Binding-Time Analysis for MetaML via Type Inference and Constraint Solving

Nathan Linger and Tim Sheard

Oregon Graduate Institute at Oregon Health & Science University

Abstract. The two predominant program specialization techniques, partial evaluation and staged programming, take opposite approaches to automating binding-time analysis (BTA). Despite their common goal, there are no systems integrating both methods. Programmers must choose between the precision of manually placing staging annotations and the convenience of automating such annotation.

We present an automatic BTA algorithm for a subset of MetaML. Such an algorithm provides a basis for a system integrating staged programming and partial evaluation because it allows programmers to switch between automatic and manual staging. Our algorithm is based on typing algorithm coupled with arithmetic-constraint solving. The algorithm decorates each subexpression of both a program and its type with numeric variables representing staging-annotations and then generates simple arithmetic constraints that describe the space of all possible stagings of the original program. Benefits of our approach include expressive BTA specifications in the form of stage-annotated types as well as support for polyvariance.

1 Background

Program specialization requires a *binding-time analysis* (BTA) to divide a program into static parts that may be computed statically and dynamic parts that must be residualized. BTA can be thought of as placing *staging annotations* on a program to partition it into static and dynamic parts. The two predominant techniques for program specialization, namely *partial evaluation* and *staged programming*, differ in where they place the responsibility for BTA. Partial evaluation performs BTA automatically, while staged programming forces the programmer to place staging annotations manually. Both approaches have merit. The automatic approach is easier on the programmer, but the manual approach can be more precise. In this paper we:

1. Bridge the gap between the two kinds of systems. We show how both automatic and manual BTA can co-exist in a single system that reaps the benefits of both approaches.

2. Reemphasize the point made by Henglein[1], that BTA can be thought of as a typing problem, rather than thinking of it exclusively as a static analysis or abstract interpretation problem. This leads us to discover two new connections between BTA and types.
   First, that the input to binding time analysis is really a *staged type*. Using staged types as BTA specifications is more precise, and more flexible than previous specifications, and integrates seamlessly with manual BTA.
   Second, we hypothesize a new relationship between polyvariance and polymorphism: that the former is simply an instance of the latter in a richer type system.
3. Identify arithmetic constraints that describe all possible ways to correctly order the stages of a program. Rather than searching for a correct staging of a program, we can infer a set of constraints that describes the entire family of solutions. This reduces a search problem to a type inference problem that is simpler and more efficient.

1.1 Staged Programming as Manual BTA

Binding time analysis can be thought of as the placement of *staging annotations* to partition a program into static and dynamic parts. The most fundamental staging annotations are bracket ($\langle e \rangle$) and escape ($\tilde{}e$). These are complementary annotations that signal transitions between the static and dynamic stages. The bracket annotation delays a computation into the dynamic stage, thus producing a code fragment (similar to the backquote in LISP's quasiquote notation). Conversely, the escape annotation returns back to the static stage to calculate a code fragment, which is then spliced into a larger code fragment (similar to comma in LISP's quasiquote notation). The cognizant difference between LISP's quasiquote and staging annotations is that staging annotations respect static scoping. We assume the reader is somewhat familiar with staged languages. If not, several good sources exist [2–4].

A staging of an unstaged term (one without any code brackets or escapes) can be obtained by strategically placing these annotations within the term. Finding the correct locations to place these annotations can be thought of as a search problem or an inference problem.

At the type level, the bracket annotation $\langle t \rangle$ is overloaded as a type constructor for code fragments. For example, $\langle \mathbf{int} \rangle$ (read "code of int") is the type of code fragments of type *int*. The type signature $f :: \mathbf{int} \rightarrow \langle \mathbf{int} \rangle \rightarrow \langle \mathbf{int} \rangle$ says that f has two parameters, the first is a static integer, the second is a dynamic integer (a code fragment of type $\mathbf{int}$), and f

returns a dynamic integer as a result. This syntax comes from the staged programming language MetaML [2].

An unstaged type (one without any code brackets) can be *staged* in many ways by adding pairs of brackets around sub-terms in the type. For example the type **int** $\rightarrow$ **bool** can be staged in all of the following ways (as well as infinitely many others).

$$\mathbf{int} \rightarrow \langle\mathbf{bool}\rangle \qquad \langle\mathbf{int} \rightarrow \mathbf{bool}\rangle \qquad \langle\langle\mathbf{int}\rangle\rangle \rightarrow \mathbf{bool} \qquad \langle\mathbf{int} \rightarrow \langle\mathbf{bool}\rangle\rangle$$

The staging of a term and its type are closely related. Placing staging annotations on a term in a legal manner corresponds to staging its type as well.

