Metacomputation: Exploring New Frontiers of Automated Software Production

「機能と構成」領域　Robert Glueck

Outline

The research explored new frontiers of automated software production. The goal is to build programs that build program. The scientific approach taken in this project is unique in that I investigate a combination of three fundamental principles: (1) three basic transformation operations on programs (program composition, program inversion, and program specialization), (2) multiple layers of these transformations, and (3) their portability to new languages via interpreters. I study these principles using semantically clean functional languages.

Motivation

The introduction of the computer was a giant step in the *execution* of programs (fast, precise, low power consumption), but the *creation* programs was not directly affected. Even today every line of software is written manually, inspected by eye, and maintained by a skilled craftsman. Most recently, the exploding demand for software has led to a dramatic lack of skilled programmers and low software quality. We all experience this, for instance, when our computer freezes or when we loose a file due to a software error. In many cases, programming errors are the primary cause of software failures.

Automatic methods for software production are urgently needed because of the time and money it saves through avoiding software errors. While certain aspects of software development are likely never to be fully automated, such as problem understanding, the method of building *programs that treat programs as data objects* is very powerful. The approach followed in this research is to apply Computer Science's own methods to its major tools: simulating, analyzing and transforming programs by means of programs. This is a challenging and long-term task because programs are semantically the *most complex* form of data objects that exist in the computer.

Thesis

Our goal is to explore the frontiers of automatic software production based on a combination of three

fundamental insights.

1. *Three operations.* Our thesis, based on a structural analysis of formal linguistic modeling as explained in our earlier publication [13], is that three fundamental operations are needed: *program composition, program inversion, and program specialization.* We found that these operations have to be performed efficiently and effectively by tools for software production to be truly powerful. Of these, program specialization, also known as partial evaluation, has been studied intensely and is the best understood method.
2. *Layers of metasystems* solve a wide spectrum of transformation problems using only the three types of operations listed in (1). A cornerstone in this development are the *Futamura projections* which make use of two metasystem layers of program specialization. We examined novel meta-system structures including the specializer projections, multi-level generating extensions and a new metasystem scheme for program composition and program inversion (cf. [2, 7, 12]).

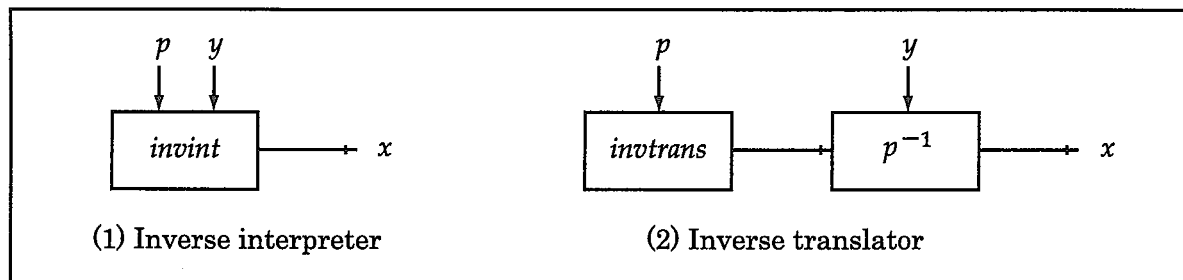

Figure 1: Two tools for solving inversion problems (where $[[p]]x = y$)

3. *New programming languages* for the construction software will continue to emerge rapidly as information technology evolves (cf. the recent phenomenal success of Java). There is no evidence that any particular programming language will be the last in this series. Solutions for (1) and (2) must be able to accommodate languages as they are needed to be truly successful. *Semantics modifiers*, a novel concept for robust semantics [2], promise language independence for composition, inversion, and specialization.

We have identified these three principles as important through our research. Existing approaches to automatic program transformation have only considered part of the operations in (1), or used only restricted forms of metasystem schemes (2). Semantics modifiers (3) are original and, thus, have not been investigated before.

Our research goal was to advance the theory and methods for automatic program transformation based on the principles identified above, and to study the computational feasibility of our scientific ideas for theoretically clean, functional languages. We approached these scientific questions partly by theoretical means and partly by experimental work. What follows is a technical overview. References to publications are provided for more detailed information.

