Evolving Domain Specific Languages
Project Summary

We believe that the use of Domain Specific Languages (DSLs) can significantly improve the ability of domain experts to use computers effectively in their work. However the cost of developing and deploying DSLs is significant, and this has severely limited their deployment.

Intellectual Merit: Like other software, DSLs have a well-defined life-cycle. SQL is a "mature" DSL: its design is specified by international standards, and its *stand-alone* implementations are carefully crafted by hand to be stable, robust and to operate with high performance in specific execution environments. The DSLs that we have designed are experimental: their designs vary week-by-week as researchers try out new ideas, their implementations must be malleable, and efficiency and the quality of the error messages are less important. The simplest way to implement an experimental DSL is by *embedding* it as a library of functions within an existing programming language. These two approaches have both advantages and disadvantages, and the point in the life cycle of a particular DSL seems to favor one approach over another.

The life cycle of a DSL has three stages, and the needs of the three stages vary. By recognizing this fact, we can design processes and tools that can dramatically change the balance point between the two choices, thus allowing the cheaper embedded approach to be viable longer. In addition, our research in meta-programming systems seems highly applicable to the opposite approach of making stand-alone implementations cheaper to build, so that they can be applicable earlier in the lifecycle. Finally, experience with some of the tools we have constructed in our meta-programming research suggest that stand-alone implementations can be extracted from embedded implementations in a semi automated way, further blending the distinction between the two approaches. This bodes well for lessening the cost of deploying a DSL since its suggests several strategies for *evolving* an embedded implementation to a stand-alone one in a cost effective way. The proposed research would investigate applying meta-programming tools to *DSL evolution*, in several different dimensions:

- Extending the capabilities of host languages for embedded implementations.
- Exploring automated techniques for converting embedded implementations to stand-alone implementations.
- Automating the generation of stand-alone implementations.

Broader Impact: The opportunity costs of being unable to utilize computers effectively has a large cost: projects that are never attempted, or whose scope is significantly reduced, because the cost of developing software is too high to contemplate, or too complex to write. DSLs can play a large role in mitigating these costs, but only if the costs of DSL implementation can be lowered. The research proposed here will make significant advances in this direction.

As a graduate-only institution, OGI is in a unique position to promote and advance the introduction of new ideas into the work force. Our students are by and large working students, uniquely motivated by the problems that they have experienced. Our close connections with industry means that our students acutely aware of the problems of constructing complex software. OGI has a tradition teaching research oriented topics in advance graduate level courses, and in building and maintaining widely distributed tools.

1 Overview

Computers are useful only if there are people to program them. In an ideal world, the end-user, who is an expert in the domain of the problem that needs to be solved, would be able to directly command the computer to do what is required. Unfortunately, even the best of modern general-purpose programming languages can be used effectively only by experts in *computing*. As computers become ever more pervasive, and the problem domains in which they are applied ever more diverse, the proportion of users who have sufficient computing knowledge dwindles.

There is growing recognition that *domain-specific languages* (DSLs) are an effective way for end-users to program computers without becoming software engineers. Although DSLs are attractive, they can be difficult and expensive to design and implement. DSL creators must combine expert knowledge of the domain, programming skills necessary for building solutions, ability to make appropriate generalizations over the domain, and programming language design and implementation skills. This makes the current cost of introducing a new DSL prohibitively high. We propose two interrelated research paths that we believe can dramatically lower this cost.

First, at OGI, we have studied scores of DSLs, and we have designed and developed a number of our own. We have discovered that DSL's (like many other systems) have a life cycle. This life cycle has three stages, and the needs of the three stages vary. By recognizing this fact, we can design processes and tools that can dramatically increase reuse amongst DSL implementations, and ease the transition of a DSL from earlier stages to later stages.

Second, our study of typeful meta-programming systems has helped us design a new generation of tools for manipulating programs as data in a way that is aware of the semantic properties of the programs being manipulated. Applying these tools to DSL design and implementation should significantly ease the burden of the DSL implementer. The aims of the research proposed include:

- **Lowering the cost of DSL development.** DSL's are effective only if they exist. The current cost for developing a DSL can be quite high. Lowering the cost for deploying a new DSL could be an effective means of increasing their use.
- **Apply meta-programming techniques to DSL development.** Much of our experience in building meta-programming systems is directly applicable to DSL design. We'd like to reuse this work in a new and exciting domain. Many of the tools we have built, such as automatic binding time analyses as a way of automatically staging programs, and the meta-programming language Ωmega seem applicable to the process of evolving DSL's.
- **Develop instructional materials on DSL development.** Without making significant efforts to educate others on the building of DSL implementations their construction will remain out of reach to most software developers.

2 Introduction to DSLs

Domain-Specific Languages (DSLs) are a promising approach to making computers accessible to end-users without the need for conventional programming. Perhaps the most widely used DSL is the formula language used inside *Microsoft Excel* to define how the content of a cell on a spreadsheet is derived from other cells. Here, the domain is calculation; the particular problems solved are expressing that one cell should be the sum, or the average, of some other cells. People who write such formulae think of themselves as solving problems in finance, or surveying, or engineering; we know that they are in fact writing computer programs, but the power of DSLs is that from the point of view of the domain expert, *the program has disappeared.*

Other widely used DSLs include *SQL*, a DSL for extracting relevant information from a relational database without explicitly specifying the physical layout or access path for the data; *Yacc*, a DSL for describing parsers by the grammar to be parsed; and *HTML*, a DSL for defining the appearance of a document without specifying explicit layout algorithms. As computers are applied to more and more aspects of our society, we believe that the ability of such DSLs to "make the program disappear" will be vital. DSLs provide many advantages over general-purpose languages. To illustrate this consider the query language SQL.

- A SQL program provides a common language that allows different database users to share precisely the description of the data they are interested in.
- A SQL program relieves the programmer from needing to know about various aspects of the database, such as the file structures that store the persistent data (sequential file, or a balanced tree?), how to open and process the different file formats (random or sequential access?), how to design and manipulate the ephemeral data structures needed to process the query (a linked list, an array, a hash table?), and what algorithm to use (a nested join, a merge join, or a hash join?).
- A SQL program encodes constraints. It restricts the complexity of the programs that the user can write. A SQL program is guaranteed to terminate, and is bounded by polynomial rather than exponential time behavior.
- A SQL program can take advantage of domain specific properties. All commercial SQL processors use an extensive query-optimization phase that attempts to aggressively minimize the resources used to answer the query.
- A SQL program is a rapid prototype. A SQL query is easy to write and lets the query writer experiment with different solutions. If the results from a query do not answer the question the query writer had in mind, he is free to throw the original query away and try another without excessive losses in time or resources.
- A SQL query can integrate with a number of existing contexts or environments. SQL is a front end to a number of different query engines.
- A SQL query is an efficient division of labor. Programming language experts designed the SQL language, algorithm experts designed the algorithms used, database experts designed the query optimizers, and domain experts ask the queries.
- A SQL query adopts an existing notation (from relational algebra) and allows database users to program queries effectively with little or no programming experience.

