# Progress Report

Research On Program Analysis System
National Creative Research Initiative Center
ropas.kaist.ac.kr
Korea Advanced Institute of Science and Technology
Director: Kwangkeun Yi

July 20, 2001

Creative Research Initiative Program
Korea Ministry of Science and Technology

# 1 Summary of Research Results

## 1.1 Research Objectives v.s. Accomplishments

### 1.1.1 Objectives of the originally proposed research

Our goal is to achieve compiler technologies suited for the global, mobile computing environment of the future. In particular, we will focus on the following three compilation problems for higher-order & typed programming languages like ML:

- compiler must generate **safe code**: not only must the compiler assure that the compiled code will not damage the host but the host must be able to verify the established safety of the incoming code.
- compiler must generate **small code**: the code size must be as small as possible, in order to minimize the delivery cost over the network. Compact code will move swiftly over the network, arriving at the host faster than other competing code.
- compiler must generate **smart code**: the code must be able to tailor itself to the most common inputs that occur during its use at the host.

Our research position is to aggressively adopt recent progress in programming language theories into a set of practical compilation techniques. The major thrust for promoting the potential synergy between the language theories and compilation practices comes from our focus on semantic-based static analysis. Static analysis is the technique of estimating the input program's run-time properties before execution. By "semantic-based," we mean that the analysis has to be based on programming language's semantic foundations (e.g. various formalisms of semantics specifications, type theories, computational logics and models, etc.).

The objectives of the first three years (phase-I) were two-fold:

- **To build our software experiment infrastructure**
  - to build an ML compiler system as a realistic workbench of our program analysis technologies
  - to develop an automatic program analyzer generator
- **To research on safe and smart code system**
  - on assuring the type and resource safety of compiled code
  - on modular static analysis techniques
  - on techniques for run-time specializations of static analyses

The compiler system will embody our research results (various static analyses for generating safe/small/smart code). The compiler must be designed such that the devised program analyses can be easily integrated to experiment with real programs.

The compiler system is for an ML-like language that has both a sound theoretical basis and a practical implmentation technique.

The static analyzer generator automates the implementation of cost-effective static analyses, and its specification language and analysis-generation engine will be key-components to check the safety of compiled mobile code.

### 1.1.2 Quantitative summary of the results

**Software:**

- A higher-order and typed programming language system nML (`ropas.kaist.ac.kr/n`)

  We have defined the nML programming language (syntax, type system, dynamic semantics) that will be the language of our experiments throughout this project. We have developed its compiler system (version 0.91) and its programming tools (lexical analyzer generator, parser generator, etc.). We have demonstrated the practicality of both the language and its compiler by implementing the compiler itself in nML and by developing some realistic applications (internet telephony system, XML parser, etc.).

- Program analyzer generator, System Zoo (`ropas.kaist.ac.kr/zoo`)

  We have defined the specification language Rabbit (syntax, type system, dynamic semantics), the system's overall architecture, and have successfully tested the specification language in expressing some existing static analyses.

Details are in Section 1.2.2 as well as in the web pages.

**Publication:** (`ropas.kaist.ac.kr/papers`)

- international journal papers:

  6 published or accepted [70, 25, 29, 71, 19, 53],

  2 invited but not published yet [57, 40],

  4 submitted and in review [28, 44, 50, 49].

- refereed international conference/workshop papers:

  9 published [56, 39, 23, 2, 4, 3, 65, 43, 47],

  2 accepted but not published yet [52, 45].

**Progress:**

| Objectives | Progress |
|---|---|
| to develop a compiler system as a realistic research workbench | 90% |
| to develop an automatic program analyzer generator | 80% |
| to research on static resource-bound checking of programs | 80% |
| to research on modular static analysis techniques | 80% |
| to research on techniques for run-time specializations of static analyses | 70% |

### 1.1.3 Self Evaluation

For the last three years (the first phase of our project), 50% of our effort has been on developing our software experiment infrastructure, and 50% on research and publishing our research results.

Implementing software infrastructure was a must for the first phase of our research, because our position for achieving competitive research is to test our analysis ideas against realistic program sets inside a practical compiler system whose target language (nML) has a sound theoretical basis.

We believe that having our own experiment software workbench (the nML language, its compiler system, and program analyzer generator Zoo) will eventually accelerate and realistically materialize the forthcoming research results.

**Building our software experiment infrastructure**

- **The nML compiler system** (`ropas.kaist.ac.kr/n`):

  We have designed our own higher-order & typed programming language nML. The nML is a Korean dialect of Standard ML [32] and Objective Caml [30], which has adopted many successes of the programming language researches for the past 30 years. The language is aimed as a complementary substitute for conventional systems-programming language (namely, C) technology that has been stagnant for the last 30 years. nML is rigorously defined [41] (see Section B) with its variant of ML's sound let-polymorphic type system [32, 31].