A. Inversion of functions is a fundamental concept in mathematics, but the inversion of programs has received little attention in software science (with the exception of logic programming). Programs that are inverse to each are often used. Perhaps the most common example are programs for compressing and decompressing files sent via networks. Today, programs for both transformations need to be written manually, but this is not necessary. One program should be sufficient, and then have a program inverter derive the other program automatically.

Inversion problems can be solved in two ways, either by an *inverse interpreter* or by a *program inverter.* Both software tools are illustrated in Fig. 1. We studied both approaches for first-order functional languages. A difficulty for program inversion is that traditional programming languages do not support computation in both directions and that there is little known about the automatic generation of inverse programs. Logic programming is suited to find multiple solutions and can be regarded as a method for inverse interpretation, but only for relational programs. A detailed description of these notions can be found in our publications [1, 2, 3].

We studied the Universal Resolving Algorithm (URA), a powerful method for inverse computation for first-order functional programs. The algorithm was implemented in Scheme for a typed dialect of S-Graph, and shows some interesting results for the inverse computation [2, 3]. The algorithm is powerful enough to deal with multiple solutions. We also showed that the algorithm is sound and complete, and computes each solution in finite time [4]. Due to the interpretive nature of the algorithm, inverse computation by URA is slower than using an inverse program.

We analyzed the Korf-Eppstein method (short, KEinv) for automatic program inversion of fi rst-order functional programs [10] and formalized the transformation using a structural operational semantics. It is one of only two existing general-purpose automatic program inverters that were ever built. This was the basis for studying the generation of inverse programs.

Recently we proposed [11] a method for automatic program inversion in a first-order functional programming language that achieves transformations beyond KEinv. One of our key observations

is that the duplication of values and testing their equality are two sides of the same coin in program inversion. This led to the design of a new self-inverse primitive function that considerably eases the inversion of programs. We illustrated the method with several examples including the automatic derivation of a program for run-length decoding from a program for run-length encoding. This derivation is not possible with other methods, such as KEinv. Another example, more theoretical in nature, is the inversion of a program fib that computes pairs of neighboring Fibonacci numbers; for instance, *fib* (2) = <2, 3>. The automatic inversion is successful and produces an inverse program *fib*$^{-1}$; for instance, *fib*$^{-1}$ (<34, 55>) =8.

B. Composition The construction of complex software by sharing and combining components in order to ease software construction is the main focus of many recent approaches. But abstraction layers do not come for free: they add redundant computations, intermediate data structures, extra run-time error checking. Program composition is a program optimization that can remove such redundancies, and allows the composition of software parts without paying an unacceptably high price in terms of efficiency.

We examines the problem to transform functional programs, which intensively use append functions into programs, which use accumulating parameters instead (like efficient list reversal) [14]. We studied an (automatic) transformation algorithm for our problem and identify a class of functional programs, namely restricted 2-modular tree transducers, to which it can be applied [15]. We showed how intermediate lists built by a selected class of functional programs, namely "accumulating maps", can be deforested using a single composition rule. For this we introduced a new function 'dmap', a symmetric extension of the familiar function 'map'. While the associated composition rule cannot capture all deforestation problems, it can handle accumulator fusion of functions defined in terms of 'dmap' in a surprisingly simple way. For this research direction we conclude, that automatic, non-trivial composition remains a challenging research problem for the future. Possibly, program composition the most difficult of the three operations to achieve in an automatic and general fashion.

C. Semantics modifiers A key ingredient of our approach are *semantics modifiers* because they allows the design of general and reusable program transformers which make use of results of task A and B, in principle, portable to other programming language.

