Walid Taha (Ed.)

LNCS 2196

Semantics, Applications, and Implementation of Program Generation

Second International Workshop, SAIG 2001
Florence, Italy, September 2001
Proceedings

Springer

Lecture Notes in Computer Science 2196

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Walid Taha (Ed.)

Semantics, Applications, and Implementation of Program Generation

Second International Workshop, SAIG 2001
Florence, Italy, September 6, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Walid Taha
Yale University, Department of Computer Science
51 Prospect St., New Haven, CT 06511, USA
E-mail: taha@cs.chalmers.se

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Semantics, applications, and implementation of program generation : second international workshop ; proceedings / SAIG 2001, Florence, Italy, September 6, 2001. Walid Taha (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2196)
ISBN 3-540-42558-6

CR Subject Classification (1998): D.3, F.3, D.1, F.4.1, D.2

ISSN 0302-9743
ISBN 3-540-42558-6 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author
Printed on acid-free paper SPIN: 10840656 06/3142 5 4 3 2 1 0

Preface

This volume constitutes the proceedings of the second International Workshop on the Semantics, Applications, and Implementation of Program Generation (SAIG 2001) held on 6 September, 2001, in Florence, Italy. SAIG 2001 was held as an ACM SIGPLAN workshop co-located with the International Conference on Principles, Logics, and Implementations of High-level Programming Languages (PLI).

As the commercial production of software systems moves toward being a traditional industry, automation will necessarily play a more substantial role in this industry, just as it plays a key role in the production of traditional commodities. SAIG aims at promoting the development and the application of foundational techniques for supporting automatic program generation. A key goal of SAIG is to provide a unique forum for both theoreticians and practitioners to present their results and ideas to an audience from a diverse background.

This year we are fortunate to have three influential invited speakers: Krzysztof Czarnecki (DaimlerChrysler), Tim Sheard (OGI School of Science and Engineering), and Mitchell Wand (Northeastern University). The proceedings include abstracts of the invited talks, and an invited paper by Tim Sheard.

Seven technical papers and two position papers were presented at SAIG 2001. The technical papers cover a wide spectrum of topics, including:

- A rigorous, operationally-based treatment of the correctness of an important program transformation (*Johann*)
- A schema-based approach to generating solutions for maximum multi-marking problems (*Sasano, Hu, and Takeichi*)
- An elegant method for the generation of machine code without chains of jumps (*Damian and Danvy*)
- A uniform approach to the compilation of goal-directed programs using partial evaluation (*Danvy, Grobauer, and Rhiger*)
- The integration of partial evaluators into interpreters (*Asai*)
- A software design method for Haskell based on the Unified Modeling Language (UML) (*Wakeling*)
- A novel approach to dynamically adaptable software using staged languages (*Harrison and Sheard*)

The two position papers tackle novel application areas:

- Global computing through meta-programming (*Ferrari, Moggi, and Pugliese*)
- Optimizing functional programs using size inference (*Herrmann and Lengauer*)

We thank the participants for their excellent contributions.

July 2001 Walid Taha

Scope

SAIG welcomes contributions on or across any of the following facets of program generation:

- Software engineering methods and processes,
- Domain specific languages,
- Deductive program synthesis methods,
- Computer algebra and symbolic computation,
- High-performance/high-reliability systems,
- Specialized support in traditional programming languages,
- Novel accounts of traditional compilation and linking techniques, and
- Specialized semantics-based methods and approaches.

Review Process

A call for papers was announced on several mailing lists and newsgroups. The workshop accepted regular technical submissions and position papers, and featured an open panel discussion at its conclusion. All papers are reviewed for presentation, clarity, interest, and coverage of related work. In addition, regular papers must contain novel technical contributions. In contrast, position papers must provide a clear description of a problem, survey existing approaches, and give a clear description of and argument for the proposed approach.