Advantages like these are not unique to SQL, but are general properties of all domain specific languages. While providing significant advantages, the DSL approach comes with a cost. Realizing a DSL requires both a design and an implementation. Such implementations can be large, expensive to produce, conceptually complex, and hard to scale. An implementation for even a simple language often does not scale as the language evolves to meet newer demands. Because of this, domain experts, and even most programmers, are not comfortable taking on the tasks of language design and implementation. Engineering issues, like these, have kept DSLs from being widely as used as they ought to be.

2.1 Domain-Specific Language Evolution

Although past research has produced methodologies and tools that aid in various stages of language design and implementation process, the level of automation available for building DSLs is low. Moreover, DSLs are seldom static. Initially, the domain may not be well-understood, so a period of experimental design is required while ideas are developed. Constraints on the implementation tend to become stronger over time, as the experimental DSL is moved into production. In addition, the problem domain itself can change over the intended lifetime of the DSL. Hence, the design and implementation of a DSL must be *evolved* over time.

DSL implementations are not only complex and difficult to build, but they are also difficult to evolve. Implementation techniques that are most cost-effective at early stages in the DSL lifecycle may be less appropriate at later stages, and vice-versa. *Embedding* the DSL as a library within an existing host programming language may work well for an initial implementation. Syntactic extensions, perhaps implemented with macros or a pre-processor, might be added later. Such implementations are quick and easier to build (than non-embedded ones), but inherit all the limitations, as well as the capabilities, of the host language and its implementation. More mature versions of the DSL may benefit from a *stand-alone* implementation, but this is typically expensive to build, as it must handle all the tasks of a conventional language translator, such as parsing, semantic analysis, optimization, and code generation. Current technologies provide no smooth transition path from embedded to stand-alone implementations. It is a goal of this proposal to identify and smooth this transition from embedded DSL to stand-alone DSL.

2.2 Supporting the DSL Lifecycle

The life cycle of a DSL may be thought of as having three stages, each calling for different implementation approaches.

Infancy. In this stage, language design ideas are in flux and domain abstractions are not well understood. While some domains have existing conceptual frameworks and notations that can be used for problem specification, many domains must be analyzed and studied to identify the key abstractions. At this stage, modeling flexibility is at a premium, performance is less important, and embedded implementations are likely to be cost-effective.

In an embedded approach, each domain concept is realized directly as a host-language construct: domain operators are host-language procedures, domain types are host-language user-defined data types, etc. Thus, creating or modifying a DSL is relatively cheap, provided a suitably powerful host language is used. Our experience with scores of DSLs has shown that successful embedding of a DSL is made much easier by the use of higher-order functions. Thus, languages such as O'Caml, Haskell, or Lisp have made suitable host languages [16, 14, 12].

Embedding may be thought of as rapid prototyping. Even if the domain ultimately requires generating code for a specialized target environment, the embedded implementation can be used for modeling and simulation. Many language features needed by a typical DSL (*e.g.*, support for procedural abstraction) will already exist in the host language; moreover, it is straightforward to integrate code from multiple DSLs if they share the same host implementation.

Adolescence. In this stage, domain abstractions stabilize and become better-understood as the language begins to be exercised by end-users. Implementation flexibility is still necessary. Actual use uncovers the limits of the implementation, which leads to new features, abstractions, and implementation techniques.

In this stage users begin to develop domain-specific logics for their DSLs. Such logics describe how the features of the DSL interact. They allow the DSL programmers to calculate equivalences, and to reason about the behavior of each solution at a high level of abstraction. Domain-specific logics also enable domain-specific optimizations, and efficient implementations.

While embedded languages remain important at this stage, their limitations begin to outweigh their advantages. As more users exercise the system, the need for specialized language syntax, type discipline, and error reporting become more apparent. These are difficult or impossible to provide in a conventional host language. Because most host languages lack the ability to reflect on their own code – *i.e.*, to analyze it as well as execute it – they make it very difficult to enhance an embedded DSL implementation with validation or optimization passes.

Few host language implementations provide much flexibility about the target environment in which they run, though this may be essential in domains involving embedded systems or requiring close integration with legacy code. Currently, these demands often force implementers to switch to a stand-alone implementation at this stage, but improvements to host languages (i.e. foreign function interfaces) may enable embedded implementation to remain a viable choice longer in the adolescence stage.

Maturity. In this stage, domain abstractions are well-understood, and the DSL design is fixed. Features that enhance usability – such as type systems, debuggers, and programming environments – become important as the DSL gains adherents. Performance of the DSL implementation itself may become a problem as DSL program size and complexity increase.

A stand-alone implementation for a DSL can have its own syntax and type system appropriate for just that domain. The DSL can be "restricted" to enforce constraints on what can be expressed, and it can have its own optimizer that relies on domain-specific knowledge so that performance bottlenecks can be addressed. If investing in a stand-alone DSL implementation ever makes sense, it will be at this phase, when the language has become relatively stable and user tools are important. Automated construction tools for interpreters and compilers can make this process cheaper; while many such tools exist, some important ones are still missing. It is still desirable to re-use any existing embedded or stand-alone implementation as far as possible; again, improvements in embedded host language design may help this goal.

2.3 Example: Maturation of the Hawk DSL

At OGI we have developed a number of DSLs, including *MSL* [35, 13], a DSL for specifying radar message formats and invariants; *Stratego* [33], a language for describing transformations on computer programs; and *Hawk* [16], a DSL for describing computer chip microarchitecture. As an example of DSL maturation we will consider the Hawk DSL, developed by Prof. John Launchbury at OGI. Hawk is used to describe computer microarchitectures at the level of abstraction used in an architecture textbook, *i.e.*, in terms of register files, memories, delays, pipeline elements and reorder buffers. A Hawk program specifies a particular microarchitecture; it can be executed to simulate the architecture; it can be transformed, say to increase throughput, and it can be analyzed for various properties. Hawk has been used to build simulators for several pipelined architectures, as well as superscalar architectures.

The infant Hawk implementation was embedded in Haskell. An embedded approach was used to help determine the most appropriate abstractions. Hawk language constructs describe circuits as a set of equations describing data flow relationships. The equations were implemented as higher-order functions in Haskell. Through experimentation we found very successful abstractions such as transactions and bypasses that succinctly capture complex functionality at the level of detail needed to understand microarchitectures.