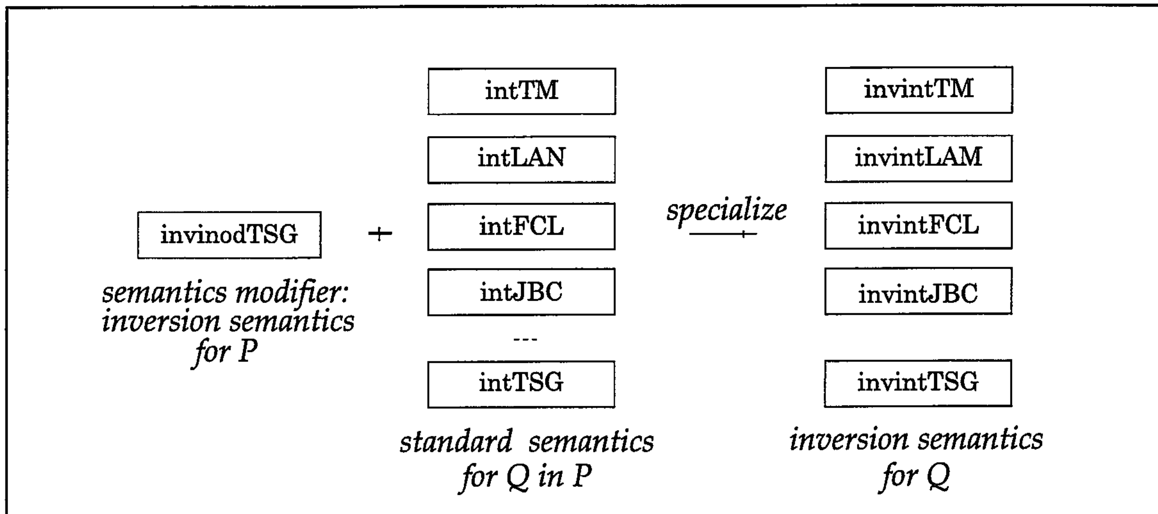

Figure 2: Semantics modifier + standard semantics = non-standard semantics.

We developed a mathematical theory for non-standard semantics and examined the meaning of several non-standard interpreter towers [1]. Our results suggest a technique for the implementation of a certain class of programming language dialects by composing a hierarchy of non-standard interpreters. Based on this theory, we experimented [12] with the Universal Resolving Algorithm (see A above) to prototype programming language tools from robust semantics: we used automatic program specialization to turn interpreters into inverse interpreters for several small languages for which no hand-written tools exist (including interpreters for an applied lambda calculus, an imperative flowchart language, and a subset of Java bytecode). This is illustrated in Fig. 2.

This application of self-applicable program specializers is remarkable since it suggest a new use of program specialization that is different from the familiar Futamura projections. Also, we studied powerful specialization methods [6], loop peeling to increase the accuracy of program analysis [16] and edited a special issue on program transformation and partial evaluation [9].

For our experiments we needed to analyze the power of program specialization and have done so for online and offline partial evaluation [5], for the Futamura projections [8] and binding-time improvements [7].

Despite practical successes with the Futamura projections, it has been an open question whether target programs produced by specializing interpreters can always be as efficient as those produced by a translator. We showed that, given a Jones-optimal program specializer with static expression reduction, there exists for every translator an interpreter which, when specialized, can produce target programs that are at least as fast as those produced by the translator. We call this class *translation*

universal specializers. We also showed that a specializer that is not Jones-optimal is not translation universal. In a second step we examined Ershov's generating extensions and introduced the class of generation universal specializers. We answered these questions on an abstract level, independently of any particular program specializer. We were interested in statements that are valid for all specializers, and have identified such conditions.

In another study about the strength of program specializers, we showed that the accuracy of online partial evaluation, or polyvariant specialization based on constant propagation, can be simulated by offline partial evaluation using a maximally polyvariant binding-time analysis [5]. We showed [7] that Jones optimality, which was originally aimed at the Futamura projections, plays an important role in binding-time improvements. The main results show that, regardless of the binding-time improvements which we apply to a source program, no matter how extensively, a specializer that is not Jones-optimal is strictly weaker than a specializer which is Jones optimal.

Future Work

Our research centered around three important principles (three program operations, metasystem layers, adaptability). In particular, we examined inverse computation theoretically and experimentally, and adapted an algorithm to several programming language subsets by automatic program specialization, including a small subset of Java Bytecode. We characterized the accuracy of online and offline specialization [5] and identified the conditions for strong binding-time improvements [7] and the translation universality [8] of Futamura projections. We proposed an automatic method for program inversion that is stronger in some important aspects than other inversion methods and shows some remarkable results. [10]. For program composition, attribute grammars are promising and we have done steps in this direction [14,15], but conclude that the fundamental problem of accumulator fusion remains a challenging research task for future work.

