Search-Based Binding Time Analysis using Type-Directed Pruning

Tim Sheard
OGI School of Engineering
Oregon Health & Science University
20000 N.W. Walker Road
Beaverton, OR 97006-8921
sheard@cse.ogi.edu

Nathan Linger
OGI School of Engineering
Oregon Health & Science University
20000 N.W. Walker Road
Beaverton, OR 97006-8921
rlinger@cse.ogi.edu

Categories and Subject Descriptors

D.1. [**Software**]: Programming Techniques—*Automatic Programming*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms

Algorithms

Keywords

Staging, Binding Time Analysis, Type-Directed Search

ABSTRACT

We introduce a new way of performing binding time analysis. Rather than analyzing the program using constraint solving or abstract interpretation, we use a method based on search. The search is guided by type information which significantly prunes the size of the search space, making the algorithm practical. Our claim is not that we compute new, or better information, but that we compute it in a new and novel way, which clarifies the process involved.

The method is based upon a novel use of higher-order multi-stage types as a rich and expressive medium for the expression of binding-time specifications. Such types could be used as the starting point in any BTA. A goal of our work is to demonstrate that a single unified system which seamlessly integrates both manual staging and automatic BTA-based staging is possible.

1. INTRODUCTION

Binding Time Analysis (BTA) can be thought of as the automatic addition of staging annotations to a well-typed, semantically meaningful term in some base language. The programmer supplies two things 1) a well-typed base language program and 2) a binding time specification (a set of instructions as to which parts of the program are static and which parts are dynamic) and the analysis produces a new program which is the old program plus staging annotations. If successful, the new program is called well- annotated, and the erasure of the staging annotations from the new program produces the original base language program.

We introduce a new kind of BTA that works by searching the space of annotated terms that can be produced by adding one or more staging annotations to a well-typed base term. These added staging annotations must be consistent with both the original type of the program, and the user supplied binding-time specification. The search space is explored lazily by adding only those staging annotations that maintain that consistency. If a path is discovered that could no longer produce a well-annotated term, the path is immediately pruned.

The search can produce more than one well-annotated term. By directing the search, the algorithm can be adjusted to produce "better" well-annotated terms first.

2. MOTIVATION

This work was motivated by our work on the MetaML meta-programming system. In a meta- programming system, *meta-programs* manipulate *object-programs*. The meta-programs may construct object-programs, combine object-program fragments into larger object-programs, and observe the structure and other properties of object-programs.

MetaML was designed to be useful as a medium for the expression of run-time code generators, but it has found many other uses as well.

MetaML is a conservative extension of core ML. It includes all the features of Standard ML, except the module system. It adds four kinds of staging annotations for constructing, manipulating, and executing code as a first class object. We discuss three of them here.

The staging annotations partition the program into stages. Brackets (`< _ >`) surrounding an expression lift the surrounded expression to the next stage. Escape (`~( _ )`) (which should only appear within brackets) drops its surrounded expression to a previous stage. Lift (`lift _`) evaluates its argument to a ground value (a first order value like `5`) and constructs a program in the next stage with that constant value. In a two stage world, bracketed code is dy-

ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.

namic, and unbracketed code is static. There is no a-priori restriction to two stages.

The most common use of MetaML is the staging of programs with interpretive overhead to improve run-time performance. This is the same reason partial evaluators are often employed. It is useful to consider how this is accomplished in MetaML. In earlier work[22] we identified a 7 step process which we greatly abbreviate here.

The example we use is the traditional one – staging the power function. But, we have used the same process on many other programs, some orders of magnitude larger.

First write an unstaged program:

```
(* power : int -> int -> int *)
fun power n x =
  if n=0 then 1 else x * (power (n-1) x);
```

Second, identify the source of the interpretive overhead. In the `power` example, it is looping on the variable representing the exponent `n`.

Third, consider the type of the unstaged function. Here, it is (`int -> int -> int`). Consider an extension of this type that can be obtained by adding the code type constructor in one or more places that makes the source of the interpretive overhead static, and the other parameters dynamic. There may be several such types. For example, two different extensions of `power`'s type are `(int -> <int> -> <int>)` and `(int -> <int -> int>)`.

Fourth, choose one of these extension types, and place staging annotations (bracket, escape, lift) on the original program to produce a well-annotated program. For example a staged version of `power` with type `(int -> <int> -> <int>)` is the function `pow1`.

```
fun pow1 n x =
  if n=0 then <1>
         else < ~x * ~(pow1 (n-1) x) >;
```

The choice of which extended type to use to guide the annotation often depends on the context that the generator is to be used in. Sometimes it is obvious, other times not so. The subtleties are beyond the scope of this short introduction, and not relevant here, since even in a automatic BTA based system the user must supply the staging specification.

The staged version is then used to produce a computation without interpretive overhead. In the examples below `pow1` is used in several contexts to generate the code fragments to the right of the *evaluates-to* arrow (`--->`)

```
pow1 3 <2>     --->  <2 * 2 * 2 * 1>

<fn z => ~(pow1 4 <z>)>
               --->  <fn z => z * z * z * z * 1>
```

3. A UNIFIED VISION

In the winter of 2002, one of the authors taught a course on staged computation. A pattern emerged. A typical assignment consisted of an unstaged function with type t, and a *target* extended type t', and the instructions, "Write a staged version of the unstaged function with the target type t'."

A discussion arose in class about the possibility of automatically producing the staged versions of some functions which could be mixed with other manually staged functions in a single system. This caused the class, and the instructor, to consider the process they followed when they staged a function. The answer was obvious – they used the type information contained in both the type of the original program and the target type, to guide the placement of the staging annotations.

We have argued[20, 21, 23] that manual annotation gives the programmer finer control than any automatic process can ever achieve. But, we must admit, that many times an automatic annotation would both suffice, and place less of a burden on the programmer. Why couldn't a tightly integrated system which supported both manual annotation, as practiced in MetaML, and an automatic BTA be built?

In our view, the key obstacle to such a system was reconciling the binding time specifications of a given BTA with MetaML's rich type structure in which it is possible to mix code types, data structures, and higher order functions. Many BTAs are driven by binding time specifications which are simple directives indicating that certain global variables and/or parameters are simply static or dynamic. Newer ones have been extended to partially static first order data. In MetaML, code types are completely first class. In MetaML it is possible for data structures to embed functions which manipulate code. This kind of partially static higher order functions (i.e. meta-functions which take meta-functions as arguments) is missing from binding-time specifications. The key insight was to *use MetaML's rich types themselves as binding time specifications.*

Once this key insight was understood, then several smaller hurdles were easily resolved.

- Most BTAs are described as part of a partial evaluation system. Such systems are almost always based upon source to source transformations, and work at the source-file level. They produce new source-files which are compiled by existing compilers. This was hard to reconcile with MetaML's view that staging annotations are first class semantic objects and are part of the language definition. By viewing MetaML types as staging specifications, it is an easy next step to view them as directions to the compiler to produce a new program whose meaning is given in terms of MetaML's semanticly meaningful staging annotations.

- The output of BTA is opaque to most users. It consists of a generating extension which when applied to the static arguments produces the residual program. The annotations are there in the generating extension, but the partial evaluation paradigm does not encourage users to look at or understand the generating extension. Most users have no idea what they look like, or how they are used. But in MetaML users already know what staged programs look like, they write them themselves when they manually annotate programs, so the conceptual barrier is lower.

The idea of using higher-order partially static types (function arrows with code types) as the directive which guides an automatic BTA in an otherwise manually staged system, provides a fine level of control that was previously missing, yet still enables automatic staging when desired. We elaborate our vision of an integrated system encompassing both manual and automatic BTA as a simple extension to MetaML.