As Hawk reached adolescence, we began to run into the characteristic limitations of the embedded approach.

- **Syntax.** We found a need for a language construct to express mutual recursion at the monadic level [10], something not available in Haskell. Such a construct would have been very useful to describe cyclic architectures with feedback loops using monadic descriptions.
- **Analysis.** Because Hawk represents its abstractions as Haskell functions (rather than data structures) we found it impossible to implement some useful analyses on Hawk programs. Performance analysis of pipelines requires knowing the length of the pipeline, and correctness analysis of feedback loops requires knowing that every feedback loop is delayed at least once by the use of a latch. Neither of these things could be discovered in the embedded Hawk implementation.
- **Error messages.** Error messages in Hawk are expressed in terms of the Haskell implementation rather than the abstractions they represent. This, coupled with extensive use of Haskell's class mechanism, makes error messages from the type checker nearly incomprehensible to end-users.

- **Interoperation.** Hawk could benefit by interoperation or integration with existing compilations of microarchitecture units, e.g., memories, written in other languages such as VHDL, or by interoperating with existing vector-based simulation tools. However, Haskell's interoperability facilities are limited.
- **Performance.** Although Hawk's expressiveness and verification ability are unparalleled, its simulation performance (in comparison with competing simulation technology such as *SimpleScalar [1]*) leaves much to be desired.

A stand-alone implementation for Hawk could address these problems directly, but a stand-alone implementation also requires an order of magnitude more work. Either we need richer host languages that solve some of these problems, delaying the need for a stand-alone implementation, or we need better tools for building stand-alone implementations.

3 Tools for DSL Design – Typeful Meta-Programming Systems

Meta-programming systems can play an important role in both building stand-alone implementations and enriching host languages. Meta-programming is the study of programs that generate and/or manipulate other programs. While our work on meta-programming systems was not explicitly motivated by the issue of DSL design and evolution, meta-programming systems seem to provide many of the capabilities that are needed to implement DSLs in a direct and efficient manner. Their ability to manipulate programs as data objects also allows them to support the evolution of existing embedded implementations into stand-alone ones. Our interests in the area of meta-programming fall roughly into two broad categories:

1. *Homogeneous meta-programming.* In homogeneous meta-programming languages (MetaML [24, 32]) the object-language and the meta-language are the same. The representation of programs (both syntactic, and semantic) are chosen by the language designer and cannot be observed by meta-programs. This relieves the meta-programmer from the responsibility to represent object-programs, and allows the meta-language itself to ensure that important properties (such as static typing) of object-languages are automatically enforced in all meta-programs. Our work in homogeneous meta-programming languages was directed primarily at staged programming languages. Staged programs proceed in stages. Each stage "writes" a program that is executed in the next stage. Staging can be exploited to build efficient implementations whenever constants or program invariants only become available after compile-time.
2. *Heterogeneous meta-programming.* In heterogeneous meta-programming, the meta- and the object-language are not (necessarily) the same. We are particularly interested in *open* heterogeneous meta-programming languages. Such languages equip the programmer with the tools to specify both the syntactic representation and semantics of arbitrary object-languages. The ability of the meta-language to ensure object-language types and properties in a heterogeneous system is considerably complicated by the difference between the meta- and object-languages. If the type system of the object-language differs significantly from the type system of the meta-language, we cannot use the single type system approach used in homogeneous meta-programming. Instead, we must rely on an extension mechanism, that allows the embedding of new (object-language) type systems inside the type system of the meta-language.

Our work on two previous NSF supported projects *Type Safe Program Generators* (CCR-9625462, 10/96-03/00), and *Heterogeneous Meta Programming Systems* (CCR-0098126 10/01-06/04) has given us the opportunity to think deeply on the issues of language representation and manipulation. We have reported on the design[32], use[24, 25], semantics[30], type systems[3, 18, 29], implementation[17, 26, 4], and open problems[27] of meta-programming systems. With the help of his colleagues the proposer has built and distributed two different meta-programming systems: MetaML[24] and Template Haskell[26].

Recently, we completed the preliminary design for Ωmega, an open heterogeneous meta-programming language. Ωmega gives special support to the description of new object-languages. One of the unique features of Ωmega is its ability to define object-language representations that use Ωmega's meta-level types to guarantee programmer-specified properties of object-language syntax. Ωmega is also a homogeneous meta-programming system that inherits MetaML's ability to write staged programs. This combination of heterogeneous and homogeneous features provides the tools needed for implementation and evolution of domain specific languages.

4 Example: A language of geometric regions.

In this section, we shall apply Ωmega to a small example as a way of presenting our ideas for evolving domain specific languages. We define a small DSL that describes geometric objects. This DSL has its own syntax and semantics. It includes a simple type system in which the type of an object specifies the measuring units (i.e. inches, centimeters, pixels) used to describe it. The type system ensures that as we combine objects, units remain consistent. There are several important points that this example makes and addresses.

- We shall illustrate the difference between an embedded and stand-alone implementation. The embedded DSL is implemented by combinators that directly manipulate the denotations of the DSL expressions. These combinators are typed in a way that forces the programmer to combine them in a way that preserves the typing of DSL expressions. This is an important advantage of using the embedding approach for many DSL's.
- The embedded Region language is easy to implement but hard to manipulate. A region program is implemented as an Ωmega function (or closure), and its internal structure cannot be observed. This denies the DSL implementer the opportunity of writing potentially useful meta-programs that manipulate and optimize DSL programs.
- Representing a DSL program as a piece of data recovers the ability to manipulate the program, but has two drawbacks. First, we lose the ability to encode the type system of the DSL in the type system of the host language. Second, implementing the DSL in the host language now introduces several layers of interpretive overhead.
- A stand-alone implementation usually involves representing the object-language as a data structure and writing separate type checking and code generation functionality. Code generation usually targets a language different from the language used to build the stand-alone implementation. This is done mostly for efficiency or interoperability/compatibility reasons.
- Reusing the work invested in an embedded implementation in a stand-alone implementation is hard because of the vastly different strategies employed.

Ωmega has several unique features that we believe allows us to overcome all of the weakness described in the bullets. The example illustrates a potential transition pathway, from an embedded implementation of the DSL to a stand-alone implementation, with many reuse possibilities. We shall also demonstrate and emphasize the importance of a meta-language that can *transform* the DSL expressions into another language to support efficient and effective implementation, thus automating some of the steps in the evolution.

