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,
WWW home page: 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 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 ef-

fect, 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.

A quasi-quote presentation is a mechanism with the benefits of both a string and an algebraic datatype representation. It guarantees the syntactic correctness of object programs using the type correctness of the meta-programs, but maintains the ease of construction of object-programs.

7 A short introduction to MetaML

MetaML is a homogeneous, manually annotated, run-time generation system. In MetaML we use angle brackets (`< >`) as quotations, and tilde (`~`) as the anti-quotation. We call the object-level code inside a pair of angle brackets, along with its anti-quoted holes a *template*, because its stands for a computation that will build an object-code fragment with the shape of the quoted code. Along with the syntactic correctness guarantee, MetaML's template mechanism also guarantees type correctness at the object-level, and treats object-level variables in a manner that respects static scoping. We illustrate these features below:

```
-| val x = <3 + 2> ;
val x = <3 %+ 2> : <int>

-| val code = <show ~x> ;
val code = <%show (3 %+ 2)> : <string>
```

In this example we construct the object-program fragment `x` and use the anti-quotation mechanism to splice it into the object-program fragment `code`. Note how the definition of `code` uses a template with a hole.

We introduce a new type constructor `< t >` (pronounced *code of t*) to type meta-level terms which evaluate to object-code. Note how each code fragment is assigned a code type, where the type inside the angle brackets indicates the type of the object-program. For example `<3+4>` has type `<int>`, because `3+4` has type `int`.

The type system helps the user construct well formed object-programs. One of its most useful features is that it tracks the level at which variables are bound. Attempts to use variables at a level lower than the level at which they are bound makes no sense, and is reported by the type checker. For example:

```
-| fun id x = x;
val id = Fn  : 'a  -> 'a

-| <fn x => ~(id x) - 4>;
Error: The term: x   Variable bound in stage 1 used too early in stage 0
```

In the above example `x` is a stage 1 variable, but because of the anti-quotation it is used at stage 0. This is semantically meaningless and is reported as a type error.

In MetaML, a quasi-quoted template involving variable binding automatically alpha-renames bound variables in a manner that precludes inadvertent name capture. Note how the bound variables `x` are alpha-renamed from `x` to `a` and `b`.

```
-| <fn x => fn x => x - 1>;
val it = <(fn a => (fn b => b %- 1))> : <'a  -> int -> int>
```

This is particularly useful when computing the body of a lambda-abstraction using an anti-quoted computation. If the computation places the variable bound by the lambda-abstraction in a context where the same variable is bound in another way, inadvertent capture can result.

```
-| fun plus x = <fn y => ~x + y>;
val plus = Fn : <int> -> <int  -> int>

-| val w = <fn y => ~(plus <y>)>;
val w = <(fn a => (fn b => a %+ b))> : <int  -> int  -> int >
```

Note how `plus` splices its argument in a context where `y` is locally bound. Yet if we use plus in another context where another `y` is bound, MetaML does not confuse the two.

MetaML also statically scopes free variable occurrences in code templates. This results when a variable is used in a level greater than the level where it was bound.

```
-|  fun f x y = x + y - 1;
val f = fn  : int -> int -> int

-| val z = <f 4 5>;
val z = <%f 4 5> : <int>

-| let fun f x y = not x andalso y in run z end;
val it = 8 : int
```

Note how the free variable `f` in the code template `z` refers to the function `f:int -> int -> int`, which was in scope when the template was defined, and not the function `f:bool -> bool -> bool` which was in scope when `z` was run. The code pretty printer places the percent-sign (%) in front of `f` in the code template `z` to indicate that this is a statically bound object-variable.

The `run` operator in MetaML transforms a piece of code into the program it represents. It is useful to think of `run` as indicating the composition of run-time compilation with execution. In the example below, we first build a generator (`power_gen`). Apply it to obtain a piece of code (`power_code`). Run the code to obtain a function (`power_fun`). And then apply the function to obtain an answer (125).

```
-| fun power_gen m =
     let fun f n x = if n = 0 then <1> else <~x * ~(f (n-1) x)>
     in <let fun power x = ~(f m <x>) in power end> end;
val power_gen = fn  : int -> <int -> int>

-| val power_code = power_gen 3;
```

```
val power_code =
<let fun power x = x * x * x * 1 in power end> : <int -> int>

-| val power_fun = run power_code;
val power_fun = fn  : int -> int

-| power_fun 5;
val it = 125 : int
```

In MetaML we use run to move from one stage to the next. Because it is legal to use the anti-quotation under a lambda-binding, there is the possibility that run may be applied to variables that will not be bound until some later stage. For example:

```
-| val bad = <fn x => ~(run <x>) + 4>;
```

will cause an error because the object-variable x will not be bound until the whole piece of code is run, and then finally applied. This kind of error just does not occur in normal programs, and complicates both the semantics and type systems of homogeneous meta-systems. There has been much thought put into devising type-systems which will disallow such programs [74, 49, 77].

MetaML has one more interesting operator lift. It evaluates its argument to a constant (any value not containing a function), and produces a piece of code that represents that value. For example:

```
-| lift (4 + 5);
val it = <9> : <int>

-| lift (rev [1,2+3]);
val it = <[5,1]> : <int list>
```

In MetaML the angle brackets, the escapes, the lifts, and the run operator are staging annotations. They indicate the boundaries within a MetaML program where the program text moves from meta-program to object-program. The staging annotations in MetaML are placed manually by the programmer and are considered part of the language. In MetaML the staging annotations have semantic meaning, they are part of the language definition, not just hints or directions to language preprocessors.

It has been argued that manually staged programs are hard to write, and are much larger than their unstaged counterparts that could be input into a partial evaluation system, thus saving the user a lot of work. With the advent of modern meta-programming systems with quasi-quote staging annotations it remains to be seen if this argument still holds. Our experience has been that manually annotated programs are (within a few percent) the same size as their unstaged counterparts. Still it would be nice to have the benefits of both systems.

8 Partial Evaluation

Off-line partial evaluation is an automatic staging meta-system. Consider a simple partial evaluation function PE. It takes the representation of a program with one static parameter and one dynamic parameter and returns an answer. It analyses this program and automatically produces an annotated program which when given the static parameter as input produces a representation of the function from dynamic parameter to answer. From this description we can infer that `PE` should have the type `(<s -> d -> a>) -> (<s -> <d -> a>>)` For example consider the program:

```
-| val ans = PE <fn s => fn d => d + (s + 3)>
ans = <fn s => <fn d => d + ~(lift (s+3))>> : <int -> <int -> int>>
```

which evaluates to a new two-stage program. This step is called binding time analysis. When this two-stage program is `run`, and then applied to `5` (the specialization stage) it produces a new specialized program:

```
-| val special = (run ans) 5;
val special = <(fn a => a %+ 8)> : <int -> int>
```

Note how the static addition (`s+3`) is performed at specialization time.

The integration of a function like `PE` into a manually staged language could provide the benefits of both manually and automatically staging a program. Since both partial evaluators and staged languages exist, it remains an interesting engineering problem to integrate the two.

9 Intensional analysis of code

MetaML, as originally designed, allowed the construction and execution (see Section 7) of code. Observation of the structure of code, and its decomposition was not originally supported, and has never been included in any of the formal property studies we have performed.

Consider the problem of implementing the `PE` function from the previous section. What tools do we need? Its obvious `PE` needs to observe the structure of its argument. Is the argument an lambda-abstraction (`<fn s => ...>`), is it an application (`<s + 3>`), is it a constant (`3`), or is it a variable (`s`)? If it is a variable, then is it statically scoped like `%+`, or is it an object-bound variable like `d`?

In MetaML, code is implemented as an abstract datatype. The interface to this datatype includes only the quasi-quote template notation, and the function `run`. Internal to the interpreter, code is implemented as an algebraic datatype along the lines of the `Exp` datatype used in Section 5. This internal representation is not visible to the programmer, as we wish to insulate the programmer from the details of the internal structure, allowing him to construct object programs using the same syntax (via the template mechanism) used for meta-programs.

In order to allow intensional analysis of code we must either make explicit the internal representation, or supply some other interface to the abstract code type. The actual MetaML implementation is far too complicated to deal with in this short section, but if we consider the `Exp` datatype as defined in Section 5 we can illustrate the key ideas.

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

Suppose we have built a quasi-quotation mechanism that internally stores templates as `Exps` (`<4>` means `(Constant 4)`, and `< ~f ~x >` means `(Apply f x)`, and `<fn x => x>` means `(Abs "x" (Variable "x")` etc.). We could support an interface to this abstract type that did not reveal the details of the actual implementation type `Exp` by supplying constants with the following types, which we display below in a Haskell-style language.

```
isApp :: <b> -> M(<a->b>,<a>)                   gensym :: M (Sym a)
isAbs :: <a->b> -> M(<a> -> <b>)                var :: Sym a -> <a>
isConst :: <Int> -> M Int                       instance Eq (Sym a)
isVar :: <a> -> M (Sym a)
```

Here `M` is a monad with failure (`fail`) and plus operation `(++)` such that `(++)` is associative and `fail` is its unit. It might help to think of `M` as the `Maybe` type constructor with additional structure. Under this interpretation, the expression `isConst <3>` evaluates to `(Just 3)`, the expression `isApp < ~f ~x >` evaluates to `(Just (f,x))`, but `isApp <3>` evaluates to `fail`. In each case the operators either succeed, producing as result some sub-structure of the object-code matched against, or they fail. The operator (++) is used to try a sequence of potentially failing expressions one after another.