We found that there is no theoretical limit to the translation power of the Futamura projections provided a specializer with static expression reduction is also Jones-optimal and introduced the class of translation universal specializers. We believe that the power to perform universal computations is another property for the theoretical power of a program specializer. Whether the results can be adapted to other non-standard interpreter hierarchies as developed in [1] is a topic for future work. It is quite possible that the results [7,8] can be carried to the next metasystem level. We also want to explore the conditions for generating translators and other program generatorsfrom generation

universal specializers.

Our experiments applied the idea of prototyping programming language tools from robust semantics [12]: we produced automatically inverse interpreters for programming languages for which no inverse interpreter existed before. Even though these languages are small, the results demonstrate that it is possible in practice with existing partial evaluators. To the best of our knowledge, these are the first results regarding this use of partial evaluation. Our results show that a speedup of an order of magnitude can be achieved for some interpreters. Limiting factors of offline partial evaluation was the need for binding-time improvements and the lack of generalization. We believe there is still more to be gained by partial evaluation and want to investigate stronger specialization techniques, such as [6].

A main difficulty in the generation of inverse programs are conditionals and recursive functions. We now try to solve some of these difficulties through the application of parsing techniques to program inversion. Tasks for future work also include the refinement of the well-formedness criteria [10]. We have not exploited mathematical properties of operators during the inversion. A possible extension of our techniques may involve the use of constraint systems for which well-estabilished theories have been developed in other areas.

We described an algorithm for inverse computation, studied its organization and structure, and illustrated our implementation with several examples [3,12]. Methods for detecting finite solution sets and cutting infinite branches can make the process of inverse computation terminate more often (while preserving soundness and completeness) and may make the method more practical. Techniques from program transformation and logic programming may prove to be useful in this context, and we are now taking first steps into this direction. We also want to explore further its portability to new languages via semantics modifies [1,2].

References

[1] S. M. Abramov and R. Glueck. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, Lecture Notes in Computer Science, Vol. 1974, pages 201–213. Springer-Verlag, 2000.

[2] S. M. Abramov and R. Glueck. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171–211, 2001.

[3] S. M. Abramov and R. Glueck. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, Lecture Notes in Computer Science, Vol. 2566, pages 269–295. Springer-Verlag, 2002.

[4] S. M. Abramov and R. Glueck.The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.

[5] N. H. Christensen and R. Glueck. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, to appear, 2003.

[6] Y. Futamura, Z. Konishi, and R. Glueck. WSDFU: Program transformation system based on generalized partial computation. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 358–378. Springer-Verlag, 2002.

[7] R. Glueck. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.

[8] R. Glueck. The translation power of the Futamura projections. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.

[9] R. Glueck and Y. Futamura. Special issue on partial evaluation and program transformation. *New Generation Computing*, 20(1):1–124, 2002.

[10] R. Glueck and M. Kawabe. An automatic program inverter for Lisp: potential and limitations. In Y. Fu and Z. Hu, editors, *Proceedings of the Third Asian Workshop on Programming Languages and Systems*, pages 230–245. Shanghai Jiao Tong University, 2002.

[11] R. Glueck and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Asian Symposium on Programming Languages and Systems. Proceedings*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.

[12] R. Glueck, Y. Kawada, and T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 10–19. ACM Press, 2003.

[13] R. Glueck and A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563–1570. World Scientific, 1994.

[14] K. Kakehi, R. Glueck. and Y. Futamura. On deforesting parameters of accumulating maps. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation. Proceedings*, volume 2372 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, 2002.

[15] A. Kühnemann, R. Glueck. and K. Kakehi. Relating accumulative and non-accumulative functional programs. In A. Middeldorp, editor, *Rewriting Techniques and Applications. Proceedings*, Lecture Notes in Computer Science, Vol. 2051, pages 154–168. Springer-Verlag, 2001.

[16] L. Song, R. Glueck and Y. Futamura. Loop peeling based on quasi-invariance/induction variables. *Wuhan University Journal of Natural Sciences*, 6(1-2):362–367, 2001.