4.1 The Region DSL

The Region DSL provides a notation to describe subsets of the two-dimensional plane, which we shall call *Regions*. Regions are specified in a planar coordinate system using magnitudes that can be measured in different kinds of units (centimeters, pixels, inches, and so on). The Region language is *typed*. Its type system is designed to prevent combining regions defined with mismatched measuring units. Once defined, a region can be drawn or displayed. The region language is very simple, but rich enough that we can illustrate the points made in the previous section.

$$
\begin{array}{llll}
u & \in U ::= & \text{cm} \mid \text{pix} & \textbf{units} \\
d & \in D ::= & n\ u & \textbf{magnitudes} \\
r & \in R ::= & \text{univ} \mid \text{empty} \mid \text{circle}\ d \mid \text{rect}\ d\ d \mid \text{trans}\ (d,d)\ r & \textbf{regions} \\
 & \quad \mid & r_1 \cup r_2 \mid r_1 \cap r_2 \mid \text{conv}\ u_1\ u_2\ r &
\end{array}
$$

$$
\frac{}{\vdash n\ u : u}
$$

$$
\frac{}{\vdash \text{empty} : R\,u} \quad \frac{}{\vdash \text{univ} : R\,u} \quad \frac{\vdash d : u}{\vdash (\text{circle}\ d) : R\,u} \quad \frac{\vdash d_1 : u \quad \vdash d_2 : u}{\vdash (\text{rect}\ d_1\ d_2) : R\,u}
$$

$$
\frac{\vdash r_1 : R \quad \vdash r_2 : R\,u}{\vdash r_1 \cup r_2 : R\,u} \quad \frac{\vdash r_1 : R \quad \vdash r_2 : R\,u}{\vdash r_1 \cap r_2 : R\,u}
$$

$$
\frac{\vdash r : R\,u_1}{\vdash (\text{conv}\ u_1\ u_2\ r) : R\,u_2} \quad \frac{\vdash d_1 : u \quad \vdash d_2 : u \quad \vdash r : R\,u}{\vdash \text{trans}\ (d_1, d_2)\ r : R\,u}
$$

Figure 1: The language of geometric regions: syntax and type system.

In Figure 1 we give the syntax and static semantics (typing rules) for the Region DSL. There are three syntactic categories: units, magnitudes (i.e. 10 *cm*, or 2 *pix*) and regions. There are four primitive regions: *empty* is the empty region; *univ* is the full plane; (circle d) is a circular region with center at the origin of the coordinate system and radius specified by d; (rect d_1 d_2) is a rectangular region centered at the origin with length d_1 and height d_2. Regions can be combined using "set-operations:" union ($r_1 \cup r_2$) and intersection ($r_1 \cap r_2$). Translation (trans (d_1, d_2) r) translates a region by an amount specified by a pair of magnitudes (d_1, d_2). Finally, a region of type u_1 can be coerced to a region of type u_2 using the explicit conversion expression (conv u_1 u_2 r).

An embedded region DSL. In Figure 2 we implement regions as an embedded DSL using Ωmega as the host language. The meaning of a region is a characteristic function: for each coordinate in the plain, the meaning function should answer whether that points lies inside the region. In the embedding, regions are defined in terms of host language functions (notice the type synonym for `Region` as a function type). Magnitudes are defined in terms of an algebraic datatype (`Mag`). The embedded implementation is just a series of definitions, one for each of the DSL's operators.

This embedding uses the type system of the host language to implement the type system of the Region language. The types of the region combinators are parameterized by the type of the unit used to construct it. For example a region of type `(Region Centimeter)` must have been constructed using centimeters as units. This requires a certain amount of overloading in the implementation. Consider the definition of the basic region for a circle.

```
circle :: Mag u -> Region u
circle r = \ x y -> leq (plus (square x)(square y)) (square r)
```

A point `(x,y)` lies within the circular region of radius `r`, if and only if $x^2 + y^2 \leq r^2$. We need some way of overloading `leq`, `plus` and `square` to work on different representations of magnitudes. There are several ways one could do this, but we choose to use two of Ωmega's unique features (kind extension and equality qualified types) because we will find them useful later on.

We use kind extension to define a new kind `Unit`. Kinds[2, 11, 19] are similar to types in that, while types classify values, kinds classify types. We indicate this by the *classifies* relation (`::`). For example: `5 :: Int :: *0` . We say `5` is classified by `Int`, and `Int` is classified by `*0` (star-zero). The kind `*0` classifies all the types that classify values that can be computed (like `Int`, `Char`, pairs, functions etc). The kind definition also introduces two new types `Pixel` and `Centimeter` that are classified by `Unit` (`Pixel::Unit`, `Centimeter::Unit`). We then define a new type constructor `Mag::Unit -> *0` which is parameterized by types of kind `Unit`. This allows us to define data- constructor functions `Pix` and `Cm` that inject `Int` values into `(Mag Pixel)` values, and `Float` values into `(Mag Centimeter)` values respectively. Consider the clause

```
-- Units and Magnitudes

kind Unit = Pixel | Centimeter
data Mag u = Pix Int where u = Pixel
           | Cm Float where u = Centimeter

-- Regions as Characteristic Function
type Region u = Mag u -> Mag u -> Bool

-- basic regions
circle::Mag u -> Region u
circle r =
  \x y -> leq (plus (square x) (square y))
              (square r)

rect::Mag u -> Mag u -> Region u
rect w h =
  \x y -> between (neg w) (plus x x) w &&
          between (neg h) (plus y y) h

univ::Region u
univ = \x y -> True

empty :: Region u
empty = \x y -> False

-- region combinators
trans::(Mag u,Mag u) -> Region u -> Region u
trans (a,b) r =
  \x y -> r (minus x a) (minus y b)

convert::(Mag v -> Mag u)-> Region u -> Region v
convert t r = \x y -> r (t x) (t y)

intersect::Region u -> Region u -> Region u
intersect r1 r2 = \x y -> r1 x y && r2 x y

union::Region u -> Region u -> Region u
union r1 r2 = \ x y -> r1 x y || r2 x y

plus :: Mag u -> Mag u -> Mag u
plus (Pix a) (Pix b) = Pix (intAdd a b)
plus (Cm a) (Cm b) = Cm (floatAdd a b)
```

Figure 2: An embedded implementation of regions with combinators.

`Pix Int where u = Pixel` The use of equality qualification in the `where` clause gives the constructor function `Pix` the qualified type `(forall (u:Unit).(u = Pixel)=>Int -> Mag u)`. On one hand whenever we apply the function `Pix` we are required to establish that `(u=Pixel)`. On the other hand, whenever we pattern match against a value constructed by `(Pix n)` we can assume that `(u=Pixel)`. These two features allow us to define type constructors whose type parameters encode information about the internal structure of the values they classify. For example, a value of type `(Mag Pixel)` must have been constructed by the `Pix` constructor function. This ability will play an important role later on in the example.