BTA produces an annotated term from a suggested staged type. To MetaML we add a new declaration form `stage`. When a user wishes to indicate that he desires an annotated version f', of a function f, at the annotated extension type t, he writes: `stage` f' `=` f `at` t. It is the compiler's job to produce such a function automatically or cause a compile-time error if it can't. Declaring the following:

```
stage pow1 = power at int -> <int> -> <int>;
```

causes the compiler to generate and compile the new function `pow1`.

```
fun pow1 n x =
  if n=0 then <1> else < ~x * ~(pow1 (n-1) x) >;
```

This is the same output that we produced by hand above. Similarly, the declaration:

```
stage pow2 =  power at int -> <int -> int>;
```

causes the compiler to generate and compile the function `pow2`.

```
fun pow2 n = <fn x =>
  ~(if n=0
       then <1> else < x *  ~(pow2 (n-1)) x >)>;
```

This scheme is quite flexible. It can be used to stage functions with partially static data, partially static higher order types, or to stage functions with more than two stages. For partially static data consider:

```
(* map : ('b  -> 'a ) -> 'b  list -> 'a  list *)
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs);

stage map2 = map at ('a-> 'b)-> 'a list-> <'b> list
```

leading to the automatic introduction of

```
fun map2 f [] = []
  | map2 f (x::xs) =  <f x> :: (map' f xs)
```

For higher order partially static types consider:

```
stage map3 = map at ('a-> <'b>)->'a list-> <'b list>
```

leading to the automatic introduction of

```
fun map3 f [] = <[]>
  | map3 f (x::xs) = < ~(f x) :: ~(map' f xs) >
```

For staging programs at more than two stages consider an inner product function staged to run in three stages[8, 14]. In the first stage, knowing the size of the two vectors offers an opportunity to specialize the inner product function on that size, removing a looping overhead from the body of the function. In the second stage, knowing the first vector offers an opportunity for specialization based on the values in the vector. If the inner product of that vector is to be taken many times with other vectors it can be specialized by removing the overhead of looking up the elements of the first vector each time. In the third stage, knowing the second vector, the computation is brought to completion.

```
(* iprod : int -> Vector -> Vector -> int *)
fun iprod n v w =
  if n  '>'  0
     then ((nth v n)*(nth w n)) + (iprod (n-1) v w)
     else 0;

stage p3 = iprod at
         int -> <Vector> -> <<Vector>> -> <<int>>
```

Here the operator `'>'` is the *greater-than* operator. MetaML uses this operator because the normal greater than operator conflicts with MetaML's use of the symbol `>` as a staging annotation. As before, the `stage` declaration would cause the compiler to automatically produce the three stage annotated version:

```
fun p3 n v w =
if n  '>'  0
   then << (~(lift (nth ~v n)) * (nth ~(~w) n)) +
           ~(~(p3 (n-1) v w)) >>
   else <<0>>;
```

4. FRAMEWORK

In this section we describe a minimal language we use to describe our BTA. The examples in the previous section are expressed in MetaML, a language considerably richer than the minimal language we will describe next. The properties we will show to hold for the minimal language will also hold for MetaML.

Base Terms. The structure of *base terms* is defined by an inductive set of productions. These productions define the set of syntactically correct terms. For example, a variant of the lambda calculus with integer constants could be defined by:

$$E \;::=\; x \;\mid\; E\;E \;\mid\; \lambda x.E \;\mid\; i \;\mid\; \text{if}\, E\, E\, E$$

Annotated Terms. Staging annotations are added to the set of productions to define the set of syntactically correct *annotated terms*. For example, we add the productions for bracket ($\langle\, E\,\rangle$), escape ($\tilde{}\, E$), and lift.

$$\begin{array}{lll} E & ::= & \cdots \\ & \mid & \langle\, E\,\rangle \;\mid\; \tilde{}\,E \;\mid\; \text{lift}\;\; E \end{array}$$

Erasure is the process of removing annotations from an annotated term to produce a base term.

Base Types. The set of *base types* for base terms is also inductively defined. The actual form of types depends upon the constructs and concepts inherent in the base language. For the lambda calculus variant, the types of base terms can be defined by introducing types for constants like I for `Int` and B for `Bool`, type constructors like *list*, and function types.

$$T \;::=\; I \;\mid\; B \;\mid\; T\;\mathit{list} \;\mid\; T \rightarrow T$$

Annotated Types. We also extend the set of base types by adding the code type constructor to produce the set of *annotated types*.

$$\begin{array}{lll} T & ::= & \cdots \\ & \mid & \langle\, T\,\rangle \end{array}$$

Note that brackets are overloaded to work both on annotated terms, and on annotated types.

Environments. We assume our language has a full complement of primitive functions that operate on the base types ($+$, $*$, $-$, etc.), and the built in data structures (*head*, *tail*,

$$\frac{}{\Gamma \vdash_n i : I} C \qquad \frac{\Gamma(x) = (t, m) \quad m \leq n}{\Gamma \vdash_n x : t} V$$

$$\frac{\Gamma, x : (t, n) \vdash_n e : t'}{\Gamma \vdash_n (\lambda x : t. e) : t \to t'} L$$

$$\frac{\Gamma \vdash_n e_1 : t \to t' \quad \Gamma \vdash_n e_2 : t}{\Gamma \vdash_n e_1\ e_2 : t'} A$$

$$\frac{\Gamma \vdash_n e_1 : B \quad \Gamma \vdash_n e_2 : t \quad \Gamma \vdash_n e_3 : t}{\Gamma \vdash_n \text{if}\, e_1\ e_2\ e_3 : t} I$$

$$\frac{\Gamma \vdash_{n+1} e : t}{\Gamma \vdash_n \langle e \rangle : \langle t \rangle} Br \qquad \frac{\Gamma \vdash_n e : \langle t \rangle}{\Gamma \vdash_{n+1} \sim e : t} Es$$

$$\frac{\Gamma \vdash_n e : c \quad c \in \{I, B\}}{\Gamma \vdash_n \text{lift}\, e : \langle c \rangle} Li$$

Figure 1: Judgments for well-typed base terms, and their extension for well-annotated terms.

cons, *nil*, *null*, etc.). Environments (Γ) map these *global* constants, and lambda-bound variables to their types.

Well-typed Terms. Type judgments select a subset of the syntactically well formed terms which are semantically meaningful. We call such terms *well-typed*. The top half of Figure 1 gives a set of judgments for base terms. The form of a judgment is $\Gamma \vdash e : (t, n)$ and can be read as *under the environment* Γ *the term* e *can be assigned the type* t *at stage* n. Here n is a natural number, and terms at level 0 are terms without any staging annotations. In general a term at level n is surrounded by n sets of matching brackets. For base terms the stage information can safely be ignored. Indeed, by erasing the stage information, the top half of Figure 1 reduces to the familiar type judgments of the lambda calculus.

Well-annotated Terms. Adding type judgments for the staging annotations to the judgments for base terms defines a new judgment that selects a subset of the syntactically correct annotated terms called the *well-annotated terms*. The bottom half of Figure 1 extends the top half with judgments for the annotations bracket, escape, and lift. The stage information (the n in the judgment) counts the number of brackets surrounding the current term. This count ensures that all escapes appear within brackets, and that variables are only used in stages later than their binding stage.

Annotated Extensions. There are two inputs to the BTA process: a well-typed base term e with base type t_1, and a target annotated type t_2. The type t_2 should be an *annotated extension* of the type t_1, and is the type of the annotated term we wish the BTA to find.