Fifteen submissions were received this year, twelve of which were technical papers and three position papers. Of these submissions seven technical papers and two position papers were accepted. There was a drop in submissions from last year, where there were 20 submissions in total. The most likely explanation for this drop is the occurrence of two related multi-day events earlier this year, namely PADO II and Reflection. That SAIG received this number of papers is an indication of a sustained interest in this research area.

All accepted papers are of the same high quality as the papers accepted last year. Most papers received three reviews. The final decisions were made collectively by the Program Committee based on all available reviews. In cases where Program Committee discussions were of benefit to the authors, the discussions were summarized and included with the reviews. One submission received a conditional acceptance, and the authors addressed the concerns of the reviewers in time for publication.

To promote further development of the works presented at the workshop, a special issue of a journal on the theme of SAIG will be organized after the event.

Organization

This year, an Advisory Committee has been formed. This committee is responsible for the long term well-being of SAIG, and for selecting the Program Committee Chair each year. The Program Committee Chair is responsible for the selection of the Program Committee members. When there is need, Program Committee members are free to solicit the help of external reviewers.

Advisory Committee

Don Batory, Texas at Austin
Eugenio Moggi, Genova
Greg Morrisett, Cornell
Tim Sheard, OGI
Walid Taha, Yale (Chair)

Program Committee

Gilles Barthe, INRIA
David Basin, Freiburg
Don Batory, Texas at Austin
Robert Glück, DIKU and Waseda
Nevin Heintze, Agere Systems
Eugenio Moggi, DISI
Greg Morrisett, Cornell
Flemming Nielson, DTU
David Sands, Chalmers
Walid Taha, Yale (PC Chair)

External Reviewers

Reiner Hänle, Chalmers
Dino Oliva, Agere Systems
Perry Wagle, OGI
Ed Walter, Agere Systems

Acknowledgments

SAIG 2001 would not have been possible without the support of the PLI organizers. We would especially like to thank Betti Veneri (PLI Workshop Chair) for her help in organizing the workshop, Carole Mann (Registration Systems Lab) for her quick response to our last minute requests, and Alfred Hofmann (LNCS Editor) and his team for their continual support during the preparation of these proceedings. Richard Gerber provided us with the START software and a lot of assistance while we tailored the system to our needs. Chuck Powel and Mark Wogahn (Yale CS Workstation Support) provided us with a lot of timely help.

Finally, we would like to thank the Università di Genova, Yale University, and ACM SIGPLAN for providing financial support for this workshop.

Table of Contents

Invited Talks

Technical Papers

Position Papers

Generative Programming and Software System Families

Abstract of Invited Talk

Krzysztof Czarnecki

DaimlerChrysler AG, Research and Technology,
Software Technology Lab, 89081 Ulm, Germany
czarnecki@acm.org
www.generative-programming.org

Today's software engineering practices are aimed at developing single systems. There are attempts to achieve reuse through object- and component-based technologies with two specific goals: to cut development costs, and time-to-market and to improve quality. But current research and practical experience suggest that only moving from the single system engineering to the system-family engineering approach can bring significant progress with respect to these goals [3,6,7].

Generative programming builds on system-family engineering and puts its focus on maximizing the automation of application development [1,2,4,5]: given a system specification, generators use a set of reusable components to generate the concrete system. Both the means of application specification, the generators, and the reusable components are developed in a domain-engineering cycle. This talk introduces the necessary techniques, notations, and processes using examples. It also outlines our vision of how the software industry can be transformed in a similar way the traditional industries moved from manual craftsmanship to automated assembly lines, and the role generative techniques can play in this transition.

References

1. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355–398.
2. J. C. Cleaveland. Building Application Generators. In *IEEE Software*, no. 4, vol. 9, July 1988, pp. 25–33.
3. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, to appear in 2001.
4. K. Czarnecki and U. Eisenecker. *Generative Programming — Methods, Tools, and Applications.* Addison-Wesley, Boston, MA, 2000.
5. J. Neighbors. Software construction using components. Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980.
6. D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1–9.
7. D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, Reading, MA, 1999.