  The nML [41] compiler system is a serious, realistic programming system that will embody our solutions for generating safe/small/smart code. This compiler system has been successfully tested by ourselves in developing the nML compiler itself, in writing some realistic nML applications such as internet telephony system, and in implementing class projects by more than 200 KAIST undergraduate students during a programming language course for the last two years.

  We recently organized NUG(nML User Group) who volunteered to write various applications in nML: from implementing an embedded operating system to writing nML primer manuals in Korean.

- **Program analyzer generator, System Zoo** (`ropas.kaist.ac.kr/zoo`):

By *System Zoo* one can quickly produce an executable program analyzers. Its user writes an analysis specification then the tool emits an nML program for the specified analysis. System Zoo is analogous to the parser generator tool *yacc* [20], but it generates semantic program analyzers, not syntax checkers.

System Zoo is not designed as an isolated system that generates just program analyzers. The nML programmers will use its specification language to annotate programs with their safety properties. The nML compiler will execute the Zoo's analysis-generation engine in order to verify the annotations.

When completed, which we expect to do by the next year, the System Zoo together with our nML compiler will give us a leverage for experimenting our static analyses, which are devised to generate safe/small/smart code from higher-order and typed, nML programs.

**Research towards safe and smart code system**

- **On type inference algorithms**: continuing from the work [25] on top-down type inference algorithm, we have generalized the ML-style let-polymorphic type inference algorithm so that it can be easily tuned for generating meaningful type-error messages. One journal paper [28, 26] on this algorithm has been submitted. This algorithm has been integrated into our nML compiler system, in order to provide a convenient programming environment for checking program's type-safety.

- **On continuation-passing-style(CPS) transformation**: CPS transformation is an important program transformation that facilitates the compilation of higher-order languages. Continuing from [23], we investigated whether it is possible to CPS-transform only a sub-part of a program, leaving others in direct-style. Such partial CPS-transformation is needed to link CPS-transformed modules with non-CPS modules, which will occur in mobile computing environment.

  We have found [56] that such partial transformation is possible based on types. Its full journal version was invited to the *Journal of Higher Order and Symbolic Computation*.

- **On theoretical study of ML-style exception-handling**: We have presented [39] a theoretical justification for using the stack mechanism to implement ML-style exceptions. We have demonstrated that the Logical Framework [38, 17] is a convenient conceptual tool in the theoretical study of programming languages. This work's full version was also invited to the *Journal of Higher Order and Symbolic Computation*.

- **On resource safety analysis**: We have been working on static memory management techniques for two languages: nML and a low-level intermediate language of our nML compiler. For nML programs, we have been investigating a combination of static analysis and dynamic hints in order to annotate programs so that either each memory object can be recycled as early as possible,

or the garbage collection overhead can be reduced. As well as inferring such annotations, we are designing a separate static system that checks the safety of such annotations. For a low-level intermediate language (first-order machine-oriented, imperative language), we have been working on static method for estimating memory consumption (resource-bound checking).

These static analyses will be implemented by our System Zoo and will be tested/integrated inside our nML compiler system. It is too early to judge the efficiency of the two techniques yet.

- **On modular static analysis**: Modular program analysis does not need the entire program text as the analysis input; it analyzes separated program sources such as modules, and later links the partial analysis results when all the modules are available. This technique is necessary for global computing environment where code fragments are linked on demand.

  We have reported [29, 27] a method for proving the correctness of a modularized version of a whole program analysis. The study is on control-flow analysis, which is a basis of almost all analyses for higher-order languages like nML.

- **On run-time specializations of static analyses**: In order to generate code that tailors to its most common inputs, we have proposed [15] a static-analysis-based technique: transforming a static analysis into one whose result can tailor to the programs' inputs at run-time. This technique's noble feature is that no profile (or trace) is collected and no probing code inside the running program is needed. This idea will be implemented in the nML compiler by a help from System Zoo.

- **On propositional program logics**: This research has focused on developing logical tools for better program analysis. We have presented on decidability [43, 44, 45] of propositional program logics, on power [52, 48, 49] of propositional program logics, and on reuse and validation [46, 51, 50] of model checkers.

- **On educating formal methods**:

  We also attempted some research on better education about foundations of formal methods. We have written one paper [71] that will appear in *Communications of the ACM*, which is the membership magazine of the oldest and biggest computer science society. This was a by-product of our research, but we think it a valuable by-product.

- **On program analysis of Java**:

  Our exception analysis for ML programs [70, 63, 69, 62, 68] has been successfully applied to Java [2, 3, 65, 66]. This work has helped us to popularize our exception analysis technology, because the Java's community is considerably larger than ML's.

**International Competitive Edges**

We have kept our research works' competitive edges by exchanging visitors and seminars with other leading research groups. All these activities are recorded on the web for later reference: `ropas.kaist.ac.kr/seminar`.