We can now see how `leq`, `plus` and `square` are overloaded. Consider the first clause of the definition of `plus` in Figure 2. To type check this clause we must be able to derive:
$\{\texttt{Pix a} :: \texttt{Mag u},\ \texttt{Pix b} :: \texttt{Mag u},\} \vdash (\texttt{Pix}(\texttt{intAdd a b})) :: \texttt{Mag u}$. Because `Pix` has an equality qualified type `Pix::(u=Pixel)=>Int -> Mag u` we can derive a type for `a` and `b` plus an equality between `u` and `Pixel`. $\{(\texttt{u} = \texttt{Pixel}),\ \texttt{a} :: \texttt{Int},\ \texttt{b} :: \texttt{Int},\} \vdash (\texttt{Pix}(\texttt{intAdd a b})) :: \texttt{Mag u}$. We can derive the type `Mag u` for the expression `(Pix (intAdd a b))` provided `(intAdd a b)::Int` and we can show `(u=Pixel)`, but this is precisely the guarantee we obtained from the pattern `(Pix a)`, so we can derive the required type.

An important feature of this embedding is that the type-system of the host language ensures type-correct combination of regions, i.e., it disallows the mixing of units of magnitude that the type system of the regions language is designed to prevent. For example, the expression `union (circle (Cm 2)) (circle (Pix 4))` is simply rejected by Ωmega's type checker as ill-typed.

Syntax as data. The essential feature of the Region embedding is that it directly manipulates the *denotations* of regions as function-values. The disadvantage of this kind of embedding is that Region programs cannot be syntactically analyzed since they are just functions. We cannot even write a pretty-printer for Region expressions. In this section we present an approach to implementing the Regions DSL that will allow us to ensure that only well-typed regions are combined, and still let us analyze Region expressions' syntax.

Usually, object-languages such as Region expressions are represented as algebraic data-types. However, a naïve data-type representation would not allow us to statically enforce the invariant that only regions with the same type of units are combined. In Figure 3, we define a new type *constructor* `RegExp` which encodes Region expressions. The important thing to note is that `RegExp` is a type constructor whose argument type

```
data RegExp u
 = Univ
 | Empty
 | Circle (Mag u)
 | Rect (Mag u) (Mag u)
 | Union (RegExp u) (RegExp u)
 | Inter (RegExp u) (RegExp u)
 | Trans (Mag u, Mag u) (RegExp u)
 | forall v. Convert (CoordTrans v u) (RegExp v)

evalTrans :: CoordTrans u v -> Mag v -> Mag u
evalTrans CM2PX = pxTOcm
evalTrans PX2CM = cmTOpx
```

```
eval :: RegExp u -> Region u
eval Univ         = univ
eval Empty        = empty
eval (Circle r)   = circle r
eval (Rect w h)   = rect w h
eval (Union a b)  = union (eval a) (eval b)
eval (Inter a b)  = intersect (eval a) (eval b)
eval (Trans xy r) = trans xy (eval r)
eval (Convert trans r)  =
  convert (evalTrans trans) (eval r)
```

Figure 3: data structure embedding.

indicates the unit of magnitude for the region it represents. In the previous section, we defined the `Mag` type with equality qualified units because we wanted to parameterize Region expressions by a unit type. As in the embedded combination approach, this parameterized data constructor approach also requires arguments of constructor functions like `Union`, to have matching units. In a similar fashion a syntactic representation of unit transformers is also defined:

```
data CoordTrans u v = CM2PX where u = Centimeter, v = Pixel
                    | PX2CM where u = Pixel, v = Centimeter
```

This uses the equality qualified types in the same way they were used in the type constructor `Mag`. The `where` clauses force the constructors `CM2PX` and `PX2CM` to have the type `CoordTrans Centimeter Pixel` and `CoordTrans Pixel Centimeter`, respectively. The values of `CoordTrans` are used to encode the `Trans` regions: given a region of some unit type `v`, and a unit transformer `CoordTrans v u`, we can obtain a region of unit type `u`.

At first glance, the technique presented here does not seem much different from a standard data-type representation of syntax available in functional languages such as ML or Haskell. This is because the type system of the Regions DSL is quite primitive. Expressing the typing invariants on the region expressions is not very difficult. Embedding richer typing invariants in algebraic syntax becomes quite hard without using equality qualified types or other mechanisms. Using this mechanism we have accumulated considerable experience in expressing significantly more complex object-language typing invariants (e.g., the simply typed λ-calculus, security type systems[22, 34], modal types[8, 9], and so on) [20].

The next step in developing a stand-alone implementation of the Region DSL is to define a function mapping Region expressions to their denotations (i.e., the appropriate characteristic functions). The type of `eval` is most interesting – it maps `(RegExp u)` to `(Region u)`. This type ensures that the meaning of regions preserves magnitude unit types. The function `evalTrans` maps a `CoordTrans u v` to a function that converts between magnitudes of type `u` and `v`. Moreover, from the definition of `eval`, we can see that each clause of `eval` simply reuses the embedding combinators to give meanings to region expressions. This style of definition provides a syntactic and semantic connection between embedded and stand-alone implementation of regions.

Using domain specific optimizations. A benefit of representing Region programs as data is the ability to improve their efficiency by applying domain specific, meaning-preserving transformations. Figure 4 shows several equivalences of Region programs that we can use to improve their efficiency. It is a simple step to code up the algebra of equivalences in Ωmega. Some sample equivalences are found in Figure 5. The function `normalize` transforms its argument by applying these equivalences to regions. It does so by replacing all of the region's constructors with *smart* constructors in a bottom-up fashion. Each smart constructor applies some domain specific equivalences for that constructor. The figure shows only one smart constructor, but the others are similarly defined.

$$\text{univ} \cup r = \text{univ} \qquad \text{empty} \cap r = \text{empty} \qquad \text{empty} \cup r = r = \text{univ} \cap r \qquad (r_1 \cup r_2) \cap (r_1 \cup r_3) = r_1 \cup (r_2 \cap r_3)$$

$$\text{conv}\ \tau_1\ \tau_2\ (\text{conv}\ \tau_2\ \tau_1\ r) = r \qquad \text{conv}\ \tau_1\ \tau_2\ (r_1 \cup r_2) = \text{conv}\ \tau_1\ \tau_2\ r_1 \cup \text{trans}\ \tau_1\ \tau_2\ r_2$$

$$\text{trans}\ \delta\ (\text{trans}\ \delta'\ r) = \text{trans}\ (\delta + \delta')\ r \qquad \text{trans}\ \delta\ (r_1 \cup r_2) = \text{trans}\ \delta\ r_1 \cup \text{trans}\ \delta\ r_2$$

Figure 4: Some algebraic identities on region expressions.

```
normalize :: RegExp u -> RegExp u
normalize (Union a b)
  = mkUnion (normalize a) (normalize b)
normalize (Inter a b)
  = mkInter (normalize a) (normalize b)
normalize (Trans xy r)
  = mkTrans xy (normalize r)
normalize (Convert t r)
  = mkConvert t (normalize r)
normalize t             = t
```

```
mkUnion :: RegExp u -> RegExp u -> RegExp u
mkUnion a     Univ        = Univ
mkUnion Univ  b           = Univ
mkUnion a     Empty       = a
mkUnion Empty b           = b
mkUnion a     (Inter x y)
  = Inter (mkUnion a x) (mkUnion a y)
mkUnion a     b           = Union a b
```

Figure 5: The expression `normalize r` puts `r` into a normal form.

