseteq \emptyset\}}
$$

수식 $\texttt{eval}\ e$의 경우는 수식 $\texttt{unbox}_k\ e$의 경우와 흡사하다. 단, 흘러들어온 코드 템플릿에는 자유 변수가 포함되어 있질 않아야 한다($\mathcal{Z}_l \Rightarrow \mathcal{Y}_{t''}^{x} \not\subseteq \emptyset$). 만약 자유 변수가 포함되어 있다면($\mathcal{Z}_l \Rightarrow \mathcal{Y}_{t''}^{x} \subseteq \emptyset$) 모순이 발생하여 제약식의 해가 존재하지 않는다. 즉, 해당 프로그램이 실행 중에 오류가 발생할 수 있다고 판단한다.

$$
\frac{\mathcal{Z}_l \Rightarrow \mathcal{Y}_t^{x} \subseteq \emptyset \qquad \mathcal{Z}_l \supseteq true}{\mathcal{Y}_t^{x} \subseteq \emptyset} \qquad \frac{\mathcal{Z}_l \Rightarrow \mathcal{Y}_t^{x} \not\subseteq \emptyset \qquad \mathcal{Z}_l \supseteq true}{\mathcal{Y}_t^{x} \not\subseteq \emptyset}
$$

$$
\frac{\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{LAMBDA} \qquad \mathcal{Z}_l \supseteq true}{\mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{LAMBDA}} \qquad \frac{\mathcal{Z}_l \Rightarrow \mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{BOX} \qquad \mathcal{Z}_l \supseteq true}{\mathcal{X}_e^{\mathcal{Y}_t} \subseteq \text{BOX}}
$$

이 규칙들은 집합 제약식을 처음 도출할 때나 푸는 과정에서 추가된 타입 검사를 보류시킨 제약식 중에서 실제로 검사를 수행할 필요가 있다고 판단된 것들에 대해 타입 검사를 수행

하도록 변환된 규칙들을 추가한다.

$$\frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq \iota}{sv_1 \supseteq \iota} \qquad \frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq sv_3 \to sv_4}{sv_1 \supseteq sv_3 \to sv_4} \qquad \frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq \Box\, sv_3}{sv_1 \supseteq \Box\, sv_3}$$

$$\frac{sv_1 \supseteq sv_2 \qquad sv_2 \supseteq true}{sv_1 \supseteq true}$$

이 규칙들은 최소 알갱이로 풀려진 제약식들의 해가 전체 집합 제약식의 해가 될 수 있도록 제약식들을 변환하는 역할을 수행한다.

이렇게 얻어진 최소 알갱이로 풀려진 제약식과 단계 0의 자유 변수에 대한 제약식($\forall x \in FV^0(l_0).\mathcal{Y}^x_{t_0} \subseteq \emptyset$)을 함께 만족하는 해가 존재하면 프로그램은 실행 중에 타입 에러가 발생하지 않는다.

## 4.4 예제

실제 예로서, $\lambda x.(x\,x)$와 $\mathtt{box}_t\ (c\,c)$에 대해 집합 제약식을 도출하고 푸는 과정을 살펴본다. 이 프로그램들은 올바른 프로그램임에도 불구하고 단순 타입 체계가 거절하는 프로그램들이다.

$\lambda x.(x\,x)$의 각 하부 수식에 $l_0 : \lambda x.(l_1 : (l_2 : x)\ (l_3 : x))$처럼 표지를 붙이면 프로그램의 각 부분에 대하여 다음과 같은 제약식이 도출된다.

$$\begin{array}{l} l_0 \ : \ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_0} \supseteq \mathcal{Y}^x_{t_0} \to \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_1},\ \mathcal{Z}_{l_1} \supseteq \mathcal{Z}_{l_0} \\ l_1 \ : \ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_1} \supseteq apply(\mathcal{X}^{\mathcal{Y}_{t_0}}_{l_2}, \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_3}),\ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_2} \subseteq \mathrm{LAMBDA},\ \mathcal{Z}_{l_1} \supseteq true \\ l_2 \ : \ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_2} \supseteq \mathcal{Y}^x_{t_0},\ \mathcal{Y}^x_{t_0} \not\subseteq \emptyset,\ \mathcal{Z}_{l_2} \supseteq true \\ l_3 \ : \ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_3} \supseteq \mathcal{Y}^x_{t_0},\ \mathcal{Y}^x_{t_0} \not\subseteq \emptyset,\ \mathcal{Z}_{l_3} \supseteq true \end{array}$$

도출된 제약식을 최대한 풀면, 최소 알갱이로 풀려진 제약식들은 다음과 같다.

$$\mathcal{X}^{\mathcal{Y}_{t_0}}_{l_0} \supseteq \mathcal{Y}^x_{t_0} \to \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_1},\ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_2} \subseteq \mathrm{LAMBDA},\ \mathcal{Y}^x_{t_0} \not\subseteq \emptyset,\ \mathcal{Z}_{l_1} \supseteq true,\ \mathcal{Z}_{l_2} \supseteq true,\ \mathcal{Z}_{l_3} \supseteq true$$

타입을 제한하는 $\subseteq$, $\not\subseteq$ 형태의 제약식의 왼쪽에 등장하는 집합 변수가 $\supseteq$ 형태의 제약식의 왼쪽에 등장하지 않으므로, 이 제약식들을 만족하는 해가 존재한다.

마찬가지로, $\mathtt{box}_t\ (c\,c)$의 각 하부 수식에 $l_0 : \mathtt{box}_t\ (l_1 : (l_2 : c)\ (l_3 : c))$처럼 표지를 붙이면 프로그램에 각 부분에 대하여 아래의 제약식이 도출된다.

$$\begin{array}{l} l_0 \ : \ \mathcal{X}^{\mathcal{Y}_{t_0}}_{l_0} \supseteq \Box\, \mathcal{X}^{\mathcal{Y}_t}_{l_1},\ \mathcal{Z}_{l_1} \supseteq \mathcal{Z}_{l_0} \\ l_1 \ : \ \mathcal{X}^{\mathcal{Y}_t}_{l_1} \supseteq apply(\mathcal{X}^{\mathcal{Y}_t}_{l_2}, \mathcal{X}^{\mathcal{Y}_t}_{l_3}),\ \mathcal{Z}_{l_1} \Rightarrow \mathcal{X}^{\mathcal{Y}_t}_{l_2} \subseteq \mathrm{LAMBDA},\ \mathcal{Z}_{l_2} \supseteq \mathcal{Z}_{l_1},\ \mathcal{Z}_{l_3} \supseteq \mathcal{Z}_{l_1} \\ l_2 \ : \ \mathcal{X}^{\mathcal{Y}_t}_{l_2} \supseteq \iota \\ l_3 \ : \ \mathcal{X}^{\mathcal{Y}_t}_{l_3} \supseteq \iota \end{array}$$

이렇게 도출된 제약식을 최대한 풀면, 아래와 같은 최소 알갱이 제약식들로 풀려진다.

$$\mathcal{X}^{\mathcal{Y}_{t_0}}_{l_0} \supseteq \Box\, \mathcal{X}^{\mathcal{Y}_t}_{l_1},\ \mathcal{X}^{\mathcal{Y}_t}_{l_2} \supseteq \iota,\ \mathcal{X}^{\mathcal{Y}_t}_{l_3} \supseteq \iota$$

타입을 제한하는 $\subseteq$ 형태의 제약식이 존재하지 않으므로, 제약식들을 만족하는 해가 존재한다.

## 5 결론 및 향후 연구

지금까지 다단계 프로그램에 대한 흐름을 고려한 정교한 분석을 고안하였다. 이 분석은 안전할 뿐만 아니라, 단순 타입 체계보다 더 많은 올바른 프로그램을 받아들일 것이라고 추측된다. 이에 대한 형식적인(formal) 증명이 진행 중이다.

## 참고문헌

[1] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[2] Neil D. Jones. The essence of program transformation by partial evaluation and driving. In *PSI '99: Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 62–79, London, UK, 2000. Springer-Verlag.

[3] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 257–268, New York, NY, USA, 2006. ACM Press.

[4] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Inf. Comput.*, 118(1):128–141, 1995.

김 덕 환

– 1997-2005 서울대학교 학사
– 2005-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

이 광 근

– 1983-1987 서울대학교 학사
– 1988-1990 Univ. of Illinois at Urbana-Champaign 석사
– 1990-1993 Univ. of Illinois at Urbana-Champaign 박사
– 1993-1995 Bell Labs., Murray Hill 연구원
– 1995-2003 한국과학기술원 교수
– 2003-현재 서울대학교 교수

<관심분야> 프로그램 분석 및 검증, 프로그래밍 언어

20-2-R04

# C 프로그램에서 사용되지 않는 자료를 찾는 정적 분석기의 개발 [1]

*Development of Static Analyzer Detecting Unused Data in C Programs*

황의권 · 이광근
서울대학교 컴퓨터공학부 프로그래밍 연구실
{neo; kwang}@ropas.snu.ac.kr

## 요 약

우리는 Airac5의 틀을 기반으로 C 프로그램에서 사용되지 않는 자료를 찾는 정적 프로그램 분석기 Umirac을 고안하고 구현하였다. Umirac은 구간 집합 도메인을 이용한 요약해석을 통하여 프로그램에서 정의되었지만 사용되지 않는 변수, 구조체의 필드, 그리고 버퍼의 구간에 대한 정보를 분석한다. 우리는 리눅스 커널 프로그램과 임베디드 시스템 소프트웨어를 대상으로 실험을 수행하여 실제 프로그램 상에서 불필요하게 정의된 자료들을 찾을 수 있었다. 또한 Umirac과 Airac7의 수행속도를 비교한 결과 구간 집합 분석을 하는 Umirac이 구간 분석을 하는 Airac7의 60–86%정도의 속도를 보여주었다.

## 1 서론

프로그램에서 메모리에 존재하지만 사용되지 않는 자료들은 프로그램의 수행성능을 저하시킨다. 이는 특히 임베디드 시스템과 같이 자원의 제약이 심한 환경에서 메모리 부족으로 인한 비정상적인 프로그램 종료를 일으키기도 하며, 불필요한 계산을 증가시켜 수행 속도에 영향을 끼치기도 한다.

프로그램의 수행성능을 향상시키기 위해서는 사용되지 않는 자료와 그것에 관련된 연산을 실행 전에 안전하게 제거하는 것이 필요하다. 컴파일러에서 코드 최적화 기법으로 널리 사용되는 살아있는 변수 분석(live variable analysis)[1]과 같은 방법은 컴파일 시간에 사용되지 않는 자료를 없애거나 레지스터를 효율적으로 사용하는 데에 유용하게 쓰이고 있다. 그러나 살아있는 변수 분석 자체는 포인터에 대한 분석을 포함하고 있지 않아 프로그램 실행 중에 각 포인터가 가리키는 대상을 알기 위해서는 별도의 포인터 분석을 사용하여야 하며, 버퍼에서 사용되지 않는 부분을 고려하지 못한다는 문제가 있다. SPLINT[9]는 버퍼 오버런과 같은 프로그램의 오류나 쓰이지 않는 변수, 구조체의 필드와 같은 불필요한 자료들을 찾아주는 분석기이다. 그러나 SPLINT는 변수나 구조체 필드에 값이 정의만 된 경우, 단순히 변수나 구조체 필드의 주소값을 취한 경우에도 해당 메모리 공간을 사용한 것으로 간주한다.

[1]본 연구는 정보통신부 선도기반기술개발사업과 교육인적자원부 두뇌한국21사업의 지원으로 수행하였다.