- **Research seminars**: we hosted 28 external visitors, each of them gave at least one technical in-depth talk or tutorial. Visitors were from Aarhus Univ.(Denmark), Carnegie-Mellon Univ.(USA), Connected Components Corp.(USA), École Normale Supérieure(France), Fredrick-Schiller Univ.(Germany), IBM T.J.Watson Research Center(USA), Kyoto Univ.(Japan), Lucent Technologies(USA), Microsoft Research(USA), National Singapore Univ.(Singapore), Oregon Graduate Institute(USA), Russia Institute of Informatics Systems(Russia), Stanford Univ.(USA), UC Berkeley(USA), Univ. of Glasgow(UK), Univ. of Illinois at Urbana-Champaign(USA), Univ. of Melbourne(Australia), and Univ. of Verona (Italy).

- **Tutorials**

  - Twelf and Linear Logical Framework

    *Instructor:* Jeff Polakow (School of Computer Science, Carnegie Mellon Univ.)

    *Benefit:* After this tutorial we started working on formal proofs that nML exceptions may be implemented using stack. The proofs were simplified by using the ordered linear logical framework (Jeff Polakow's Ph.D. thesis). This work is a sequel to [14, 23], was presented [39], and its full journal version paper was invited [40].

  - Program Logics

    *Instructor:* Nikolay Shilov (Ershov Institute Of Informatics Systems (IIS), Siberian Branch of Russian Academy of Science, Novosibirsk, Russia)

    *Benefit:* This tutorial extended our static analysis techniques to the modal-logic-based software verifications. For example, our System Zoo supports CTL(Computational Tree Logic) as the specification language for expressing queries on the program analysis results. Standard model-checking procedure is adapted to check the validity of such queries (program property descriptions).

- **Survey Workshops**

  We have had two survey workshops: one about resource bound checking (techniques for estimating the upper bound of the needed computing resources such as CPU cycles, memory, and network bandwidth), and the other about typeful compilation (techniques to interact between type systems and the compilation process). Both workshops were internal to ROPAS members, each of which lasted for 11 and 12 days, respectively.

  *Benefit:* We have built a collective common knowledge in these areas and identified research directions that are relevant to our research goals. By the second

workshop, we became acquainted with problems that we have to expect when we implement typeful intermediate transformation phases of our nML compiler system.

**Industrialization Efforts**

Even though our project is just in half way toward the final goal, our current nML programming system and its safe typeful programming technologies already attracted industry's attention. We have started working with two industry companies. These activities are small at the moment, but we expect it to grow when our research matures further.

- LG Telecom (`www.lg019.co.kr/english`): LG Telecom is one of the three mobile telecommunication providers in Korea. LG Telecom is eager to have a safe and smart software execution platform for their cellular phones. We exchanged our mutual interest and arranged that LG Telecom provide us with their CDMA cellular phones and their related systems knowledge on which we can install a miniaturized version of our nML compiler system.

- Medison Inc. (`www.medison.com`): Medison is a leading company for medical ultra-sound scanning systems. Our nML programming language system is currently being tried by Medison's software engineers to implement some of their embedded softwares. Even one feature of our nML compiler (the proven-sound type-inference phase) is expected to improve their softwares' quality. In the future, our System Zoo will help them to detect more sophisticated bugs than just type-errors.

## 1.2 Achievements

### 1.2.1 Publications (`ropas.kaist.ac.kr/papers`)

Papers that are published or accepted by SCI journals/series during July 1998 till July 2001. The SCI impact factors of the journals/series are:[1]

| | |
|---|---|
| *ACM Transactions on Programming Languages and Systems* | 1.162 |
| *Communications of the ACM* | 1.779 |
| *Information Processing Letters* | 0.242 |
| *Lecture Notes in Computer Science* | 0.872 |
| *Science of Computer Programming* | 0.63 |
| *Theoretical Computer Science* | 0.41 |

- "A Cost-Effective Estimation of Uncaught Exceptions in Standard ML Programs", Kwangkeun Yi and Sukyoung Ryu, *Theoretical Computer Science*, Vol.273, 2001 (to appear)

[1] `library.kaist.ac.kr/Ko/main/sci.html`

**Note:** In computer science and in the programming language area in particular, it is rare that prestigious journals and series have the "SCI impact factor" larger than 1.00. Moreover, publishing/presenting papers in top conferences or workshops is valued at least as much as in journals.

**abstract:** We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML(SML) programs. This analysis will predict abrupt termination of SML programs, which is SML's only "safety hole."

Our implementation of this analysis has been applied to realistic SML programs and shows a promising cost-accuracy performance. For the ML-Lex program, for example, the analysis takes 1.36 seconds and it reports 3 potentially-uncaught exceptions, which are exactly the exceptions that can really escape.

- "A Proof Method for the Correctness of Modularized 0CFA", Oukseh Lee and Kwangkeun Yi, *Information Processing Letters*, (to appear)

  **abstract:** We reports two findings: 1) deriving a modular version from a whole-program monovariant (or context-insensitive) CFA makes the resulting analysis polyvariant (or context- sensitive) at the module-level, 2) a convenient stepping-stone to prove the correctness of a modularized version (instead of proving it against the program semantics) is a whole-program CFA that is polyvariant at the module-level.