Removing interpretive overhead. Using the syntax as data approach injects a layer of interpretive overhead into the implementation. Executing a Region program now involves a call to the `eval` function. This call must walk over the structure of the data representing the Region program. In the embedded approach this extra overhead was not present. To remove this overhead[31] we employ the staging capabilities of Ωmega.

There are four staging annotations in Ωmega. Bracket `[| _ |]`, escape `$( _ )`, lift `(lift _)` and run `(run _)`. Brackets introduce a new code template, and specify that the expression inside the brackets should be generated as a program for potential execution in the next stage. Within brackets, escape specifies a hole within a template. The escaped expression is executed (resulting in a piece of code), and the resultant code is spliced into that hole. The escape annotation may only appear inside code brackets. Escape is the mechanism used for building larger pieces of code from smaller pieces. The `lift` annotation evaluates an expression of primitive type to a literal value and builds some trivial code that returns that value. The final annotation is `run`. It evaluates its argument – the result should then be a piece of code, and this code is then run. The run annotation is how Ωmega transitions from one stage to the next.

The staged version appears in Figure 6. The staged interpreter is implemented by the function `comp` which is very similar in structure to the unstaged interpreter `eval`. The transition from `eval` to `comp` is accomplished in a few simple steps:

1. Stage the `Mag` datatype so that it contains `(Code Int)` instead of `Int` and `(Code Float)` instead of `Float`. This allows us to know the units of a magnitude at "compile time", even though we will not know its value until "run time". The definition of `CodeMag` accomplishes this staging.
2. Stage the primitive functions on `Mag` values to work with this new type. We follow the convention that the staged versions of helper functions are given the same name but suffixed with an "S". Each of these staged functions is derived from the original in a systematic way. For example, the pixel versions of the original and staged version of the circle combinators are as follows:

```
circle :: Mag u -> Region u
circle (Pix r) = (\x y -> ((x * x) + (y * y)) <= (r * r))

circleS :: Mag u -> CodeMag u -> CodeMag u -> Code Bool
circleS (Pix r) = \ (CodePx x) (CodePx y) -> [| (($x * $x) + ($y * $y)) <= $(lift (r*r)) |]
```

```
-- staged magnitude datatype
data CodeMag u
  = CodePix (Code Int) where u = Pixel
  | CodeCm (Code Float) where u = Centimeter

-- compiled region type synonym
type CReg u =
        CodeMag u -> CodeMag u -> Code Bool

-- helper function
compTrans :: CoordTrans u v ->
                 CodeMag v -> CodeMag u
compTrans CM2PX (CodePix i) =
        CodeCm [| rem $i $(lift scale) |]
compTrans PX2CM (CodeCm i) =
        CodePix [| $i * $(lift scale) |]

-- staged interpreter
comp :: RegExp u -> CReg u
comp Univ           = univS
comp Empty          = emptyS
comp (Circle r)     = circleS r
comp (Rect w h)     = rectS w h
comp (Union a b) = unionS (comp a) (comp b)
comp (Inter a b) = intersectS (comp a) (comp b)
comp (Trans xy r)   = transS xy (comp r)
comp (Convert tr r) = convertS (compTrans tr)
                               (comp r)
```

Figure 6: Staged implementation.

Note that these differ only by the addition of the staging annotations `[| _ |]`, `$( _ )`, and `(lift _)`.

3. Stage the `Region` type so that it is defined in terms of `CodeMag` instead of `Mag`. The definition of `CReg` (Compiled `Region`) accomplishes this staging.

4. Redefine the region combinators and interpreter `eval` to manipulate `CReg` values instead of `Region` values. The updated definitions is given in terms of the staged primitives from step 2. This is accomplished by `comp` and its helper functions.

This figure demonstrates that the structure of the staged implementation is exactly the same as the unstaged implementation modulo staging annotations (compare Figure 6 with Figures 2 and 3. Once we stage the types to reflect what information should be static (known at compile time) versus dynamic (known at run time), the changes made to the definitions are completely systematic. In fact, we have had some success in building automatic tools to accomplish this[28, 15]. These tools borrow from research on staging and research on binding-time analysis from the partial evaluation community. We believe it is possible to completely automate this task once the types are staged. It is a research goal to demonstrate this.

Lessons of the Region DSL example. By using a meta-programming system with the right features we have accomplished several things.

- We built an intensional representation for *typed* region expressions. Type-equality constraints and user-defined kinds allowed us to define a data structure for well-typed abstract syntax[23], thus combining the type-safety of the combinator approach with the optimization possibilities of a data structure approach.
- The intensional approach is still embedded, so we can apply the full power of the host language in constructing region programs. For example the code below constructs three adjacent circles. The resulting data structure `r` is relatively large and complicated.

```
f :: [RegExp Centimeter] -> RegExp Centimeter
f []     = Empty
f (x:xs) = Union x (Trans (Cm 2,Cm 0) (f xs))

c,r :: RegExp Centimeter
c = Circle (Cm 1)
r = Convert CM2PX (Trans (Cm (0-2),Cm 0) (f [c,c,c]))
```

```
prompt> r
(Convert CM2PX (Trans ((Cm -2),(Cm 0))
    (Union (Circle (Cm 1))
           (Trans ((Cm 2),(Cm 0))
                  (Union (Circle (Cm 1))
                         (Trans ((Cm 2),(Cm 0))
                                (Union (Circle (Cm 1))
                                       (Trans ((Cm 2),(Cm 0)) Empty)))))))) : RegExp Pixel
```