To illustrate this idea concretely, consider the specialization of the power function obtained by annotating its unstaged definition. We obtain such a definition by strategically placing $\langle - \rangle$ around subterms in the function's type, and $\langle - \rangle$ and $\tilde{}\,-$ around subterms in the function's body.

$$\begin{array}{l} pow_1 :: \mathbf{int} \rightarrow \langle\mathbf{int}\rangle \rightarrow \langle\mathbf{int}\rangle \\ pow_1\ n\ x = \mathbf{if}\ n = 0\ \mathbf{then}\ \langle 1\rangle\ \mathbf{else}\ \langle \tilde{}x * \tilde{}(pow\ (n-1)\ x)\rangle \end{array}$$

Note that this staged version of *pow* has a staged type: $\mathbf{int} \rightarrow \langle\mathbf{int}\rangle \rightarrow \langle\mathbf{int}\rangle$ which is a staging of its original type $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$. We can use the staged power function to residualize a dynamic code fragment: $pow_1\ 3\ \langle i\rangle \hookrightarrow \langle i * i * i * 1\rangle$.

Placing the staging annotations on the *pow* function in a different manner may lead to different staged types. Perhaps surprisingly, the staged type is an excellent specification for how to stage the term. This is illustrated in Figure 1 where we list all possible two-level stagings of the power function. Of all the staged types given in Figure 1, only a few lead to multiple valid terms. Those that do, differ only in whether the test of the conditional is static or dynamic. Of these two choices one is always "better" (more static) than the other. This leads us to consider staged types as a specification of what we want BTA to accomplish. We further discuss the expressiveness of such specifications in Section 1.4.

1.2 Automatic vs. Manual BTA

Since fully automatic methods for BTA exist, why do we bother with manual methods? Because automation often comes at the price of precision.

1. Automatic BTA can be too aggressive: Systems based upon automatic BTA may build infinite programs by indefinitely inlining recursive

1. $\langle int \to int \to int \rangle$
   a. $\langle \mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * pow\ (n-1)\ x \rangle$
2. $int \to \langle int \to int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \langle \lambda x\ .\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * \tilde{}(pow\ (n-1))\ x \rangle$
   b. $\mu\, pow\ .\ \lambda n\ .\ \langle \lambda x\ .\ \tilde{}(\textbf{if}\ n = 0\ \textbf{then}\ \langle 1 \rangle\ \textbf{else}\ \langle x * \tilde{}(pow\ (n-1))\ x \rangle) \rangle$
3. $int \to int \to \langle int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \langle \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * \tilde{}(pow\ (n-1)\ x) \rangle$
   b. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \textbf{if}\ n = 0\ \textbf{then}\ \langle 1 \rangle\ \textbf{else}\ \langle x * \tilde{}(pow\ (n-1)\ x) \rangle$
4. $int \to \langle int \rangle \to \langle int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \langle \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ \tilde{}x * \tilde{}(pow\ (n-1)\ x) \rangle$
   b. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \textbf{if}\ n = 0\ \textbf{then}\ \langle 1 \rangle\ \textbf{else}\ \langle \tilde{}x * \tilde{}(pow\ (n-1)\ x) \rangle$
5. $\langle int \rangle \to \langle int \to int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \langle \lambda x\ .\ \textbf{if}\ \tilde{}n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * \tilde{}(pow\ \langle \tilde{}n - 1 \rangle)\ x \rangle$
6. $\langle int \rangle \to int \to \langle int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \langle \textbf{if}\ \tilde{}n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * \tilde{}(pow\ \langle \tilde{}n - 1 \rangle\ x) \rangle$
7. $\langle int \rangle \to \langle int \rangle \to \langle int \rangle$
   a. $\mu\, pow\ .\ \lambda n\ .\ \lambda x\ .\ \langle \textbf{if}\ \tilde{}n = 0\ \textbf{then}\ 1\ \textbf{else}\ \tilde{}x * \tilde{}(pow\ \langle \tilde{}n - 1 \rangle\ x) \rangle$

Fig. 1. Every possible two-stage version of the power function. (The expression form $\mu\, x\ .\ e$ is a notation for fixed points which is equivalent to $\textbf{fix}\ (\lambda x\ .\ e)$.) Programs types with staging annotations can be considered a *specification* of the program's staging behavior.

functions. In Figure 1, programs 2a, 3a, 4a, 5, 6, and 7 exhibit this problem. In each case, the problem is that the conditional expression's test is dynamic but the enclosed recursive call is static: so every call to *pow* results in another recursive call to *pow*.

We see that partial evaluation of a terminating program may not terminate. This is not particularly surprising since the purpose of specialization is to change a program's evaluation order. Ensuring termination of specialization is a serious problem, and is undecidable in general.