Umirac[우미락]은 요약해석 틀(abstract interpretation framework)[3, 4, 2]에 기반하여 C 프로그램[5]에서 사용되지 않는 자료를 찾아주는 정적 분석기이다. Umirac은 세 가지의 정보를 사용자에게 알려준다:

– 정의되었으나 사용되지 않는 변수(UV : unused variable)
– 사용되지 않는 구조체의 필드(UF : unused field)
– 사용되지 않는 버퍼 구간(BB : bubble in buffer)

Umirac의 설계와 구현은 Airac5[8]의 틀을 바탕으로 이루어졌다. 즉, Umirac은 Airac5와 마찬가지로 프로그램이 실행 중에 겪을 수 있는 모든 상황을 포섭하며, 정확도와 속도를 향상시키기 위한 많은 기술들이 적용되어 있다. Airac5의 구간 도메인을 분석에 보다 적합한 구간 집합 도메인으로 확장하였고, 요약 메모리에서 특정 메모리 공간이 사용되었는지의 여부를 표현하는 요소를 추가하였다.

## 2 분석기 설계

Umirac은 Airac5에서 사용한 G[7]의 요약해석을 구간 집합 도메인(set of interval domain)으로 확장한 설계를 사용하였다. 구간 분석(interval analysis)[3]이 사용되지 않는 버퍼 구간을 찾는 분석에서 유용하지 않은 예는 다음과 같다:

```
if (rand()%2)
    index = 0;
else index = ARRAY_SIZE - 1  /* 9 */;
return array[index];
```

구간 분석을 사용하였을 경우 마지막 `return`문에서의 `index`의 값은 [0, 9]로 요약된다. 만약 프로그램의 다른 부분에서 버퍼 `array`를 사용하는 부분이 없다면 버퍼 `array`에 사용하지 않는 구간 [1, 8]이 있음은 명백하지만 구간 분석으로는 이러한 정보를 알아낼 수 없다.

구간 집합 도메인은 요약 세계에서의 값을 정수 구간의 집합으로 표현한다. 예를 들어, 위의 예제 마지막 줄에서의 `index`의 값은 구간 집합 {[0, 0], [9, 9]}로 요약된다. 따라서 `array`에는 사용하지 않는 구간 [1, 8]이 있음을 알아낼 수 있다.

구간들의 집합을 $\hat{\mathbb{Z}}$라고 할 때, 구간 집합 도메인 $\hat{Int}$는 다음과 같이 정의할 수 있다:

$$2^{\mathbb{Z}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} 2^{\hat{\mathbb{Z}}} = \hat{Int}$$

$$\begin{aligned} \gamma\, A &= \bigcup\{\gamma_{\hat{\mathbb{Z}}}\, a \mid a \in A\} \\ \hat{\bot}_{\hat{Int}} &= \emptyset \\ \hat{\top}_{\hat{Int}} &= \{[-\infty, \infty]\} \end{aligned}$$

여기서 두 구간 집합 간의 순서(order)는 다음과 같이 정의된다:

$$A, B \in \hat{Int}, A \sqsubseteq B \iff (\forall a \in A. \exists b \in B \,:\, a \sqsubseteq b)$$

이러한 구간 집합의 정의에 있어서 중요한 것은, 한 구간 집합의 원소인 구간들 사이에는 겹치거나 이어질 수 있는 관계가 존재하면 안 된다는 것이다. 이러한 성질이 유지되도록 각 연산의 마지막 단계에서 구간 집합을 정규화(normalize)할 필요가 있다. 예를 들어 두 구간 집합 $A$와 $B$의 합치기(join) 연산은 다음과 같이 정의할 수 있다:

$$A \sqcup B = |(A \cup B)|$$

정규화는 구간 집합 안에서 서로 겹치거나 이어질 수 있는 구간들을 하나의 구간으로 합치는 역할을 한다. 두 구간 집합 $A$와 $B$의 일반적인 연산은 다음과 같이 두 구간 집합이 포함하는 각 구간들의 연산 결과를 모으는 형태로 정의된다.

$$\hat{\diamond} \in \{\hat{+}, \hat{-}, \hat{*}, \hat{/}, \sqcap\} \ : \ \hat{Int} \times \hat{Int} \to \hat{Int}$$
$$\hat{\underline{\diamond}} \in \{\hat{\underline{+}}, \hat{\underline{-}}, \hat{\underline{*}}, \hat{\underline{/}}, \underline{\sqcap}\} \ : \ \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$$

$$A \hat{\diamond} B = |\bigcup \{a \hat{\underline{\diamond}} b \mid a \in A, b \in B\}|$$

구간 도메인과 마찬가지로 구간 집합 도메인에서도 분석을 유한시간에 끝내기 위한 넓히기(widening)연산과 정확도를 복구하는 좁히기(narrowing) 연산이 필요하다. Umirac에서는 구간 집합을 하나의 구간으로 요약한 후 구간 사이의 넓히기/좁히기 연산을 수행하도록 구간 집합 사이의 넓히기/좁히기 연산을 설계하였다. $\underline{x}$와 $\underline{y}$가 각각 구간 집합 $x$와 $y$를 하나의 구간으로 요약한 것이라 할 때 넓히기 연산자 $\nabla$와 좁히기 연산자 $\Delta$는 다음과 같이 정의된다.

– 넓히기(widening)

$$\begin{aligned}
x \nabla y &= \{\underline{x} \underline{\nabla} \underline{y}\} \\
\underline{\bot} \underline{\nabla} \underline{y} &= \underline{y} \\
\underline{x} \underline{\nabla} \underline{\bot} &= \underline{x} \\
[l_0, u_0] \underline{\nabla} [l_1, u_1] &= [0 \le l_1 < l_0 \ ? \ 0 : (l_1 < l_0 \ ? \ -\infty : l_0), \\
&\quad\ 0 \ge u_1 > u_0 \ ? \ 0 : (u_0 < u_1 \ ? \ +\infty : u_0)]
\end{aligned}$$

– 좁히기(narrowing)

$$\begin{aligned}
x \Delta y &= x \sqcap \{\underline{x} \underline{\Delta} \underline{y}\} \\
\underline{\bot} \underline{\Delta} \underline{y} &= \underline{\bot} \\
\underline{x} \underline{\Delta} \underline{\bot} &= \underline{\bot} \\
[l_0, u_0] \underline{\Delta} [l_1, u_1] &= [((l_0 \le 0 \le l_1) \vee (l_0 = -\infty)) \ ? \ l_1 : l_0, \\
&\quad\ ((u_1 \le 0 \le u_0) \vee (u_0 = +\infty)) \ ? \ u_1 : u_0]
\end{aligned}$$

사용되지 않는 변수와 구조체 필드 분석을 위해, G의 요약공간을 어떤 메모리가 사용될 수 있는지의 여부를 표현하도록 확장하였다:

$$\hat{Map} = \hat{Addr} \xrightarrow{\text{fin}} (\hat{Value} \times \mu)$$

$\hat{Map}$은 메모리의 요약을 의미하며, $\mu$는 $\bot$(사용되지 않음)과 $\top$(사용될 수 있음)의 두 가지 상태를 표현한다. 두 요약된 메모리를 합칠 때, 같은 주소에 대해 서로 $\mu$값이 다르다면 합친 메모리의 $\mu$값은 $\top$으로 정의한다. 그리고 요약 의미 공간에서 값을 계산하는 함수 $\hat{\mathcal{V}}$를 다음과 같이 정의한다:

$$\hat{\mathcal{V}} : \hat{Map} \to \hat{Expr} \to (\hat{Value} \times \hat{Map})$$

Umirac의 $\hat{\mathcal{V}}$는 메모리와 수식(expression)을 받아 계산한 값과 메모리를 내놓는다. 메모리를 반환하는 이유는 $\hat{\mathcal{V}}$는 메모리에서 값을 꺼냈을 경우 해당 메모리 공간에 있는 $\mu$값이 $\bot$이었으면 $\top$으로 바꾸는데, 이 때 바뀐 메모리를 물려주기 위함이다.

Umirac은 고정점을 구한 분석결과를 가지고 프로그램의 실행 흐름을 따라가며 반드시 사용되지 않을 자료만을 찾는다. 요약해석의 틀에서 설계된 Umirac은 어떤 메모리 공간이 실제 실행 중에 사용될 수 있는지의 여부와 버퍼의 인덱스 값으로 사용될 수 있는 값들을 모두 포섭한 분석을 수행한다. 따라서 이 분석결과를 마탕으로 Umirac이 사용되지 않는 자료라 판단한 것은 항상 실제로 사용되지 않는 자료들이라 할 수 있다.

하지만 분석 대상 프로그램의 실행의미가 충분하지 않다면 정확하지 않은 경보를 발생시킬 수 있다. 예를 들어 변수나 버퍼를 사용하는 부분이 다른 언어로 기술되어 있거나, 외부함수로 정의되어 있어 분석 대상 프로그램에 나타나 있지 않은 경우 Umirac은 해당하는 변수나 구조체 필드, 버퍼 구간을 사용하지 않는다고 분석한다.

# 3 실험 결과

실제 임베디드 시스템 소프트웨어와 리눅스 커널 코드를 대상으로 Umirac의 성능을 측정하였다. 수행속도의 대조군으로 Airac5를 더욱 다듬은 Airac7[6]을 설정하였다. 표 I, II는 각각 임베디드 시스템 소프트웨어와 리눅스 커널 코드의 일부를 대상으로 한 Umirac의 실험결과와 Airac7와의 수행 속도 비교를 보여준다.

실험은 모두 인텔 펜티엄4 3.2GHz 프로세서와 4GB 메모리가 탑재된 리눅스 시스템에서 수행하였다. Umirac과 Airac7 모두 분석 시나리오를 만들지 않고 프로그램의 모든 함수를 차례대로 분석하도록 하였으며, Airac7의 정확도 옵션은 가장 세밀한 분석을 수행하는 3으로 지정하였다. 함수 호출 펼치기(function inlining)는 두 분석기 모두 한 차례 하는 것으로, 그리고 분석 시간은 전처리와 변환을 제외한 분석에만 소요된 CPU 시간으로 측정하였다. 또한 프로그램의 크기는 프로그램을 전처리 하기 전의 것을 기록하였다.

| 프로그램 (LOC) | Umirac alarm | | | Umirac | Airac7 |
|---|---|---|---|---|---|
| | UV | UF | BB | 시간(sec) | 시간(sec) |
| AviReader (1486) | 5(0) | 0(0) | 6(3) | 6 | 4 |
| H263FRDivx (3944) | 7(0) | 4(0) | 5(0) | 300 | 257 |
| MADStream (8171) | 17(1) | 0(0) | 31(7) | 3615 | 2681 |
| software1 (4725) | 7(0) | 1(0) | 8(0) | 344 | 291 |
| software2 (21653) | 16(2) | 2(0) | 30(10) | 6510 | 3948 |

[표 I]임베디드 시스템 소프트웨어

| 프로그램 (LOC) | Umirac alarm | | | Umirac | Airac7 |
|---|---|---|---|---|---|
| | UV | UF | BB | 시간(sec) | 시간(sec) |
| vmax301.c (246) | 31(8) | 2(2) | 1(1) | 101 | 82 |
| cdc_acm.c (849) | 48(12) | 4(3) | 4(3) | 187 | 139 |
| atkbd.c (944) | 40(7) | 3(3) | 3(2) | 333 | 268 |
| eata_pio.c (984) | 105(21) | 6(5) | 11(6) | 1120 | 803 |
| ip6_output.c (1110) | 140(29) | 2(2) | 24(13) | 6430 | 4773 |
| xfrm_user.c (1201) | 148(35) | 3(1) | 19(9) | 5199 | 3794 |
| keyboard.c(1256) | 105(26) | 0(0) | 17(10) | 443 | 359 |
| af_inet.c (1273) | 150(38) | 3(3) | 15(10) | 6590 | 4917 |
| usb_midi.c (2206) | 104(17) | 2(1) | 13(5) | 2301 | 1619 |
| aty128fb.c (2466) | 104(25) | 1(1) | 14(10) | 680 | 525 |
| mptbase.c (6158) | 150(53) | 3(2) | 27(20) | 10817 | 8804 |

[표 II]Linux Kernel 2.6.4의 일부

Umirac의 분석 결과에서 UV, UF, BB는 각각 사용하지 않는 변수, 구조체의 필드, 그리고 버퍼 구간의 개수를 나타내며, 괄호 안의 숫자는 외부 함수 또는 다른 언어로 구현된 부분에서 사용되어지는 것으로 보이는 부정확한 경보의 수이다. 부정확한 경보의 수는 표 II에서 더욱 많이 관찰되는데, 이는 프로그램의 특성에 기인한 것으로, 리눅스 커널 프로그램은

많은 시스템 헤더 파일을 사용하여 감추어진 외부함수가 많고 어셈블리어 등 다른 언어로 기술된 부분이 많아 정확하지 않은 경보가 최대 50% 가까이 발생하는 것을 볼 수 있다. 한편 표 I의 임베디드 시스템 소프트웨어의 경우 다른 언어로 구현된 부분이 없고 외부함수 호출이 적어 부정확한 경보가 전체 경보의 0 – 32% 정도로 나타났다.