W. Taha (Ed.): SAIG 2001, LNCS 2196, p. 1, 2001.

Accomplishments and Research Challenges in Meta-programming

Invited Paper

Tim Sheard

Pacific Software Research Center
OGI School of Science and Engineering
Oregon Health & Science University
sheard@cse.ogi.edu,
http://www.cse.ogi.edu/~{}sheard

1 Introduction

In the last ten years the study of meta-programming systems, as formal systems worthy of study in their own right, has vastly accelerated. In that time a lot has been accomplished, yet much remains to be done. In this invited talk I wish to review recent accomplishments and future research challenges in hopes that this will spur interest in meta-programming in general and lead to new and better meta-programming systems.

I break this paper into several sections. As an overview, in Section 2, I try and classify meta-programs into groups. The purpose of this is to provide a common vocabulary which we can use to describe meta-programming systems in the rest of the paper.

In Section 3, I describe a number of contexts in which the use of meta-programming has been found useful. Some knowledge of the areas where meta-programming techniques have been developed helps the reader understand the motivation for many of the research areas I will discuss.

In Section 4, I motivate why meta-programming systems are the right tools to use for many problems, and I outline a number particular areas where I believe interesting research has been accomplished, and where new research still needs to be done. I do not claim that this set of issues is exclusive, or that every meta-programming system must address all of the issues listed. A meta-programming system designer is like a diner at a restaurant, he must pick and choose a full meal from a menu of choices. This section is my menu.

In the following Sections I elaborate in more detail on many of the areas outlined in Section 4. I will discuss many ideas from many different researchers that I think are important in the overall meta-programming picture. For some areas, I have outlined proposed research projects. My proposals are at the level of detail I would assign to a new student as a project, and I have not personally carried the research to its conclusions.

If we continue using the food metaphor, in these sections we discuss the general preparation of menu items; which ingredients need special handling; and

W. Taha (Ed.): SAIG 2001, LNCS 2196, pp. 2–44, 2001.

special techniques that apply to broad areas of cooking. Not so much a cookbook that describes how to make each item on the menu, but a cooking class in general techniques of building and understanding meta-programming systems.

Finally, in Section 21, I discuss a number existing meta-programming systems. My understanding of what they were designed to do, and where in the taxonomy of meta-programming systems they lie. In this section I also discuss the MetaML system, which is my attempt at building a useful meta-programming system. In the world of meta-programming meals, MetaML is only one full course meal. I look forward to many other delightful meals in the future, especially those where I am the diner and not the chef.

2 Taxonomy of Meta-programs

In a meta-programming system, *meta-programs* manipulate *object-programs*. A meta-program may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs. We use the term object-program quite loosely here. An object-program is any sentence in a formal language. Meta-programs include things like compilers, interpreters, type checkers, theorem provers, program generators, transformation systems, and program analyzers. In each of these a program (the meta-program) manipulates a data-object representing a sentence in a formal language (the object-program).

What kind of meta-programs are there? Meta-programs fall into two categories: program generators and program analyzers. A program generator (a meta-program) is often used to address a whole class of related problems, with a family of similar solutions, for each instance of the class. It does this by constructing another program (an object-program) that solves a particular instance. Usually the generated (object) program is "specialized" for a particular problem instance and uses less resources than a general purpose, non-generated solution.

A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kind of meta-systems are: program transformers, optimizers, and partial evaluation systems. In addition to this view of meta-programs as generators or analyzers (or a mixture of both), there are several other important distinctions.

- **Static vs. run-time.** Program generators come in two flavors: static generators, which generate code which is then "written to disk" and processed by normal compilers etc. and run-time code generators which are programs that write or construct other programs, and then immediately execute the programs they have generated. If we take this idea to the extreme, letting the generated code also be a run-time code generator, we have *multi-stage programming*.