2. Automatic BTA can be too conservative: It may fail to recognize parts of the program as static and thereby unnecessarily delay computation. In Figure 1, programs 2b, 3b, and 4b exhibit this behavior: the test in the conditional expression is made unnecessarily dynamic (a fact witnessed by the alternate stagings 2a, 3a, and 4a). In a manually staged system the programmer is responsible for avoiding these situations.

   In an automatically staged system, the expert who understands the inner workings of an automatic BTA algorithm can refactor the original program so that the BTA algorithm identifies more static computations. Such refactorings are called *binding-time improvements*. The need for binding-time improvements shows that even automatic BTA is not always so automatic.

A system combining automatic and manual BTA could reap the benefits of both worlds.

1.3 Combining Automatic and Manual BTA

One way to integrate manual and automatic BTA is to add a **stage** keyword to a staged programming language such as MetaML. The programmer could then make the following declaration:

$$\textbf{stage}\ pow_2 = pow\ \textbf{at}\ \textbf{int} \rightarrow \langle\textbf{int}\rangle \rightarrow \langle\textbf{int}\rangle;$$

Such a declaration would instruct the compiler to calculate a staged version of *pow* (either program 4a or 4b in Figure 1) and assign that function to pow_2. For simple programs like *pow*, automatic BTA can construct the correct version. For more complex examples, the programmer can take control by manually placing the staging annotations.

Behind this declaration lies an automatic BTA algorithm that adds staging annotations to the definition of *pow* as specified by the staged type $\textbf{int} \rightarrow \langle\textbf{int}\rangle \rightarrow \langle\textbf{int}\rangle$ to obtain a new function pow_2.

1.4 Staged Types As Expressive BTA Specifications

As BTA specifications, staged types are much more expressive than the standard distinction between static and dynamic. In many systems a BTA specification is nothing more than a partition of a function's arguments into static and dynamic.

There are two refinements to this basic form of BTA specifications. One refinement is partially static data. Here, some part a function's argument is statically known, but other parts are not. For example, an argument of type $\textbf{list}\ \langle\textbf{bool}\rangle$ is a list whose structure is known statically, but whose elements are dynamic. The second refinement is to allow staging a function into more than two stages. Thus a simple static vs. dynamic dichotomy is no longer sufficient. Here, a BTA specification may specify a function's argument as: $\langle\langle\textbf{int}\rangle\rangle$, the type of an integer that will not be calculated until stage 3.

Staged types can describe both these refinements. They can also specify staging behaviors that go beyond these simple refinements. For example, consider a function of type $(\textbf{int} \rightarrow \textbf{list}\ \langle\textbf{bool}\rangle) \rightarrow \langle\textbf{int}\rangle$. Its argument is neither static nor dynamic, but something much more refined. Only staged types can describe this kind of higher-order partially-static data.

2 Previous Work

In earlier work [5], we developed a search based approach to BTA for a staged language. The key idea is to walk over the body of a function, considering the addition of staging annotations $\langle - \rangle$ and $\tilde{}\ -$ to every subterm. One can use type information to prune the search space of all annotations that lead to ill-typed programs. The search can be specified in a non-deterministic way by a set of inference rules for the judgment $\Gamma \vdash_n (e : t) \sqsubseteq (e' : t')$ whose key inference rules appear in Figure 2.

$$[\textsc{Int}]\ \frac{}{\Gamma \vdash_n (i : \mathbf{int}) \sqsubseteq (i : \mathbf{int})} \qquad [\textsc{Var}]\ \frac{\Gamma(x) = (t_2, m) \quad n \geq m}{\Gamma \vdash_n (x : t_1) \sqsubseteq (x : t_2)}$$

$$[\textsc{Lam}]\ \frac{\Gamma, x : (t_2, n) \vdash_n (e_1 : s_1) \sqsubseteq (e_2 : s_2) \quad t_1 \sqsubseteq t_2}{\Gamma \vdash_n ((\lambda x : t_1 \,.\, e_1) : t_1 \to s_1) \sqsubseteq ((\lambda x : t_2 \,.\, e_2) : t_2 \to s_2)}$$

$$[\textsc{App}]\ \frac{\begin{array}{l} s_1 \sqsubseteq s_2 \\ \Gamma \vdash_n (e_1 : s_1 \to t_1) \sqsubseteq (e_2 : s_2 \to t_2) \\ \Gamma \vdash_n (e_1' : s_1) \sqsubseteq (e_2' : s_2) \end{array}}{\Gamma \vdash_n (e_1\ e_1' : t_1) \sqsubseteq (e_2\ e_2' : t_2)}$$