Airac7와의 수행 속도 비교에서는 Umirac이 60 – 86% 정도의 속도를 보여준다는 것을 알 수 있다. 확장된 요약 도메인과 요약 실행 공간이 분석의 비용을 증가시킨 것으로 보인다.

# 4 결론 및 앞으로

Umirac은 Airac5의 틀을 바탕으로 하여 만들어진, C 프로그램에서 사용되지 않는 변수, 구조체의 필드, 그리고 버퍼의 구간을 찾아내는 요약해석기이다. 이를 위해 구간 집합 도메인을 사용한 요약해석을 설계하고 구현하였다.

요약해석의 틀에서 설계된 Umirac은 반드시 사용되지 않을 자료들만을 찾아내지만 프로그램의 모든 실행의미가 드러나있지 않은 경우 정확하지 않은 경보를 발생시킬 수 있다.

리눅스 커널과 임베디드 시스템 소프트웨어와 같은 실제 프로그램에 대한 실험을 통해 사용되지 않는 자료들을 찾음으로써 요약해석이 메모리 사용량 최적화에 유용한 정보를 제공할 수 있다는 것을 보였다. Umirac의 분석을 위해 확장한 설계는 기존의 구간 분석 요약해석에 비해 60 – 86% 정도의 수행속도를 보였다.

앞으로의 연구는 분석기가 부정확한 분석을 하는 경우를 줄이는 방법과 분석 결과를 활용하는 방법에 대한 것이다. 먼저 분석기의 정확도를 높이기 위해 자주 사용되는 외부 함수들의 의미를 Umirac에 맞게 정의하여 분석기에 탑재하는 과정을 구현하는 중이다. 또한 Umirac의 분석 정보를 활용하여 프로그램에서 사용되지 않는 자료를 자동으로 없애는 프로그램 변환 규칙을 정의하는 연구를 진행하고 있다. 예를 들어 어떤 버퍼에서 사용하지 않는 구간이 발견되었다면, 사용하지 않는 구간이 없게끔 버퍼의 크기를 줄이고 프로그램의 의미가 변하지 않게 버퍼를 접근하는 수식들을 바꾸어주는 것을 생각할 수 있다.

## 감사의 글

본 연구에 있어 Umirac의 초기 설계 및 구현에 도움을 준 오학주에게 감사의 뜻을 전한다.

# 참고문헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Patrick Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM*, 28(2):324–328, June 1996.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[5] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[6] http://ropas.snu.ac.kr/airac7.

[7] Jaeho Shin. An abstract interpretation with the interval domain for C-like programs. Master's thesis, Seoul National University, 2006.

[8] Jaeho Shin, Jaehwang Kim, Hakjoo Oh, Yungbum Jung, and Kwangkeun Yi. Abstact interpreter AiracV. In *Transactions on Programming Languages*, volume 19, pages 11–17. Korea Information Science Society Special Interest Group on Programming Languages, Nov 2005.

[9] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

## 황 의 권

– 2001-2005 연세대학교 학사
– 2005-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

## 이 광 근

– 1983-1987 서울대학교 학사
– 1988-1990 Univ. of Illinois at Urbana-Champaign 석사
– 1990-1993 Univ. of Illinois at Urbana-Champaign 박사
– 1993-1995 Bell Labs., Murray Hill 연구원
– 1995-2003 한국과학기술원 교수
– 2003-현재 서울대학교 교수

<관심분야> 프로그램 분석 및 검증, 프로그래밍 언어

20-2-R05

# 가벼운 모양분석을 통한 메모리 누수 탐지[1]

*Memory-Leak Detection by Lightweight Shape Analysis*

이욱세·정승철
한양대학교 안산캠퍼스 컴퓨터공학과
{oukseh; scjung}@pllab.hanyang.ac.kr

김태호
한국전자통신연구원
taehokim@etri.re.kr

요 약

최근 독보적인 연구들을 통해서 포인터를 다루는 복잡한 프로그램에 대한 포인터 오류 및 메모리 누수 탐지 기술이 획기적으로 진화하여, 복잡한 포인터 프로그램의 오류를 감지할 수 있는 기술이 마련되었다. 그러나, 복잡한 자료구조를 위해 비용이 비싸게 설계되었는데, 비교적 단순한 메모리 구조만을 사용하고 있는 실제 현장에 적용하기 어려운 점이 있다. 본 논문에서는 트리 정도 단순한 모양의 단순한 자료구조를 사용하는 포인터 프로그램의 메모리 오류 발견 및 누수 탐지를 위한 저렴한 분석기를 제안하고자 한다. 문법기반 모양분석 기술을 단순화하여 트리에 대해서만 정밀하게 분석하는 분석기를 고안하였고, 실제 코드에 적용한 사례를 보아 가격과 성능이 만족스러울 것이라고 예상된다.

## 1 서론

최근 독보적인 연구들[3, 4, 7, 8, 9]을 통해서 포인터(pointer)를 다루는 복잡한 프로그램에 대한 포인터 오류 및 메모리 누수 (memory leak) 탐지 기술이 획기적으로 진화하였다. 힙 메모리 (heap memory) 분석은 그 복잡성으로 인하여 한동안 분석 기술이 답보되어 있다가 모양 분석(shape analysis)[4, 9]과 분리 로직(separation logic)[3, 7, 8] 같은 독보적인 기술들이 제안됨으로써 분석기술이 다시 주목받고 있다. 이 기술들은 검증하기 어렵다는 Deutsch-Schorr-Waite 트리 검색 알고리즘[10]이 완전하게 옳다는 것(total correctness)을 검증해 내기도 하였다[6, 11]. 문제는 이러한 검증의 비용은 저렴하지는 않다는 것이다.

실제 현장에서 필요한 기술은 복잡하지 않은 자료구조를 사용하는 프로그램 코드에 대해 저렴한 비용으로 검증하는 기술이다. 실제 리눅스(linux) 운영체제에 사용되는 디바이스 드라이버 (device driver) 소스 코드를 검토해 본 결과, 복잡한 재귀적 자료 구조를 사용하는 경우는 적었다. 재귀적이지 않은 자료 구조를 사용하거나, 재귀적 자료 구조라 하더라도 비교적 단순한 리스트(list)와 유사한 형태의 자료 구조를 사용하고 있다. 복잡한 자료구조를 검증해 내는 것도 분명 필요한 기술이지만 현장에서 사용되는 단순한 자료구조에 대해 효율적인 검증 엔진을 고안하는 것 또한 매우 중요한 일이다.

본 논문에서는 트리 (tree) 모양의 단순한 자료구조를 사용하는 포인터 프로그램의 메모리 오류 발견 및 누수 탐지를 위한 저렴한 모양 분석기를 제안하고자 한다. 제안하는 분석

[1]본 연구는 한국전자통신연구원의 지원으로 이루어졌음.

기는 문법기반 모양분석(grammar-based shape analysis)[4]을 트리 모양의 구조만 정밀하게 분석할 수 있도록 단순화한 형태이고, 분리로직의 프로그램 성질 유추엔진[3]을 리스트 이상의 구조를 분석할 수 있도록 확장한 형태이다.

## 2 분석 대상 언어

분석 대상 언어는 일반적인 포인터 연산을 하는 프로그램이다.

$$
\begin{array}{lll}
\textit{Variable} & x & \\
\textit{Field} & f & \in \{\texttt{1}, \texttt{2}\} \\
\textit{Expr} & e & ::= x\texttt{==}x \mid x\texttt{==0} \mid \texttt{!}e \mid \texttt{true} \\
\textit{Statement} & t & ::= x\texttt{=}x \mid x\texttt{=0} \mid x\texttt{=}x\texttt{.}f \mid x\texttt{.}f\texttt{=}x \mid x\texttt{=new} \mid \texttt{free}\ x \\
 & & \quad \mid\ t\texttt{;}\ t \mid \texttt{if}\ e\ t\ t \mid \texttt{while}\ e\ t
\end{array}
$$

0는 널 포인터(null pointer)를 의미한다. 명령문의 new는 새로운 힙 셀(heap)을 할당(allocation)해 주며, 할당된 힙 셀은 두 개의 필드(field)로 구성되어 있다고 가정한다. 즉, $x\texttt{.1}$는 변수 $x$가 가리키는 힙 셀의 첫 번째 필드를 의미한다. $\texttt{free}\,x$는 $x$가 가리키는 힙 셀만을 반환(disposal)하는 명령문이다.

## 3 분석 도메인

분석 도메인은 기본적으로 문법기반 모양분석[4]의 것을 단순화한 것이다. 문법기반 모양분석은 힙 메모리를 두 요약 수준으로 요약하는데, 구체적인 부분과, 요약된 부분으로 나누어 요약한다. 구체적인 부분은 실제 힙 메모리를 그대로 표현한 부분이고, 요약된 부분은 힙 메모리의 객체를 문법으로 표현하면서 실제를 뭉뚱그린 부분이다.

문법기반 모양분석에서 단순화한 부분은 문법으로 기술된 요약 부분이다. 문법을 사용하면 다양하고 복잡한 자료구조를 표현할 수 있다. 그러나, 현장에서 사용되는 코드가 그 표현력만큼 복잡한 자료구조를 사용하고 있지 않고, 또한 비용도 저렴하지 않아서, 특정한 형태의 문법 형태만 사용하도록 고안하였다.