The Relation $\sqsubseteq$. Base types and their annotated extensions are related by the relation $\sqsubseteq$. The relation $\sqsubseteq \in \mathcal{P}(\mathit{basetype} \times \mathit{annotatedtype})$ intuitively means *Can be obtained by removing staging annotations from.* It can be made precise by the following inductive rules, where we use a notation in which $b_i \in \mathit{basetype}$, and $a_i \in \mathit{annotatedtype}$ to remind the reader that the two arguments to the relation come from different sets.

$$\frac{t \in \{I, B\}}{t \sqsubseteq t} \qquad \frac{b \sqsubseteq a}{b \sqsubseteq \langle a \rangle} \qquad \frac{b \sqsubseteq a}{b\ \mathit{list} \sqsubseteq a\ \mathit{list}} \qquad \frac{b_1 \sqsubseteq a_1 \quad b_2 \sqsubseteq a_2}{b_1 \to b_2 \sqsubseteq a_1 \to a_2}$$

The relation $\sqsubseteq$ simply formalizes the notion of erasure on types. All it does is describe in a precise manner when one type is an erasure of another. If $t_1 = \mathit{erase}\ t_2$ then $t_1 \sqsubseteq t_2$. Note that *erase* acts homomorphically on all type constructors except the code (bracket) type constructor. Adding annotations to a program e with type t_1 cannot produce another program of arbitrary type t_2. The types t_1 and t_2 must be related in the fashion made precise by $\sqsubseteq$.

Given two partial functions Γ_1 and Γ_2, representing environments, where Γ_1 maps term variables to base-types, and Γ_2 maps term variables to annotated-types, both with the same domain, we lift the relation $\sqsubseteq$ pointwise to environments. $\Gamma_1 \sqsubseteq \Gamma_2 \equiv \forall x \in \mathit{Dom}(\Gamma_1, \Gamma_2). \Gamma_1\, x \sqsubseteq \Gamma_2\, x$.

Overloading $\sqsubseteq$ on Terms. We overload the relation $\sqsubseteq \in \mathcal{P}(\mathit{terms} \times \mathit{annotatedterms})$ on terms as well as types. The two meanings are so similar that this shouldn't be a problem conceptually.

$$\frac{}{i \sqsubseteq i} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{b_1 \sqsubseteq a_1 \quad b_2 \sqsubseteq a_2}{b_1\ b_2 \sqsubseteq a_1\ a_2}$$

$$\frac{b_1 \sqsubseteq a_2 \quad t_1 \sqsubseteq t_2}{\lambda x : t_1. b_1 \sqsubseteq \lambda x : t_2. a_2} \qquad \frac{b_1 \sqsubseteq a_1 \quad b_2 \sqsubseteq a_2 \quad b_3 \sqsubseteq a_3}{\text{if}\ b_1\ b_2\ b_3 \sqsubseteq \text{if}\ a_1\ a_2\ a_3}$$

$$\frac{b \sqsubseteq a}{b \sqsubseteq \langle a \rangle} \qquad \frac{b \sqsubseteq a}{b \sqsubseteq \text{lift}\ \ a} \qquad \frac{b \sqsubseteq a}{b \sqsubseteq \sim a}$$

Again we use $b_i \in \mathit{baseterm}$ and $a_i \in \mathit{annotatedterm}$. The lifted relation simply formalizes the notion of erasure on terms. If e_1 is an erasure of e_2 then $e_1 \sqsubseteq e_2$.

5. A STAGING CHECKING SYSTEM

Given a term e, its type t, and a target type extension t', we wish to find an annotated term e' such that $t \sqsubseteq t'$ and $e : t$ and $e' : t'$ and $e \sqsubseteq e'$. We need some adjustments to our notation to capture this precisely.

First, the extension of $\sqsubseteq$ to terms may seem too syntactically rigid. For example, terms which are α equivalent are not related if the bound variable is not the same. This should not be a problem since we intend to generate the annotated terms on the right-hand-side of the relation, and we can always use the same name for the bound variable.

Second, the ordering on terms is not quite satisfactory. It is possible to relate annotated terms that are not well-annotated to well-typed base terms. Since we only care about relating well-typed terms to well-annotated terms, we'll define a new relation which captures this distinction. Overloading $\sqsubseteq$ once again we combine the typing judgments, and the type and term relations into one:

$$\begin{array}{c} \Gamma_2 \vdash_n (e_1 : t_1) \sqsubseteq (e_2 : t_2) \\ \equiv \\ \exists \Gamma_1 . (\Gamma_1 \vdash_0 e_1 : t_1) \land (\Gamma_2 \vdash_n e_2 : t_2) \land \\ (e_1 \sqsubseteq e_2) \land (t_1 \sqsubseteq t_2) \land (\Gamma_1 \sqsubseteq \Gamma_2) \end{array}$$

Given e_1, t_1, e_2, and t_2 infer if e_2 could both be well-annotated at type t_2, and be related to e_1 by the term overloading of $\sqsubseteq$. We formalize this by writing down a new set of judgments which appear in Figure 2.

$$\frac{}{\Gamma \vdash_n (i : I) \sqsubseteq (i : I)}\text{Int} \qquad \frac{\Gamma(x) = (t_2, m) \qquad m \le n}{\Gamma \vdash_n (x : t_1) \sqsubseteq (x : t_2)}\text{Var}$$

$$\frac{\Gamma, x \mapsto (t_2, n) \vdash_n (e_1 : s_1) \sqsubseteq (e_2 : s_2) \qquad t_1 \sqsubseteq t_2}{\Gamma \vdash_n ((\lambda x : t_1.e_1) : t_1 \to s_1) \sqsubseteq ((\lambda x : t_2.e_2) : t_2 \to s_2)}\text{Lam}$$

$$\frac{\begin{array}{c} s_1 \sqsubseteq s_2 \\ \Gamma \vdash_n (e_1 : s_1 \to t_1) \sqsubseteq (e_2 : s_2 \to t_2) \\ \Gamma \vdash_n (e_1' : s_1) \sqsubseteq (e_2' : s_2) \end{array}}{\Gamma \vdash_n (e_1\ e_1' : t_1) \sqsubseteq (e_2\ e_2' : t_2)}\text{App}$$

$$\frac{\begin{array}{c} \Gamma \vdash_n (e_1 : B) \sqsubseteq (e_4 : B) \\ \Gamma \vdash_n (e_2 : t_1) \sqsubseteq (e_5 : t_2) \\ \Gamma \vdash_n (e_3 : t_1) \sqsubseteq (e_6 : t_2) \end{array}}{\Gamma \vdash_n (\text{if } e_1\ ; e_2\ e_3 : t_1) \sqsubseteq (\text{if } e_4\ e_5\ e_6 : t_2)}\text{If}$$

$$\frac{\Gamma \vdash_{n+1} (e_1 : t_1) \sqsubseteq (e_2 : t_2)}{\Gamma \vdash_n (e_1 : t_1) \sqsubseteq (\langle e_2 \rangle : \langle t_2 \rangle)}\text{Code}$$

$$\frac{\Gamma \vdash_n (e_1 : t_1) \sqsubseteq (e_2 : \langle t_2 \rangle)}{\Gamma \vdash_{n+1} (e_1 : t_1) \sqsubseteq ({\sim}\, e_2 : t_2)}\text{Escape}$$

$$\frac{\Gamma \vdash_n (e_1 : c) \sqsubseteq (e_2 : c) \qquad c \in \{I, B\}}{\Gamma \vdash_n (e_1 : c) \sqsubseteq (\text{lift } e_2 : \langle c \rangle)}\text{Lift}$$

Figure 2: Relating well-typed terms to well-annotated terms.

The judgments are derived in a straightforward manner from those in Figure 1, and the relation $\sqsubseteq$ on types and terms.