- "Proofs about a Folklore Let-polymorphic Type Inference Algorithm", Oukseh Lee and Kwangkeun Yi, *ACM Transactions on Programming Languages and Systems*, Vol.20, No.4, pp.707-723, 1998

  **abstract:** The Hindley/Milner let-polymorphic type inference system has two different algorithms: one is the *de facto* standard Algorithm $\mathcal{W}$ that is bottom-up (or context-insensitive), and the other is a "folklore" algorithm that is top-down (or context-sensitive). In this article, we formally define the context-sensitive, top-down type inference algorithm (named "$\mathcal{M}$"), prove its soundness and completeness, and show a distinguishing property that $\mathcal{M}$ always stops earlier than $\mathcal{W}$ if the input program is ill typed. Our proofs can be seen as theoretical justifications for various type-checking strategies being used in practice.

- "Puzzles for Learning Model Checking, Model Checking for Programming Puzzles, Puzzles for Testing Model Checkers", Nikolay Shilov and Kwangkeun Yi, *Electronic Notes in Theoretical Computer Science*, Volume 43, 2001

  `www.elsevier.nl/gej-ng/31/29/23/72/23/show/Products/notes`

  **abstract:** We study issues related to model checking of the $\mu$-Calculus in finite models while a focus is on: importance of games for validation of model checkers, utility of model checking and games for solving puzzles, and educational aspects of model checking and games.

- "Proving Syntactic Properties of Exceptions in an Ordered Logical Framework", Jeff Polakow and Kwangkeun Yi, *Lecture Notes in Computer Science*, Vol.2024, pp. 61-77, Proceedings of the 5th International Symposium on Functional and Logic Programming, March, 2001

  **abstract:** We formally prove the stackability and linearity of exception handlers of ML-style semantics using a novel proof technique via an ordered logical framework (OLF). Our work can be seen as two-fold: we present a theoretical justification of

using the stack mechanism to implement exceptions of ML-like semantics; and we demonstrate the value of an ordered logical framework as a conceptual tool in the theoretical study of programming languages.

- "Engaging Students with Theory through ACM Collegiate Programming Contests", Kwangkeun Yi and Nikolay Shilov, *Communications of the ACM.* (to appear)

  **abstract:** The role of formal methods in the development of computer hardware and software increases since systems become more complex and require more efforts for their specification, design, implementation and verification. Simultaneously formal methods become more and more complicated since they have to capture real properties of real systems for sound reasoning.

  In this article we try to attract the attention of people from academia and industry to a popular presentation of theoretical research for better computer science education. Simultaneously we would like to present some experience on popularizing program logics via training undergraduates for ACM Collegiate Programming Contests.

- "An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs", Kwangkeun Yi, *Science of Computer Programming*, Vol. 31, pp. 147-173, 1998

  **abstract:** We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML (SML) programs. This analysis enhances the software safety by predicting, *prior to* the program execution, the abnormal termination caused by unhandled exceptions. Our analysis is limited to SML programs that are type-correct and are operationally invariant even if the generative nature of SML's data-type and exception declarations is not considered.

### 1.2.2 Software

- The Definition of nML [41]

  nML is our own higher-order and typed programming language which will be "the" language of our software results. The definition of its syntax, static semantics(type system), and dynamic semantics are shown in Appendix B or available at `ropas.kaist.ac.kr/n/doc/n-e.pdf`.

- nML Compiler System version 0.91 (`ropas.kaist.ac.kr/n`)

  - The nML compiler in nML: we have been developing the nML compiler in nML. For this bootstrapping to be possible, our nML compiler is implemented on top of the OCaml compiler [30]. We defined a translator from nML into OCaml and use the OCaml compiler to generate executable code. The compiler runs on Unix, Linux, MS Windows 98/2000/ME/NT.
  - We have completed the compiler's first four phases: lexical analysis, parsing, type inferencing, type checking, and pattern compilation. Remaining compilation phases, which we currently use OCaml's, will be completed in nML by the next year.

  - The nML compiler system has been used by KAIST's programming languages class (CS320, Spring 2000 and 2001). Total of 95 (Spring 2000) + 122 (Spring 2001) sophomore and junior computer science students programmed their homeworks in nML. Homeworks were implementing interpreters for languages that range from C-like imperative languages to ML-like higher-order applicative language. This extensive test of our nML compiler system has been quite successful.

    (`ropas.kaist.ac.kr/~kwang/320/01`)

- nML programming tools and applications:
  - `nlex`: tool to generate an nML program that implements the specified lexical analyzer.
  - `nyacc`: tool to generate an nML program that implements the specified context-free-grammar parsing.
  - XML parser in nML: XML is an extensible mark-up language that was used in web-based documents. This parser will be used later for applications handling WWW databases.
  - `netPhone`: an internet telephony system.