$$[\textsc{Code}]\ \frac{\Gamma \vdash_{n+1} (e_1 : t_1) \sqsubseteq (e_2 : t_2)}{\Gamma \vdash_n (e_1 : t_1) \sqsubseteq (\langle e_2 \rangle : \langle t_2 \rangle)} \qquad [\textsc{Escape}]\ \frac{\Gamma \vdash_n (e_1 : t_1) \sqsubseteq (e_2 : \langle t_2 \rangle)}{\Gamma \vdash_{n+1} (e_1 : t_1) \sqsubseteq (\tilde{}\, e_2 : t_2)}$$

$$[\textsc{Lift}]\ \frac{\Gamma \vdash_n (e_1 : c) \sqsubseteq (e_2 : c) \quad c \in \{\mathbf{int}, \mathbf{bool}\}}{\Gamma \vdash_n (e_1 : c) \sqsubseteq (\mathbf{lift}\ e_2 : \langle c \rangle)}$$

Fig. 2. Inference rules for type-directed BTA based on search. The meaning of the judgment $\Gamma \vdash_n (e : t) \sqsubseteq (e' : t')$ is that e' has type t' in Γ at level n, e has type t in $erase(\Gamma)$, $e = erase(e')$, and $t = erase(t')$ (where $erase$ is the operation of erasing all staging annotations).

These rules are doing two typing derivations in parallel: one with annotations and one without. Doing the derivations in parallel allows the rules to ensure that all the annotated expressions and types have the same underlying structure as the corresponding unannotated ones.

The algorithm has Γ, e, t, and t' as inputs and searches for an e' such that $\Gamma \vdash_0 (e : t) \sqsubseteq (e' : t')$. However, the rules are not syntax directed on the inputs, and this leads to an implementation based upon back-tracking search. Such an algorithm works [5], but can be inefficient in many cases.

The fundamental problem is that information is flowing in the wrong direction. We want information to flow from the type to the program: deriving program annotations from type annotations. Type inference (which, for functional languages, is a sub-problem of type checking) moves information in the opposite direction, types are derived from programs. Annotation information flowing in the wrong direction means we sometimes have to guess when to add program annotations. Is there any way around this problem?

3 Annotation Variables

One way to resolve this problem of the direction of information flow is introduce *annotation variables* and generalize the type system to generate constraints on those variables. Solutions to generated constraints determine valid (concrete) staging annotations for a program. Then all the choices that forced guessing in the search-based approach can be expressed as constraints and delayed until constraint-solving time.

What sorts of values should an annotation variable have? Any given subterm e can be annotated with an escape $\tilde{}\, e$ or brackets $\langle e \rangle$, or any combination of the two $\tilde{}\langle \tilde{} e \rangle$. However, since the two annotations cancel each other, we can normalize annotation sequences of this sort to be either all escapes or all brackets (or empty). We may encode all such annotations as integers: positive integers say how many brackets to insert; negative integers say how many escapes to insert; and zero says insert nothing. Since MetaML is a *multistage* language (it can support more than 2 program stages), annotation variables can take on values other than 1, 0, and -1. The following table shows the concrete annotations corresponding to different values of an annotation variable j that is annotating an expression e.

$$\begin{array}{rcccccc} j \text{ value}: & 0 & 1 & -1 & 2 & -2 & \cdots \\ e^j \text{ value}: & e & \langle e \rangle & \tilde{} e & \langle\langle e \rangle\rangle & \tilde{}\tilde{} e & \cdots \end{array}$$

Given a specific type with concrete annotations as a specification, matching it against the inferred type with variable annotations gives values for the annotation variables. If these values satisfy the constraints, we use them to instantiate the variable-annotated program with concrete annotations. We give a concrete example of this in Section 5.

4 Generalized Typing Rules

Figure 3 shows the subset of MetaML we will be working with and gives generalized typing rules. The superscript j represents a relative difference between levels: If e^j is at level n, then e is at level $n-j$. The main contribution of these typing rules is the [Ann] rule that generalizes both the [Code] and [Escape] rules from MetaML.

$$[\textsc{Ann}]\ \frac{C; \Gamma \vdash_{n+j} e : t^{j'-j}}{C \cup \{n+j \geq 0, j' \geq j\}; \Gamma \vdash_n e^j : t^{j'}}$$

$$[\textsc{Code}]\ \frac{\Gamma \vdash_{n+1} e : t}{\Gamma \vdash_n \langle e \rangle : \langle t \rangle} \qquad [\textsc{Escape}]\ \frac{\Gamma \vdash_n e : \langle t \rangle}{\Gamma \vdash_{n+1} \tilde{}\, e : t}$$

When $j = j' = 1$, the [Ann] rule instantiates to the [Code] rule. When $j = -1$ and $j' = 0$, it instantiates to the [Escape] rule. When $j = 0$, the [Ann] rule becomes vacuous.