6. FROM CHECKING TO INFERENCE

Can we move from a type checking system that checks if two terms are related by staging annotation, to an algorithm that computes the staged program from the unstaged program? The judgments in Figure 2 describe several rules for checking relationships between e_1, e_2, t_1, and t_2, when all four are known. When e_2 is unknown, we can use the rules to guide a search. A slight restructuring of the rules helps illustrate this. Let the notation $e_2 \leftarrow \mathcal{A}_n^\sigma\ {e_1}^{t_1}\ t_2\ s$ denote a search for a well- annotated term e_2 whose erasure is e_1. Let t_1 be the type of e_1 and t_2 be the target type of the sought-after term e_2., and s be the current state. We'll discuss states in a moment. Most of the rules proceed by searching for annotations for the subterms of e_1, which will be combined to form e_2. Occasionally we search for the annotation of a subterm, fully intending to wrap a new set of brackets around this result when we get it. We call the number of "pending" brackets the level of the search, and it is part of the state of the search. Let n be the level of the search, then a checking rule like **If**

$$\frac{\begin{array}{c} \Gamma \vdash_n (e_1 : B) \sqsubseteq (e_4 : B) \\ \Gamma \vdash_n (e_2 : t_1) \sqsubseteq (e_5 : t_2) \\ \Gamma \vdash_n (e_3 : t_1) \sqsubseteq (e_6 : t_2) \end{array}}{\Gamma \vdash_n (\text{if } e_1\ e_2\ e_3 : t_1) \sqsubseteq (\text{if } e_4\ e_5\ e_6 : t_2)}\text{If}$$

leads naturally to a search rule like:

$$\frac{\begin{array}{c} e_4 \leftarrow \mathcal{A}_n^\sigma\ {e_1}^B\ B\ \mathit{clear} \\ e_5 \leftarrow \mathcal{A}_n^\sigma\ {e_2}^{t_1}\ t_2\ \mathit{clear} \\ e_6 \leftarrow \mathcal{A}_n^\sigma\ {e_3}^{t_1}\ t_2\ \mathit{clear} \end{array}}{(\text{if } e_4\ e_5\ e_6) \leftarrow \mathcal{A}_n^\sigma\ (\text{if } e_1\ e_2\ e_3)^{t_1}\ t_2\ s}$$

Note how the environment Γ in the checking rule changes to an environment σ in the search rule. In the checking rules we map term variables to types, but in a search algorithm we map term variables to annotated terms. In the inference algorithm where we compute an annotated term for every unannotated counterpart term, σ maps unannotated term variables and to well-annotated terms.

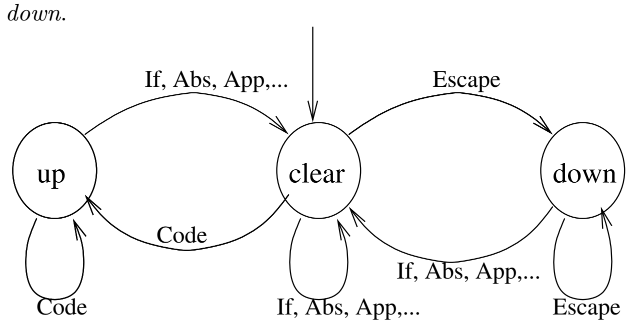
What the checking rules don't tell us is in what order to apply the search rules, or what to do if a rule fails. At some point we need to make a choice about what implementation mechanism we will use to implement our search. Prolog comes immediately to mind, and would probably have made a good choice, but further investigation suggested several annoying details, and we choose the functional language Haskell because of the ease with which we could modify our program even after drastic changes to what we considered a search. This will become more clear in what follows.

Except for the rules **Code**, **Escape** and **Lift**, the rules are syntax directed over the structure of e_1. These rules are driven by the syntactic structure of t_2 and n. We can use this structure to decide what rules are applicable. Unfortunately, at any point, more than one rule is usually applicable. We must also be careful because the rules *Code* and *Esc* are circular, and could lead to derivations of infinite height, and hence search of infinite depth.

$$\frac{\frac{\frac{\cdots}{\Gamma \vdash_{n+1} (e : t_1) \sqsubseteq (?_2 : t_2)}}{\Gamma \vdash_n (e : t_1) \sqsubseteq (?_1 : \langle t_2 \rangle)}}{\Gamma \vdash_{n+1} (e : t_1) \sqsubseteq (?_0 : t_2)}$$

An algorithm will control each of these parameters in some way. The key to the algorithm is controlling two important aspects: the search is partial – it may fail. And, the search is non-deterministic – there may be more than one annotated term with the given type.

An effective way to controlling the search is to attempt the **Escape** rule first, followed by the syntax directed rules over the structure of e_1 next, and to apply the **Lift** and **Code** rules only if all the other rules fail. Why this is an effective strategy is discussed in Section 13. To control the circularity of **Code** and **Escape** we embed the algorithm in a small state machine with three states: *clear*, *up*, and *down*.



The algorithm starts in state *Clear*. Any use of rule *Code* moves to state *Up*, where *Escape* is not allowed, and any use of *Escape* moves to state *Down* where *Code* is not allowed. Applying any rule which recurses on a sub- term of the current term, moves the machine back to state *clear*.

For example, $i \leftarrow \mathcal{A}^{\sigma}_{n}\ i^{I}\ I\ s$ expresses that if we have an integer constant i, and we're searching for a term of type I, in any state s, then the same term i will suffice.

For terms with sub-structure we need to search for annotations, of a particular type, of some of the given term's sub-terms. For example:

$$\cfrac{\cfrac{x' \leftarrow \mathcal{A}^{\sigma}_{(n+1)}\ x^{s}\ w\ clear \qquad f' \leftarrow \mathcal{A}^{\sigma}_{(n+1)}\ f^{s\rightarrow t}\ (w \rightarrow z)\ clear}{f'\ x' \leftarrow \mathcal{A}^{\sigma}_{n+1}\ (f^{s\rightarrow t}\ x^{s})^{t}\ z\ up}\ \text{App}}{\langle f'\ x' \rangle \leftarrow \mathcal{A}^{\sigma}_{n}\ (f^{s\rightarrow t}\ x^{s})^{t}\ \langle z \rangle\ clear}\ \text{Code}$$

If we can find an annotated version f' of f with the given type, and an annotated version x' of x, then we can find an annotated version of the application $f\ x$.

If a search on a sub-term fails, then the search for the whole term fails. But, if a search on a sub-term produces more than one result, then the search on the whole term may produce more than one result. If a term has two subterms A and B, and the search on A produces n results, and the search on B produces m results, the search on the whole term will produce $n \times m$ results.

Fortunately there is a well-studied formalism for describing such algorithms. This is the notion of monadic computation. Several papers give good overviews of monadic computation [24, 25, 26], and we assume some familiarity with monadic programs.

In our case the monad is the *non-determinism* monad (some times called the *list* monad because of the data structure on which it is based, or the monad of *multiple-results*).

Consider the search rule specified above. Which of the sub-searches do we perform first? How do we specify what to do if one of them fails? If they both succeed, how do we specify the combination of the two sets of results? This is the job of the monad. We express the search implicit in the rule above explicitly in an equation that uses the monadic `do` notation.

$$\begin{array}{ll} \mathcal{A}^{\sigma}_{n}\ (f^{s\rightarrow t}\ x^{s})^{t}\ z\ state & = \\ do\quad x' \leftarrow \mathcal{A}^{\sigma}_{(n+1)}\ x^{s}\ w\ clear & \\ \quad\quad\ f' \leftarrow \mathcal{A}^{\sigma}_{(n+1)}\ f^{s\rightarrow t}\ (w \rightarrow z)\ clear & \\ \quad\quad\ return\quad (f'\ x') & \end{array}$$

Perform the search for an annotation of the subterm x first, if that succeeds search for an annotation of f next, if that succeeds combine all the results from both searches pair wise, building the newly annotated term $f'\ x'$.