Examples of run-time program generators are the multi-stage programming language MetaML [69,79], run-time code generation systems like the Synthesis Kernel [46,67], 'C [64], and Fabius [44]. An example of a static program generator is Yacc [39].

- **Manually vs. automatically annotated.** The body of a program generator is partitioned into static and dynamic code fragments. The static code comprises the meta-program, and the dynamic code comprises the object-program being produced. Staging annotations are used to separate the pieces of the program.
  We call a meta-programming system where the programmer places the staging annotations directly a manually staged system. If the staging annotations are place by an automatic process, then the meta-programming system is an automatically staged system.
  Historically, the area of partial evaluation pioneered both the technique and terminology of placing the staging annotations in an automatic way without the intervention of the programmer. Write a normal program, declare some assumptions about the static or dynamic nature of the programs inputs, and let the system place the staging annotations. Later, it became clear that manually placing the annotations was also a viable alternative.
- **Homogeneous vs. heterogeneous.** There are two distinct kinds of meta-programming systems: homogeneous systems where the meta-language and the object language are the same, and heterogeneous systems where the meta-language is different from the object-language.
  Both kinds of systems are useful for representing programs for automated program analysis and manipulation. But there are important advantages to homogeneous systems. Only homogeneous systems can be n-level (for unbounded n), where an n-level object-program can itself be a meta-program that manipulates $n+1$-level object-programs. Only in a homogeneous meta-system can a single type system be used to type both the meta-language and the object-language. Only homogeneous meta-systems can support reflection, where there is an operator (`run` or `eval`) which translates representations of programs, into the values they represent in a uniform way. This is what makes run-time code generation possible. Homogeneous systems also have the important pedagogical and usability property that the user need only learn a single language.

3 Uses of Meta-programs

Meta-programming provides various benefits to users. We explain some of these benefits here.

- **Performance.** A common objective of many meta-programming systems is performance. Meta-programs provide a mechanism that allows general purpose programs to be written in an interpretive style but to also perform without the usual interpretive overhead. Rather than write a general purpose but inefficient program, one writes a program generator that generates

an efficient solution from a specification. The interpretive style eases both maintenance and construction, since a single program solves many problems. One program is easier to maintain than many similar individual programs. The use of the parser generator Yacc is an illustrative example. Rather than using a general purpose parsing program, we generate an efficient parser from a specification, i.e. a language grammar.

- **Partial evaluation.** Partial evaluation is another meta-programming technique used to improve performance. Partial evaluation optimizes a program using a-priori information about some of that program's inputs. The goal is to identify and perform as many computations as possible in a program before run-time. The most common type of partial evaluation, *Off-line* partial evaluation, has two distinct steps, *binding-time analysis* (BTA) and *specialization*. BTA is an analysis that determines which computations can be performed in an earlier stage given only the names of inputs available before run-time (the static inputs). Specialization uses the values of the static inputs to produce an improved program.
- **Translation.** Perhaps the most common use of meta-programming is translation of one object-program to another object-program. The source and target languages may or may not be the same. Examples of translators are compilers and program transformation systems.
- **Reasoning.** Another important use of meta-programs is to reason about object-programs. If the object-programs are sentences in a formal language, an analysis can discover properties of the object-program. These properties can be used to improve performance, provide assurance about the object-programs behavior, or validate meaning preserving transformations. Examples of reasoning meta-programs are program analyses such as flow analyses and type checkers. Reasoning meta-programs are also used to build theorem proving systems such as LEGO [66], HOL [33], Coq [7] and Isabelle [57] and the study [36] and implementation [24] of logical frameworks such as Elf [60], Twelf [62], LF [61].
- **Pedagogy.** A pedagogical use of meta-programs is program observation. Computation often proceeds in stages. Inputs arrive in several stages and the computation comprising each stage is a program that incorporates the current inputs, and anticipates the next stage. Higher order functions provide a convenient mechanism for structuring staged programs. In a higher-order language solutions proceed by accepting some input, then producing as a result, a function that can deal with the next stage. Since functions are extensional (they can be observed only by noticing how they behave when applied to inputs), it is hard to explain or understand programs written in this style, since the intermediate stages cannot be observed.
  A meta-programmed solution alleviates this problem. Instead of each stage producing a function as output, each stage can produce the code of a function as output. The code is observable and its structure is often quite illuminating. We have used this to illustrate several very complex algorithms to great effect, for example, the continuation-passing-transform, monad-transformers, and combinator parsers