In contrast to the previous set of inference rules, these typing rules are syntax directed. This fact is due to variable annotations. While we may not know the concrete annotations in an expression, we at least have handles for referring to them.

There are two binding constructs in the language, λ and μ. The former binds function parameters, and the latter binds fixed point values. The expression $\mu\, x\, .\, e$ is equivalent to the more familiar notations **fix** $(\lambda x\, .\, e)$ or **let** $x = e$ **in** x. Its purpose is for defining recursive functions.

The typing rules treat λ-bound and μ-bound variables differently. This is because of the MetaML feature called *cross stage persistence* that allows values bound in one stage to be used in later stages. Since this feature is not used in practice for recursive function variables, we tag this special case with a "μ" in the environment. All other variables are tagged with a "λ". The difference shows up in the rules: μ-bound variables are used only in the same stage at which they were bound, but λ-bound variables may also be used at later stages.

Each constraint in these typing rules serves a purpose. The constraint $(n \geq m)$ in the [λVar] rule ensures that each variable is not used at an earlier stage than the one in which it is bound. The fact that this constraint is an inequality rather than an equality allows for cross stage persistence. The constraints $valid(s)$ in the [App] rule and $(j' \geq j)$ in the [Ann] rule ensure that there are no escapes in types. The constraint $(n + j \geq 0)$ in the [Ann] rule ensures that there are no negative levels in expressions (i.e. no escapes at level zero).

An invariant guides the choice of where constraints go in the above rules. For the judgment $C; \Gamma \vdash_n e : t$, the invariant property is that the following annotations are all non-negative: the level n, all annotations

$$
\begin{aligned}
s, t &::= \mathbf{int} \mid \mathbf{bool} \mid s \to t \mid \mathbf{list}\ t \mid t^{j} \\
e &::= i \mid x \mid e\ e \mid \lambda x\ .\ e \mid \mu\, x\ .\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid e^{j} \\
\Gamma &::= \cdot \mid \Gamma, x : (t, n, \lambda) \mid \Gamma, x : (t, n, \mu) \\
C &::= \varnothing \mid \{n \geq m\} \mid C \cup C \\
n, m, j &::= 0 \mid v \mid n + m \mid n - m
\end{aligned}
$$

$$
[\textsc{Int}]\ \frac{}{\varnothing; \Gamma \vdash_n i : \mathbf{int}} \qquad [\textsc{If}]\ \frac{C_1; \Gamma \vdash_n e : \mathbf{bool} \quad C_2; \Gamma \vdash_n e' : t \quad C_3; \Gamma \vdash_n e'' : t}{C_1 \cup C_2 \cup C_3; \Gamma \vdash_n \mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' : t}
$$

$$
[\lambda\textsc{Var}]\ \frac{\Gamma(x) = (t, m, \lambda)}{\{n \geq m\}; \Gamma \vdash_n x : t} \qquad [\mu\textsc{Var}]\ \frac{\Gamma(x) = (t, n, \mu)}{\varnothing; \Gamma \vdash_n x : t}
$$

$$
[\textsc{Lam}]\ \frac{C; \Gamma, x : (s, n, \lambda) \vdash_n e : t}{C; \Gamma \vdash_n \lambda x\ .\ e : s \to t} \qquad [\textsc{Fix}]\ \frac{C; \Gamma, x : (s \to t, n, \mu) \vdash_n e : s \to t}{C; \Gamma \vdash_n \mu\, x\ .\ e : s \to t}
$$

$$
[\textsc{App}]\ \frac{C_1; \Gamma \vdash_n e' : s \quad C_2; \Gamma \vdash_n e : s \to t}{C_1 \cup C_2 \cup \mathit{valid}(s); \Gamma \vdash_n e\ e' : t}
$$

$$
[\textsc{Ann}]\ \frac{C; \Gamma \vdash_{n+j} e : t^{j' - j}}{C \cup \{n + j \geq 0, j' \geq j\}; \Gamma \vdash_n e^{j} : t^{j'}}
$$

$$
\mathit{valid}(\mathbf{int}) = \mathit{valid}(\mathbf{bool}) = \varnothing \qquad \mathit{valid}(\mathbf{list}\ t) = \mathit{valid}(t)
$$

$$
\mathit{valid}(s \to t) = \mathit{valid}(s) \cup \mathit{valid}(t) \qquad \mathit{valid}(t^{j}) = \mathit{valid}(t) \cup \{j \geq 0\}
$$

Fig. 3. Generalized syntax and typing rules with Constraints. The judgment $C; \Gamma \vdash_n e : t$ means "under constraints C, in typing environment Γ, at level n, expression e has type t". By convention, n and m denotes absolute levels and j denotes differences in levels.

in t, and all annotations in all types in Γ. It is easily checked that for every rule, if the property holds for the concluded judgment (below the line), it is also true for all the antecedent judgments (above the line). So if a derivation tree with conclusion $C; \Gamma \vdash_n e : t$ exists, we now know $C \cup \mathit{valid}(t)$ guarantees that every subexpression in e has a non-negative stage and a valid type.

