An Efficient Staging Algorithm for Binding-Time Analysis

Takuma Murakami[1], Zhenjiang Hu[1,2],
Kazuhiko Kakehi[1], and Masato Takeichi[1]

[1] Department of Mathematical Informatics,
Graduate School of Information Science and Technology, University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
{murakami,kaz}@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp
[2] PRESTO 21, Japan Science and Technology Corporation

1 Introduction

Binding-Time Analysis (BTA) is one of the compile time program analyses which is a general framework for program optimization and program generation [1]. BTA divides a whole program into static and dynamic parts from a binding-time specification so that partial evaluators can perform static computations in compile time [2]. A binding-time specification gives information about availability of data: *static* data are available at compile time while *dynamic* data are available at run time. From a binding-time specification of the input BTA determines which computation can be done statically by propagating information on static data. Partial evaluators specialize the program for the static data by using the information from BTA to generate a more efficient program than the original one.

By now several algorithms for BTA such as abstract interpretation [3], type inference [4], constraint solving [5] or type based search [6] have been developed. Looking at the type directed searching approach proposed by Tim Sheard and Nathan Linger [6], we found two unclear points. First, it is not obvious how the algorithm can cut off the searching space, and the precise running time is left unestimated. The second point is the accuracy of the solution. The result obtained by the algorithm is good, but may not be best. These two kinds of uncertain are due to the existence of heuristics. We make use of the established method of program transformation, *the optimization theorem* for the maximum marking problem [7] [8]. It transforms a naive algorithm to an efficient one by means of dynamic programming. Because it formally gives the estimation of computation complexity and guarantees of optimality of solutions with respect to given weight functions, we get a clear algorithm.

The main contribution of our algorithm is to avoid iterations in BTA. Because of the finiteness of binding-time, we are able to compute all possible binding-time annotations by only one traversal over the given program. The naive algorithm in this approach obviously requires a large table to memoize intermediate results but we address the problem by applying the optimization theorem. The theorem

drastically reduces the intermediate table if a problem is in a suitable form in which we are to formalize the original BTA problem.

Moreover, our algorithm computes the optimal solution under some given measurement of optimality. Generally an expression has multiple ways of binding-time annotations that conform to given binding-time specification. In such cases we want to select the optimal one as the result. Though the measurement of optimality is up to users, circumstances and purposes, a result which computes more parts in static is normally better than the others. Our algorithm allows users to define their own measurement, and it derives the optimal solution under the measurement.

In this paper we formalize the staging problem that we solve in Section 2. In Section 3 we restate the staging problem from the perspective of the maximum marking problem to apply the optimization theorem. Section 4 briefly shows the derived efficient staging algorithm. Section 5 concludes.

2 The Staging Problem

In this section we describe the problem to be solved. After introducing the subject language of our algorithm in Section 2.1 we state staging problem in Section 2.2.

2.1 A Simple Staged Language

In this paper we deal with the staged ML of MetaML [9] as the subject language. MetaML is a dialect of Standard ML with staged computation. In short, a piece of code in stage i is data in stage $i+1$ in MetaML. It adds two notations to operate on stages. First, angle brackets (`<>`) are called *brackets* and bring up surrounded expression into one upper stage. For instance the expression `<5>` denotes stage 1 code which results in the value `5` if executed. Types also reflect stages, hence the type of the expression is `<int>`. Second, tildes (`~`) are called *escapes*. Placing an escape just before an expression drops down its stage. It extracts the piece of code from the expression and replaces original expression with that. It should be noted that escapes must occur in stage 1 or higher because their purpose is to construct a new piece of code from other ones. For example in the case `x = <5-2>`, the expression `<3 + ~x>` evaluates to `<3 + 5 - 2>`.

The abstract syntax of the subject language is defined by BNF as in Fig. 1. There are staged expressions ($\bar{e}$) and staged types ($\bar{t}$) in addition to familiar expressions (e) and types (t). Here we consider only integers (I) and booleans (B) as values.

For example of the language, a staged version of an increment function is defined as follows (the concrete syntax of MetaML is based on Standard ML).

```
val inc' = fn x:<int> => < ~x + 1 >;
```

Its type is

```
<int> -> <int>
```

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x : t.\bar{e} \mid \bar{e}\ \bar{e} \mid \text{if } \bar{e}\ \bar{e}\ \bar{e} \\
t & ::= & I \mid B \mid \bar{t} \to \bar{t} \\
\\
\bar{e} & ::= & e \mid \langle e \rangle \mid \tilde{}\, e \\
\bar{t} & ::= & t \mid \langle t \rangle
\end{array}
$$

Fig. 1. The subject language

which means it accepts a piece of code yielding an integer and returns another piece of code yielding the incremented integer.

2.2 Staging

The staging problem is essentially much like BTA problem, except that it uses *bracket* and *escape* notations of MetaML rather than static and dynamic notations.