The operators `isAbs`, `isVar`, `gensym`, and `var` are used to deal with variables and binding operators such as lambda-abstraction. Because the actual name of a bound variable does not matter, we would like our implementation to respect this constraint.

The type constructor `Sym` is an abstract type which is a member of the `Eq` class. Its only operators (beside equality testing with (==)) are `gensym` and `var`. The operator `Abs` works in conjunction with the operators over `Sym`bols to analyze the structure of lambda-abstractions and variables. The operator `isAbs`, when applied to a piece of code which is an object-level abstraction, returns a meta-level function. Given an "argument", this meta-level function re-produces the body of the abstraction with all occurrences of the bound variable replaced by the "argument". Under this scheme the user cannot observe the actual names of object-variables. The operator `gensym` produces new symbols that can only be observed through the `Sym` abstract type interface. This interface is designed to avoid harmful use of variables.

For example to test if some code term `x` matches the code value `< \f -> f 5 >` one writes as follows. We use a Haskell-like syntax because of its excellent support for monadic computation using the `do` and `return` operators.

```
test :: <(Int -> a) -> a> -> M Bool
test x =
  do { g <- isAbs x
     ; y <- gensym
     ; (fpart,arg) <- isApp (g (var y))
     ; n <- isConst arg
     ; z <- isVar fpart
```

```
  ; return ( n==5 && y==z )
  }
```

Using the do notation, a successful match binds the elements to the left of the arrow (<-), causing the evaluation to proceed to the next clause. A single failure causes the complete do expression to fail.

Not the use of gensym and var to generate new object-level variables. A key observation is that all production of variables happens within the monad M. One can write an equality function that compares two pieces of code for syntactic equality (modulo alpha-equivalence) as follows:

```
eq :: <a> -> <a> -> M Bool
eq x y =
  (do { n <- isConst x; m <- isConst y; return(n==m)) ++
  (do { (f,x) <- isApp x; (g,z) <- isApp y; b1 <- eq f g
      ; b2 <- eq x z; return (b1 && b2) }) ++
  (do { f <- isAbs x; g <- isAbs y; x' <- gensym
      ; eq (f (var x')) (g (var x')) }) ++
  (do { x' <- isVar x; y' <- isVar y; return(x'==y')}) ++
  (return False)
```

The purpose of the monad is to put structure on the possibility of failure, and to delimit the scope where object-bound variables may live. The problem is that the use of gensym may produce free variables which are never eliminated. For example:

```
do { x <- gensym
   ; return (var x)
   }
```

returns a variable which has no binding location. One saving grace here is that the expression above has type M <a>, not <a>. One purpose of the monad is to prevent the escape of such variables. Thus an important open question is "How do you get out of the monad?"

An operation with type M a -> a is clearly too general, since it might allow the abstract Sym values to escape the monadic computation. One solution might be to use a qualified type, qualifying a as follows:

```
run :: NotCode a => M a -> Maybe a
```

If a cannot contain code then things will be safe. But clearly one often wants to write code analyzers that return code. How to accomplish this is an interesting open question.

This approach can even be lifted to complex object-code expressions such as case-expressions. For example:

```
isCase :: <a> -> M([Exists x . (<x -> b>,<x> -> <a>)],<b>)
```

Applying isCase to an object-level case, if it succeeds returns a pair. The first part of the pair is a list corresponding to each arm of the case, and the second part of the pair corresponds to the argument of the case. For example:

```
test = <case g x of
          Cons (x,xs) => x + sum xs
        | Nil () => 0>

 do { ([(c1,f),(c2,h)],arg)  <- isCase test
    ; ...
    }
```

Each item in the list, corresponding to the arms of the case, is a pair. The first element corresponds to the constructor function, and the second element (as in the `isAbs` example) is a meta-function that could be used to obtain the right-hand-side of the arm.

For example the binding of `(isCase test)` in the example above, binds the meta-variables whose types and bindings are indicated in the table below:

```
arg :: <[Int]>                            arg       --> <g x>
c1  :: <(Int,[Int]) -> Int>               c1        --> <Cons>
c2  :: <() -> Int>                        c2        --> <Nil>
f   :: <(Int,[Int])> -> <Int>             f <(a,b)> --> <a + sum b>
h   :: <()> -> <Int>                      g <()>    --> <0>
```

Clearly, there remains considerable work to extend and polish this proposal so that it would be robust, and apply to a realistic size object-language. In the case of MetaML one needs to deal with statically scoped variables like `%length`, and deal with typing issues for constructors like `Tuple`.

10 Higher level interfaces to code analysis

Quasi-quoted templates were introduced as an abstract way to construct code, but can also be used as patterns against which code can be matched. This supplies a more abstract interface to intensional analysis of code than the interface in the previous section. If a quasi-quoted template is used as a pattern, then the anti-quoted variables in the template are meta-variables which are bound to object-code fragments during pattern matching. We have built a simple prototype implementation of this into MetaML.

```
-| fun decompose <(~x,~y)> = (x,y)
     | decompose _ = error "bad"
val decompose = Fn  : <('b  * 'a )> -> (<'b > * <'a >)

-| decompose <(3,5)>;
val it = (<3>,<5>) : (<int> * <int>)

-| decompose <(fn x => (x,x)) 5>;
Error: bad
```

Several engineering problems remain to be solved to fully use templates as patterns. Pattern templates use antiquotation to indicate the meta-variables destined to be bound when the pattern matches. Most templates include enough context (in the form of concrete-syntax) to disambiguate which kind of object-level term is being matched. Two cases remain problematic: constants and sequence-based constructs. If one wants to match a particular constant, it is straightforward to just wrap it in quoting brackets to make an object-language pattern that matches code containing that constant.

```
fun is_five < 5 > = true
  | is_five _     = false
```

But, how does one write a pattern that matches against all code constants, and when doing this binds a meta-variable to the value of the constant in the code matched against? This is easy to do using algebraic datatypes. Using the algebraic datatype `Exp` introduced in Section 5, we write:

```
fun  f2 (Constant n) = n
```

Using templates, there is not enough context inside the brackets to distinguish constants. For example `< ~n >` matches any code, not just constants.

A similar problem occurs when matching against language constructs which are sequence-based; i.e., consist of an unbounded number of similar sub-constructs. Examples include tuples and let-expressions with multiple bindings.

For example, the pattern below matches all tuples with three components.

```
fun g <(~x,~y,~z)> = 3
```

What pattern will match all tuples? Again this is easy using an algebraic datatype approach. The pattern `(Tuple xs)` matches all tuples, and the meta-variable `xs` is bound to a *list* of sub-expressions. Typing such an interface is also problematic. A robust template-based approach to pattern matching against code will need to address these issues.

To use quasi-quoted templates to match against binding constructs such as function abstraction and case-expressions requires solving the same problems of variable escape that we saw in Section 9.

11 Quasi-quotes distinguish stages

A valuable property of quotes and anti-quotes is that they serve as staging annotations that distinguish the meta-program from the object-program. Such annotations are important because they remove ambiguity. Consider the program transformation that replaces exceptions with `Either` types[1] in a Haskell-like object-program. The meta-program below defines the *magic-brackets* (`[| _ |]`) which specify the transformation.

```
[| x |]                  = Inl x
[| lambda x . e |]       = Inl (lambda x . [| e |])
[| e1 e2 |]              = match [| e1 |]
                               (lambda v1 . match [| e2 |] v1 Inr) Inr
[| raise |]              = Inl Inr
[| try e3 |]             = Inl (lambda h . match [| e3 |] Inl  h)
[| let x = e4 in e5 |] = match [| e4 |] (lambda x .[| e5 |]) Inr
```

[1] `data Either a b = Inl a | Inr b`

This transformation is a meta-program. Meta-programs described in this format are typically found in research papers. It manipulates two separate object-languages. It does a case analysis over the structure of the *source object-language*, and builds an element of the *target object-language*. It contains both object-variables and meta-variables. Can you tell which variables are which? If you are a skilled programming language researcher you can probably use your past experience, and context to figure out which is which. But as an unambiguous algorithm it leaves much to be desired.