    (`ropas.kaist.ac.kr/~bskim/netphone`)

    It supports both one-to-one connection and many-to-many connection (for conference calls). All the voice data managements, quality of service, and graphic user interface has been implemented in nML. Its normal operation is possible if the network bandwidth is at least 150kbps.

    This application in nML showed that the nML language can be a serious language to write non-trivial systems applications.
  - `cloneChecker`: software similarity checker.

    (`ropas.kaist.ac.kr/~abyss/clonecheckr`)

    This program quantifies the similarity of nML programs and groups them into "cliques." This software has been used in KAIST undergraduate programming language classes for checking pirate copies of homeworks. We are currently building its sister softwares for checking C, Java, and Scheme programs.
  - `vmc`: model-checker validating system.

    (`ropas.kaist.ac.kr/~poisson/vmc`)

    This system consists of several modules against which existing model checkers can be tested on automatically generated test suites.

- Program analyzer generator, System Zoo (`ropas.kaist.ac.kr/zoo`)

  System Zoo has been designed as an evolutionary descendent of System Z1 [67] and Z2 [24]. It supports an ensemble of four frameworks of static analysis: abstract interpretation [8, 7, 9, 10, 11], data-flow analysis [22, 21, 42], constraint-based analysis [1, 18], and model checking [5]. The first three frameworks are

variations in presentation among which the user chooses one to conveniently specify how to set-up static equations from the input programs. Zoo system transforms this specification into a combination of a data-flow equation extractor and a specialized fixpoint iterator (equation solver). The last framework, model checking is used to specify queries on the equation solutions. The user specifies static properties as computational-tree logic (CTL) formula. Zoo system generates a model checking procedure that checks this formula against the equation solutions.

We have defined the specification language called Rabbit [64] (see Appendix C or `ropas.kaist.ac.kr/zoo/doc/rabbit-e.pdf`). The expressive power of the Rabbit language is being investigated by defining existing static analyses in it. So far, our exception analysis [70, 63, 69] and other standard analyses have been successfully defined.

### 1.2.3 Patents

N/A

## 1.3 International Reputation of Principal Investigator

**Seminars and Invited Talks**

- 2 months among September 2001 - August 2002: visiting professorship, Computer Science Department, École Normale Supérieure (Paris, France) (host: Patrick Cousot)
- 1 month in 2002: visiting and seminar invitation from Computing Laboratory, Oxford University (host: Luke Ong)
- spring 2002, seminar invitation from Basic Reserch In Computer Science, Aarhus University, Denmark (host: Olivier Danvy)
- July 13, 2001: seminar, "System Zoo: towards a realistic program analyzer generator", Computer Science Department, École Normale Supérieure, Paris (host: prof. Patrick Cousot)
- March 17-21, 2000: two invited seminars, Department of Information Science, University of Tokyo (host: Naoki Kobayashi)
- March 17-19, 1999: invited talk, "Static Value Slicing", The First Japanese Programming and Programming Language Workshop, Japan
- March 15-16, 1999: invited seminar, "Static Analysis for Code Compaction and Safety Assurance ", Research Institute of Mathematical Sciences, Kyoto University (host: Atsushi Ohori)
- July 30 - August 25, 1998: consulting scientist, Software Principles Research Department, Bell Laboratories, Murray Hill (host: David MacQueen)

**Program Committee Member**

- SAS 2001: The 8th International Static Analysis Symposium 2001 (`www.ens.fr/sas01`)
- ICFP 2001: ACM International Conference on Functional Programming 2001 (`cristal.inria.fr/ICFP2001`)
- FLOPS 2002: The 6th International Symposium on Functional and Logic Programming 2002
- Program Chair: The 2nd Asian Workshop on Programming Languages and Systems, December 2001, Korea

Both SAS and ICFP are considered the top conferences in the areas of static analysis and functional programming, respectively. Every year the paper acceptance rate is about 20% - 30%. For FLOPS, it is about 50%. The committee members consists of about 12 members.

**Advise Visiting Students**

- Andrzej Murawski, 7/6/2001 - 8/5/2001, Ph.D. student, Oxford Univ.
- Jeff Sarnat, 6/4/2001 - 8/19/2001, senior undergraduate, CMU
- Jeff Polakow, 6/20/2000 - 8/20/2000, Ph.D. student, CMU
- Fermin Reig, 7/21/2000 - 10/20/2000, Ph.D. student, Univ. of Glasgow
- Charles Hymans, 8/1/1999 - 1/30/2000, Ph.D. student, École Normale Supérieure

**Send Visiting Ph.D. Students**

- 2 students (Oukseh Lee and Sukyoung Ryu) to Yale University (host: Zhong Shao), 7/27/1999 - 8/26/1999
- 1 student (Sunae Seo) to École Polytechnique (host: Radhia Cousot), 4 months from October 2001