Using the checking rules from Figure 2 we build search rules. We use the *do* notation to control the search elements. Each rule leads to a small search component. A complete search is constructed using the *do* notation to control, sequence, and combine the component searches. The complete algorithm can be found in Appendix A written as a working Haskell program. The program uses the following definitions for types and terms.

```
data T = I          -- int
       | B          -- bool
       | Arr T T    -- T -> T
       | Code T     -- < T >
       | L T        -- T list

data E = EA E E           -- f x
       | EL String T E   -- (fn x => E)
       | EV String       -- x
       | EI Int          -- 5
       | Eif E E E       -- if x then y else z
       | EB E            -- < E >
       | ES E            -- ~ E
       | ELift E         -- lift E
```

For each checking rule in Figure 2, we derive one or more search components written as Haskell functions. Each component has type

```
type M x = [ x ]  -- the Monad of Multiple Results
type Env1 = String -> T
type Env2 = String -> T -> Int -> M E
type Component =
     State -> Int -> Env1 -> Env2 -> (T,T,E) -> M E
```

Most rules lead to a single component, but consider the rule **App**. Its goal is

$$\Gamma \vdash_n (e_1\ e_1' : t_1) \sqsubseteq (e_2\ e_2' : t_2)$$

When searching $(e_1\ e_1')$, t_1, and t_2 are known, and $(e_2\ e_2')$ are not. We know $(e_1\ e_1')$ is well-typed at type t_1, thus it is possible to compute the type of e_1, which is $s_1 \rightarrow t_1$. It is not possible to compute the domain of e_2 which is labeled s_2 in the rule. All we know is that $s_1 \sqsubseteq s_2$. So a search rule based on this checking rule will have to choose some s_2 that is correctly related to s_1. Some simple choices are $s_2 = s_1$ and $s_2 = \langle s_1 \rangle$. This leads to two different search rules in our program. Additional rules are possible, and a generic treatment is discussed in Section 14.

To give a taste of how a component is constructed, we discuss these two rules. Two ways to stage an expression $(f\ x : t)$ would be to stage the whole term as if f had type $s \rightarrow t$, or to assume f has type $\langle s \rangle \rightarrow t$. The first way is captured by the following component.

```
appCase1 :: Component
appCase1 _ n sig phi (t1,t2,e @ (EA e0 e1)) =
   trace "App1" n e t2 $
   do let s1 = dom sig e0 -- Compute domain type of e0
      e2 <- a1 Clear n sig phi (Arr s1 t1) (Arr s1 t2) e0
      e3 <- a1 Clear n sig phi s1 s1 e1
      return (EA e2 e3)
appCase1 _ _ _ _ _ = mzero
```

Note how the state of the search on the sub-terms is reset to clear, and how the component follows the structure of the typing judgment and the relation $\sqsubseteq$. This is one of the syntax directed rules, and if applied to an non-application, the default clause (`appCase1 _ _ _ _ _ = mzero`) causes the component to fail.

In the second component, we account for the code property of the argument.

```
appCase2 _ n sig phi (t1,t2,e @ (EA e0 e1)) =
   trace "App2" n e t2 $
   do let s1 = dom sig e0 -- Compute domain type of e0
      e2 <- a1 Clear n sig phi (Arr s1 t1)
                               (Arr (Code s1) t2) e0
      e3 <- a1 Clear n sig phi s1 (Code s1) e1
      return (EA e2 e3)
appCase2 _ _ _ _ _ = mzero
```

The main algorithm is composed of a search strategy applied to the individual search components. In the next section we comment on the control mechanisms used to direct the strategy of the main algorithm.

7. CONTROL OF THE SEARCH

The *do* notation is used to control the order of the search, but its action on failure is to propagate failure once it arises. This means a single failure, anywhere, causes the whole algorithm to fail. What is needed is a mechanism to set up several searches and to combine the successful results into one large set of results. The monad of multiple results (M), supports several operations that facilitate this.

```
leftChoice :: M a -> M a -> M a
leftChoice [] ys = ys
leftChoice xs ys = xs

first :: [M a] -> M a
first [] = fail "first done\n"
first [x] = x
first (x:xs) = leftChoice x (first xs)

many :: [M a] -> M a
many [] = fail "many done\n"
many [c] = c
many (x:xs) = mplus x (many xs)
```

The operation `leftChoice` takes two computations producing multiple results. If the first succeeds, it returns the result and ignores the second. If the first fails, it runs and then returns the results of the second computation.

The operation `first` iterates `leftChoice` over a list of computations. It returns the results of the first successful computation in the list.

The operation `many` runs all the computations in the list, and returns the concatenation of all the results. It is the operation used to specify a branching search.

8. THE MAIN ALGORITHM

The main algorithm is called `a1` and regroups the syntax directed arguments into a tuple, and then calls `a2`. The algorithm `a2` is defined as large search, whose search strategy is constructed using `first` and `many` on the component searches. This strategy is only one possible strategy. Other strategies are possible. We discuss strategies in Section 13

The function `a1` takes as input the level (`n`), environments mapping term variables to types (`sig`), and term variables to annotated terms (`phi`), two types (`t1` and `t2`), and a term (`e1`). It produces multiple results, hence its return type (`M E`). It is meant to correspond roughly to the notation $e_2 \leftarrow \mathcal{A}^{\sigma}_{n}\, {e_1}^{t_1}\, t_2$.

```
a1 :: State -> Int -> Env1 -> Env2 -> T -> T -> E -> M E
a1 step n sig phi t1 t2 e1 = a2 step n sig phi (t1,t2,e1)
a2 step n sig phi x =
 first
 [ escCase  step n sig phi x -- note Esc case first
 , intCase  step n sig phi x
 , varCase  step n sig phi x
 , absCase  step n sig phi x
 , appCase1 step n sig phi x
 , appCase2 step n sig phi x
 , appCase3 step n sig phi x
 , ifCase   step n sig phi x
 , liftCase step n sig phi x
 , codeCase step n sig phi x
 ]
```

9. EXAMPLE TRACE

In this section we show a trace of the search to stage the term $\lambda f.\lambda x.f\ x$ at the type $\langle I \rightarrow I\rangle \rightarrow I \rightarrow \langle I\rangle$

```
Abs {0} fn f => fn x => f x : <int -> int> -> int -> <int>
  Abs {0} fn x => f x : int -> <int>
    App1 {0} f x : <int>
      Var {0} f : int -> <int>
      Var failed
    App1 failed
    App2 {0} f x : <int>
      Var {0} f : <int> -> <int>
      Var failed
    App2 failed
    Lift {0} f x : <int>
      App1 {0} f x : int
        Var {0} f : int -> int
        Var failed
      App1 failed
      App2 {0} f x : int
        Var {0} f : <int> -> int
        Var failed
      App2 failed
    Lift failed
    Code {0} f x : <int>
      App1 {1} f x : int
        Esc {1} f : int -> int
          Var {0} f : <int -> int>
          Var succeeded f
        Esc succeeded ~f
        Esc {1} x : int
          Var {0} x : <int>
          Var failed
          Lift {0} x : <int>
            Var {0} x : int
            Var succeeded x
          Lift succeeded lift x
        Esc succeeded ~(lift x)
      App1 succeeded ~f ~(lift x)
    Code succeeded <~f ~(lift x)>
  Abs succeeded fn x => <~f ~(lift x)>
Abs succeeded fn f => fn x => <~f ~(lift x)>
```

10. CORRECT BY CONSTRUCTION

We believe that the soundness of the search algorithm with respect to the checking rules can be proved, although we have not yet done so. Every node of the search space considered by the search algorithm is generated from the well-typed source expression by a checking rule. So the search program never considers an invalid term. The algorithm may return multiple results, and some of these results may be "better" than others, but they will all be valid extensions of the base term, at the type given. In Section 13 we discuss the use of strategies to order the returned solutions, best ones first.