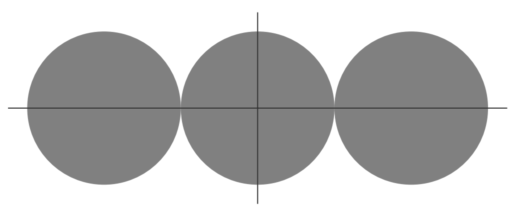
- The intensional representation allows for domain specific transformations like our `normalize` function. Such transformations can dramatically reduce the size and complexity of the region's representation.

```
prompt> normalize r
(Union (Trans ((Pix -144),(Pix 0)) (Circle (Pix 72)))
       (Union (Trans ((Pix 0),(Pix 0)) (Circle (Pix 72)))
       (Trans ((Pix 144),(Pix 0)) (Circle (Pix 72))))) : RegExp Pixel
```

- Staging allows the removal of all interpretive overhead incurred by creating an intensional representation, so that our technique is no less efficient than the combinator approach (in fact, if domain-specific optimizations are used, DSL programs can be implemented more efficiently this way).

```
prompt> [| \(Pix a) (Pix b) ->
           $(comp (normalize r) (CodePix [|a|]) (CodePix [|b|])) |]
[| \ (Pix x) (Pix y) ->
     (x + 144) * (x + 144) + (y * y) <= 5184 ||
     (x * x) + (y * y) <= 5184              ||
     (x - 144) * (x - 144) + (y * y) <= 5184
|] : Code ((Mag Pixel) -> (Mag Pixel) -> Bool)
```

- The Region DSL can be interpreted in other ways (besides its denotation as a characteristic function). For example, we can translate the `RegExp`s into PostScript. This picture was generated in Ωmega from the region `r` defined above.



The ability to interpret a DSL program in several ways (as a characteristic function, or as a PostScript file) is an important benefit of this approach.

5 DSL Evolution Research Problems

The research proposed falls roughly into three categories. First, research into improving host languages: since embedded DSLs are by far the most cost-effective mechanism for DSL implementation, improvements to host languages that allow DSLs to survive longer as embedded languages should pay off easily. Second, techniques that support the transition of embedded languages into stand-alone ones: automation of the process of transforming embedded implementations into stand-alone ones should make this transition cheaper. Third, new tools for stand-alone language construction: existing automatic tools for describing compiler stages such as parsing, syntax-directed translation, and code generation can be augmented by new tools for other key stages.

Improving Host Languages. The embedded approach to DSL implementation requires a feature-rich host language. Many of the features we developed for meta-programming (e.g., extensible kinds, equality qualified types), in retrospect, have been very useful as host language features. We propose research into transferring these techniques, and developing new techniques, for enhancing the hosting facilities of existing languages.

- **Syntactic Support.** Within the embedded approach, DSL programs are written as host language programs, so it is difficult to cope with syntactic structures in the DSL that are radically different from those of the host language. To alleviate this problem we propose *typed* macro-like extensions to the host language. By embedding the macro system within the host language (rather than using a "string-based" preprocessor), and by using typed macros, we can alleviate many of the problems associated with existing macro systems.

- **Extensible Type systems.** In an embedded DSL implementation it is difficult to enforce a type discipline that is richer, or more restricted than, or incomparable with, the host language's type system. Embedding of new (object-language) type systems inside the type system of the meta-language is one way to make this possible. Our experience [20] with extensible kinds and equality qualified types [5], leads us to believe that this is a viable approach to user-defined type-systems. We must also research ways to structure such systems modularly.

- **Language Restriction.** In an embedded DSL implementation it is also difficult to restrict the expressiveness of the DSL (since the host language is almost certainly Turing-complete). We propose to develop ways to define restricted subsets of a host language, which can be respected by error reporting, documentation, etc.

- **Staging.** Embedded DSLs must rely on the host language implementation and target environment, which may be inefficient, or incompatible for a particular DSL, adversely affecting performance. We propose adding staging facilities (including dynamic loading and run time code generation) to the host language to permit programs to specialize themselves to run-time invariants, as a mechanism for improving performance problems in embedded DSLs.

- **Reflection.** Embedded DSLs do not usually have the ability to manipulate the

  DSL program symbolically, since it is a host language program and not a data structure. As we have shown, the use of typed abstract syntax supports the intentional analysis of DSL programs. We propose adding support for reflective meta-programming to the host language. The reflection supports the ability to construct typed abstract syntax automatically, from a embedded DSL. Recent result in the semantics of staged programs[21] and normalization by evaluation [6, 7] suggest that this is not only possible, but actually practical.

Extracting Stand-alone Implementations from Embedded Implementations. If a DSL has an embedded implementation, it would be useful to reuse it by transforming it into a stand-alone implementation. We propose several tools and mechanisms to make this possible.

- **Intermediate Language Extraction.** As we showed in Section 4.1 an embedded DSL and staged implementation can have a lot in common when viewed through the right set of rose colored glasses. By using the reflective capabilities of the host language (the ability to build a data structure from the host language program) and binding-time analysis tools we believe we can extract a staged implementation from an embedded one in a mostly automated way. Using such a tool a stand-alone implementation for the DSL can then be constructed that completely shares the front-end part of the embedded implementation, including parsing, type checking, and semantic analysis.

- **Deriving DSL Type Systems by Specialization.** An embedded DSL automatically inherits many features from its host language, including a type system: a DSL program can be treated as type correct if, and only if, its translation into the host language is type correct. In practice, however, it can be difficult for users of the DSL to understand such a type system, or to make sense of type error diagnostics at the level of the DSL. Instead, users will often need detailed knowledge of the mapping from DSL to host, and of the host's type system, which may have many complications that are not relevant to the domain to which the DSL is targeted. We propose tackling these problems by developing new techniques for type system specialization that will allow specialized versions of the host's type system to be generated automatically, trimming out as many of the unnecessary features as possible. This specialized type system could then be applied to a stand-alone implementation.

- **Host Language Retargeting.** We propose developing new methods for easily retargeting host language translators to generate code for new execution environments. Existing retargetable compilers support code generation for multiple machine architectures, typically with the aid of declarative specifications (e.g., instruction selection templates). Our goal is to extend this approach to handle a much wider variety of execution environment parameters, including target language (*e.g.*, machine code *vs.* C or other high-level language code), memory model (*e.g.*, virtually unbounded stack and heap *vs.* tightly constrained physical memory), procedure model (*e.g.*, stack-based activations *vs.* completely unfolded), and data layout (*e.g.*, optimized by compiler *vs.* dictated by interoperability constraints). These retargetable code generators will be highly modularized and easily specified using a variety of specialized formalisms.

Stand-alone Phases and Tools. Parser generation was one of the first great successes of computer science. The theories of regular expressions and context free grammars provide a firm foundation for practical tools like `lex` and `yacc`. These are both prime examples of DSLs in their own right, each allowing a particular aspect of programming language syntax to be described at a high-level of abstraction and then translated automatically into a working recognizer or parser. There are similar gains to be made in other areas of compiler implementation.