사용하는 분석 도메인은 다음과 같다.

$$
\begin{array}{lllll}
\textit{Location} & l & & & \\
\textit{Pointer} & p & ::= & 0 \mid - \mid l & \\
\textit{Cell} & c & ::= & \langle o, o \rangle & \\
\textit{Object} & o & ::= & p \mid c \mid \star p & \\
\textit{Stack} & s & \in & \textit{Variable} \to \textit{Object} & \\
\textit{Heap} & h & \in & \textit{Location} \to \textit{Cell} & \\
\textit{Memory} & m & \in & \textit{Stack} \times \textit{Heap} & \\
\textit{AbstractDomain} & M & \in & \mathcal{P}(\textit{Stack} \times \textit{Heap}) + \{\top\} &
\end{array}
$$

포인터는 주소 또는 널포인터 0, 그리고 끊어진 포인터(dangling pointer) $-$이다. 힙 셀은 두 필드를 위한 객체를 갖고 있는데, 객체는 포인터, 셀, 아니면 요약객체 (abstract object) $\star p$이다. 요약객체 $\star p$는 도달가능한 셀들의 모임인데 반드시 출구는 하나 이하인 것만 가능하다. 출구란 요약객체가 의미하는 셀들의 모임에서 모임 바깥을 가리키는 포인터를 뜻한다. 예를 들어 $\star 0$는 출구가 없는 것을 말하고, $\star l$은 출구가 반드시 하나 있고, 그 출구가 주

소 $l$을 가리킨다는 뜻이다.[2] 요약 메모리는 두 부분으로 나뉘어 있는데 스택 부분은 변수에서 객체로 가는 맵이고, 힙은 주소에서 셀로 가는 맵이다.[3] 요약 도메인은 메모리의 집합으로 여러 경우를 표현할 수 있도록 고안되었다. 분석이 어려운 경우를 위해 모든 가능성을 의미하는 $\top$을 두었다.

요약객체는 단순하지만 여러 상황을 표현할 수 있다. 예를 들어, $x$가 트리를 가리키고 있고, $y$가 트리의 중간을 가리키고 있는 상황을 다음과 같이 기술할 수 있다.

$$s = \{x \mapsto *l, y \mapsto l\}, h = \{l \mapsto \langle *0, *0\rangle\}$$

스택에서 보면 $x$는 출구가 있는 요약객체를 가리키는데 그 출구가 주소 $l$을 가리키고, $y$는 $l$을 직접 가리키고 있다. 주소 $l$에는 출구가 없는 요약객체를 가리키고 있는 셀이 있다.

요약 도메인에서의 순서 관계는 $\top$이 가장 높고 나머지는 집합의 포함관계를 따른다. 단지, 같은 모양을 갖고 주소만 다른 두 메모리는 같다고 판정하다.

$$M \sqsubseteq \top$$
$$M_1 \sqsubseteq M_2 \Longleftrightarrow M_1, M_2 \neq \top \wedge \forall m_1 \in M_1.\exists m_2 \in M_2.\exists R.m_1 =_R m_2$$

여기서 $m_1 =_R m_2$는 $R$이 $m_1$에 나오는 주소들과 $m_2$에 나오는 주소들의 일대일 관계(one-to-one relation)이고 $m_1$의 모든 주소를 관계되는 주소로 변경했을 때 $m_2$와 동일할 때 성립한다. 예를 들어, $m_1 = (\{x \mapsto 1\}, \{1 \mapsto \langle *0, *0\rangle\})$와 $m_2 = (\{x \mapsto 2\}, \{2 \mapsto \langle *0, *0\rangle\})$에 대해 $m_1 =_{\{(1,2)\}} m_2$가 성립한다.

# 4 분석기 (Analysis)

분석 엔진은 문법기반 모양분석[4]과 같은 구조를 가지고 있다. 모양분석 엔진의 특징은, 힙 셀에 연산을 수행할 때 초점을 맞추어 요약된 메모리를 구체화하는 것 (focus라 부른다[9]), 그리고, 반복문 등에서 고정점 (fixed-point) 계산이 무한히 반복되는 것을 막기 위해 확장 (widening) 연산[1, 2]을 수행하여 구체화된 메모리를 요약시키는 것이다. 우선, 초점 맞추기 연산과 확장 연산을 설명하고, 실제 분석 엔진을 기술하도록 하겠다.

## 4.1 초점 맞추기 (Focus)

초점 맞추기는 사용할 메모리를 구체적으로 변환하여 메모리에 대한 코드 수행에 대한 분석이 정확하게 이루어질 수 있도록 준비하는 연산이다. 포인터 값을 읽거나 메모리 내용의 변경은 반드시 구체적인 부분에서 이루어져야 완전히 이루어질 수 있다 (strong update). 본 분석기에서는 두 가지 경우의 초점 맞추기가 있는데 하나는 변수에 접근할 때 초점을 맞추는 것과, 셀의 필드에 접근할 때 초점을 맞추는 것이다. 스택 또는 힙에서 연산을 취하는 점을 제외하고는 비슷하게 정의되었다.

초점 맞추기 연산은 그림 1에 있다: 변수 초점 맞추기($\mathsf{Focus}_x$)와 필드 초점 맞추기($\mathsf{Focus}_{x.f}$). 두 연산 모두 초점 대상을 보고 세 가지 경우로 나뉘는 데, 초점 대상이 포인터이면 초점이 이미 맞추어진 것이고, 초점 대상이 셀인 경우 셀을 독립시키게 되고, 초점 대상이 요약객체인 경우 요약객체를 한 번 풀어서 경우를 구체화시킨다. 요약객체 $\star p$를 풀 때는 세 가지 경우가 생기는데, 첫 번째 경우는 요약객체가 셀 없이 바로 출구인 경우 포인터 $p$로 치환하고, 두번째 경우는 요약객체의 입구 셀을 꺼내는 데 출구가 첫 번째 필드쪽

[2] 문법기반 모양분석의 수식을 빌리면 $\star 0$는 "$\alpha \rightarrow \mathrm{nil} \mid \langle \alpha, \alpha\rangle$" 문법의 $\alpha$가 의미하는 객체이고, $\star l$은 "$\beta(n) \rightarrow n \mid \langle \beta(n), \alpha\rangle \mid \langle \alpha, \beta(n)\rangle$" 문법의 $\beta(l)$이 의미하는 객체이다.

[3] 요약 메모리의 스택, 힙은 실제 의미구조의 스택, 힙과 정확히 일치하지 않는다.

$\mathsf{Focus}_x M = M \neq \top$이면 $\bigcup \{\mathsf{focus}_x\, m \mid m \in M\}$, 그 외의 경우, $\top$

$$\mathsf{focus}_x\,(s,h) = \begin{cases} \{(s,h)\}, & s(x) = p\text{인 경우} \\ \{(s[l/x], h[c/l])\},\ \text{새로운 주소 } l, & s(x) = c\text{인 경우} \\ \bigcup \{\mathsf{focus}_x\,(s[p'/x], h \cup h') \mid (p',h') \in \mathsf{unfold}\,\star p\}, & s(x) = \star p\text{인 경우} \end{cases}$$

$$\mathsf{unfold}\,\star p = \{(p,\emptyset), (l, \{l \mapsto \langle \star p, \star 0\rangle\}), (l, \{l \mapsto \langle \star 0, \star p\rangle\})\},\ \text{새로운 주소 } l$$

$\mathsf{Focus}_{x.1} M = M' \neq \top$이며, $\forall m \in M'.\mathsf{focus}_{x.1}\, m$이 정의되면 (여기서 $M' = \mathsf{Focus}_x M$)
$\bigcup \{\mathsf{focus}_{x.1}\, m \mid m \in M'\}$, 그 외의 경우, $\top$
($\mathsf{Focus}_{x.2} M$도 유사하게 정의)

$$\mathsf{focus}_{x.1}\,(s,h) = \begin{cases} \{(s,h)\} & s(x) = l,\ h(l) = \langle p, o\rangle\ \text{인 경우} \\ \{(s, h[\langle l', o\rangle / l, c/l'])\},\ \text{새로운 주소 } l' & s(x) = l,\ h(l) = \langle c, o\rangle\ \text{인 경우} \\ \bigcup \{\mathsf{focus}_{x.1}\,(s, h[\langle p', o\rangle] \cup h') \mid (p',h') \in \mathsf{unfold}\,\star p\} & s(x) = l,\ h(l) = \langle \star p, o\rangle\ \text{인 경우} \\ \text{정의 안됨}, & \text{그외의 경우} \end{cases}$$

[그림 1] 초점 맞추기 연산 (focus operators).

에 있는 경우, 마지막 경우는 출구가 두 번째 필드쪽에 있는 경우이다. 요약객체의 구체화는 풀기 연산(unfold)을 통해 수행된다.

예를 살펴보면 다음과 같다. 앞서 소개했던 예제, 즉, $x$가 트리를 가리키고 있고, $y$가 트리의 중간을 가리키고 있는 상황에서 시작하겠다.

$$s = \{x \mapsto *l, y \mapsto l\}, h = \{l \mapsto \langle *0, *0\rangle\}$$

$x$에 대해 연산을 하기 위해 초점을 맞추어 보면, $x$가 요약객체를 가리키고 있으므로, 풀어서 세가지 상황이 생긴다.

$$s = \{x \mapsto l, y \mapsto l\}, h = \{l \mapsto \langle *0, *0\rangle\}$$
$$s = \{x \mapsto l', y \mapsto l\}, h = \{l' \mapsto \langle *l, *0\rangle, l \mapsto \langle *0, *0\rangle\}$$
$$s = \{x \mapsto l', y \mapsto l\}, h = \{l' \mapsto \langle *0, *l\rangle, l \mapsto \langle *0, *0\rangle\}$$

첫번째 경우는 $x$, $y$가 동일 셀을 가리키는 경우, 두번째 경우는 $x$의 첫번째 필드를 통해 공유하는 경우, 마지막 경우는 $x$의 두번째 필드를 통해 공유하는 경우이다.

## 4.2 확장 (Widening)

확장 연산은 구체적인 객체를 요약함으로써 분석이 무한히 끝나지 않음을 막아준다 [1, 2]. 제안하는 분석기의 확장 연산은 다음과 같이 정의할 수 있다. 문법기반 모양분석[4]의 경우와 매우 유사하지만, 각 연산이 보다 단순하고 문법을 정리하는 연산이 없다.

$$\mathsf{Widen} = \mathsf{Unify} \circ \mathsf{Bound}_k \circ \mathsf{Fold} \circ \mathsf{Checkleak}$$

첫 번째 연산 Checkleak은 메모리 누수가 있는 경우 분석 결과를 $\top$으로 해 준다. 두 번째 연산 Fold는 구체적인 셀들을 접어서 메모리 객체를 그룹화한다. 세 번째 연산 $\mathsf{Bound}_k$는 그래도 해결할 수 없는 복잡한 자료구조인 경우 깊이를 제한해 준다. 마지막 연산 Unify는 비슷한 경우를 합쳐주어 경우의 수를 줄인다. 확장 연산의 부속 연산자들은 Checkleak을 제외하고 그림 2에 있다.