Staging annotations remove this ambiguity. Below we use MetaML-like staging annotations.

```
[| < %x > |]                    = <Inl ~x>
[| <lambda x . ~(e <x>)> |]     = <Inl (lambda y . ~[| e y |])>
[| < ~e1 ~e2 > |]               = <match ~[| e1 |]
                                       (lambda v1 . match ~[| e2 |] v1 Inr)
                                      Inr>
[| <raise> |]                   = <Inl Inr>
[| <try ~e3> |]                 = <Inl (lambda h . match ~[| e3 |] Inl  h)>
[| <let x = ~e4 in ~(e5 x)> |]  = <match ~[| e4 |]
                                         (lambda y . ~[| e5 y |]) Inr>
```

Note how the quotations (< and >), delimit the object-code, and how the anti-quotations (~) indicate the meta-computations that compute object-code sub-terms. It is possible to distinguish object-variables from meta-variables by noting whether their binding occurrence is within quotations. Several unresolved problems remain. The percent (%) in front of `x` is an attempt to indicate that this pattern should match only variables, and that the meta-variable `x` should be bound to the object-variable the pattern matches. The other problem concerns manipulating binding occurrences of object-variables (like in `lambda`). In the example above we have used a solution which is similar in feel to the use of the function `isAbs` of Section 9. Patterns matching lambdas bind variables which are functions from terms to terms. This is discussed in further detail in Section 13.

12 Templates and heterogeneous meta-systems

In the MetaML implementation the object-language and the meta-language share the same fixed representation, the same parser, and the same type checker etc. The representation of both the meta and object languages was chosen *once* by the MetaML system's developers.

In a heterogeneous system, such templates become problematic. Instead of a single, fixed object-language, the user can define multiple object-languages. Thus neither a fixed strategy, nor shared representation is possible. In a heterogeneous system meta-programmers will have to develop their own representations.

A good step in this direction is addressed by the *conctypes* of Annika Aasa [1–3]. She describes how an algebraic datatype facility can be extended to one that allows concrete syntax descriptions. In her work a fixed parsing strategy is embedded into the extension, and the users write *conctype* specifications which behave like both a grammar specification, and an algebraic datatype definition.

Integrating this with a system that treats object-bound variables sensibly, handles the ambiguity problems of constants and sequence-based constructs, and that allows object-language typing as well remains an open problem.

13 Manipulating binding constructs

As we have seen, meta-programs are particularly difficult to write correctly if they must manipulate object-terms that have a notion of statically scoped variables. There are two related problems. The first occurs when generating programs that include new binding occurrences. When generating a new binding construct, some meta-systems use a *gensym* operator that produces a new unique name that will never be introduced again. This strategy is used to avoid inadvertent name capture of free variables in the scope of the binding construct.

Such a solution is awkward to use since it separates the introduction of the name from its binding, and prescribes a stateful implementation from which new names can be selected. In Section 7 we discussed how a quasi-quote template based interface can hide this gensym process from the programmer.

The second problem occurs when deconstructing programs that include binding occurrences. The exact representation of the bound variable is generally uninteresting, and the meta-program must make subtle "administrative changes" to the object-program so that it maintains its original "meaning". Such names also need special attention to prevent the escape of bound variables from their scope. In our discussion of intensional analysis of code, the handling of variables was a major complication.

A representation where the meta-program can be freed from the responsibility of concretely representing bound variables and their names, yet which facilitates correct program manipulations would be quite useful indeed!

An interesting representation technique based upon an idea that goes back, at least, to Alonzo Church [16] exploits the binding mechanism of the meta-language to implement the binding mechanism(s) of the object-language. We saw an inkling of this in the use of our function `isAbs`.

To illustrate the elegance of the approach contrast the definition of `Term` and `Term'` below. In `Term'` we represent the object-language lambda abstraction (`Abs'`) using the meta-language function abstraction. Note, in our examples, how the `id'` term and the `apply'` term are represented by applying the `Abs'` constructor to a meta-language function.

```
data Term = App Term Term
          | Abs String Term
          | Const Int
          | Var String
-- \ x -> x
id = Abs "x" (Var "x")
-- \f -> \ x -> f x
apply = Abs "f" (Abs "x"
        (App (Var "f") (Var "x")))
```

```
data Term' = App' Term' Term'
           | Abs' (Term' -> Term')
           | Const' Int
-- \ x -> x
id' = Abs' (\ x -> x)
-- \f -> \ x -> f x
apply' =
  Abs' (\ f ->
          Abs' (\ x -> (App' f x)))
```

The higher-order abstract syntax representation (HOAS) `Term'` is elegant in that a concrete representation for variables is not needed, and that it is not necessary to invent unique, new names when constructing lambda-expressions which one can only hope do not clash with other names. Unfortunately, there are drawbacks as well. HOAS works fine for constructing statically known representations, but quickly breaks down when trying to construct or observe a representation in an algorithmic way. There are four problems that we illustrate in the examples below:

– **Opaqueness.** HOAS bindings are "opaque". We cannot pattern match or observe the structure of the body of an `Abs'`, or any object-level binding, because they are represented as functions in the meta-language, and meta-level functions are extensional. We can observe this by casting our `Term'` example above into a simulated evaluation session in Haskell, and noticing that `id'` prints as `Abs' fn`.

```
id' = Abs'(\ x -> x);

Main> id
Abs' fn
```

– **Junk.** HOAS admits *junk* [13]. I.e. there are terms in the meta-language with type `Term'` that do not represent any legal object-program. Consider the example:

```
junk = Abs'(\ x -> case x of
                     App' f y -> y
                   ; Const' n -> x)
```

No legal object-abstraction behaves in this way, analyzing its bound variable.

– **Loss of expressivity.** Using HOAS, there exist meta-functions over object-terms that cannot be expressed. Consider writing a `show` function for `Term'` that turns a `Term'` into a `string` suitable for printing. What legal meta-program value do we use for `?v` ?

```
show (App' f x) = (show f) ++ " " ++ (show x)
show (Const' n) = toString n
show (Abs' g) = "\\ "++ ?v ++ " -> " ++ (show (g ?v))
```

Since `g` is a `Term'` to `Term'` function, we need some sort of "variable" with type `Term'`, to which we can apply `g`. Unfortunately, no such thing can be created (this was solved in Section 9 by the use of `gensym` and `var`). There are other "tricks" for solving this problem [26], but in the end, they only make matters worse.

– **Latent effects.** HOAS delays non-termination and other effects. This problem is especially obvious in a strict language. Computational effects of the meta-language are introduced into the purely syntactic representation of the object language. Even worse, the effects are only introduced when the object-term is observed. If a term is observed multiple times, it causes the effects to be introduced multiple times.
For example, because functions delay computation, a non-terminating computation producing a `Term'` may delay non-termination until the `Term'` object is observed. This may be arbitrarily far from its construction, and can make things very hard to debug. Consider the function `bad`.

```
bad (Const' n) = Const' (n+1)
bad (App' x y) = App'(bad x)(bad y)
bad (Abs' f) = Abs'(\ x -> diverge(bad (f x)))
```

`bad` walks over a `Term'` increasing every explicit constant by one. Suppose the programmer made a mistake and placed an erroneous divergent computation in the `Abs'` clause. Note that `bad` does not immediately diverge.

We believe the trick to representing object-level binding is to use a binding mechanism of the meta-language. I.e. higher-order abstract syntax. The catch 22 - the function (λ) abstraction mechanism is not the right binding mechanism. And, function abstraction is often the *only* binding mechanism the meta-language has. The solution is to introduce a *new* binding mechanism.

Ten years ago Dale Miller informally proposed a simple and elegant extension to SML for supporting higher-order abstract syntax [47] using a new kind of "variable binding" we call *object-level binding.* His proposal illustrated the simplicity and elegance of HOAS as a means for representing object-languages but left the formal semantics, the typing of such a system, and demonstration of its practical usefulness as open problems. We illustrate a variation of Miller's extension as a basis for representing object-languages with binders. We have presented a simple operational semantics for our variation of Miller's language [59, 58], where we established that this operational semantics is sound and adequate with respect to a simple, natural reduction semantics. We outline here how object-level binding works.