**Host Research Visitors and Their Seminars**

Hosted 28 seminars: 4 by domestic researchers, 24 by international researchers. They are all very active world-class researchers in static analysis, compilers, and programming languages. (`ropas.kaist.ac.kr/seminar`)

**Paper Invitations from International Journals**

Our papers presented in international conferences and workshops have been invited for publication in journals:

- 1 paper [70] from *Theoretical Computer Science*
- 1 paper [53, 47] from *Electronic Notes of Theoretical Computer Science*
- 2 papers [39, 40] and [56, 57] from *Journal of Higher-Order and Symbolic Computation*

**Domestic Award**

The principal investigator, with his Ph.D. student, was awarded the Gaheon Academic Achievement Award 2001 from the Korea Information Science Society. This annual award is in recognition of the recipient's academic excellence.

# 2 Summary of Future Research Plans

## 2.1 Current Trends in Related Research Areas

Two camps of researches are the most related with our project: those on compilation with proofs (the proof-carrying code system (PCC) [37, 6, 35, 36] of CMU and UC Berkeley) and on compilation with types (elaborate type systems [34, 33, 60, 61, 13, 12, 34, 54, 59] of Cornell and CMU). Both approaches are similar in their overall architecture: the code producer (compiler) constructs a formal proof that the code respects a safety policy. The code consumer then uses a simple proof checker to check the proof that accompanies the code.

The two approaches differ in what representations they use for proofs and in how they check the proof's integrity. In early PCC [37] systems the proof was represented as a proof tree in Twelf [38, 17]. The proof checking is then a specialized cross-checking (by means of the Curry-Howard isomorphism [16]) of the code and its accompanying proof. In type-disciplined approaches, the proof corresponds to type expressions, and the proof checking to conventional type-checking of the type-annotated programs. Both approaches strike a slightly different balance between the size of the proofs and the complexity of the proof checker. PCC simplifies more the proof checker, while the type-based approach simplifies more the proof. The most recent PCC system [36] substantially reduced the proof size without selling much the proof-checking simplicity. The original idea of using the Curry-Howard Isomorphism (proof checking = type checking) has been given up.

Their current researches are two-fold: to find a better balance between the proof's complexity and checker's complexity, and to extend the property classes they prove and check. At the moment the safe property they can check is restricted to those in the first-order logic [37, 36] or those that can be solvable in linear constraints [60, 61].

Other than these researches on certifying compilers, there are works on building mobile-computing infrastructure. These works on next generation network architectures will lay the stepping-stones to the high-speed global network computing environment, which we assume in our project. The goal of UPenn's SwitchWare [55] project is to make the network nodes programmable so that the network service can become selectable on a per user(or even per packet) basis. This programmable network architecture would reduce the network evolution cycle(standardization $\rightarrow$ development $\rightarrow$ service deployment). Mathematically rigorous models of such switches are also being studied. The goal of MIT's Active Network [58] project is to allow the network users to inject customized programs into the switching nodes. This network architecture will permit a massive increase in the sophisticated computation performed within the network.

## 2.2 Research and Development Goals

During the second three years we will focus on

- completing the System Zoo
- completing the nML compiler system that embodies the three research results that have been accumulated for 6-year research:
  - nML compiler system will generate code whose memory-consumption properties can be statically confirmed.
  - nML compiler system will minimize the generated code
  - nML compiler system will generate code that tailors to their popular inputs

Such nML compiler system will stand on our idea of "fixpoint-carrying code." For each safety property (e.g. a resource safety or an invariant safety), there exists a static analysis to safely estimate the property of the input programs. Every static analysis consists of two phases: setting up data-flow equations from the input programs, and solving the equations. The compiler completes the two phases for the input programs then provides the compiled code together with the analysis solution. The code consumer, upon receiving the pair of the compiled code and a safety solution, confirms the code's safety simply by setting-up equations again (by the first phase of the same static analysis) then checking whether the accompanied solution is really a solution of the equations. This technique is named "fixpoint-carrying" because the solution of data-flow equations is a fixpoint of the monotonic function that corresponds to the equations.

Our System Zoo will play the key-role here: the safety policy, to be shared between the code provider and code consuer, is an analysis definition written in Zoo's specification language Rabbit. Using Zoo we translate the analysis specification into executable code for setting-up equations and for computing the fixpoint iteration (checking).

There are two key sub-problems we have to solve:

- Fixpoint engineering: there exists a tradeoff between the fixpoint-size and the fixpoint checker's complexity. The fixpoint is always an element of a lattice. Exact representation of a lattice element can be traded for a more compact representation. Approximate but compact representation of fixpoint will complicate the fixpoint checking because the fixpoint iteration code from Zoo must be extended to safely manipulate "approximate" lattice elements.

  Depending on how we represent the fixpoints (lattice elements) a right trade-off can vary.

- How to compile fixpoints of source programs into those of corresponding low-level code.