If the algorithm uses the strategy employed in Appendix A, it is easy to argue that the algorithm always terminates. A measure function[4] can easily be constructed on the arguments `n`, `e1`, `t2`, and `state` that decreases by a well-founded relation on every recursive call of `a1`. While `n` may increase in the **Code** rule, it is always accompanied by a decrease in the size of `t2`. When `t2` has all its brackets stripped off, there must be a decrease in the size of `e1` by one of the syntax directed rules, or a failure. The three state automata encoded in `state` enforces this. Since the measure function decreases on every recursive call, and cannot fall below zero, the algorithm must terminate.

11. POLYVARIANCE

Polyvariance allows a single function to be used at multiple binding types. The algorithm needs no changes to support polyvariance. Recall that one of the parameters to the search algorithm is an environment mapping term variables and types to annotated terms. By allowing the environment to map the same term variable at different types to different annotated terms, polyvariance is achieved. For example consider the function f in an environment where the function h has type int -> int -> int.

```
fun f x y = h x y + h y x;
stage f' = f at int -> <int> -> <int>
```

producing

```
fun f' x y = <h x ~y + h ~y x>;
```

If the BTA environment had several stagings of h we could do better.

```
stage h1 = h at int -> <int> -> <int>
stage h2 = h at <int> -> int -> <int>
fun f x y = h x y + h y x;
stage f' = f at int -> <int> -> <int>
```

With these stagings of h we could do more work statically by using h1 and h2 in the annotated version of f produced automatically by the compiler.

```
fun f' x y = <~(h1 x y) + ~(h2 y x)>;
```

Although the definition of f mentions only h, the automatic BTA can make use of all declared stagings of h, and generates a staging of f with polyvariant uses of h.

A tight integration of both automatic and manual staging can be used to the mutual advantage of both. Consider manually staged versions of (op *), the primitive multiplication operator in MetaML. These manually staged versions exploit the arithmetic identities $x * 0 = 0$ and $x * 1 = x$, and are beyond the scope of any automatic BTA without using semantic information. Yet a programmer can easily stage them manually, injecting this semantic information into the system.

```
(* times1 : (int * <int>) -> <int> *)
fun times1 (0,x) = <0>
  | times1 (1,x) = x
  | times1 (y,x) = < y * ~x >;

(* times2 : (<int> * int) -> <int> *)
fun times2 (y,0) = <0>
  | times2 (y,1) = y
  | times2 (y,x) = < ~y * x >;
```

To inform the automatic BTA of times we use a variant of the stage declaration.

```
stage times1 = ( op * );
stage times2 = ( op * );
```

without the at type suffix, the stage declaration checks that times1 and times2 are manually staged versions of (op *) with consistent staged type (as is made precise by the relation $\sqsubseteq$), and adds them both to the environment of the BTA process.

Thus manually staged versions of functions, which use semantic information to force computation to earlier stages, can be used polyvariantly.

12. POLYMORPHISM

Our techniques easily extend to a language with Hindley-Milner (rank-1) polymorphism. In Hindley-Milner polymorphism all universally quantified types appear at the outer most level. This allows a simple preprocessing step to extend the BTA to a language with Hindley-Milner polymorphism. Consider the example where the programmer wants to stage the standard function

```
map :: ('a -> 'b) -> ['a] -> ['b]
```

at the type <I -> 'c> -> [I] -> <['c]>. Consider each of these types to be universally quantified (at the outer-most level) over the free type variables (those with tick marks, like 'a). The staged type is more annotated, but less general, than the type of map. In order to handle a staging declaration like

```
stage mapI = map at <int -> 'c> -> [int] -> <['c]>;
```

we first unify the **erasure** of the target type (<I -> 'c> -> [I] -> <['c]>) with the type of map (('a -> 'b) -> ['a] -> ['b]) to obtain a substitution σ ($'\mathtt{a} \mapsto \mathtt{I}, '\mathtt{b} \mapsto '\mathtt{d}, '\mathtt{c} \mapsto '\mathtt{d}$) which we then apply to both the source type and (unerased) target type before proceeding. The algorithm must treat type variables as unknown types, and hence only rules where the actual structure of the type is immaterial may apply. The BTA will then do the right thing.

13. STRATEGIES

The strategy used to order the individual components matters. There are two different properties of the output that we are trying to achieve simultaneously: *minimality* and *optimality.* We are working on precise definitions of these properties, but do not have them completely worked out yet. Informally, minimality means that the answer produced has the minimum number of staging annotations. For example both <f x> and <f ~<x>> are extensions of (f x) where f has type a -> b and x has type a. But the first is preferred because it has fewer staging annotations than the second. We wish to arrange our search strategy so that minimal annotations are found before larger ones, or better yet, so that the search spaces containing non-minimal ones are pruned completely.

The three state automata that guides the use of the **Code** and **Escape** rules, not only prevents infinite derivations, but also prunes non-minimal ones. The annotation ~<x> is pruned since the **Code** rule cannot be applied immediately after the **Escape** rule without first descending into a subterm.

Informally, optimality means that all computation is performed at the earliest stage possible. Static ifs, such as (if x then <y> else <z>), are preferred over dynamic ones such as: <if x then y else z>, because the first allows the test of the if to be performed in an earlier stage.

Currently, our search strategy uses a heuristic to push optimal solutions to the front of the result list: *Try the* **Esc** *rule first.* The component escCase always fails at level 0, but at levels greater than zero it searches for solutions at lower levels, i.e. earlier stages. By placing it first in a first or many strategy, we order stages with the earliest possible computations first. This explains why the component **Escape** is first in function a2. An analysis or proof that this always leads to optimal results remains as future work.

14. COMPLEXITY

Its hard to estimate the complexity of the algorithm, without knowing quite a bit about the search strategy used. The strategy used in Appendix A is extremely simple. It consists of a single `first` control operator. This will cause a search whose maximum depth is proportional to the depth of the term e plus the number of brackets in the target type. If the environment maps each term variable to a single annotated term at each type, then the algorithm will always find exactly 0 or 1 results. The breadth of the search is not so easy to estimate, and depends upon both the term being annotated, and the type at which the annotation is sought, and is the source of most of the algorithms complexity.

We have identified two places where clever implementation techniques can overcome some of this complexity. First, the naive algorithm performs redundant computations. These redundant computations are possible because the search space of annotated programs is a directed acyclic graph. There are often two (or more) paths to the same subproblem. Here is an example of algorithm trace showing this behavior.

```
...
staging (if large_calc then e1 else e2) :: <Bool>
If rule fired
  staging (large_calc) :: Bool
  ... large calculation ...
If rule failed
Lift rule fired
  staging (if large_calc then e1 else e2) :: Bool
  If  rule fired
    staging (large_calc) :: Bool
    ... same large calculation ...
  If   failed
Lift failed
...
```

This is solved by the standard technique of dynamic programming. We memoize away results in a table as we compute them, and then do a table lookup when attempting sub-searches to make sure we're not recalculating anything.