Consider the lambda calculus example once again. We will use the infix operator (`a => b`) as both a meta-language term constructor (for our new binding mechanism) and as a type constructor. Using this we define the algebraic datatype `Term2` analogous to `Term'` but using the new object-level binding construct rather than function abstraction.

```
datatype Term2 = App2 Term2 Term2 | Abs2 (Term2 => Term2) | Const2 Int
```

Terms of type `(a => b)` are introduced using the meta-language construct for object-binding introduction. For example: `(#x => App2(#x,Const2 0)) :: (Term2 => Term2)`. Here we use the convention that hashed variables (`#x`) denote object-level variables to distinguish them from meta-level variables. This new operator has the following properties:

- **Evaluation under binding.** Latent effects and junk arise because the body of an object-binding is a computation (i.e. a suspended function), rather than a constant piece of data. To solve both these problems, object-level binding evaluates under the binding operator `=>`. Below are two attempts to construct an object-language program:

  ```
  Abs' (\x -> bottom)                    Abs2 (#x => bottom)
  ```

  The expression on the left (in the `Term'` language) uses a meta-language binding mechanism (λ abstraction). It succeeds in representing an object-language program which obviously has no meaning. The expression on the right (in the `Term2` language), however, does not represent any object-language program, since by our semantics, evaluation proceeds "under" the object-level binding operator and thus never terminates. Note that the effect on the left has seeped into the object-language program representation (junk), while on the right non-termination occurs before the object-language program is constructed and thus is never present in the object-language program itself.
  In a lazy setting, the similar problems occur, but they manifest themselves differently. Pattern matching repeatedly against `Abs' (\x-> e)` will cause `e` to be re-computed. But pattern matching against `Abs (#x => e)` will only evaluate `e` once.
  An intuitive way to think about object-level binding, is to think of it having an underlying first order implementation supplied with a rich interface. Inside this first order implementation object-level bindings are represented by pairs. A construction like: `(Abs2 (#x => e)) :: Term2` is translated into the underlying representation (e.g. `Term`) by using a *gensym* construct to provide a "fresh" name for the required object-bound variable:

  ```
  let y = gensym () in Abs y ((\ #x -> e)(Var y))
  ```

  It is important to emphasize that both the *gensym* and the underlying first-order implementation are hidden from the user. Earlier, we criticized the use of a *gensym* construct since such a construct is stateful, and hence forces our meta-language to be stateful, rather than purely functional. Fortunately, the statefulness of this use of gensym can only be observed if one can observe the value of the name produced by gensym. If the interface to the hidden underlying implementation allows access to variables only in restricted ways, it is possible to mask this statefulness.

- **Higher-order pattern matching.** To solve the problem of opaqueness the new binding mechanism should support higher-order pattern matching. We use a higher-order pattern when we pattern match against a constructor like

`Abs` which takes an object-level binding as an argument. Like all patterns, a higher-order pattern "binds" a meta-variable, but the meta-variable bound by a higher-order pattern does not bind to an object-term, but instead binds to a meta-level function with type `Term2 -> Term2`.
To illustrate this consider the rewrite rule **f** for object-terms `Term2`, which might be expressed as: **f** : `Abs2(#x => App2` e' `(Const2 5))` $\rightarrow$ (e'[0/`#x`])
Here, the prime in e' indicates that e is a meta-variable of the rule, and e'[0/`#x`] indicates the capture free substitution of `(Const2 0)` for `#x` in e'. The subtlety that e' might have free occurrences of `#x` inside is what higher-order pattern matching makes precise. The bound meta-variable introduced by a higher-order pattern is a function from `Term2 -> Term2`, and this function behaves like the function $\lambda y.e'[y/$`#x`$]$.
We make this idea concrete by extending the notion of pattern in our meta-language. Patterns can now have explicit object-level abstractions, but any pattern-variables inside the body of an object-level abstraction are higher-order pattern-variables, i.e. will bind to functions. Thus the rewrite rule **f** can be specified as follows:

```
f (Abs2(#x => App2(e' #x)(Const2 5))) = e'(Const2 0)
```

In this example the meta-function `f` matches its argument against an object-level abstraction. The body of this abstraction must be an application of a term to the constant 5. The function part of this object-application can be any term. This term may have free occurrences of the object-bound variable (which we write as `#x` in the pattern, but which can have any name in the object-term it matches against). Because of this we use a higher-order pattern `(e' #x)` composed of an application of a meta-variable to an object-bound variable. This application reminds the user that `e'` is a function whose formal parameter is the object-bound variable `#x`. If the underlying implementation is first order (like `Term`), patterns of this form (in `Term2`) have an efficient and decideable implementation, which one could visualize as follows:

```
f (Abs x e) = let e' y = subst [(x,y)] e  in  e' (Const 0)
```

– **Loss of expressivity.** Many simple programs can be expressed simply by using the meta-language construct for object-binding (`x => e`), to introduce object-variables. For example the identity function over `Term2` can be expressed as:

```
identity (App2 f x) = App2 (identity f) (identity y)
identity (Const2 n) = Const2 n
identity (Abs2(#x => e' #x)) = Abs2(#y => identity (e' #y))
identity (x @ #_) = x
```

The fourth clause of the `identity` function illustrates the pattern matching construct for object-level variables introduced by the `x => e` operator. Similar to the `isVar` function from Section 9, a pattern `(x @ #_)` matches against any object-level variable and binds the meta-variable `x` to the `Term` matched.
It is sometimes necessary to introduce a new object-variable simply as a place holder, and to then eliminate it completely from a computation. This

was the problem with the `show` function from Section 13, and the reason for the `gensym` and `var` functions in Section 9. The solution to this problem is a new language construct *discharge*. The construct `(discharge #x => e1)` introduces a new object-level variable `(#x)`, whose scope is the body `e1`. The value of the discharge construct is its body `e1`. The body `e1` can have any type, unlike an object-level binding `(#x => e2)`, where `e2` must be an object term.
In addition, discharge incurs an obligation that the variable `(#x)` does not appear in the value of the body `(e1)`. An implementation must raise an error if this occurs.
For example consider a function which counts the number of `Const2` sub-terms in a `Term2`.

```
count :: Term2 -> Int
count (Const2 _) = 1
count (App2 f x) = (count f) + (count x)
count (Abs2(#x => e' #x)) =  discharge #y => count (e' #y)
count #_ = 0
```

Note how the fourth clause conveniently replaces all introduced object-bound variables with 0, thus guaranteeing that no object-variable appears in the result. The obligation that the variable does not escape the body of the discharge construct may require a run-time check (though in this example, since the result has type `Int`, no such occurrence can happen).

14 FreshML – an alternative approach to binding

In contrast to higher-order abstract syntax approaches, which use the binding mechanisms of the meta-language to represent binding constructs in the object-language, an alternative approach has recently emerged in work of Andrew Pitts and Murdoch J. Gabbay [27, 28, 63].

Their work is based upon Frankel-Mostovsky set theory (which dates back to the 1930's). They use this theory to model and reason about datatypes that represent first order terms (with variable names) modulo α convertibility. FreshML [27] introduces several language constructs for correctly manipulating such terms, and a type system that ensures that one is indeed manipulating α equivalence classes. Intuitively, their approach resembles a *nameful* version of the well-known "nameless" de Bruijn style of representing binding constructs[21]. The main advantage for the programmer is that the burden of reasoning about (as well as complicated algorithmic interface to) nameless terms is cast into a more user-friendly setting.

We shall present an example of such a language (with slightly modified syntax from [63]), to demonstrate the concepts.

```
data LambdaTerm = Abs of [atm]LambdaTerm
                | App of LambdaTerm * LambdaTerm
                | Var of atm
```

The expression `(a.Var a)` of type `[atm]LambdaTerm` denotes an atom abstraction. One can think of an atom abstraction as consisting of a pair of an *atom* (representing an object-variable name) and a lambda term that may contain the name as a subterm.

The language also contains constructs for ensuring "freshness" of atoms with respect to bindings inside terms.

```
val Apply = Abs (x => (Abs (y => App (Var x, Var y))));
```

The term `Apply :: LambdaTerm` represents the well-known lambda calculus term $\lambda x.\lambda y.x\ y$. We use the `a => e` construct to build an atom abstraction. The construct introduces a *fresh* atom which is bound to the meta-variable a and the whole construct evaluates to an atom abstraction with type `[atm]t` if `e :: t`. The freshness construct assures that the actual name of the atoms in atom abstractions are irrelevant and can be freely transposed to obtain equivalent values.

The language also supplies facilities for pattern-matching against atom abstractions. Consider the following two examples

```
fun right (Abs (a. App(x,y))) = Abs (a. App(y,x))
fun wrong (Abs (a. App(x,y))) = App(x,Abs (a . y))
```

The function `right` analyzes a `LambdaTerm` abstraction. The pattern matching is similar to taking apart a pair. However, the type system ensures that the result of the function is a lambda term which does not depend on the actual name of the object-variable `a`.

The second function, `wrong`, is rejected by the type checker because the resulting term may indeed result in terms belonging to different α equivalence classes under different transpositions of the atom `a` (this happens when the atom, denoted by `a` occurs as a subterm of `x`).

Both FreshML and HOAS seek to model classes of α equivalent terms. The FreshML approach has the advantage of a well-formalized set theoretical foundation as well as supporting first-order, datatype-style induction over its terms.

Both HOAS and the FreshML approach lack efficient and robust implementations in a programming language setting with which to experiment.

Open problems in this area include the further development of operational semantics and the development of a calculus for performing equational reasoning.

15 Manipulating typed object-programs

If meta-programmed systems are to gain wider acceptability, especially run-time code generators, then systematic efforts to guarantee safety properties of the object-programs produced must be developed. One example of this is type-safety.