– **Mobile code.** Recently meta-programming has been used as means of program transportation. Instead of using networks to bring the data to the program, networks are used to bring the program to the data. Because of security reasons, intensional representations of programs are transported across the network [38,82]. These representations can be analyzed for security and safety purposes to ensure that they do not compromise the integrity of the host machines they run on. The transported programs are object-programs and the analyses are meta-programs.

Why is meta-programming hard? Meta-programming is hard because programs are complex. Large computer programs may well be the most complex entities ever constructed by humans. Programmers utilize many features to manage this complexity. These features are often built-in to programming languages and include: type-systems (to catch syntactically correct, yet semantically meaningless programs), scoping mechanisms (to localize the names one needs think about), and abstraction mechanisms (like functions, object hierarchies, and module systems to hide irrelevant details). These features add considerably to the complexity of the languages they are embedded in, but are generally considered worth the cost. When we write programs to manipulate programs we must deal with this complexity twice, once in the programs we write, and again in the data they manipulate.

A good meta-programming system knows about and deals directly with the complexities of the object-language. If the meta-language does not deal directly with the type system, scoping discipline, and abstraction mechanisms of the object-language, the meta-programmer must encode these features using some lower level mechanism, complicating an already difficult task.

How can a meta-programming system help? A meta-programming system is supposed to make the manipulation of object programs easier. It should be easy to use and understand by the programmer; interface the meta- and object-languages seamlessly; provide high-level abstractions that capture both the details of the object-language and the patterns used by the meta-programs; be blindingly fast; and in the case of program generators, generate blindingly fast code. There is utility in providing general purpose solutions to these needs that can be reused across many systems. There has been lots of good work in addressing many of these issues, but it is not always clear how they fit together, or what is still missing.

4 Research Areas

Meta-programming as an area has been around for a long time. The LISP hackers had it right. Programs are data. But as a formal area of study, meta-programming has become an active area of research only in the last decade or so. There is a huge amount of work that remains to be done. Enough to supply legions of Ph.D. students with thesis topics for the next 10 years. The work

varies from highly theoretical proofs to the nitty-gritty engineering problems of systems building. All of it is important, and worth doing. An overview of a few areas is presented in the itemized list below. In the rest of the paper we discuss many of these items in more detail.

- **Representing Programs (Section 5).** Programs are data, but they are complex entities. How do we represent them to hide unnecessary details, yet make their important structure evident, and their common operations easy to express, and efficient to implement?
- **Presentation (Section 6).** Presentation is the interface to the object-language that the meta-programming system provides to the programmer. Presentation can have immense effect on the usability of a system. Our experience with MetaML shows that object-language templates that "look like" object language programs are a great boon to the meta-programmer.
- **Integrating automatic and manual annotation (Section 8).** Partial evaluation systems save the user the bother of placing staging annotations, but the user of an automatic system loses some control over the structure of the output. Manually placing staging annotations provides complete control, but in many cases is tedious and error prone. Can the two techniques be married in a harmonious relationship?
- **Observing the structure of code (Sections 9 & 10).** There are many techniques for representing code as data. Many make code an abstract type. This is done to support its dual nature, or to provide a more usable presentation. Since these representations hide the internal structure of code, some other interface to the internal structure of code is necessary if code is to deconstructed or observed. A good interface that is both easy to use, and which reflects the user's logical view of the object-code's structure (rather than a view of how it is implemented) is hard to obtain.
- **Manipulating Binding constructs (Section 13 & 14).** Many object-languages include binding constructs which introduce and delineate the scope of local variables. As far as the meta-program is concerned, the actual name of these variables is immaterial. Any name could be used as long as it is used in a consistent manner. When generating programs one needs to invent new local names that are guaranteed to be different from existing names. This is necessary to prevent the possibility that one will introduce a scope that will inadvertently hide a necessary variable. In program transformation or analysis, the flip side of the coin must be dealt with. When deconstructing programs how can we ensure that locally scoped variables never escape their scope and become unbound?
- **Manipulating typed object-programs (Sections 15, 17, & 18).** When we write programs, many of us find the discipline of a typed programming language too valuable to ignore. Type systems catch errors sooner and facilitate writing good programs. Why wouldn't we want the same benefits for our object-programs? A typed meta-programming system can catch type errors in object-programs at the compile-time of the meta-program. Our MetaML system (see Sections 7 & 21) has made a good start in this area. Problems