The second problem is caused by multiple `App` rules (mentioned in section 6). The type checking rule **App** (Figure 2) leads to many possible search rules.

$$\frac{\begin{array}{c} s_1 \sqsubseteq s_2 \\ \Gamma \vdash_n (e_1 : s_1 \to t_1) \sqsubseteq (e_2 : s_2 \to t_2) \\ \Gamma \vdash_n (e_1' : s_1) \sqsubseteq (e_2' : s_2) \end{array}}{\Gamma \vdash_n (e_1\ e_1' : t_1) \sqsubseteq (e_2\ e_2' : t_2)}\text{App}$$

One search rule for every possible solution to the side condition $s_1 \sqsubseteq s_2$,

The more deeply applications are nested, and the more deeply nested code types appear in the target type, the more this branching blows up. By observing the structure of the two search rules `appCase1` and `appCase2`, we see that they have much in common. Clever programming can merge all these rules into a single case, thus drastically reducing the branching, and hence the size of the search space. The trick is to perform sub-searches with s_2 instantiated to a type variable , and to maintain a list of constraints on all type variables. Thereafter, whenever the algorithm calls for an equality check on two types , the algorithm employs unification on types. A failure to unify becomes a failure in the search. The constraints are initially of the form $t \sqsubseteq \alpha$ where α is a type variable and t is a base type. These inequality constraints can either be strengthened to stronger inequalities or collapsed to equalities upon unification.

The following single `App` rule now encompasses the previous two and many others. The unification, new variable generation, and constraint maintenance, is all handled in the underlying monadic machinery. Thus the structure of the algorithm changes only slightly.

```
appCase _ n sig phi (t1, t2, e@(EA e0 e1)) =
  trace "App" n e t2 $
  do let s1 = typeof e1
     s2 <- freshWC s1
     e2 <- a1 Clear n sig phi (Arr s1 t1)
                              (Arr s2 t2) e0
     e3 <- a1 Clear n sig phi s1 s2 e1
     return (EA e2 e3)
appCase _ _ _ _ _ = mzero
```

$s_2 \leftarrow freshWC\ s_1$ generates a new type variable s_2 and adds the constraint $s_1 \sqsubseteq s_2$

Both of these changes, dynamic programming, and use of unification, have been implemented by reworking the underlying monad upon which the search is implemented. We have observed that the traversed search space is notably smaller. We currently working on quantifying a precise complexity bounds on an algorithm which employs these two techniques.

We believe the importance of this work lies in that it describes a *new and very simple framework* to describe BTAs with such advanced features as Higher-order functions, Polymorphism, Polyvariance, Partially static first-order data, Higher-order partially static functions, and an Unbounded number of stages. We believe the potential for tuning the algorithm to a highly efficient one remains.

15. RELATED WORK

Mogensen [16], Bondorf [2, 3] and Consel [5] present BTAs for higher order languages which are based upon abstract interpretation.

Three papers on BTAs for typed lambda calculi are closely related to the work presented here because they express BTA as a type inference problem. Nielson and Nielson [18] give a BTA based upon type inference for the two-level simply- typed lambda-calculus. Given simple binding times (compile-time or run-time) for all free variables in a typed expression, they show that their algorithm computes an unique expression with an optimal set of annotations that minimizes run-time computation. The complexity of their algorithm is worst case exponential on the size of the expression.

Gomard [11] presents an $O(n^3)$ algorithm for annotating an untyped lambda-calculus term in a similar manner. His work is interesting because he uses a crude type system to perform BTA for an un-typed language. The type system treats all second stage terms as having a single type "code" and all other terms (except functions) as having type "value", and uses arrows over these simple types to specify binding-times for functions.

Henglein [12] presents an efficient algorithm which uses a similar trick for expressing binding-time in a similarly simple type system. His algorithm has complexity $O(n\,\alpha(n,n))$, where α is an inverse of the Ackerman's function, and $\alpha(n,n)$ is for all intents and purposes a small constant. His algorithm uses a constraint solving system to determine where annotations should be placed.

While our algorithm may look like a type inference problem, it is really a search based algorithm. It is easy to identify both the search space, and the search strategy employed. The possibility of multiple solutions also separates it from type inference.

The use of types as BTA specifications can be found in the work of Le Meur, Lawall, and Consel[15]. They describe a module based system for writing binding time specifications for programs in C. The system allows the programmer to name multiple binding time specifications for each function and global variable in a module, and to use these named specifications in other specifications. Their specifications are written as stage-annotated types. The module system can then propagate this information to multiple use-sites of the annotated functions allowing different specializations at different occurrences. Unlike our use of annotated types as binding time specifications, theirs is limited to first order functions.

Many BTAs are based upon abstract analysis. BTAs for partially static data have been presented by Launchbury[13] and Mogensen[17], and polyvariant BTAs have been presented by Consel [6, 1], Dussart et. al. [7], Rytz & Gengler [19], amongst others.

Glueck and Joregensen [8, 9] pioneered the use of BTA for multi-level languages. Their work generalizes a standard abstract interpretation technique to multiple levels. We show that search based techniques can also generalize to multiple levels.

The techniques described here incorporate all these features in a simple framework based upon search.

16. CONTRIBUTIONS

In this paper we have described a radically new approach to the construction of binding time analyses. The approach is based upon exploring the search space of well-annotated extensions of well-typed terms. Type information is an effective means to prune the search space, and makes the algorithm practical. The algorithm is surprisingly simple yet supports such advanced features as Higher-order functions, Polymorphism, Polyvariance, Partially static first-order data, Higher-order partially static data, and an Unbounded number of stages. The complexity of the algorithm has not been fully analyzed and remains as future work.

The algorithm is based upon the use of code-annotated extensions of base-types as a binding time specification. Such types are a rich and expressive mechanism which subsumes all other mechanisms known to the authors for the expression of binding-time specifications.

We have argued that an integrated system combining both manually staged functions, and automatically staged functions, eases the burden on the programmer. Yet, it allows the fine control that only manually staged systems have supported until now. This includes the use semantic information in staged versions that could never be fully automated.

Our proposed system integrates BTA as a semantic part of the language, and does not depend upon the intervention of some external tool, whose semantics are separate from the language.

17. ACKNOWLEDGMENTS

The work described here was supported by NSF Grant CCR-0098126, the M.J. Murdock Charitable Trust, and the Department of Defense. The authors would also like to thank the students of the class *CSE583 – Fundamentals of Staged Computation*, in the winter of 2002, who participated in many lively discussions about the uses of staging.

18. REFERENCES

[1] J. M. Ashley and C. Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1331–1448, Sept. 1994.

[2] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Berlin: Springer-Verlag, May 1990.

[3] A. Bondorf. *Self-Applicable Partial Evaluation (Revised Version)*. PhD thesis, DIKU, University of Copenhagen, Denmark, Dec. 1990. DIKU-report 90/17.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[5] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming*, pages 264–272. ACM, ACM Press, June 1990.

[6] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, 14-16 June 1993.

[7] D. Dussart, E. Bevers, and K. D. Vlaminck. Polyvariant constructor specialisation. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–65, La Jolla, California, 21-23 June 1995.

[8] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[9] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.

[10] R. Glueck and J. Joergensen. Efficient multi-level generating extensions. *Lecture Notes in Computer Science*, 982:259–278, 1995.

[11] C. K. Gomard and N. D. Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[12] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming & Computer Architecture*, pages 448–472. Springer Verlag, LNCS 523, Aug. 1991.

[13] J. Launchbury. *Projection Factorizations in Partial Evaluation*. PhD thesis, University of Glasgow, Nov.

1989.

[14] M. Leone and P. Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU-CS-93-225, Carnegie-Mellon, Department of Computer Science, Pittsburgh, PA 15212, Dec. 1993.

[15] A.-F. L. Meur, J. Lawall, and C. Consel. Towards bridging the gap between programming language and partial evaluation. In P. Thiemann, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–18, Portland, OR, USA, Jan. 2002. ACM Press.