The key to tracking the type-safety of object-programs, by static analysis of the meta-programs that manipulate them, is the embedding of the type of the object-program in the type of object-terms. In MetaML this is straight forward to accomplish. As explained above, a new type constructor (`< t >`) is used to give every object-level term a *code* type. Because MetaML is a homogeneous system, the type of object-level terms (the `t` in `< t >`) can be captured by the same type system used to capture types in the meta-language. The quasi-quotation mechanism can can be exploited to infer the type of the object-language fragments within the meta-language, since both languages have the same structure, and are typed by the same type system. This is a principal design feature of MetaML.

```
-| val x = <3 + 2> ;
val x = <3 %+ 2> : <int>

-| val code = <show ~x> ;
val code = <%show (3 %+ 2)> : <string>
```

Note how `x` has type `<int>` (pronounced *code of int*), and `code` has type `<string>`. In MetaML the type correctness of a meta-program guarantees the type correctness (as well as the syntactic correctness) of the object-programs it manipulates [74].

16 Polymorphism and Staging

In this section we discuss the interaction of staging with polymorphism. In Hindley-Milner style type inference, `let` bound variables can be generalized to have polymorphic types. The introduction of staging introduces a new possibility for generalization. Should templates be a new generalization point? Should homogeneous meta-systems be designed such that:

```
< \ x -> x > :: all a . < a -> a>
```

or, should code templates be given rank-2 polymorphic types?

```
< \ x -> x > :: < all a . a -> a>
```

This question has been studied in the context of inferring static types for dynamic documents by Mark Shields in his Ph.D. thesis [71]

In his thesis Shields also studies the interaction of staging with implicit parameters [45]. This is interesting because implicit parameters in a staged context could be interpreted as parameters that will be supplied by the environment that the generated code will execute in. The interaction of staging, implicit parameters, and polymorphism that is discussed there is quite interesting. For example what type should be assigned to the template below:

```
<fn x => ?y x> :: ??
```

The design issues implicit in this problem are quite complex, and beyond the scope of this short synopsis. I refer you to the thesis for full details.

17 Dependently typed meta-programs

It is not unusual to have meta-programs which when given different inputs of the same type, produce object-programs with different types. This behavior is usually typed by a dependent type system. For example we might like to write the following meta-program using MetaML style quasi-quotations.

```
f 0 = < () >
f n = <(1, ~(f (n-1)) )>
```

The result of applying `f` to a few values is given below:

```
f 0 -->  < () >
f 1 -->  < (1, ()) >
f 2 -->  < (1,(1,())) >
```

Each of which has a different type

```
 f 0 :: < () >
 f 1 :: <(Int,())>
 f 2 :: <(Int,(Int,()))>
```

The type of `f n` depends upon the value of `n`. One normally indicates this using a Pi type ($\Pi x : t \,.\, s$). A Pi type is like a function arrow ($t \rightarrow s$), only the type to the right of the dot (s) can depend upon the value of the bound variable (x) to the left of the dot. We can give the function `f` the dependent type: $\Pi n : \mathit{Int} \,.\, \texttt{<}g\ n\texttt{>}$ where the function g is a function from `Int` to types.

```
g 0 = ()
g n = (Int,g (n-1))
```

In a language with dependent types, programs with dependent types must be given explicit type signatures. These signatures are notoriously difficult to discover and to reason about, so dependent types are not normally used in normal programming languages. Unfortunately, useful meta-programs with dependent types occur all the time. It is an open problem to discover how to mix dependent types and meta-programming in usable manner.

One approach to writing dependently typed meta-programs is to delay some typing till run-time [72]. In this approach the code type constructor (`< _ >`) is no longer a type constructor, but simply a type (`<>`). As in MetaML, code is constructed by templates, but the representation of code at run-time carries not only information about the structure of the code, but also information about its type. Each quasi-quoted template (without anti-quotations) can be statically typed, but is assigned the type `<>`.

```
-| val good = <length [1,2]>
good = <%length [1,2] > :: <>

-| val bad  = <5 0>
Error: The sub term: "5" is not a function.
```

Any use of code, either via `run`, or via splicing into another piece of code (via anti-quotation), must meet a run-time check that the code is used in a type meaningful context. Thus templates can be statically typed, but splicing and run must wait until run-time for type checking. For example consider an algebraic data type representing the syntax for a simple functional language:

```
data Term = Cnat Nat | Cbool Bool | Var String | Abs String Term
          | App Term Term | Cond Term Term Term | Eq | Plus
```

It is possible to write an interpreter `interp :: Term -> Env -> <>` where the code constructed is dynamically type checked as it is constructed.

```
interp (Cnat n) env = < n >
interp (Cbool b) env = if b then <True> else <False>
interp (Var s) env = env s
interp (Abs x e) env = <\ y -> ~(interp e (extend e x <y>))>
interp (App f x) env = < ~(interp f env) ~(interp x env) >
interp (Cond x y z) env =
  <if ~(interp x env) then ~(interp y env) else ~(interp z env) >
interp Eq env = <(==)>
interp Plus env = <(+)>
```

Thus `(interp (App (Cnat 5) (Cnat 0)) env)` causes a run-time type error, since the code fragment `<5>` is spliced into a context that requires a function. A similar kind of error can occur when `run`ning a piece of code. Thus, the term

```
(0 + (run (interp (Cbool True) env)))
```

causes a run-time error, even though the result of `interp` is a well typed program (of type `Bool`), because it is used in a context that requires an `Int`.

A rich dependent-type system could infer some kinds of errors like these statically. This remains an open problem.

18 Typing heterogeneous meta-programs

In heterogeneous systems, the introduction of a simple code type constructor is no longer possible, since the type system of the meta-language may be completely different from the type system needed to type the object-language. Nevertheless, the idea of typing every object-level term with a *type constructor* applied to an argument which encodes the type of the object-level term is a good one. We need to broaden the notion of a legal argument to a type constructor.

Kinds. Functional languages have used the notion of *kind* to make the same fine distinction on types, that types make on values.

type grammar	kind grammar	example types	example kinds

$t \to$ `Int`
 $\mid$ t `->` t
 $\mid$ $[t]$
 $\mid$ (t, t)
 $\mid$ `Tree` t
	$k \to$ `Star`
	 $\mid$ k `-k->` k
		`5 :: Int`
		 `(5,2) :: (Int, Int)`
		 `\x->x+1 :: Int->Int`
		 `[1,2,3] :: [ Int ]`
		 `Tip 4 :: Tree Int`
			`Int :k: Star`
			 `(Int, Int) :k: Star`
			 `Int -> Int :k: Star`
			 `Tree :k: Star -k-> Star`
			 `(->) :k:`
			 `Star -k->`
			 `(Star -k-> Star)`

In a typed language, types partition the value space into sets of values with similar properties. Kinds partition the type space into sets of types with similar properties. All types that fall in the partition of values have (for historical reasons) kind `Star`. Example kinds are given in the table above. Because type constructors construct types from types, we give type constructors higher-order kinds. We use `-k->` to distinguish the kind arrow from the type arrow, and we use `:k:` to distinguish the *has kind* relation from `::`, the *has type* relation.

For 20 years functional languages have supported extensible type systems. The user can add to the well formed types by defining new types or type-constructors by using algebraic datatype definition facilities. For example, in Haskell, one can add the type constructor `Tree` by writing:

```
data Tree a = Tip a | Fork (Tree a) (Tree a)
```

Such definitions add the new type constructor (`Tree`) to the grammar of well formed types, and new values such as `(Tip 5)` and `(Fork (Tip 1) (Tip 5))` to the well formed values.

The addition of extensible kinds as well as extensible types, is one way to attack the problem of object-code types. We will introduce a new kind, that will be an algebraic structure which models the type of object-terms. We will then use this new kind as an index to the object-term type constructor. By writing:

```
datakind T = TypConst | TypPair T T
          -- comment   TypConst :k: T,
          --           TypPair :k: T -k-> T -k-> T
```

We add the new kind `T` to the grammar of well formed kinds, and new types like `TypConst` and `(TypPair TypConst TypConst)` to the grammar of well formed types. But unlike types we are familiar with, types `TypConst` and `(TypPair TypConst TypConst)` have kind `T` rather than kind `Star`. Notice there are no program values with these types. Types with kind other than `Star` are used only to encode the object-level types of object programs.

Indexed Object Types. We have now enriched our type system with sufficient power to encode an object-language type constructor, which takes a T kind as a parameter, indicating the type of the object-term encoded. We call such a type an *indexed type*, where the index set is the set of well-formed type terms of some kind (like T).