– 누수 검사 (Checkleak): 메모리 $(s,h)$에 대해 주소 $l$이 도달가능하다는 것은 어떤 변수 $x \in \mathrm{dom}(s)$에 대해 $s(x)$에 $l$이 나타나거나, 도달가능한 주소 $l' \in \mathrm{dom}(h)$에 대해 $h(l')$에

$$\mathsf{Fold}\ M = M \neq \top \text{이면 } \{\mathsf{fold}\ m \mid m \in M\},\ \text{그 외의 경우, } \top$$

$$\mathsf{fold}\ (s,h) = \begin{cases} \mathsf{fold}\ (s\{c/l\}, h'\{c/l\}), & \text{어떤 } l \in \mathrm{dom}(h)\text{에 대해, } h = h' \uplus \{l \mapsto c\} \text{ 이고,} \\ & l\text{이 한 번 } (s,h')\text{에 나타나며 } \mathrm{exit}(c)\text{가 정의된 경우} \\ (s,h), & \text{그외의 경우} \end{cases}$$

$$\mathrm{exit}(o) = \begin{cases} p, & o = p \text{ 또는 } o = \star p\text{인 경우} \\ \mathrm{exit}(o_2), & o = \langle o_1, o_2\rangle \text{ 이고 } \mathrm{exit}(o_1) = 0\text{인 경우} \\ \mathrm{exit}(o_1), & o = \langle o_1, o_2\rangle \text{ 이고 } \mathrm{exit}(o_2) = 0\text{인 경우} \\ \text{정의 안됨}, & \text{그외의 경우} \end{cases}$$

$$o\{c/l\} = \begin{cases} \langle o_1\{c/l\}, o_2\{c/l\}\rangle, & o = \langle o_1, o_2\rangle \text{ 인 경우} \\ c, & o = l\text{인 경우} \\ \star p, & o = \star l\text{이고 } \mathrm{exit}(c) = p\text{인 경우} \\ o, & \text{그 외의 경우} \end{cases}$$

$$(s,h)\{c/l\} = (\{x \mapsto (s(x))\{c/l\} \mid x \in \mathrm{dom}(s)\}, \{l \mapsto (h(l))\{c/l\} \mid l \in \mathrm{dom}(h)\})$$

$$\mathsf{Bound}_k\ M = M \neq \top \text{이면 } \{\mathsf{bound}_k\ m \mid m \in M\},\ \text{그 외의 경우, } \top$$

$$\mathsf{bound}_k\ (s,h) = (s,h')\{-/l_1\}\cdots\{-/l_n\}$$

$$\text{단, } l_i \in \left\{ l \in \mathrm{dom}(h) \quad \mathrm{depth}_{(s,h)}(l) > k \right\} \text{이고 } h = h' \uplus \{l_1 \mapsto c_1, \cdots, l_n \mapsto c_n\}$$

$\mathrm{depth}_{(s,h)}(l)$ = 변수에서 $l$까지의 경로 중 가장 짧은 것의 길이
경로는 $\rightsquigarrow$로 연결된 것의 나열: $s(x)$에 $l$이 있으면 $x \rightsquigarrow l$, $h(l)$에 $l'$이 있으면 $l \rightsquigarrow l'$

$$\mathsf{Unify}\ M = \begin{cases} (M \setminus \{m_1, m_2\}) \cup \{m_1 \oplus_S m_2\}, & M \neq \top\text{이고, } M\text{에 } m_1 \sim_R m_2\text{인 } m_1, m_2, R\text{이 있으면} \\ M & \text{그 외의 경우} \end{cases}$$

$$\begin{aligned} o \sim_R o' \quad &\Longleftrightarrow\ o = \langle o_1, o_2\rangle \wedge o' = \langle o_1', o_2'\rangle \wedge o_1 \sim_R o_1' \wedge o_2 \sim_R o_2' \\ &\quad\ \vee\ (o \notin \mathit{Cell} \vee o' \notin \mathit{Cell}) \wedge (\mathrm{exit}(o), \mathrm{exit}(o')) \in R \\ (s,h) \sim_R (s',h') \ &\Longleftrightarrow\ \mathrm{dom}(s) = \mathrm{dom}(s') \wedge R\text{이 } \mathrm{dom}(h) \times \mathrm{dom}(h')\text{에 대한 일대일 관계} \\ &\quad\ \wedge\ \forall x \in \mathrm{dom}(s).s(x) \sim_R s'(x) \wedge \forall (l,l') \in R.h(l) \sim_R h'(l') \end{aligned}$$

$$o \oplus o' = \begin{cases} \langle o_1 \oplus o_1', o_2 \oplus o_2'\rangle, & o = \langle o_1, o_2\rangle,\ o' = \langle o_1', o_2'\rangle \text{ 인 경우} \\ o, & o, o' \in \mathit{Pointer}\text{인 경우} \\ \star(\mathrm{exit}(o)), & \text{그 외의 경우} \end{cases}$$

$$(s,h) \oplus_R (s',h') = (\{x \mapsto s(x) \oplus s(x') \mid x \in \mathrm{dom}(s)\}, \{l \mapsto h(l) \oplus h'(l') \mid (l,l') \in R\})$$

[그림 2] 확장(widening)을 위한 연산자: Fold, $\mathsf{Bound}_k$, Unify.

$l$이 나타나는 경우이다. 메모리 $(s,h)$에 누수가 없다는 것은 모든 주소 $l \in \mathrm{dom}(h)$가 도달가능하다는 것이다. 요약도메인 $M$에 대해 누수가 없다는 것은 $M$이 $\top$이 아니고, $M$에 속한 모든 메모리 $(s,h)$가 누수가 없다는 것이다.

$$\mathsf{Checkleak}(M) = \begin{cases} M, & M\text{에 누수가 없을 때} \\ \top, & M\text{에 누수가 있을 때} \end{cases}$$

– 접기 (Fold): 접기 대상은 매우 한정적인 객체에 대해서만 행해진다. 자신을 가리키는 포인터가 하나이고, 객체가 가리키는 포인터(출구)가 하나 이하일 때에만 접는다. 함수 exit를 통해서 객체의 출구를 찾는데, 출구가 하나 일 때는 그 포인터, 출구가 없을 때는 널포인터(0)를 결과로 하고, 출구가 여러 개일 때는 정의되지 않는다. 접는 방법은 자신을 가리키는 포인터를 자기자신으로 치환하는 것이다. 치환 과정에서 요약객체의 출구에 치환될 때 객체는 요약되어 치환된다. 즉, 객체의 구체적인 구조는 무시되고 탈출 포인터만 남게된다. 예를 들어, $x$는 출구가 $l$인 요약객체를 가리키고, 주소 $l$에는 첫번째 필드가 0이고, 두 번째 필드가 포인터 $l'$이면, 그 셀은 $x$의 요약객체와 통합되어, 결과적으

로 $x$는 출구가 $l'$인 요약객체를 가리키게 되는 것이다.

접는다고 모든 셀이 요약되지는 않는다. 문법기반 모양분석에서는 모든 접어진 셀은 문법으로 변환되어 요약될 가능성이 높아진다. 그러나, 본 논문에서 제시한 접기는 요약객체의 출구로 접을 때를 제외하고는 바로 요약되지는 않는다. 단지 주소만 사라지고 그 셀의 모양이 그대로 객체 내에서 유지되게 된다. 예를 들어, $x$가 길이 3인 리스트를 가리키고 있다고 하자. 이런 경우 문법기반 모양분석에서는 접기 과정을 통해 3개의 셀이 문법으로 변환되어, 마지막 문법 요약 과정에서 $x$가 임의의 길이의 리스트를 가리키고 있다고 요약되게 된다. 그러나, 본 접기 방법은 셀들의 주소만 사라지고 $x$가 길이 3의 리스트를 가리키고 있다는 정보가 남게 된다. 특별히 $x$가 길이가 다른 리스트를 가리키는 추가의 경우가 없을 때는 이후에도 요약되지 않는다.

– 깊이 제한 ($\mathsf{Bound}_k$): 요약 메모리를 접어서 단순화하더라도 분석 도메인 자체가 단순한 트리 모양 구조에만 적합하기 때문에 요약 메모리의 크기가 무한히 증가하는 경우가 발생한다. 이 경우에는 어쩔 수 없이 깊이 제한을 할 수 밖에 없다. 예를 들어, 프로그램이 필드 1, 2가 같은 셀을 가리키도록 셀을 생성하고, 다시 셀을 생성하되 필드 1, 2가 그 전에 생성된 셀을 가르키도록 하기를 반복하면, 전혀 접어지지 않아 무한한 그래프를 만들어 낼 수 있다. 이런 최악의 경우에는 깊이를 재서 깊이가 너무 깊은 것($k$ 초과)은 끊어진 포인터로 치환해 준다.

– 유사 메모리 통합 (Unify): 유사 메모리를 통합하여 경우의 수를 줄여 준다. 유사하다는 것은 각 스택의 도메인이 같고 힙의 주소간에 일대일대응 관계가 있을 때, 스택의 대응되는 객체끼리 유사하고, 힙의 대응되는 객체끼리 유사하다는 것이다. 두 객체가 유사하다는 것은, 둘 다 셀인 경우 각 필드들이 끼리끼리 유사하여야 하고, 모두 셀이 아닌 경우는 출구가 대응되는 주소로 같아야 하는 것이다. 예를 들면, 포인터 $p$와 출구가 $q$인 요약객체는 $p$, $q$가 대응되는 주소이면 유사하다.

두 유사한 메모리의 통합은 유사함을 판단하는 것과 비슷하게 통합된다. 두 셀은 각 필드 별로 통합되고, 두 포인터는 포인터로 통합되는데, 그 외의 경우는 모두 요약객체로 통합된다. 즉, 출구 정보만 남기고 모두 요약되는 것이다. 예를 들어, 유사한 메모리에 한쪽은 $x$가 널 포인터 0를 갖고 있고, 다른 쪽은 $x$가 모든 필드가 0인 셀을 갖고 있다면, 통합되면서 $x$는 요약객체를 갖게 되고, 그 출구는 공통된 출구인 0가 되는 것이다. 결국 크기가 다른 객체들은 유사 메모리 통합 과정을 통해 임의의 크기를 가지는 요약객체가 되는 것이다.

## 4.3 분석 엔진

분석 엔진 또한 문법기반 모양분석의 것[4]과 유사하게 그림 3에 정의되었다. 메모리에 대한 연산을 수행할 때 해당 변수 또는 해당 필드에 대해 초점을 맞춘 후 수행한다. 초점이 맞추어진 부분은 항상 포인터 값을 가지게 된다는 것을 기억하기 바란다. 메모리를 반환하는 `free` 명령문에 대해서는 양쪽 필드 모두에 대해 초점을 맞춘 후 수행한다. 이유는 `free` 명령문은 셀이 소멸되는 것으로 단지 그 변수에 대해서만 메모리 명령을 수행하는 것이 아니라 필드에 있는 포인터들도 동반 소멸되는 것이기 때문이다. 역시 반복문인 `while` 문에서는 확장연산을 수행하여 무한히 반복하지 않도록 한다.

조건문이나 반복문의 조건계산식에 대해서는 가능한 경우를 줄여서 수행한다 (context pruning). 이 때도 역시 조건 대상이 되는 변수에 대해 초점을 맞춘 후 조건을 만족하는지 파악하다. 초점을 맞추고 나면 대상이 모두 포인터가 되기 때문에 조건 만족을 파악하기 용이하다. 끊어진 포인터($-$)는 어떤 것과도 같을 수도 있고 다를 수도 있다. 그 외의 경우는 같고 다름이 실제 의미구조와 일치한다. 즉, 요약 도메인에서의 같은 주소는 실제 의미구조에서도 같은 주소이고, 서로 다른 주소는 실제 의미구조에서도 서로 다른 주소를 의미한다.