still to be addressed include meta-programs that generate (or manipulate) object-programs with different object-types, that depend upon the meta-program's input, issues involving polymorphism inside object programs, and the issue of type-safety and the use of effects in meta-programming.

- **Heterogeneous meta-programming systems (Section 12, 18, & 19).** What do we do about heterogeneous systems where the meta-language and object-language are different? Heterogeneous systems bring a whole new set of challenges to the forefront. For every object-language, a new and different type system must be defined. If the type of the meta-program is to say something about the type of the object-programs it produces, then it must be possible to embedded the type system of the object-languages into the type-system of the meta-language. There is no guarantee that the object-language type systems is in any way similar or compatible to the meta-language type system. The two type systems may be incommensurate. For example, the meta-language may be the polymorphic lambda calculus, and the object-language Cardelli's object calculus [4], so a simple embedding is not always possible.
- **Building good implementations (Section 21).** Implementations of meta-programming systems are rare. We especially lack good implementations of run-time code generating systems. There are many competing needs in such systems. Should we generate code quickly? Or should we generate fast code? Either is hard to do. Sometimes, doing *both* seems nigh on impossible. How can we control the tradeoff?

  General purpose program generators should be able to generate code that interacts with several different object-program environments. Do we really need a Yacc for every language? How do we build meta-systems with the ability to produce object-programs that interact with differing environments?
- **The theory of meta-programs (Section 20).** Reliable systems are only possible when we understand the theory behind the systems we wish to build. Meta-programs have subtle semantic difficulties that just do not occur in ordinary programs. A good theory of meta-programming systems is necessary to isolate these problems and understand their interaction with other system features. Good theory leads to good tools.

In the following sections I discuss many of these areas in more detail. In many of the sections I give example programs, some times in imaginary, not-yet-existing languages. I have a choice here of which language style to use for these examples. On one hand, I have built a large system (MetaML) with great care to adhere to Standard ML's syntax and semantics, so I might like to give my examples in an ML style language. On the other hand, ML's style of using postfix application for type-constructor application (i.e. `int list`), its use of quoted variables (i.e. `'a`) to represent type variables, and its inability to give type declarations (where the programmer gives a type for a function, but not its definition), push me towards using a style more akin to Haskell, with its typing prototypes and qualified type system.

So in the end I choose to do both. When giving examples that are actual MetaML programs, I adhere to the ML style. In all other places I adhere to the Haskell style of program definition.

5 Representing Programs

Many meta-systems represent object-programs by using strings, graphs, or algebraic data-structures. With the string encoding, we represent the code fragment `f(x,y)` simply as `"f(x,y)"`. While constructing and combining fragments represented by strings can be done simply, due to their lack of internal structure, deconstructing them is quite complex. More seriously, there is no *automatically verifiable* guarantee that programs thusly constructed are syntactically correct. For example, `"f(,y)"` can have the static type string, but this clearly does *not* imply that this string represents a syntactically correct program. The problem is that strings have no internal structure corresponding to the object-language's structure.