In order to represent such types, we need to generalize our algebraic datatype definition facility. We do this by explicitly kinding the type (or type constructor) being defined, as well as the full type of each of the datatype's constructor functions. For example the familiar `Tree` definition: `data Tree a = Tip a | Fork (Tree a) (Tree a)` could be considered as a short hand for the the more verbose, and precise definition below:

```
data Tree :k: Star -k-> Star where
  Tip :: a -> Tree a
  Fork :: Tree a -> Tree a -> Tree a
```

Let's use this power to define an object-language for an expression language with products. Values of this algebraic datatype will have a type which is a type constructor applied to a kind `T`.

```
data Exp :: T -k-> Star where
  ExpConst :: Int -> Exp TypConst
  ExpPair :: Exp a->Exp b->Exp(TypPair a b)
  ExpPi1 :: Exp(TypPair a b) -> Exp a
  Pi2 :: Exp(TypPair a b) -> Exp b
```

$$\frac{x :: a \quad y :: b}{\texttt{ExpPair}\ x\, y\ :: \texttt{TypPair}\ a\, b}$$

$$\frac{x :: \texttt{TypPair}\ a\, b}{\texttt{ExpPi1}\ x\ :: a}$$

Object-languages are usually specified by their syntax (which specifies their form) and by type judgments (which specifies the membership of the set of well formed object-terms). The enriched algebraic datatype definition mechanism in cooperation with the extensible kind mechanism is a very powerful mechanism since it allows us to do both within a single framework. Note how the meanings of the type judgments on the right are captured by the type system of the meta-language in the types of the value constructor functions of the object-type `Exp`.

It is an open problem to construct a system where any well typed meta-program of type `Exp a`, is not only a specification of the object-term described, but also a proof that it has object-type `a`.

Functions over indexed-typed terms. Object-language representations with indexed types can be manipulated with meta-programs in the normal way. The type system of the meta-language will maintain the well typedness of the object-language terms. A simple example is given below.

```
data Value :k: (T -k-> Star) where
  ValConst :: Int -> Value TypConst
  ValPair :: Value a -> Value b -> Value(TypPair a b)

eval :: Exp a -> Value a
eval (ExpConst n) = ValConst n
eval (ExpPair x y) = ValPair (eval x) (eval y)
eval (ExpPi1 x) = case eval x of ValPair a b -> a
eval (ExpPi2 x) = case eval x of ValPair a b -> b
```

Here, we have introduced a second object-language we call `Value`. A `Value` is an indexed type with the same index (`T`) as `Exp`. The Meta-program `eval` transforms an `Exp a` into a `Value a`.

Meta-programs which manipulate indexed object-level terms need a richer type checking system than those used to type more traditional programs. There are two problems here. First, the function `eval` can be type checked only if it is given a polymorphically recursive type. In the clause:

```
eval (ExpPair x y) = ValPair (eval x) (eval y)
```

there are 3 occurrences of `eval`. In the terms

`eval (ExpPair x y)`	*eval has type*	`Exp(TypPair a b) -> Value(TypPair a b)`
`eval x`	*eval has type*	`Exp a -> Value a`
`eval y`	*eval has type*	`Exp b -> Value b`

Both the second and third instances cannot be reconciled with the first without giving `eval` a polymorphically recursive type. This has important considerations since type inference of polymorphically recursive functions is not, in general, possible. This means meta-programs which manipulate indexed object-level terms must be given explicit type signatures (like `eval :: Exp a -> Value a`).

The other problem concerns the way such functions are type checked. Usually, every clause in a multiple clause definition must have exactly the same type. With indexed object-level terms this rule must be relaxed. Study each of the clauses below on the left, and its type on the right.

```
eval (ExpConst n) = ValConst n                      --  eval :: Exp TypConst -> Value TypConst
eval (ExpPair x y) = ValPair (eval x) (eval y)      --  eval :: Exp(TypPair a b) ->
                                                              Value(TypPair a b)
eval (ExpPi1 x) = case eval x of ValPair a b -> a  --  eval :: Exp c -> Value c
eval (Pi2 x) = case eval x of ValPair a b -> b      --  eval :: Exp d -> Value d
```

Note that the type of the first (`Exp TypConst -> Value TypConst`) and second (`Exp(TypPair a b) -> Value(TypPair a b)`) clause are incompatible and cannot be unified, because the indices (`TypConst` and `(TypPair a b)`) are incompatible. The indexed *type constructors*, are identical, and with out the indexing, each of the clause gives the typing `eval :: Exp -> Value`. Because each clause differs only in its indices, and each indices is instance of the declared indices, and because the clauses are mutually exclusive and exhaustive, typing each clause independently should be a sound mechanism. It is an open problem to find the right combination of type system and theory to prove such a soundness property.

19 Indexed types and dependently typed programs

In the world of indexed object-terms, there is a special class of programs with dependent types, that are easy to understand and which can be handled quite efficiently: those programs where the dependency is on the index of a type, and not on its value. Such programs can be handled quite easily using a constrained (or qualified) type system.

In a constrained type system, a function f can have a constrained type of the form $\forall a. Ca \Rightarrow a \to b$. Interpret this as: for all types a which meet the constraint

Ca, the function f has type $a \to b$. The constraint *qualifies* the quantification. Such types are used in the type class system of Haskell [40, 41].

A simliar mechanism can be used to track dependencies on indexes. For example, consider an alternative to the `eval` function above, which rather than returning another kind of object-term like `Value`, returns an actual tuple. Note the qualified type of `eval2` below:

```
eval2 :: Encodes a b => Exp a -> b
eval2 (ExpConst n) = n
eval2 (ExpPair x y) = (eval2 x,eval2 y)
eval2 (ExpPi1 x) = case eval2 x of (a,b) -> a
eval2 (ExpPi2 x) = case eval2 x of (a,b) -> b
```

The function `eval2` has a dependent type, since the type of the result depends upon the value of the `Exp` it consumes. This dependency is actually weaker than it seems at first. The type of `eval2` depends only on the index parameter (`a` in the type `Exp a`), and not on the value. We can express this in the type of `eval2`. Interpret `eval2`'s type as, if the constraint `Encodes a b` is met, then `eval2` takes an `Exp a` as input and produces a value of type `b`.

```
rule Encodes :: T -> Star -> Rule
Encodes TypConst Int
Encodes (TypPair a b) (x,y) <- Encodes a x,
                               Encodes b y
```

`Encodes` is a predicate on types that can be given a simple Prolog-like definition. The constraint `Encodes a b` can be used to track the type dependency of the result of `eval2` on its input.

Tracking dependencies. There is a well understood theory of constraint propogation and management that can be brought to bear on meta-programs such as `eval2` above [40, 41]. Consider a defintion of a function `f`.

```
f x y = eval2 x + y
```

In a constrained type system, the constraint `Encodes a b` in the type of `eval2` is propagated by type inference into the type of `f`, which can be given by `f :: Encodes a Int => Exp a -> Int -> Int`

Further more, using the Prolog-like rules for constraint resolution, we can solve the constraint `Encodes a Int` by unifying `a` and `TypConst`. Thus we infer the new type `f :: Exp TypConst -> Int -> Int`.

A sound system built upon these ideas would have to answer many questions, and remains an open problem. Recapping, we discussed 4 ways in which we might deal with dependently typed meta-programs:

- First, require full dependent type declarations on all meta-programs and use a language like Cayenne [5] to do the type checking.
- Second, punt. Put off some type checking to run-time [72]. Thus some program errors will only be caught at run-time.

- Third, use extensible kinds to implement indexed types. Use the index to track value information (like the length of a list) in the type of objects. Then the dependency can be on the index and not the value.
- Fourth, extend the third mechanism with qualified types to track more sophisticated dependencies.

20 The theory of meta-programs

The foundations for meta-programming systems were laid by the programming language community. Most theoretical so far work has concentrated on meta-programming systems for generating code. Both denotational, operational and type-theoretic treatments abound. In the area of code analyzers there has been considerably less work.

Flemming Nielson and Hanne Nielson [51–54] have thoroughly studied the denotational semantics and abstract interpretation of two-level languages in the context of compiler design and specification.

The area of partial evaluation has attacked the performance problems of interpreted solutions. Of particular interest to the meta-programming community is the work of of Robert Glück and Jesper Jørgensen [30–32]. They study untyped, multi-stage languages in the context of binding time analysis for offline partial evaluation.

Important, early formal investigations into lanuages for staging computation were carried out by Rowan Davies and Frank Pfenning. They studied the *typed multi-stage languages* MiniML$^{\Box}$ [20] and MiniML$^{\bigcirc}$ [19]. These languages explore type systems (and find connections to intuitionistic modal logic, and linear-time constructive modal logic) for languages with type-constructors for code.

An important work relating these areas is the thesis of Walid Taha [74]. His thesis explains the utility of program generation in general and the importance of a type-safety guarantee in a generation paradigm. The thesis explains in detail, the difficulties that arise in specifying a mathematical semantics of staged programs, and then presents one along with two proofs.

The first proof justifies the claim that only type safe programs are ever generated. This proof is a “subject reduction” proof. A key technical contribution of the work provides a type system where the generated code can also be “run” or executed in the same framework where the code was constructed. Taha’s thesis explains how MetaML [80] combines features of both MiniML$^{\Box}$ and MiniML$^{\bigcirc}$ together with other useful features. This is an important generalization from the work of Davies and Pfenning. Such a guarantee is highly desirable in a system with run-time code generation.

In addition to the type-saftey properties, Taha’s work is the first to provide an equational theory for a staged programming language [76]. This theory can be used to prove equivalencies between two staged programs, or between a normal program and its staged (or partially evaluated) counterpart. Taha’s equational theory is built upon an equivalence proof between a small-step reduction semantics, and operational big-step semantics.