$\mathcal{B} : Expr \to AbstractDomain \to AbstractDomain$

$$
\begin{array}{lcl}
\mathcal{B}[\![e]\!]M_0 & = & M_n \neq \top\text{인 경우 } B[\![e]\!]M_n \\
 & & \text{여기서, } \{x_1, \cdots, x_n\} = V(e)\text{이고 } M_i = \mathsf{Focus}_{x_i} M_{i-1} \text{ for } 1 \leq i \leq n \\
 & & V(x\texttt{==}0) = \{x\},\ V(x\texttt{==}y) = \{x, y\},\ V(\texttt{!}e) = V(e) \\
\mathcal{B}[\![e]\!]M & = & \top\ (\text{그 외의 경우}) \\
B[\![x\texttt{==}0]\!]M & = & \{(s,h) \in M \mid s(x) = 0 \vee s(x) = -\} \\
B[\![\texttt{!}(x\texttt{==}0)]\!]M & = & \{(s,h) \in M \mid s(x) \neq 0\} \\
B[\![x\texttt{==}y]\!]M & = & \{(s,h) \in M \mid s(x) = s(y) \vee s(x) = - \vee s(y) = -\} \\
B[\![\texttt{!}(x\texttt{==}y)]\!]M & = & \{(s,h) \in M \mid s(x) \neq s(y) \vee s(x) = - \vee s(y) = -\} \\
B[\![\texttt{!!}e]\!]M & = & B[\![e]\!]M
\end{array}
$$

$\mathcal{S} : Statement \to AbstractDomain \to AbstractDomain$

$$
\begin{array}{lcl}
\mathcal{S}[\![t]\!]M_0 & = & t\text{가 } x\texttt{=}y,\ x\texttt{=}0,\ x\texttt{=}y\texttt{.}f,\ x\texttt{.}f\texttt{=}y,\ x\texttt{=new}\text{에 한해} \\
 & & M_n \neq \top\text{인 경우 } \bigcup\{S[\![t]\!]m \mid m \in M_n\} \\
 & & \text{여기서, } \{a_1, \cdots, a_n\} = E(e)\text{이고 } M_i = \mathsf{Focus}_{a_i} M_{i-1} \text{ for } 1 \leq i \leq n \\
 & & E(x\texttt{=}0) = E(x\texttt{=new}) = \{x\},\ \text{그외의 경우, } E(a\texttt{=}a') = \{a, a'\} \\
\mathcal{S}[\![\texttt{if } e\ t_1\ t_2]\!]M & = & \mathcal{S}[\![t_1]\!](\mathcal{B}[\![e]\!]M) \sqcup \mathcal{S}[\![t_2]\!](\mathcal{B}[\![\texttt{!}e]\!]M) \\
\mathcal{S}[\![t_1\texttt{;}t_2]\!]M & = & \mathcal{S}[\![t_2]\!](\mathcal{S}[\![t_1]\!]M) \\
\mathcal{S}[\![\texttt{while } e\ t]\!]M & = & \mathcal{B}[\![\texttt{!}e]\!](\text{fix } \lambda M'.\mathsf{Widen}(M \sqcup \mathcal{S}[\![t]\!](\mathcal{B}[\![e]\!]M'))) \\
\mathcal{S}[\![t]\!]M & = & \top\ (\text{그 외의 경우})
\end{array}
$$

$$
\begin{array}{lcl}
S[\![x\texttt{=}y]\!](s,h) & = & (s[s(y)/x], h) \\
S[\![x\texttt{=}0]\!](s,h) & = & (s[0/x], h) \\
S[\![x\texttt{=}y\texttt{.}i]\!](s,h) & = & (s[p_i/x], h) \text{ where } h(s(y)) = \langle p_1, p_2 \rangle \\
S[\![x\texttt{.1=}y]\!](s,h) & = & (s, h[\langle s(y), o \rangle / s(x)]) \text{ where } h(s(y)) = \langle p, o \rangle \\
S[\![x\texttt{.2=}y]\!](s,h) & = & (s, h[\langle o, s(y) \rangle / s(x)]) \text{ where } h(s(y)) = \langle o, p \rangle \\
S[\![x\texttt{=new}]\!](s,h) & = & (s[\langle -, - \rangle / x], h) \\
S[\![\texttt{free } x]\!](s,h) & = & (s[-/x], h') \text{ where } h = h' \uplus \{s(x) \mapsto \langle p_1, p_2 \rangle\}
\end{array}
$$

[그림 3] 분석기.

# 5 구현 및 분석 사례

분석기의 프로토타입(prototype)은 구현이 되었다. Objective Caml 언어[5]로 구현되었고, 입력으로 받는 언어는 C를 단순화한 유사한 중간 언어이다. 향후 C 프로그램을 직접 입력받을 수 있는 분석기로 확장 구현할 예정이다. 구조체의 필드 수가 여러 개인 경우로 확장하였고, 스택 셀의 주소도 사용할 수 있도록 확장하였다. C 프로그램의 메모리 연산들 중 현재 지원하지 않는 기능은, 메모리 필드의 주소를 사용하는 경우, 배열을 사용하는 경우, 공용체(union)을 사용하는 경우, 포인터 정수 연산(pointer arithmetic)을 사용하는 경우 등이다.

프로시저 호출(procedure call)에 대해서는 지역 계산(local reasoning)의 원리[7]를 따라 구현되었다. 기본적으로는 문맥둔감 (context-insensitive) 분석을 수행한다. 그러나, 호출 당시의 모든 메모리를 모두 모아 분석하지는 않는다. 호출된 프로시저가 사용할 메모리 부분만 골라내서 모아 프로시저를 분석하고 결과와 나머지 메모리를 결합하여 호출 후를 분석한다. 이 때 호출된 프로시저가 사용할 메모리는, 전역 변수에서 도달 가능한 메모리와 인자를 통해 전달되는 값에서 도달 가능한 메모리를 말한다. 이 분리 과정에서 분리가 깔끔

하게 처리되지 않는 경우, 즉, 프로시저에 전달할 메모리의 중간 부분을 전달되지 않는 메모리가 가리키고 있는 경우는 정확하게 분석하지 못한다.

디바이스 드라이버 소스 코드 중 필립스 웹 캠에 대한 드라이버 소스 코드에 대해 분석해 본 결과를 사례로 살펴 보기로 한다. 해당 드라이버 소스의 `pwc_handle_frame`이라는 프로시저는 `pdev` 타입의 구조체를 전달 받는다. 이 구조체에는 네 개의 리스트 구조체 포인터 타입의 필드가 있다: `full_frame`, `read_frame`, `empty_frames`, `empty_frames_tail`. 메인 함수를 작성해서 `pwc_handle_frame`을 호출하도록 하였는데 이 때 네 개의 필드를 독립적인 길이 0이상의 리스트를 가리키도록 초기화하였다. 분석 결과 널포인터 접근 오류가 발생하였다. 문제의 소스는 다음과 같다.

```
1:    if (pdev->empty_frames == NULL) {
2:       pdev->empty_frames = pdev->read_frame;
3:       pdev->empty_frames_tail = pdev->empty_frames;
4:   }
5:   else {
6:       pdev->empty_frames_tail->next = pdev->read_frame;
7:       pdev->empty_frames_tail = pdev->read_frame;
8:   }
```

분석해 보면, `empty_frames`가 널이 아니라고 해서 `empty_frames_tail`가 널이 아닌 것은 아니므로, 6번 줄에서 오류가 발생하였다. 두 번째 분석에서는 `empty_frames_tail`을 길이 1이상의 리스트로 초기화하였다. 결과는 6번 줄에서 메모리 누수가 발생할 수 있다고 보고하였다. 이유는 `empty_frames_tail->next`가 가리키고 있는 셀 들이 도달 불가능해 질 수 있기 때문이다. 소스 코드의 초기화 코드를 보고 다음과 같이 제대로 초기화 해 주었다. 두 경우가 있는데 `empty_frames`, `empty_frames_tail` 두 필드가 모두 널인 경우, 또 하나는 다음과 같은 경우이다.

$$l \mapsto \langle \star l' \rangle, l' \mapsto \langle 0 \rangle$$

여기서 $l$은 `empty_frames`의 주소이고, $l'$은 `empty_frames_tail`의 주소이다. 즉, $l$은 리스트를 가리키는 데 그 끝 셀을 $l'$이 가리킨 다는 것이다. 제대로 초기화한 경우 메모리 오류 및 누수가 없다고 검증해 주었다.

## 6 향후 연구과제

향후 연구과제로는 실제 사용되는 소스 코드에 대해 분석을 적용하여 분석기가 실용적임을 입증할 필요가 있다. 현재 구현된 프로토타입은 제약이 있지만 많은 기능에 대해 확장되어 있어 일반적인 C 프로그램에 대해 분석할 준비가 되었다. C 전단부와 중간 언어로의 변환기를 구현하여 실제 C 프로그램을 분석할 수 있는 분석기를 구현하고, 실제 성능과 정확도를 파악하는 일이 필요하다. 그 이후에 분석 설계의 경중을 조정할 필요가 있다.

향후 이론적으로 해결할 과제는 양방향 리스트(doubly-linked list) 등 실제 사용되는 조금 더 복잡한 자료 구조에 대한 분석 확장과 프로시저 호출 부분의 개선이다. 트리 모양의 자료 구조외에 자주 사용하는 양방향 리스트까지는 분석할 수 있는 것이 실용적인 메모리 누수 탐지기를 위해 필요한 일이다. 프로시저 호출 부분에는 인자로 전달되는 메모리와 남는 메모리 간에 간섭이 있는 경우 정확도가 떨어지는 요인이 있는데 이를 개선하는 일이 필요하다.

# 참고문헌

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[2] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[3] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302, April 2006.

[4] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Proceedings of the European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 124–140, April 2005.

[5] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.09. Institut National de Recherche en Informatique et en Automatique, October 2005. `http://caml.inria.fr`.

[6] A. Loginov, T. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Proceedings of the International Symposium on Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 261–279, 2006.

[7] P. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the CSL*, Lecture Notes in Computer Science, pages 1–19, 2001.

[8] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[9] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[10] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communication of the ACM*, 10(8):501–506, August 1967.

[11] H. Yang. *Local Reasoning for Stateful Programs.* PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, June 2001.

이욱세

– 1991–1995 한국과학기술원 전산학과 학사
– 1995–1997 한국과학기술원 전산학과 석사
– 1997–2003 한국과학기술원 전산학과 박사
– 2003–2004 서울대학교 전자컴퓨터공학부 박사후연구원
– 2004–현재 한양대학교 컴퓨터공학과 교수

<관심분야> 프로그램 분석 및 검증, 타입 시스템, 컴파일러

정승철

– 2002–2006 한양대학교 전자컴퓨터공학부 학사
– 2006–현재 한양대학교 컴퓨터공학과 석사과정

<관심분야> 프로그램 분석 및 검증, 메모리 분석

김태호

– 1991–1995 성균관대학교 정보공학과 학사
– 1995–1997 한국과학기술원 전산학과 석사
– 1997–2005 한국과학기술원 전산학과 박사
– 2005–현재 한국전자통신연구원 임베디드SW연구단 선임연구원

<관심분야> 정형 명세 및 검증, 임베디드 소프트웨어

20-2-R06

# XML 기반의 문서통합 시스템 설계
## (Designed of Contents Integration System based on XML)

유 재 규, 박 재 훈, 전 양 승, 정 영 식, 한 성 국
원광대학교 전기전자및정보공학부 컴퓨터공학과
(jkyoo82; pjh98; globaljeon; ysjeong; skhan)@wonkwang.ac.kr

요 약

국내외적으로 정보통신망의 구축이 활발히 진행됨에 따라 이를 이용하여 개인이나 기업과 기관에서 업무의 효율성을 높이기 위하여 전자적으로 문서를 교환하여야 할 필요성이 증대되고 있다. 본 논문에서는 XML기반으로 문서를 통합하는 시스템을 설계하였다. 문서를 개인별로 산출물을 작성하는 문서의 경우 개별적으로 작성한 후 작성된 파일을 종합하여 하나의 문서로 통합하는 작업을 거치게 된다. 이러한 수동적인 방법을 컴퓨터 시스템을 이용하여 통합하는 과정을 자동화하고자 한다. 본 논문에서는 내용과 관련된 태그를 직접 만들 수 있고, XML 문서들을 하나의 큰 문서로 병합할 수 있으며, 어떠한 종류의 응용프로그램과도 통합될 수 있는 범용적 데이터베이스라고도 할 수 있는 XML의 특성을 이용하여 좀 더 확장성이 좋아지게 하였다.