Using PERL as a meta-language for object-language (say HTML) manipulation moves all the work of implementing language manipulations to the user. It is better to move common tasks into the meta-language implementation so the programmer does not need to solve the same problems over and over again.

The lack of internal structure is so serious that my advice to programmers is unequivocal: *No serious meta-programmer should ever consider representing programs as strings.*

LISP systems, and their use of S-expressions, add internal structure to program representations, but this does not really solve the syntax correctness problem. Not all S-expressions are legal object-programs (unless, of course, the object-language is S-expressions). LISP also lacks a static typing mechanism which is extremely useful when meta-programming.

An algebraic approach can be used to capture an object-language's structure in a more rigorous manner. One solution in this vein is to use a data-structuring facility akin to the algebraic datatype facility of Standard ML or Haskell. With the datatype encoding, we can address both the destructuring problem and the syntactic correctness problem. A datatype encoding is essentially the same as *abstract syntax*. The encoding of the fragment `"f(x,y)"` in an Haskell algebraic datatype might be:

```
Apply (Variable "f") (Tuple[Variable "x" ,Variable "y"])
```

using a datatype declared as follows:

```
data Exp = Variable String | Apply Exp Exp | Tuple [ Exp ]
         | Constant Int    | Abs String Exp
```

Using a datatype encoding has an immediate benefit: *correct typing for the meta-program ensures* correct syntax for all object-programs. For languages which support pattern matching over datatypes, like Haskell and Standard ML, deconstructing programs becomes easier than with the string representation. However, constructing programs is now more verbose because we must use the

cumbersome constructors like `Variable`, `Apply`, and `Tuple`. The drawback is the requirement that the meta-programmer must be aware of the detailed mapping of the concrete syntax of the object language into the data structuring component of the meta-language. If at all possible, it is better to manipulate a representation with the more familiar feel of the concrete syntax of the object language.

6 Presentation

A *quasi-quote* representation is an attempt to make the user's interface to the object-language as much like the object-language concrete syntax as possible. Here the actual representation of object-code is hidden from the user by the means of a quotation mechanism. Object code is constructed by placing "quotation" annotations around normal object-language concrete syntax fragments. Inside quotations, "anti-quotation" annotations allow the programmer to splice in computations that result in object-code.

I am told that the idea of quasi-quotation originates in the work of the logicians Willard V. Quine in his book *Mathematical Logic* [83], and Rudolph Carnap in his book *The Logical Syntax of Language* [15].

A description of the early use of quasi-quotation appears in Guy Steele's *The evolution of LISP* [73] where he describes various dialects of MacLISP which supported a feature he calls *pseudo-quoting*

– *Pseudo-quoting allowed the code to compute a replacement value, to occur within the template itself. It was called pseudo-quoting because the template was surrounded by a call to an operator which was "just like quote" except for specially marked places within the template.*

In LISP back-quote begins a quasi-quotation, and a comma preceding a variable or a parenthesized expression acts as an anti-quotation indicating that the expression is to be treated, not as a quotation, but as a computation that will evaluate to a piece of object-code. A short history of *Quasiquotation in LISP* [8] can be found in an article of that name by Alan Bawden in the 1999 PEPM proceedings as an invited talk, and describes this in much more detail.

In LISP, quasi-quotation is unaware of the special needs of variables and binding forms. Quasi-quotation does not ensure that variables (atoms) occurring in a back-quoted expression are bound according to the rules of static scoping. For example `‘(plus 3 5)` does not bind `plus` in the scope where the back-quoted term appears, nor does it treat the `x` in `‘(lambda (x) `*exp*`)` in any reasonable way that respects it as a binding occurrence.

It wasn't until much later in the design and implementation of the Scheme Macro system [23,22], that quasi-quotation dealt properly with these issues. MetaML also fixes this problem and employs a static typing discipline, which types quasi-quoted expressions with object-level types, a useful and important extension.