5 The Algorithm in Action

Here we run through the BTA algorithm for the power function. First, we do regular type inference on the unannotated program to obtain its type, which we then annotate with fresh annotation variables.

$$
(\mathbf{int}^{a} \to (\mathbf{int}^{b} \to \mathbf{int}^{c})^{d})^{e}
$$

Then we annotate each sub-expression in the the original program with fresh annotation variables.

$$
\begin{array}{l}
(\mu pow\ .\ (\lambda pow\ .\ (\lambda n\ .\ (\lambda x\ . \\
\quad (\mathbf{if}\ ((=^f\ n^g)^h 0^i)^j \\
\qquad \mathbf{then}\ 1^k \\
\qquad \mathbf{else}\ ((*^l x^m)((pow^o((-^p n^q)^r 1^s)^t)^u x^v)^w)^x \\
\quad)^y)^z)^{a'})^{b'}
\end{array}
$$

We call annotation variables placed on the body of a function *slack* variables. Then we check that the body of the function has type $(\mathbf{int}^a \rightarrow (\mathbf{int}^b \rightarrow \mathbf{int}^c)^d)^e$. This generates constraints between the slack variables (from the body) and the variables placed on the type. The algorithm generates 31 equations and 57 inequations. We solve the equations for as many slack variables as possible (thus eliminating them) and then simplify the remaining inequations. The following inequations are left over

$$
a, b, c, d, e \geq 0 \qquad\qquad c \geq b \qquad\qquad -a + c + d \geq k \geq 0
$$

along with the following annotated program

$$
\begin{array}{l}
(\mu pow\ .\ \lambda n\ .\ (\lambda x\ . \\
\quad (\mathbf{if}\ n^{-a} = 0 \\
\qquad \mathbf{then}\ 1^k \\
\qquad \mathbf{else}\ (x^{-b} * ((pow\ \ (n^{-a} - 1)^a)^{-d}\ x)^{-c})^k \\
\quad)^{c-k})^d)^e
\end{array}
$$

At this point, we are done with the first phase of our algorithm. The result is *the complete space of valid stagings of the original program and its type*. For a single function, the results of this phase are reusable across BTA calls. This approach makes polyvariance very cheap to implement.

In phase two we pick out a single program from this space by picking out a single type. For example, if we want the annotated type $int \rightarrow \langle int \rangle \rightarrow \langle int \rangle$, we match this against $(\mathbf{int}^a \rightarrow (\mathbf{int}^b \rightarrow \mathbf{int}^c)^d)^e$, yielding the substitution $\{a, d, e \mapsto 0; b, c \mapsto 1\}$ which simplifies the constraint set to just $\{0 \leq k \leq 1\}$. Thus by picking a single (valid) staged type we have narrowed the space of possible programs down to these two

$$
\begin{array}{l}
k = 0 : \mu\ pow\ .\ \lambda n\ .\ \lambda x\ .\ \langle \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ \tilde{}x * \tilde{}(pow\ \ (n-1)\ \ x) \rangle \\
k = 1 : \mu\ pow\ .\ \lambda n\ .\ \lambda x\ .\ \mathbf{if}\ n = 0\ \mathbf{then}\ \langle 1 \rangle\ \mathbf{else}\ \langle \tilde{}x * \tilde{}(pow\ \ (n-1)\ \ x) \rangle
\end{array}
$$

We see that the “slack variable” k is not always determined by concrete annotations in the type. This happens because conditional expressions of code type sometimes have a choice between being performed statically or dynamically. If we always want the if statement to execute as soon as possible, we can maximize k symbolically by setting it to $-a+c+d$, yielding the following program

$$
\begin{array}{l}
(\mu pow\ .\ \lambda n\ .\ (\lambda x\ . \\
\quad (\mathbf{if}\ n^{-a} = 0 \\
\quad\quad \mathbf{then}\ 1^{-a+c+d} \\
\quad\quad \mathbf{else}\ (x^{-b} * ((pow\ (n^{-a} - 1)^{a})^{-d}\ x)^{-c})^{-a+c+d} \\
\quad)^{a-d})^{d})^{e}
\end{array}
$$

and the simpler constraint set

$$a, b, c, d, e \geq 0 \quad c \geq b \quad c + d \geq a$$

In general, there may be several upper bounds given for a slack variable s, in this case we replace s with a max (or min) operation on all of s's lower bounds (or upper bounds).