## 1. 서 론

현재 공공기관, 업체, 학교 등에서는 수많은 형태의 문서 작업이 이루어지고 있다. 하지만 문서 작업 시 사람이 절대적으로 개입되어져야 하는 실정이다. 또한 IT 기술이 발전되어지고 있음에도 불구하고 이러한 일련의 작업들은 비효율적인 형태로 진행되어지고 있다.

이러한 문제점을 해결하기 위하여 CMS (Contents Management System) 등에서 이러한 문제를 해결하고자 하는 시도가 있었지만, 많은 문제점에 봉착하였다. 그 문제점은 문서(콘텐츠)의 통합, 재활용 등이 있다.

따라서 본 논문에서는 XML/Schema를 이용하여 문서의 구조적인 정보 모델링을 하고, 문서의 각 요소들을 기술하기 위하여 XML 문법을 준수하여 표현될 수 있도록 문서 구조 정의를 한다. 이러한 문서의 본 시스템 표준 문서 템플릿을 정의하고, 공동으로 문서를 작업하기 위해서 문서 템플릿을 각각 사용자에게 배포한다.

그리고 각 사용자가 작성하고자 하는 부분을 작성하면, 사용자는 업로드(upload) 인터페이스를 통해서 업로드를 하게 되고, 업로드된 파일은 정해진 템플릿에 복사되어 저장되어 진다. 이때, 사용자가 업로드 하는 문서의 부분을 체크하고, 버전 관리를 해야 한다.

마지막으로 각 작성자가 최종 파일을 올리고서 관리자에게 업로드 완료 메시지를 보내게 되면 관리자는 최종적인 문서의 형태를 점검하는 것으로 문서의 통합 작업이 끝나게 된다.

## 2. 관련 연구

### 2.1. XML

초기에 웹은 단순히 정보를 제공하기 위하여, 예를 들면 문자나 그림을 표현하기 위한 도구정도만 사용되어져왔다. 인터넷상의 대부분의 정보는 HTML문서로 구성되어있으며, HTML은 단지 문서의 재현을 위한 정보를 나타내는 하나의 정의된 DTD(Document Type Definition)를 사용하기 때문에 각 문서의 엘리먼트를 의미 있는 정보를 표현하는 기능이 부족하다.

이에 W3C(World Wide Web Consortium)에서는 차세대 웹 문서의 표준으로 XML (Extensible Markup Language)을 1998년에 지정하였다[1]. XML은 확장성 마크업 언어로서 이름 그대로 HTML같은 고정형식이 아니라 확장이 가능한 언어이며 문서의 내용과 관련된 태그를 직업 정의할 수 있으며 그 태그를 다른 사람들이 사용할 수 있다. XML은 본질적으로 다른 언어를 기술하기 위한 언어, 즉 메타언어이다. 또한 구조적 데이터를 표현할 수 있으며 사용자가 정의한 DTD를 만족하는 트리 구조를 가지고 있어서 XML은 구조적 문서를 표현하는데 유용하다.

또한 XML 기반언어로 RDF/S, RSS, MathML, XHTML, SVG 등이 있으며, 이들 언어들은 단일하게 규정된 방식으로 정의되었기 때문에, 사전정보 없이도 이들 언어로 작성된 문서에 대해 수정이나 적합성 검사를 하는 프로그램의 제작도 가능하다.

### 2.2. XML 스키마

XML 스키마는 XML 문서의 내용을 제안하고 기술하는 XML언어이다[2,3,4]. XML 문서의 항목과 포함할 수 없는 항목에 대한 규칙을 정의한다. 예를 들어 날짜 필드에 단어를 입력할 수 없도록 스키마를 정의할 수 있다.

XML 스키마는 DTD의 단점을 보완하기 위해 만들어 졌다. DTD는 내용 모델을 정의하는데 사용되었으며, 다음과 같이 뚜렷한 한계를 가지고 있다[3,4].

- DTD는 XML과 다른 문법을 사용한다.
- DTD는 이름공간(Namespace)을 지원하지 못한다.
- DTD는 데이터형이 제한적이다.
- DTD는 복잡하고 느슨한 확장 메카니즘을 가지고 있다.

XML 스키마는 DTD의 이런 단점을 보완하기 위하여 다음과 같은 특징을 가지고 있다.

- 45가지의 다양한 데이터타입을 지원한다.
- 사용자 정의 데이터타입을 선언할 수가 있다.
- 이름공간을 지원한다.
- 사용자 정의 데이터타입이나 상속을 재정의 한다.
- 속성을 그룹핑한다.
- 모듈화와 재사용이라는 객체 지향적 개념을 제공한다.

### 2.3. XSLT

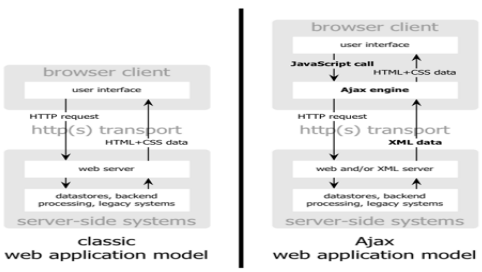
XSLT(XSL Transformations)는 한 XML 문서 구조를 다른 구조를 갖는 XML 문서로 변환하는 방법을 기술하기 위한 표준적인 방법으로서, W3C의 권고사항 중 하나이다[5]. XSLT는 XSL의 확장판이라고도 생각할 수 있다. XSL은 XML 문서, 예를 들어 XML 문서 내에 기술되어 있는 데이터가 어떤 방식으로 웹페이지 내에 표현되어야 하는 가를 보여주는 문서를 형식화하기 위한 언어이다. XSLT는 XML 문서가 다른 데이터 구조로 어떻게 재구성되어야 하는지를 보여준다.

### 2.4. AJAX

AJAX[1),2)]는 대화식 웹 애플리케이션의 제작을 위해 다음과 같은 기술 조합을 이용하는 웹 개발 기법을 통칭한다[5,6].

- 문서 표현을 위한 XHTML(또는 HTML)과 CSS표준
- 동적인 화면 출력 및 정보와의 상호작용을 위한 DOM, JavaScript
- 웹 서버와 비동기적으로 데이터를 교환하고 조작하기 위한 XML, XSLT, XMLHttpRequest (XML/XSLT 대신 미리 정의된 HTML이나 일반 텍스트, JSON-RPC를 이용할 수도 있음)

AJAX 기술을 활용한 웹 응용한 ActiveX 기반의 응용과 달리 XML 처리를 할 수 있는 DOM 엔진과 JavaScript 엔진을 가진 대부분의 브라우저나 플랫폼에서 이용가능 가능하다. 또한 비동기적인 데이터 교환을 가능하기에, 요청에 대한 서버의 응답을 기다리지 않고 다음 작업이 가능하므로 대기시간이 줄어들고, 이에 따라 서버의 부담을 줄이고 사용자의 체감속도를 높일 수 있다. 또한 이벤트 처리를 할 수 있으므로 효과적인 사용자 인터페이스 구현이 가능하다.




(그림 1) 동기 통신, 비동기 통신

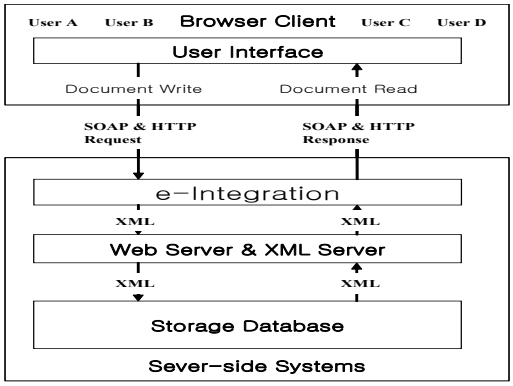
1) AJAX라는 명칭의 기원은 제시 제임스 가렛(Jesse James Garrett)이 2005년 2월 18일 쓴 'A New Approach to Web Application'이라는 에세이에서 'Ajax(Asynchronous JavaScript + XML)'라는 낱말로 이 기술을 소개한 이후로 알려지게 됨

2) AJAX에 대한 발음은 '에이잭스'나 '에작크스', '아약스', '아작스'등 여러 가지가 있지만, 최근에는 '에이잭스'로 통일되어 가고 있는 추세임

## 3. 문서통합 시스템 설계

### 3.1. 문서통합 시스템의 개요

본 논문에서 설계한 XML 기반의 문서통합 시스템은 XML, XML 스키마, XLST, AJAX를 이용하여 시스템을 설계하였다. 그 전체적인 수행 구조는 다음과 같다.

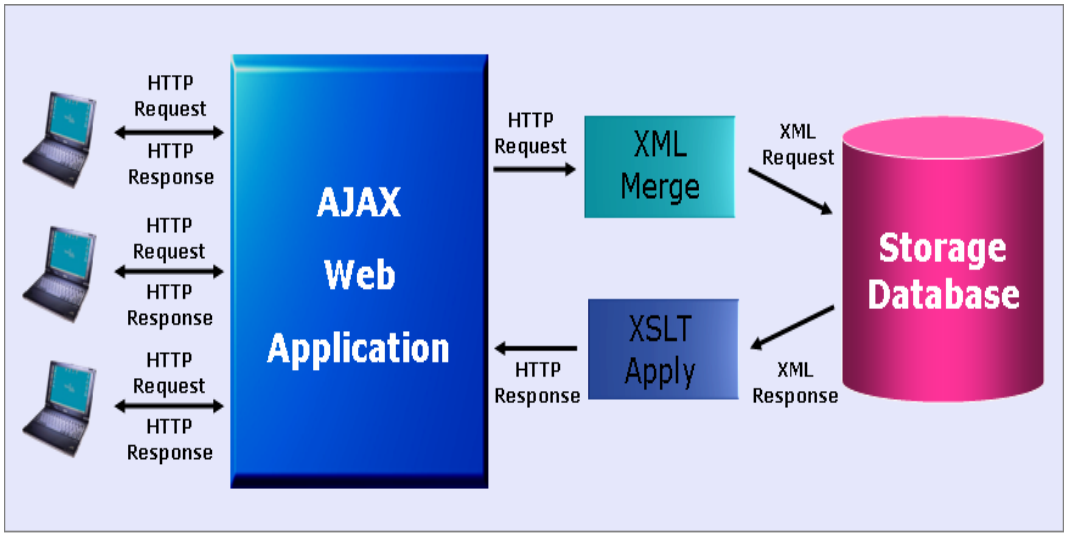



(그림 2) 문서통합 시스템의 구조도

### 3.2. 통합문서 등록

통합문서 시스템의 서비스를 이용하기 위해서는 통합문서 시스템 문서를 XML과 XSLT를 작성하여 통합문서 서버에 등록하여야 한다. 또한 하나의 XML 문서는 다수의 XSLT(문서의 템플릿)를 등록할 수 있다. 등록하는 과정은 SOAP과 HTTP 프로토콜을 이용하여 서비스에 등록하게 된다. 아래의 (그림 3)은 과정을 간략하게 그림으로 표현하였다.


(그림 3) 통합문서 서버에 통합문서 등록과정




(그림 4) 통합문서 시스템 구성도

### 3.3. 통합문서 이용

각 사용자는 일종의 AJAX 웹 어플리케이션에 접속하여 문서를 작성, 수정을 한다. 등록된 통합문서 시스템에 접속하여 작성, 수정할 수 있으며, 작성이 완료된 문서는 출력, 모든 문서 파일로 변환하여 준다. 즉 HWP, DOC, PPT, XLS 상에 존재하는 다양한 형태의 파일로 출력이 가능할 수 있다. 최근 이슈가 되고있는 AJAX 기법을 이용하여 사용자와 상호 대화적이기 문서의 추가, 변경이용이하다.