[16] T. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In J. Diaz and F. Orejas, editors, *TAPSOFT'89, Barcelona, Spain*, volume 352 of *LNCS*, pages 298–312. Springer-Verlag, Mar. 1989.

[17] T. A. Mogensen. *Partially static structures in a self-applicable partial evaluator*. North-Holland, 1988.

[18] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda-calculus. *Science of Computer Programming*, 10(2):139–176, Apr. 1988.

[19] B. Rytz and M. Gengler. A polyvariant binding time analysis. In *Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28, San Francisco, U.S.A., June 1992. Association for Computing Machinery.

[20] T. Sheard. *Advanced Functional Programming*, chapter Using MetaML: A staged Programming Language, pages 207–240. Springer, 1999.

[21] T. Sheard. Accomplishments and research challenges in meta-programming. In W. 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.

[22] 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.

[23] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

[24] P. Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.

[25] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, Jan. 1992.

[26] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

APPENDIX

A. HASKELL INSTANTIATION

In this code we define the monad of multiple results (`M`) to be a richer monad than the list monad. This new monad supports tracing as well as the control operations listed in the main text. Tracing was instrumental in perfecting a search strategy for the algorithm.

```
data T = I          -- int
       | B          -- bool
       | Arr T T    -- T -> T
       | Code T     -- < T >
       | L T        -- T list
   deriving (Eq)

data E = EA E E           -- f x
       | EL String T E    -- (fn x => E)
       | EV String        -- x
       | EI Int           -- 5
       | Eif E E E        -- if x then y else z
       | EB E             -- < E >
       | ES E             -- ~ E
       | ELift E          -- lift E
   deriving (Eq)

data State = Up | Down | Clear  deriving Eq

type Env1 = String -> T
type Env2 = String -> T -> Int -> M E
type Component =
  State -> Int -> Env1 -> Env2 -> (T,T,E) -> M E

a1 :: State -> Int -> Env1 -> Env2 -> T -> T -> E -> M E
a1 step n sig phi t1 t2 e1 = a2 step n sig phi (t1,t2,e1)
a2 step n sig phi x =
 first
 [ escCase  step n sig phi x -- note Esc case first
 , intCase  step n sig phi x
 , varCase  step n sig phi x
 , absCase  step n sig phi x
 , appCase1 step n sig phi x
 , appCase2 step n sig phi x
 , ifCase   step n sig phi x
 , liftCase step n sig phi x
 , codeCase step n sig phi x
 ]

dom :: Env1 -> E -> T
dom sig e0 = case typeof sig e0 of Arr s1 s2 -> s1

intCase _ n sig phi (I,t2 @ I,e @ (EI i)) =
  trace "Int " n e t2 $ return (EI i)
intCase _ _ _ _ _ = mzero

varCase _ n sig phi (t1,t2,e @ (EV x)) =
  trace2 "Var " n e t2 $ phi x t2 n
varCase _ _ _ _ _ = mzero

ifCase step n sig phi (t1,t2,e @ (Eif e0 e1 e2)) =
 trace "If  " n e t2 $
 do e3 <- a1 Clear n sig phi B B e0
    e4 <- a1 Clear n sig phi t1 t2 e1
    e5 <- a1 Clear n sig phi t1 t2 e2
    return (Eif e3 e4 e5)
ifCase _ _ _ _ _ = mzero

absCase _ n sig phi (Arr t1 s1,
                     s @ (Arr t2 s2),
                     e @ (EL x t e1)) =
 trace "Abs " n e s $
 do e2 <- a1 Clear n
              (ext sig x t1)
              (ext phi x (varTest x t2 n ))
              s1 s2 e1
    return (EL x t2 e2)
absCase _ _ _ _ _ = mzero

appCase1 _ n sig phi (t1,t2,e @ (EA e0 e1)) =
```

```
   trace "App1" n e t2 $
   do let s1 = dom sig e0 -- Compute domain type of e0
      let s2 = s1
      e2 <- a1 Clear n sig phi (Arr s1 t1)
                               (Arr s2 t2) e0
      e3 <- a1 Clear n sig phi s1 s2 e1
      return (EA e2 e3)
appCase1 _ _ _ _ _ = mzero

appCase2 _ n sig phi (t1,t2,e @ (EA e0 e1)) =
   trace "App2" n e t2 $
   do let s1 = dom sig e0 -- Compute domain type of e0
      let s2 = Code s1
      e2 <- a1 Clear n sig phi (Arr s1 t1)
                               (Arr s2 t2) e0
      e3 <- a1 Clear n sig phi s1 s2 e1
      return (EA e2 e3)
appCase2 _ _ _ _ _ = mzero

codeCase Down _ _ _ _ = mzero
codeCase prevStep n sig phi (t1,t @ (Code t2),e1) =
 trace "Code" n e1 t $
 do e2 <- a1 Up (n+1) sig phi t1 t2 e1
    return (EB e2)
codeCase _ _ _ _ _ = mzero

escCase Up  _ _ _ _ = mzero
escCase _ (n+1) sig phi (t1,t2,e1) =
 trace "Esc " (n+1) e1 t2 $
 do e2 <- a1 Down n sig phi t1 (Code t2) e1
    return (ES e2)
escCase _ _ _ _ _ = mzero

liftCase _ _ _ _ (_,_, EI _)       = mzero
liftCase _ _ _ _ (_,_, EV "true")  = mzero
liftCase _ _ _ _ (_,_, EV "false") = mzero
liftCase _ n sig phi (I,t2 @ (Code I),e1) =
  trace "Lift" n e1 t2 $
    fmap ELift (a1 Clear n sig phi I I e1)
liftCase _ n sig phi (B,t2 @ (Code B),e1) =
  trace "Lift" n e1 t2 $
    fmap ELift (a1 Clear n sig phi B B e1)
liftCase _ _ _ _ _ = mzero

---------------- The Monad -----------------------

data M a = M ([a],String)

instance Monad M where
  return x = M([x],[])
  fail s = M([],s)
  (M f) >>= g =
     M(case f of
        (as1,cs1) ->
          let acc [] cs = ([],cs)
              acc (x:xs) c0 =
                 case (g x) of     { M (ys,c1) ->
                 case acc xs c1 of { (zs,c2) ->
                    (ys ++ zs,c0 ++ c2) }}
          in acc as1 cs1)

instance MonadPlus M where
  mzero = (M ([],[]))
  mplus (M f) (M g) =
    M(case f of
         (xs1,cs1) ->
            case g of
              (xs2,cs2) -> (xs1 ++ xs2,cs1 ++ cs2))

leftChoice (M f) (M g) =
  M(case f of
      (triple @ (xs1,cs1)) ->
        if null xs1
          then case g of (xs2,cs2)->(xs2,cs1++cs2)
          else triple)

first []     = fail "first done\n"
first [x]    = x
first (x:xs) = leftChoice x (first xs)

many []     = fail "many done\n"
many [c]    = c
many (x:xs) = mplus x (many xs)

eval (M (xs,s)) = (xs,s)

ext f x y s = if x==s then y else f s

----------------------------------------
varTest s t1 = varTest' s [(t1,EV s)]

varTest' :: String -> [(T,E)] -> Int -> T -> Int -> M E
varTest' s _ m _ n | m > n =
   fail ("variable bound in phase " ++ show m ++
         " used too early in phase " ++ show n )
varTest' s t1e2s m t2 n =
  case lookup t2 t1e2s of
   Just e2 -> return e2
   Nothing ->
     fail ("Fails at: " ++
           show (head(map fst t1e2s)) ++ "\n")
```