  Translating the properties about a source program into the corresponding properties of the compiled target code is a challenging problem, which is unexplored.

Morrisett [34]'s work is only for the types: a translation from source (ML) types into target (assembly language) types. We have to investigate generalizing this technique for static properties in general, not just types.

## 2.3 Plan

By year 4: Completing System Zoo, and completing nML compiler's conventional back-end phases.

By year 5: Integrating the resource safety mechanism into the nML compiler.

By year 6: Integrating the code minimization and self-optimizing mechanism into the nML compiler.

Meanwhile, we will keep driving the nML programming system to the limit by developing realistic applications in nML (such as embedded operating systems, numerical computation packages, bio-computing applications, and etc.) and by working closely with industries.

## 2.4 Significance

In the future global computing environment where all the computers around the globe are connected via a light-speed network an extremely large number of code fragments will compete for providing the optimal functionality to the hosts in need. This is because code fragments that constitute a program are dispersed across the globe and will be brought to a host on demand.

In such a new computing environment, generating safe/small/smart code is a key technology for nurturing a world-class competitive software industry. Our research drives a potential synergy between the language theories and compilation practices by focusing on semantic-based static analysis. We emphasize the formal foundations of the source programming languages because it will enable us to clearly understand the fundamental structures of our problems and hence help us to avoid unsound and ad-hoc solutions.

# References


[1] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, January 1995.

[2] Byeong-Mo Chang, Jangwoo Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for java. In *The Proceedings of the 16th ACM Symposium on Applied Computing*, March 2001.

[3] Byeong-Mo Chang and Kwangkeun Yi. Constraint-based analysis for java. In *Proceedings of the SSGRR Conference on Computer and E-Business*, L'Aquila, Italy, August 2000 (invited presentation).

[4] Eunsun Cho and Kwangkeun Yi. Escape analysis for stack allocation in java. In *ECOOP'00 Workshop on Formal Techniques for Java Programs*, June 2000.

[5] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[6] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, and Mark Plesko. A certifying compiler for java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–107, June 2000.

[7] Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10. Prentice-Hall, 1981.

[8] 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 The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[10] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.

[11] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 170–181, 1995.

[12] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, September 1999.

[13] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, January 2000.

[14] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of cps programs. In Andrew Gordon and Andrew Pitts, editors, *Typed Lambda Calculus and Applications*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 147–163. Proceedings of the third international workshop on higher order operational techniques in semantics, hoot'98 edition, September 1999.

[15] Hyunjun Eo and Kwangkeun Yi. Preparing set-based analysis for run-time specialization. Technical Memorandum ROPAS-1999-1, Research on Program Analysis System, Notional Creative Research Center, Korea Advanced Institute of Science and Technology, November 1999.

[16] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[17] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[18] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[19] Jay Hoeflingr, Yunheung Paek, and Kwangkeun Yi. Unified interprocedural parallelism detection. *Journal of Parallel Programming*, (accepted, invited submission).

[20] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Supplementary Documents 1*. 1986.

[21] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithm. *Journal of the ACM*, 23(1):158–171, 1976.

[22] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[23] Jungtaek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ml exceptions by selective cps transformation. In *The Proceedings of the ACM SIGPLAN Workshop on ML*, pages 103–114, September 1998.

[24] Seong-Hoon Kim, Kwangkeun Yi, Hyun jun Eo, and Kwang-Moo Choe. SUIF program analysis using System Z2. In *Proceedings of the Second SUIF Compiler Workshop*, August 1997.

[25] Oukseh Lee and Kwangkeun Yi. Proofs about a Folklore Let-polymorphic Type Inference Algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[26] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Notional Creative Research Center, Korea Advanced Institute of Science and Technology, March 2000.

[27] Oukseh Lee and Kwangkeun Yi. Modularization of 0-cfa makes it polyvariant. Technical Memorandum ROPAS-2000-4, Research on Program Analysis System, Notional Creative Research Center, Korea Advanced Institute of Science and Technology, February 2000.

[28] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. *Higher-Order and Symbolic Computation*, (submitted).

[29] Oukseh Lee and Kwangkeun Yi. A proof method for the correctness of modularized 0cfa. *Information Processing Letters*, (to appear).

[30] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system (relase 3.00), documentation and user's manual. http://caml.inria.fr/ocaml/htmlman/index.html, 2000.

[31] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[33] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[34] Greg Morrisett, Dvaid Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[35] George Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[36] George Necula and S. Rahul. Oracle-based checking of untrusted software. In *popl*, pages 142–154, January 2001.

[37] George C. Necula. Proof-Carrying Code. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[38] Frank Pfenning, Carsten Schürmann, and Roberto Virga. The twelf project. School of Computer Science, Carnegie-Mellon University.

[39] Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. In *Proceedings of the 5th International Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 61–77. Springer-Verlag, March 2001.

[40] Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. *Higher-Order and Symbolic Computation*, (invited submission).