아래의 (그림4)는 통합문서를 이용하는 과정을 간략하게 그림으로 표현하였다.

## 4. 결론 및 전망

기존의 문서 작업은 여러 문서를 다수의 사용자가 협업 형태로 하나의 산출물을 도출한다. 이럴 경우 각 사용자가 작성한 문서는 일관성이 없고 새로운 문서를 만들 때마다 마찬가지이다. 이런 일관성이 없는 문서는 최종 통합 책임자가 별도로 일정한 문서 형식에 맞게 수정 및 보완작업을 거쳐 하나의 문서로 통합해야 하는 비효율적인 과정을 거치게 된다.

각각의 사용자가 자유롭게 만든 문서는 본 통합시스템을 통해 일관된 형식으로 자동 변환 및 보관되어 부가적으로 발생하는 문서의 통합과 관련된 비용 및 시간을 절감할 수 있다. 그리고 각 문서의 일관된 형식을 사용함으로써 키워드 도출이 쉬워지므로 효율적인 문서 검색이 가능해진다.

본 통합시스템은 문서 작성을 위하여 사용자가 개별적으로 대기하지 않고 아무 때나 작성할 수 있으며, 기업 인트라넷에서 업무보고 시스템을 획기적으로 변화할 수 있다.

## Acknowledgement

"이 논문은 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(원광대학교, 헬스케어기술개발사업단)."

## 참 고 문 헌

[1] Extensible Markup Language(XML), http://www.w3.org/XML/, 2006.
[2] 한성국, XML 워크샵, 2004.
[3] Eric van der Vlist, "Using W3C XML Schema“, http://www.xml.com, 2000.
[4] Norman Walsh, "Understanding XML Schemas", http://www.xml.com, 2000.
[5] Jesse James Garrett, "Ajax: A New Approach to Web Applications", http://www.adaptivepath.com/publications/essays/archives/000385.php.
[6] David Heller, Ajax for Designers, 2005

유 재 규
2002~현재 원광대학교 전기전자 및 정보공학부
관심분야: 시맨틱 웹서비스, 온톨로지 공학

박 재 훈
2004 원광대학교 컴퓨터공학과 (공학사)
2005~현재 원광대학교 컴퓨터공학과 (석사과정)
관심분야: 웹서비스, 온톨로지 공학

전 양 승
2001 원광대학 컴퓨터공학과 (공학사)
2006 원광대학교 컴퓨터공학과 (석사과정)
2006~현재 원광대학교 컴퓨터공학과(박사과정)
관심분야: 시맨틱 웹서비스, 온톨로지 공학, 지능형 e-Business

정 영 식
1993 고려대학교 전산학(박사)
1993~현재 원광대학교 컴퓨터공학부 교수
1997 미시간 주립대학교 전산학과 객원교수
2004 웨인 주립대학교 컴퓨터공학과 객원교수
관심분야: 그리드컴퓨팅, LBS, 분산병렬처리

한 성 국
1979 인하대학교 전자공학과 (공학박사)
1984~현재 원광대학교 컴퓨터공학부 교수
1989 University of Pennsylvania 방문교수
2003~2004 University of Innsbruck와 DERI 연구교수
2004~현재 대한전자공학회 컴퓨터소사이어티 감사
2005~현재 한국정보과학회 호남・제주지부장
관심분야: 시맨틱 웹서비스, 온톨로지 공학, 웹서비스, 의료정보, e-Learning

# 프로그래밍언어연구회 회칙

제 1 조 (명칭) 본 회는 한국정보과학회 프로그래밍언어연구회라 칭한다.

제 2 조 (목적) 본 회는 회원 상호간의 학술 활동과 친목을 목적으로 한다.

제 3 조 (회원의 자격) 본 회의 회원은 한국정보과학회 회원으로서 프로그래밍 언어 분야에 관심이 있는 자로 한다.

제 4 조 (회원의 의무) 회원은 소정의 회비를 납부하고 본 회의 사업에 적극 참여할 의무를 갖는다.

제 5 조 (기구) 본 회는 총회와 운영위원회를 둔다.

제 6 조 (총회) 총회는 본 회의 최고 의결기관이다. 총회는 정기총회와 임시총회로 구분하며, 정기총회는 매년 1회 소집하는 것을 원칙으로 하고 임시총회는 운영위원장(이하 "위원장"이라 한다)이 소집할 수 있다. 총회의 의장은 위원장이 겸한다.

제 7 조 (고문) 본 회의 고문을 추대할 수 있다. 고문의 자격, 대상, 인원수, 혜택 등은 운영위원회에서 결정하고 고문의 추대는 총회의 의결을 거친다. 고문은 본 회 운영에 관하여 위원장의 자문에 응하여야 한다.

제 8 조 (운영위원회) 본 회의 회무를 수행하기 위하여 다음과 같이 운영위원회를 둔다.

1. 위원장: 1인
2. 부위원장: 1인
3. 편집위원장: 1인
4. 운영위원: 편집위원(4인 이내)을 포함하여 15인 이내
5. 감사: 2인 이내

제 9 조 (운영위원의 직무) 위원장은 본 회를 대표하며, 회무를 통괄한다. 부위원장은 위원장을 보좌하여 위원장 유고시 그 직무를 대행한다. 편집위원장은 편집위원회를 구성하여 본 회의 발행지 편집에 관한 직무를 관할한다. 운영위원은 총무, 재무, 학술사업, 편집 등의 회무를 담당한다. 감사는 본 회의 업무 및 회계를 감사한다.

제10조 (운영위원의 선출) 위원장은 정기총회 출석회원의 과반수의 득표로 선출한다. 부위원장, 편집위원장 및 운영위원은 위원장이 임명하되 편집위원은 편집위원장의 추천을 받아 위원장이 임명한다. 감사는 정기총회에서 전임 및 전전임 위원장으로 선임한다.

제11조 (임기) 위원장의 임기는 2년으로 한다. 단 1회에 한하여 연임할 수 있다.

제12조 (재정) 본 회의 재정은 회원의 회비, 찬조금, 기타 수입금으로 충당한다.

제13조 (예산과 결산) 위원장은 매년 정기총회에서 예산과 결산의 승인을 얻어야 한다.

제14조 (기타) 기타 사항은 한국정보과학회 회칙에 따른다.

부 칙

1. (효력발생) 본 회의 규정은 2005년 11월 19일부터 효력을 발생한다.

# 프로그래밍언어연구회 논문지 투고 규정

제 1 조 연구회지에 게재할 원고의 종류는 프로그래밍 언어에 관련한 논문(정규논문, 초청논문, 학술대회논문의 수정 및 증보판 등), 학술강좌(기술해설, 기술소개, 기술보고, 튜토리얼 등), 특별기고(서평, 학술대회 참관기, 연구기관 방문기 등) 및 기타 편집위원회가 인정하는 것으로 한다.

제 2 조 투고자는 원칙적으로 본회 회원에게 한한다. 단 회원과의 공동기고자 및 초청기고자는 예외로 한다.

제 3 조 국내외 타학술지에 게재되었던 원고는 원칙적으로 투고할 수 없다. 단 편집위원회가 인정하는 경우는 예외로 한다.

제 4 조 투고된 원고는 다음과 같은 조건을 구비하여야 한다. 이 조건을 갖추었는지의 여부는 편집위원회에서 결정한다.

제 5 조 원고 접수는 전자 우편을 이용하여 수시로 하며 접수일은 본 연구회지 편집위원들 중 일인 이상에게 도착한 날로 한다.

제 6 조 원고는 원칙적으로 한글로 쓰되 가능한 한 널리 쓰이는 한글 워드프로세서를 시용하여 A4용지에 한 줄 건너서(double sapcing) 편집하고, 그림 및 표를 포함하여 가급적 30면 이내로 한다.

제 7 조 연구 내용에 직접 관련이 있는 문헌에 대해서는 이들 문헌에 관련이 있는 본문 중에 참고 문헌 번호를 쓰고 그 문헌을 참고문헌란에 반드시 기술한다.

참고문헌은 학술지의 경우는 저자, 표제, 학술지명, 권, 호, 면수, 발행년의 순서로, 단행본은 저자, 서명, 면수, 발행소, 발행년의 순서로 기술한다.

제 8 조 본문의 경우 한글과 영문으로 작성된 제목, 저자성명, 초록을 포함해야 한다.

제 9 조 논문의 경우 편집위원회가 2인의 심사위원을 선정하여 심사를 거쳐 게재 여부를 결정하며 필요하면 수정을 요구할 수 있다.

제10조 본 규정은 2000년 9월 1일부터 효력을 발생한다.

# 프로그래밍언어연구회 소식 및 기사투고 안내

프로그래밍언어연구회에서는 회원 여러분의 유익한 소식 및 기사들을 기다리고 있습니다. 프로그래밍 언어 이론 및 응용 연구 결과, 논문, 특별 기 고(본인의 견해, 전망, 분야별 개관(survey), 특정 분야의 소개, 제품 소개), 국내 기사(회원 동정, 회의 결과, 제품 개발 안내), 해외 기사, 연구기관 소개, 책 리뷰, 특별 모임 안내, 세미나 안내 등에 대한 기사를 보내주시기 바랍니다. 아직 제대로 운영되고 있지는 못하지만, 회원들간의 소식을 전달하는 매체로서 연구회지와 연구회 web 페이지를 병행할 계획을 하고 있습니다. 원고를 web을 통하여 접수하고, 회원들에게 필요한 web을 이용하여 수시로 제공하는 체제를 만들려고 합니다. 이러한 것이 가능하기 전까지는 기존의 방법대로 원고를 접수하겠습니다. 아래 편집위원 중 한 분에게 원고를 보내주시기 바랍니다.

- 한국항공대학교 안준선 : jsahn@hau.ac.kr (Tel. 02-300-0144)
- 한양대학교 이욱세 : oukseh@hanyang.ac.kr (Tel. 031-400-5234)
- 동덕여자대학교 이은영 : elee@dongduk.ac.kr (Tel. 02-940-4588)

# 프로그래밍언어연구회 연회비 안내

연회비는 프로그래밍언어논문지 발행 및 운영에 필요한 경비입니다. 본 회의 회원께서는 다음과 같이 납부하여 주시기 바랍니다.

- 연회비 : 일반회원 10,000원/년
  학생회원 5,000원/년
- 납부처 : 조흥은행 313-03-002919 (예금주 : 사단법인 한국정보과학회)
- 문  의 : 서울대학교 이재진 : jlee@cse.snu.ac.kr (Tel. 02-880-1863)

# 프로그래밍언어연구회 가입 안내

입회 원서를 작성하신 후 입회비(일반 : 10,000원, 학생 : 5,000원)를 다음의 방법으로 지불하고 입회원서 및 회비 납부 사본을 같이 본 연구회 총무에게 보내주시기 바랍니다. 특정 단체의 회원 가입에 대해서도 총무에게 문의하시길 바랍니다.

숙명여자대학교 창병모 : chang@sookmyung.ac.kr (Tel. 02-710-9378)

| 입 회 원 서 | | |
|---|---|---|
| 성 명 | 한 글 | |
| | 영 문 | |
| 연 락 주 소 | | |
| 근 무 처 | 기 관 | |
| | 주 소 | |
| 정보과학회 회원입니까? | | 예( ) 아니오( ) |
| 관 심 분 야 | | |

본인은 프로그래밍언어연구회의 취지에 회원으로 가입하고자 합니다.

20 년 월 일

성명 : 인

| 변경사항 통지서 | | |
|---|---|---|
| 성 명 | 한 글 | |
| | 영 문 | |
| 변경사항 | (주소, 근무처, 전화번호) | |