6 Polymorphism and Polyvariance

MetaML is a much richer language than the small one we have shown so far. Many features will need to be added to our language and BTA algorithm before we can handle full MetaML. The most significant extension is handling polymorphism.

In order to extend our language to handle Hindley-Milner polymorphism, we will need to extend the notion of a type scheme to include a constraint set. A type scheme would then have the form

$$
\begin{array}{rl}
\text{Type Scheme} & \sigma ::= \forall\{\alpha_1, \ldots, \alpha_n\}\ .\ \forall\{j_1, \ldots, j_m\}\ .\ C \Rightarrow \tau \\
\text{Type} & \tau ::= \mathbf{int} \mid \mathbf{bool} \mid \tau \rightarrow \tau \mid \mathbf{list}\ \tau \mid \tau^{a} \mid \alpha \\
\text{Constraints} & C ::= \varnothing \mid C, a \geq 0 \\
\text{Arithmetic Expression} & a ::= j \mid a + a \mid a - a
\end{array}
$$

Instantiating a type scheme requires fresh type variables as well as fresh annotation variables. The constraint set C is instantiated to these fresh annotation variables and then asserted. This is where a constraint solver/simplifier is important. Type schemes should carry as few constraints as possible so that there is less work to do when instantiating them. We need to remove all (or most) redundant constraints.

An interesting consequence of this approach is that polyvariance collapses to qualified polymorphism. Polyvariance is an advanced BTA feature that allows a function to be used at different staging signatures in different locations. Polymorphism is a feature of type systems that allows functions to have different types at different locations. The approach outlined here handles both features with a single mechanism.

These type schemes are reminiscent of the work on qualified types [6, 7]. However, the constraints C are not on types, but on annotations. The implications of this distinction are not yet clear.

7 Type Inference Implementation

We have a Haskell program that does type inference according to the generalized typing rules of the previous section. The algorithm produces many redundant inequalities so we have written an ad-hoc arithmetic inequality simplifier that eliminates many of them. This section addresses the issues that arise when moving from the typing rules to an actual algorithm.

The grammar in Figure 3 does not tell the whole story. In order to make sure we catch every possible annotation of the program, all subexpressions of terms and types should be annotated with a fresh variable initially. Figure 4 shows Haskell datatypes for annotated and unannotated types and expressions that capture this restriction. Similar datatypes are used in the implementation. If there is no annotation possible for a certain position, the generated constraints will constrain that variable to be zero.

```
data Expr e = Lit Int | Var Name | App e e | Lam Name e | ...
data Type t = TInt | TBool | TArr t t | TList t | TVar Name
data Base s = Base (s (Base s))
data Annotated s = Ann Step (s (Annotated s))
```

annotated expressions:	`Annotated Expr`
annotated types:	`Annotated Type`
unannotated expressions:	`Base Expr`
unannotated types:	`Base Type`

Fig. 4. Haskell two-level datatypes for annotated and unannotated expressions. Values of type **Step** are linear arithmetic expressions involving annotation variables. The form of **Annotated s** ensures that every **s** sub-structure is annotated.

Constraints coming from the *valid* operator are generated on the fly by annotating every freshly generated type variable with a fresh annotation variable that is asserted to be non-negative.

The typing rules contain implicit *equality* constraints. For example, in the [μVAR] rule, there are two occurrences of the stage expression n. Each may be represented by a different arithmetic expression. In this common case, there is an implicit equality constraint generated. A similar situation occurs whenever two types are unified – corresponding annotations must be set equal to each other.

8 Future Work

Correctness Proof. A correctness theorem akin to the "minimal completion" result of Henglein [1], would say that in staged programs resulting from BTA, all work is done at the earliest possible stage. This is a more difficult result to obtain in a multistage language.

Obtaining such a result for our BTA algorithm means solving the problem of slack variables. These slack variables represent a choice between multiple valid program annotations. We believe that the algorithm places slack variables in such a way that we can always either maximize or minimize them to obtain a minimal completion of the original program. This remains to be seen.

Efficient Solver. One benefit of using a standard form of constraints is that existing efficient constraint solvers can be plugged in to replace our ad-hoc solver. What existing constraint solvers are more efficient than our ad-hoc one?

William Pugh has developed the Omega test for solving Presburger arithmetic formulas [8]. Presburger formulas are first order logical formulas with arithmetic terms built from the symbols $+$, $-$, $=$, $\leq$, and variables ranging over integers. Though the general problem is NP-Complete, the Omega test has been found to be efficient in practice (of low order polynomial time complexity).