[41] Research On Program Analysis: National Creative Research Center, KAIST. *The Definition of nML*, July 2001. `http://ropas.kaist.ac.kr/n/doc/n-e.ps`.

[42] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[43] Nikolay Shilov. A new decidability proof for full branching time logic ctl*. In *International Conference on Temporal Logic 2000*, pages 145–154, University of Leipzig, Germany, 2000.

[44] Nikolay Shilov. Program schemata technique revised. Submitted to *Journal of Logic and Computation*, May 2000.

[45] Nikolay Shilov. Games with second-order quantifiers which decide propositional program logics. Accepted for presentation on Logic and Games , a Satellite Workshop of ESSLLI, August 20-24, 2001, Helsinki, Finland, April 2001.

[46] Nikolay Shilov and Kwangkeun Yi. Model checking puzzles in mu-calculus. *Joint Bulletin of the Novosibirsk Computing Center and A.P.Ershov Institute of Informatics Systems*, (13), 2000.

[47] Nikolay Shilov and Kwangkeun Yi. Puzzles for learning model checking, model checking for programming puzzles, puzzles for testing model checkers. In *Formal Methods *ELSEWHERE* , a Satellite Workshop of FORTE-PSTV-2000*, pages 18–37, Pisa, Italy, 2000. Final version in v. 43 of *Electronic Notes in Theoretical Computer Science* [53].

[48] Nikolay Shilov and Kwangkeun Yi. Model checking power of propositional program logics. Submitted for presentation on Fixed Points in Computer Science FICS'2001, a Satellite Workshop of PLI'2001, September 8, 2001, Florence, Italy, April 2001.

[49] Nikolay Shilov and Kwangkeun Yi. Model checking vs. expressive power. Submitted to a special issue on model checking of *Journal of Logic and Algebraic Programming*, June 2001.

[50] Nikolay Shilov and Kwangkeun Yi. A note on model checkers reuse. Submitted to *Information Processing Letters*, January 2001.

[51] Nikolay Shilov and Kwangkeun Yi. A note on model checking reuse. Has been accepted for presentation on $2^{nd}$ Australasian Workshop on Computational Logic AWCL'01, 31 January - 1 February 2001, Gold Cost, Australia, August 2001.

[52] Nikolay Shilov and Kwangkeun Yi. On expressive power of second order propositional program logics. Accepted for presentation on A. Ershov $4^{rd}$ International Conference on Perspectives of System Informatics, July 3-6, 2001, Novosibirsk, Russia (to appear in Springer Lecture Notes in Computer Science), February 2001.

[53] Nikolay Shilov and Kwangkeun Yi. Puzzles for learning model checking, model checking for programming puzzles, puzzles for testing model checkers. *Electronic Notes in Theoretical Computer Science*, 43, 2001.

[54] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the European Symposium on Programming*, March 2000.

[55] J. Smith, D. Farber, C. Gunter, S. Nettles, D. Feldmeier, and W. Sincoskie. SwitchWare: Accelerating network evolution(white paper), 1996.

[56] Jung taek Kim and Kwangkeun Yi. Interconnecting between cps terms and non-cps terms. In *The Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, pages 7–16. ACM, January 2001.

[57] Jung taek Kim and Kwangkeun Yi. Interconnecting between cps terms and non-cps terms. *Higher-Order and Symbolic Computation*, (invited submission).

[58] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[59] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types In Compilation*, September 2000.

[60] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of The IEEE Symposium on Logic in Computer Science*, June 2000.

[61] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, January 1999.

[62] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864, pages 238–254. Springer-Verlag, proceedings of the first international static analysis symposium edition, 1994.

[63] Kwangkeun Yi. An abstract interpretation for estimating uncaught exception in Standard ML programs. *Science of Computer Programming*, 31(1):147–173, 1998.

[64] Kwangkeun Yi. *Program Analysis System Zoo.* Research On Program Analysis: National Creative Research Center, KAIST, July 2001. `http://ropas.kaist.ac.kr/zoo/doc/rabbit-e.ps`.

[65] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for java. In *ECOOP'99 Workshop on Formal Techniques for Java Programs*, June 1999.

[66] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for java. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology. ECOOP'99 Workshop Reader (Formal Techniques for Java Programs)*, volume 1743 of *Lecture Notes in Computer Science*, pages 111–112. Springer-Verlag, June 1999. Extended version of this paper is available from `ropas.kaist.ac.kr/~kwang/paper/99-ecoop-yich.ps.gz`.

[67] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[68] Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analyzer 0.98. `http://compiler.kaist.ac.kr/pub/exna/exna-README.html`.

[69] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1997.

[70] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. (to appear).

[71] Kwangkeun Yi and Nikolay Shilov. Engaging students with theory through acm collegiate programming contests. *Communications of the ACM* (accepted), October 2000.

# A Three Selected Papers and Their Reviews

# B Definition of nML

# C Definition of Rabbit

# D Miscellanies