- **Types.** Tools for the design and realization of type systems have not yet reached the maturity level of syntax tools. For example, there is no uniformly accepted, general framework for describing type systems. Moreover, practical type inference or type checking engines are still written as monolithic units that are constructed in an ad-hoc fashion and at a low level of abstraction. However, based on recent experience, we believe that a more generic approach to type system construction, together with corresponding tool support, is now well within reach. We therefore propose building a type checker/inferencer generator that will turn high-level descriptions of type systems into executable type checkers. The technical starting point for this work is a combination of constraint-based techniques and generalized relations over term and type algebras.
- **Translation via Semantics Combinators.** It has been a long unrealized dream of language implementers to define a language by writing a set of semantic equations that define its meaning, and then using these semantic equations to define the translation of abstract syntax into an *efficient* target language program. We propose to explore a rapid prototyping mechanism using a library of semantics combinators to define the meaning of a new DSL. In this approach, a fixed library of combinators will be used to define those aspects of a DSL that do not depend upon domain-specific algorithms, while a domain-specific library will provide its specialized algorithms. A compiler for the DSL can be obtained by re-interpreting its semantics combinators as code generators rather than interpretation functions. Recent work on modular semantics using monads provides the basis for this approach.
- **Strategy-directed rewriting systems for optimization.** Conventional optimizers are complicated, hand-crafted pieces of code, which form the bulk of a modern language translator implementation. Automating the generation of optimizers is therefore key to simplifying the production of translators. A promising approach is to specify optimizations by a set of rewrite rules, which describe valid program transformations, and a separate rewriting strategy, which is used to describe when and how the various rules should be applied. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer in a modular fashion, compared to traditional monolithic optimizer implementations. To date, only small languages and optimizations have been handled in this fashion; we are ready to extend these ideas to more substantial examples.

These three broad classes of research provide a road map to the grand vision of making the introduction of a new DSLs feasible. A single investigator proposal must focus on a small subset of these ideas. Our current research on typeful meta-programming systems has generated some unique new approaches to some of the problems outlined above. We plan to concentrate our efforts on the following: (1) extensible type systems; (2) staging; (3) reflection; (4) host language retargeting; and (5) translation via semantic combinators.

6 Broader Impacts

The OGI School of Engineering is in a unique position to make a broad impact in the design and use of Domain Specific Languages. The feature that distinguishes OGI's CSE department from other leading departments with similar levels of research activity is our close connection with industry. This is reflected not only in industrial support for our research, but also in our student body. Many of our Ph.D. students come to us from industry, and the majority of the students in our MS program are either part-time students currently employed in industry, or full- time students who have worked for some years in industry and have decided to continue their studies at a higher level. As such our students are motivated by the real problems of commercial software development, and are open to trying new and better ways of deploying software.

The P.I has taught a graduate level course *The Design and Development of Domain Specific Languages* twice in the past, and is scheduled to teach it again in the spring of 2004. The current plan is to use the preliminary version of the Ωmega interpreter as a candidate host language. The plan is to develop a set of course notes as the start to a set of instructional material on DSL development. The experience of using Ωmega as a host language should motivate new research directions.

7 Conclusion

We propose three activities that will help lower the cost of DSL deployment. (1) Improving host languages. (2) Extracting stand-alone implementations from embedded ones. (3) Improving stand-alone phases and tools.

Our experience with meta-programming languages suggests that much of the research in that area can be directly applied to the first activity. The features of typed abstract syntax trees and staging supports an approach to DSL implementation that provides many of the advantages of both the embedded and stand alone approaches. This is important because host language extension seems to be the most cost effective way of evolving domain specific language implementations. Many of the features of meta-programming systems seem ideal as host language extensions.

Advanced meta-programming techniques such as automatic binding time analysis and reflection seem promising approaches to extraction of stand-alone implementations from embedded ones. Finally, Building stand-alone implementations via semantic combinators is also supported by research into meta-programming systems.

References

[1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[2] H. P. Barendregt. Lambda calculi with types. In D. M. Gabbai Samson Abramski and T. S. E. Maiboum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.

[3] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(12):545–572, May 2003.

[4] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, Rice University, 2002.

[5] James Cheney and Ralf Hinze. Phantom types. Available from `http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf.`, 2003.

[6] Olivier Danvy. Type-directed partial evaluation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[7] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

[8] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey*, pages 184–195. IEEE Computer Society Press, July 1996.

[9] Rowan Davies and Frank 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, January 1996.

[10] Levent Erkök and John Launchbury. Recursive monadic bindings. *ACM SIGPLAN Notices*, 35(9):174–185, September 2000.

[11] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, June 1993.

[12] Simon Peyton Jones. Composing contracts: An adventure in financial engineering. *Lecture Notes in Computer Science*, 2021:435–??, 2001.

[13] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering*, pages 542–552. IEEE Computer Society Press / ACM Press, 1996.

[14] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. *Lecture Notes in Computer Science*, 2257:137–??, 2001.

[15] Nathan Linger and Tim Sheard. Binding-time analysis for metaml via type inference and constraint solving. In *Tenth International Conference On Tools And Algorithms For The Construction And Analysis Of Systems*. LNCS, 2004.

[16] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.

[17] MetaOCaml Hompage. Available online from `http://cs-www.cs.yale.edu/homes/taha/metaocaml/`.

[18] 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.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):528–569, May 1999.

[20] Emir Pasalic. *Heterogeneous Meta-programming*. PhD thesis, Oregon Health & Sciences University, OGI School of Science and Engineering, 2004. Forthcoming.

[21] Andrew Pitts and Tim Sheard. On the denotational semantics of staged execution of open code. In *Submitted to – Nineteenth Annual Ieee Symposium On Logic In Computer Science*, 2004.

[22] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[23] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. *ACM SIGPLAN Notices*, 34(1):313–323, January 1999.

[24] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–239, 1999.

[25] 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.

[26] T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 1–16. ACM, 2002.

[27] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 2–44, Berlin, September 2001. Springer Verlag. Invited talk.

[28] Tim Sheard and Nathan Linger. Search-based binding time analysis using type-directed pruning. In *Proceedings of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM)*, pages 20–31. ACM Press, 2002.

[29] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[30] Walid 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 Manipulation (PEPM'00)*, January 2000.

[31] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and jones-optimality. *Lecture Notes in Computer Science*, 2053:257–??, 2001.

[32] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[33] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357–??, 2001.

[34] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[35] L. Walton and J. Hook. Message Specification Language (MSL): A Domain Specific Design Language for Message Translation and Validation. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Portland, OR, August 1994.