Our form of constraints is much more restricted than Presburger formulas. Our constraints contain no quantifiers, disjunctions, or implications. For this reason, we may be able to get away with a simpler solver.

9 Related Work

In the tradition of the seminal works of Nielson and Nielson [9] and Gomard and Jones [10], Henglein builds a BTA for a two-level language based on type inference and constraint solving [1]. His main contribution is to provide the first efficient implementation of the type inference approach to BTA. Henglein is the first to separate BTA into separate type inference and constraint solving phases. As in our case, the solution of these constraints gives annotations to place in the original program. His constraints are rather ad-hoc: their form is specifically crafted for the single purpose of constraining binding times. In contrast, our much simpler constraint system arises naturally from the realization that the stage of a program is precisely captured by its type in a typed, staged language. Abstracting the nesting of the code type constructor into a single integer-valued stage variable leads naturally to a type system based upon simple arithmetic constraints.

Heldman and Hughes [11] present a polychronic BTA algorithm that handles parametric polymorphism in the source language. Their approach is similar to ours in that it is based on type inference and constraint solving and they handle both polyvariance and polymorphism with the same construct. However, they only consider a two-level language and ignore partially static data. They do handle a **lift** annotation that we do not, and conversions between $\langle a \to b \rangle$ and $\langle a \rangle \to \langle b \rangle$ via a form of subtyping constraints. Also, their annotations represent stages rather than transitions between stages: an unwieldy syntactic convention for programmers.

The seminal work in multistage BTA is that of Glück and Jørgensen [12]. They generalize the constraint-based approach of Henglein to generate multistage generating extensions. Their constraints, like Henglein's, are ad-hoc. Davies [13] presents the staged language $\lambda^{\bigcirc}$ based on the $\bigcirc$ modality from linear temporal logic. He proves that $\lambda^{\bigcirc}$ is a refinement of the multistage language of Glück and Jørgensen [14].

10 Conclusions

We have developed a novel approach to BTA which is based on the well established method of type inference. We have made the following contributions:

1. The novelty of our algorithm is the expression of a program's well-stagedness in the form of simple arithmetic constraints.
2. Staged types were found to be extremely expressive as BTA specifications, much more so than anything from the partial evaluation literature. Staged types can express higher-order partially static data and arbitrarily many stages of execution.
3. Our deepest insight is that polyvariance reduces to polymorphism when stage ordering constraints are built into type schemes. This does not seem to have been addressed in the literature to date. Identifying an advanced BTA feature (polyvariance) with an advanced typing feature (polymorphism) reduces the number of concepts an algorithm must handle.

4. Our work enables a system to integrate manual staging with automatic BTA. A BTA algorithm based on our work here would enable adding the `stage` command described in Section 1.3 to MetaML.

References

1. Henglein, F.: Efficient type inference for higher-order binding-time analysis. In: Functional Programming Languages and Computer Architecture. Volume 523 of Lecture Notes in Computer Science., Springer-Verlag (1991) 448–472
2. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press (1997) 203–217
3. Sheard, T.: Accomplishments and research challenges in meta-programming. In: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG). Volume 2196 of Lecture Notes in Computer Science., Springer-Verlag (2001) 2–44
4. Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: ACM SIGPLAN Workshop on Haskell, ACM Press (2002) 1–16
5. Sheard, T., Linger, N.: 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), ACM Press (2002) 20–31
6. Jones, M.P.: Qualified Types: Theory and Practice. PhD thesis, Oxford University (1992) Also available as Programming Research Group technical report 106.
7. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems **5** (1999) 35–55
8. Pugh, W.: The omega test: A fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **38** (1992) 102–114
9. Nielson, H., Nielson, F.: Automatic binding time analysis for a typed λ-calculus. Science of Computer Programming **10** (1988) 139–176
10. Gomard, C., Jones, N.: A partial evaluator for the untyped lambda-calculus. Journal of Functional Programming **1** (1991) 21–69
11. Heldal, R., Hughes, J.: Binding-time analysis for polymorphic types. In: Andrei Ershov Fourth International Conference on Perspectives of System Informatics. Volume 2244 of Lecture Notes in Computer Science., Springer-Verlag (2001) 191–201
12. Glück, R., Jørgensen, J.: Fast binding-time analysis for multi-level specialization. In: Andrei Ershov Second International Conference on Perspectives of System Informatics. Volume 1181 of Lecture Notes in Computer Science., Springer-Verlag (1996) 261–272
13. Davies, R.: A temporal-logic approach to binding-time analysis. In: Proceedings of the 11th Annual Symposium on Logic in Computer Science. (1996) 184–195
14. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: Programming Languages, Implementations, Logics, and Programs. Volume 982 of Lecture Notes in Computer Science., Springer-Verlag (1995) 259–278