Taha's reduction semantics relies only on the standard notion of substitution, and unlike much of the earlier work[2], does not need any additional machinery for performing renaming at "run-time". This insight is a major advance in the theory of program generators. It opens the way to using standard techniques for manipulating program generators as formal objects.

In addition to Taha's thesis much work was done in improving the preciseness of type systems for MetaML [79–81]. These type systems and are designed to prevent *phase errors*, i.e., situations when values are used at a stage prior to their definition (as illustrated in Section 7).

Other papers [49, 75, 77–79] develop reduction and natural semantics for MetaML and prove important properties about them. Denotational semantics of MetaML from the perspective of categorical analysis were studied by Eugenio Moggi, Tim Sheard, Walid Taha and Zine El-Abidine Benaissa [9, 10, 48] yielding a semantics and a class of categorical models for MetaML.

Motivated by the categorical analysis, the same group did a more rigorous formulation of MetaML, supporting both open and closed code [50]. A theoretical study of multi-stage imperative programming languages has been undertaken by Christiano Calgano, Eugenio Moggi and Walid Taha [14]. They study operational semantics and type systems for a language that safely combines imperative state with multi-stage programming.

Practical applications of these semantic approaches to meta-programming systems have yet to bear much fruit. They supply a firm foundation, but the engineering work necessary to realizing robust implementations for staged programming languages has yet to be done.

21 Building good implementations

Meta-programming in general, and staged programming in particular is an important way to think about computation. For a long time we have been hampered in our attempts to promote this view of computation because of a lack of vocabulary: we lack running systems that treat object-computations in a first class manner. The challenge of building robust implementations of program generation systems remains. A few attempts have been made, but none has really caught the attention of the community at large.

MetaML: A First Attempt. The MetaML interpreter is an attempt to provide a tool where representations of programs (called `code`) are first class objects. MetaML is an *excellent tool for demonstrating that meta-computation can be expressed at a high-level of abstraction in a type-safe manner*, but it is still an interpreter, the code it produces is too slow, and it cannot interact with other languages or systems.

From this exercise in staged language implementation, I have learned many valuable lessons. Not the least of those is that building any system, polishing it so that other people may use it, and supporting it, is a huge amount of work.

[2] Notable exceptions are MiniML$^{\Box}$ and MiniML$^{\bigcirc}$.

MetaML is the largest system I have ever worked on, yet in many respects it is still a toy.

Despite these limitations building MetaML has been an extremely valuable experience which is worth sharing. Some of the important lessons learned while building the MetaML system are summarized below.

- **Homogeneous System.** MetaML is a homogeneous system because both the meta-language and the object-language are the same. Homogeneity plays an important role in many aspects of MetaML, but experience has shown that heterogeneous systems also have an important role to play. A heterogeneous system with a fixed meta-language, in which it is possible to build multiple systems each with a different object-language, or one system with multiple object-languages would be equally useful.
- **Template Based Presentation.** MetaML constructs object-code using pattern-based object-code templates. Templates look like the object language they represent. Program fragments are easy to combine into larger program fragments using templates with "holes". Templates make MetaML really easy to use to construct object-programs. Much remains to be done to use templates to pattern match against code when deconstructing object-programs.
- **Explicit Annotations.** Manually placing staging annotations is not a burden when supported by a quasi-quote presentation. Manually annotated programs are within a few percent of their non-staged versions. Annotations should me semantically meaningful, not just ad-hoc suggestions to the compiler or preprocessor. If you cannot define precisely what they mean, then their implementation will certainly be problematic.
- **Staged Generators.** MetaML is an N-stage meta-programming *generation* system. MetaML produces code generators. MetaML generators produce object-code as output, which itself can be another generator in a future stage. There is no limit to the number of stages in a MetaML program. This has been useful theoretically, but has found very little practical use. Programmers find it hard to write programs with more than a few stages.
- **Intensional analysis of code.** MetaML was *not* designed to produce code analyzers, but the ability to analyze the internal structure of code objects is an important capability that should be designed into any meta-programming system. Analysis of code is important even in the domain of program generation, since it allows generators to optimize code as it is generated.
- **Static Scoping.** MetaML handles free variables in object-code templates by building code where the *free* variables obey the rules of static scoping. Variables free in a template are bound in the scope where the template appears, not in the scope where the resulting code is executed. MetaML also handles *bound* variables in object-code templates in a way which guarantees no name clashes (inadvertent variable capture).
- **Type Safety.** In MetaML type-safe meta-programs are guaranteed to manipulate and produce only type-safe object-programs. This is the most important lesson learned. Writing meta-programs is hard. There are too many

dimensions in which it is possible to introduce errors. Tools, such as object-level typing, which help the programmer in this task are worth their weight in gold.

- **Observability.** Object-programs can be observed and printed. This is essential for debugging of meta-programs. In MetaML object-programs are pretty-printed in the manner in which they are normally written by programmers. They are not represented by some abstract representation unfamiliar to the meta-programmer.
- **Reflection.** In MetaML object-programs can be constructed, typed, and run, all in a single system. This ability to test generated code without leaving the meta-system is a great boon to programmers. Such a facility ought to be made available in any meta-programming system, even a heterogeneous system. The meta-language ought to embed an interpreter for the object-language to facilitate object-language testing.

Other attempts have emphasized different aspects of the code generation paradigm. In particular speed is an important concern of most other run-time generation systems. There is a tradeoff in this dimension, one can either generate code quickly or generate code that runs fast. Techniques which support one of these strategies often get in the way of addressing the other.

There is basically two strategies to achieving run-time code generation. Either generate source code (or some abstraction of it) at run-time, or generate machine code directly at run-time.

The first can be implemented in a straightforward manner using a *run-time compiler*. This lends itself to post-processing phases such as code transformations and register allocation. Since such processing requires time, the goal of generating code quickly is thwarted. It also requires larger run-time systems as the whole compiler must always be available.

Generating machine code directly is more complex, but potentially quicker to perform, since an intermediate step is not required. Unfortunately important optimizations like register allocation are harder to perform under this scheme. Generating machine code is hard to make generic and cross platform compatible since the machine architecture is intimately involved in the code generation phase.

Other issues to be addressed include allocation and garbage collection for generated code (constructing generated code of an unknown and unbounded size, and collecting generated code when its no longer needed). Most garbage collection systems are designed to collect data only, not code, which are often resident in different portions of memory[3]. In a staged system the maxim that *code is data* was never more true.

One problem with run-time code generation is that different kinds of variables (let bound, lambda bound, global, external) often use different access methods depending on the kind of variable (they may reside on the stack, in the heap, or in

[3] An exception to this is: SML/NJ, that collects code since code is allocated in the heap.

a register). Keeping track of all this at run-time is difficult, and general purpose solutions that work for all kinds of variables often introduce inefficiencies.

In this light we discuss several dynamic code generation systems.

- **'C and tcc.** ‘C (pronounced Tick C), and its compiler `tcc` [25,64] were designed to generate C code at run-time. ‘C is a two-stage language with no type system at the object-level. It uses a template based approach to construct object-level C code in an implementation independent manner. A goal of ‘C is to generate code quickly, yet still generate code that runs fast. They claim that tcc “generates code at a rate on the order of 60-600 instructions per generated instruction, depending on the level of dynamic optimization employed.” A sophisticated run-time register allocation scheme is used [65] to increase the performance of the generated code.
- **Tempo.** Tempo [17] is a large project investigating the use of partial evaluation strategies on C code. One aspect of the Tempo is dynamic code generation via specialization [18,55]. This capability is relies on the binding time analysis of the C partial evaluator. Given the static parameters as input, the dynamic code generation partitions the C code into two stages. The static stages are compiled in the normal fashion, and the dynamic stages are reduced to code templates. Each template is compiled to a sequence of instructions that generates code at run-time.

  In Tempo the staging is performed in a semiautomatic manner by the BTA. The Tempo implementation is strongly tied to the code generation capabilities of the Gnu C Compiler, and is not very portable. A strong advantage of the Tempo system is its ability to be applied to an existing code base.
- **The Dynamo Project.** The Dynamo [11,12] project works on implementing an infrastructure for building programs with *dynamic optimizations* which use run-time information unavailable to static optimizers, to perform further optimizations on a program as it is executing.

  The *Dynamo* compiler is broken down into a number of stages. User annotations determine which stages are run statically (at compile-time) or dynamically (at run-time) for each annotated code fragment. Thus the user can make specific tradeoffs of speed vs. time, or space vs. time depending upon which stages of the compiler are delayed until run-time.

  Each stage uses a different run-time representations of code, each of which is better suited to certain kinds of optimizations. Annotations (somewhat adhoc in nature) are placed by the programmer. These annotations, together (potentially) with profiling information, are used to decide which representation of code should be used for what particular piece of source program, and when it will (finally) be fully compiled.
- **Fabius.** Fabius [42–44] is a compiler for a staged language. Fabius uses explicit annotations to stage a program in an ML-like language. The staging annotations of Fabius are based upon the type systems devised by Frank Pfenning and Rowan Davies [20,19]. Other than MetaML, Fabius is the only dynamic code generator that types object-level code.

- **The UW Dynamic Compilation Project.** DyC [34, 6, 35] is a C based dynamic compilation system. In DyC regions of code are marked for specialization, and variables in those regions are declared either dynamic or static. An analysis then determines which code can be compiled at compile-time, and which code must be deferred for run-time compilation. Regions can be nested or even overlap.
  Annotations can be provided by the programmer to control a number of tradeoffs in both space and time. Implementations have been generated for a number of different architectures

22 Applications of meta-programming

Recently I have worked on, or read about several interesting applications of meta-programming techniques which I think are worth mentioning.

- **Macro design.** Macros are an application of staged programming, because macros define computation intended to run at compile-time. Most macro systems have been string based (except for the LISP and Scheme macro systems) and have been designed to run before type checking. Macros have also traditionally been used to define *new* binding constructs. This use is beyond the scope of the meta-systems discussed in this note. Recently, Steven Ganz, Amr Sabry, and Walid Taha have designed a macro system that is both strongly typed, and capable of introducing new binding constructs [29]. They do this by providing a translation of their macro language into a two stage MetaML program.
- **Language implementation.** Staged programming, combined with monads, can be used to implement Domain Specific Languages [70]. Staging a semantics-based interpreter produces a compiler for the language interpreted. Monads complement staging by describing control structure and effects in a manner that is orthogonal to the staging issues present in compiler implementation. The equational logic of MetaML can be used to prove that the compiled code produces the same outputs as the non-staged reference interpreter.
- **Dynamically adaptable software.** Staged programming has recently used to specify adaptive systems that, based on profiling information gathered at run-time, can recompile themselves to adapt better to their current running environment [37]. MetaML's staging annotation allow the programmer to express this rather difficult concept at a very high-level of abstraction that frees the user completely from the minutiae of runtime code generation.
- **Interacting with other systems.**
  Generators are often designed to generate code for a single environment. How do we build meta-systems with the ability to produce object-programs that interact with differing environments? The environment of a program generator is often a fixed, legacy system, and beyond the clients control. Wouldn't it be nice to construct a generator that could be configured to generate code for multiple environments? Norman Ramsey has recently written a clear

exposition of the many interesting issues that need to be addressed when building such a system [68].

23 Conclusion

I hope that these notes inspire discussion, research, and implementation efforts in the field of meta-programming.

References

1. A. Aasa. *User Defined Syntax.* PhD thesis, Chalmers University, Dept of Computer Science, Chalmers University, Sweden, 1992.
2. A. Aasa. Precedence for conc types. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 83–91, New York, June 1993. ACM Press.
3. A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 96–105. ACM, ACM, July 1988.
4. M. Abadi and L. Cardelli. An imperative object calculus (invited paper). *Theory and Practice of Object Sytems*, 1(3):151–166, 1995.
5. L. Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, Jan. 1999.
6. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996.
7. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
8. A. Bawden. Quasiquotation in LISP (invited talk). In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12. ACM, BRICS Notes Series, january 1999.
9. Z. E.-A. Benaissa, E. Moggi, W. Taha, and T. Sheard. A categorical analysis of multi-level languages (extended abstract). Technical Report CSE-98-018, Department of Computer Science, Oregon Graduate Institute, Dec. 1998. Available from [56].
10. Z. E.-A. Benaissa, E. Moggi, W. Taha, and T. Sheard. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, July 1999. To appear.
11. R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University Computer Science Department, March 1997.
12. R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 240–249. IEEE Computer Society Press, 1998.

13. R. Burstall and J. Goguen. An informal introduction to specifications using Clear. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.
14. C. Calcagno, E. Moggi, and W. Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *Automata, Languages and Programming*, pages 25–36, 2000.
15. R. Carnap. *The Logical Syntax of Language*. Kegan Paul, Trench and Trubner, 1937.
16. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
17. Consel, Hornof, Marlet, Muller, Thibault, and Volanschi. Tempo: specializing systems applications and beyond. *CSURVES: Computing Surveys Electronic Section*, 30, 1998.
18. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 Jan. 1996.
19. R. Davies. A temporal-logic approach to binding-time analysis. In *Proceedings,* 11^{th} *Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.
20. R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, Jan. 1996.
21. N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
22. R. K. Dybvig. From macrogeneration to syntactic abstraction. *Higher-Order and Symbolic Computation*, 13(1–2):57–63, Apr. 2000.
23. R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *LISP and Symbolic Computation*, 5(4):295–326, Dec. 1992.
24. C. M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.
25. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for efficient, machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 131–144, St. Petersburg Beach, Florida, January 1996. An earlier version is available as MIT-LCS-TM-526.
26. L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan 1996*, pages 284–294. ACM Press, New York, 1996.
27. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.
28. M. J. Gabbay. *Theory of Inductive Definitions With* α*-equivalence: Semantics, Implementation, Programming Language*. PhD thesis, Cambridge University, 2000.

29. S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-2001)*, New York, September 2001. ACM Press.
30. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
31. R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.
32. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *LISP and Symbolic Computation*, 10(2):113–158, 1997.
33. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
34. Grant, Mock, Philipose, Chambers, and Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *TCS: Theoretical Computer Science*, 248, 2000.
35. B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, Atlanta, Georgia, May 1–4, 1999.
36. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, June 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.
37. B. Harrison and T. Sheard. Dynamically adaptable software with metacomputations in a staged language. In *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, September 2001. Appearing in this proceedings.
38. L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, Dec. 1999.
39. S. Johnson. Yacc – yet another compiler compiler. Technical Report 32, Bell Labs, 1975.
40. M. P. Jones. *Qualified Types: Theory and Practice.* Programming Research Group, Oxford University, July 1992.
41. M. P. Jones. A theory of qualified types. In B. Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, New York, NY, 1992.
42. P. Lee and M. Leone. Optimizing ML with run-time code generation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, 1996.
43. M. Leone and P. Lee. Lightweight run-time code generation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 97–106, 1994.
44. M. Leone and P. Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):??–??, Sept. 1998. Article 23.

45. J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 108–118, N.Y., Jan. 19–21 2000. ACM Press.
46. H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.
47. D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.
48. E. Moggi. Functor categories and two-level languages. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
49. E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
50. E. Moggi, W. Taha, Z. El-Abidine Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. *Lecture Notes in Computer Science*, 1576:193–207, 1999.
51. F. Nielson. Program transformations in a denotational setting. *ACM Trans. Prog. Lang. Syst.*, 7(3):359–379, July 1985.
52. F. Nielson. Correctness of code generation from a two-level meta-language. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP 86)*, volume 213 of *Lecture Notes in Computer Science*, pages 30–40, Saarbrücken, Mar. 1986. Springer.
53. F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, Dec. 1989.
54. F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, Jan. 1988.
55. F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–142. IEEE Computer Society Press, 1998.
56. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`. Last viewed August 1999.
57. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
58. E. Pašalić, T. Sheard, and W. Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Submitted to ICFP2000.
59. E. Pašalić, T. Sheard, and W. Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, 2000. Available from [56].
60. F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
61. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

62. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.
63. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.
64. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
65. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.
66. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.
67. C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Usenix Journal, Computing Systems*, 1(1):11, Winter 1988.
68. N. Ramsey. Pragmatic aspects of reusable software generators. In *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG)*, pages 149–171, September 2000. Workshop held in collaboration with the International Conference on Functional Programming (ICFP).
69. T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.
70. T. Sheard, Z. Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, October 1999. USEUNIX.
71. M. Shields. *Static Types for Dynamic Documents.* PhD thesis, Department of Computer Science, Oregon Graduate Institute, Feb. 2001. Available at `http://www.cse.ogi.edu/~mbs/pub/thesis/thesis.ps`.
72. M. Shields, T. Sheard, and S. P. Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, Jan. 1998.
73. G. L. Steele, Jr. and R. P. Gabriel. The evolution of LISP. *ACM SIGPLAN Notices*, 28(3):231–270, 1993.
74. W. Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).
75. W. Taha. A sound reduction semantics for untyped CBN multi-stage computation: Or, the theory of MetaML is non-trivial. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM-00)*, pages 34–43, N.Y., Jan. 22–23 2000. ACM Press.
76. W. Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM'00)*, Jan. 2000.
77. W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, July 1998.

78. W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. Technical Report CSE-98-002, Oregon Graduate Institute, 1998. Available from [56].
79. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [81].
80. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, Jan. 1999. Extended version of [79]. Available from [56].
81. W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. Revised version of [80].
82. S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Seventeenth IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, pages 135–143, Washington - Brussels - Tokyo, Oct. 1998. IEEE.
83. W. van Orman Quine. *Mathematical Logic*. Harvard University Press, Cambridge, 2 edition, 1974.