In order to formalize our staging problem we first introduce the notion of *erasure*. Intuitively, we can gain the erasure of a given expression by removing brackets and escapes. The function $\mathcal{E}$ defined below recursively computes the erasure of a given expression.

$$
\begin{array}{ll}
\mathcal{E}(c) = c & \mathcal{E}(\langle e \rangle) = \mathcal{E}(e) \\
\mathcal{E}(x) = x & \mathcal{E}(\tilde{}\, e) = \mathcal{E}(e) \\
\mathcal{E}(\lambda x : t.\bar{e}) = \lambda x : t.\mathcal{E}(\bar{e}) & \\
\mathcal{E}(\bar{e}_1\ \bar{e}_2) = \mathcal{E}(\bar{e}_1)\ \mathcal{E}(\bar{e}_2) & \\
\mathcal{E}(\text{if } \bar{e}_1\ \bar{e}_2\ \bar{e}_3) = \text{if } \mathcal{E}(\bar{e}_1)\ \mathcal{E}(\bar{e}_2)\ \mathcal{E}(\bar{e}_3) &
\end{array}
$$

Staging problem is formalized in [6] as follows: Given an unstaged expression e of type t and a desired staged type $\bar{t}$, staging is to find the optimal staged expression $\bar{e}$ which has the type $\bar{t}$ and its erasure is e. In general, such $\bar{e}$ is not unique so we intend to give the optimal one under an estimation of optimality.

For example of staging, consider the ML power function which computes n-th power of x:

```
fun pow n x = if n = 0 then 1 else x * pow (n-1) x
```

Note that its type is

```
int -> int -> int
```

The left tree in Fig. 2 shows the body of this `pow` function as a syntax tree.

Assume that we want to fix the first parameter n to a value and call it repeatedly with many different x's. Then we can give the binding-time specification of the desired function as

```
int -> <int> -> <int>
```

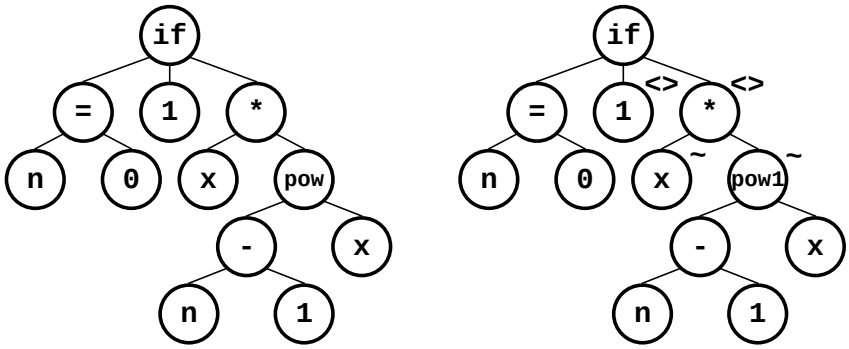

Fig. 2. Syntax tree of `pow` and `pow1`

It means that the first parameter to this function is a normal value whereas the second parameter is a piece of code which yields an integer value when executed. From the definition of `pow` and the binding-time specification we expect to get the staged version of the power function

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

which is the right tree of Fig. 2. Note that stage notations like brackets and tildes are not nodes in the figure; instead they are represented as marks on nodes. The figure illustrates the basic idea of our approach to regard the staging problem as a marking problem. Our algorithm indeed derives the definition of `pow1` above.

3 An Optimal Staging Algorithm

Our basic policy is to develop an algorithm to solve the staging problem in terms of algorithm for the maximum marking problem and applying the optimization theorem [7] to make the algorithm efficient. To achieve this goal, our first algorithm must satisfy conditions required by the optimization theorem. In this section we design our first algorithm for the maximum marking problem in Section 3.1 and apply the optimization theorem to the algorithm to derive the efficient algorithm in Section 3.2.

3.1 Staging as a Maximum Marking Problem

Maximum Marking Problem (MMP for short) [7] is a class of problems on recursive data structures. To focus on our algorithm we discuss the syntax tree of the subject language here. MMP is the problem to find a marking on the data structure that accomplishes maximum weight by means of a weight function.

Mark	Weight	
None	0	no staging
<>	1	brackets
~	3	escapes

Fig. 3. Mapping from marks to weights

We can consider multiple *marks*, a *predicate* to define acceptable markings and a *weight function* to evaluate total weight of marked data structures. A naive solution of MMP is

1. to generate all possible markings,
2. to take valid markings which satisfy the predicate p, and
3. to return the maximum one according to the weight function w.

As we regard the staging problem as a marking problem like Fig. 2, we construct an algorithm to solve the problem by suitably selecting marks, a predicate and a weight function.

First, considering bracket and escape as the two marks for syntax tree is quite natural. A syntax tree with no marks represents the unstaged expression. We can get erasure of an expression by stripping all marks from the syntax tree of it.

Second, the predicate p judges validity of markings. Only valid markings can be solution of problem. We employ type judgment as the predicate in our algorithm so that it always returns correctly typed expressions as the result. The type judgment includes staged type judgment as well as ordinary ML type judgment. Section 4.1 illustrates some more about the staged type judgment.

Finally, we consider the weight function w which determines the way to select the optimal solution. It is basically composed of two parts: a mapping from each mark to weight and a recursive function that calculates the weight of a parent node from the weights of child nodes. Since the goal of staging is making computations in earlier stages as possible, escapes are preferred to brackets which make computations dynamic. We give higher weight to escape marks than bracket marks to represent our intention in the algorithm. Our current system gives the weight 3 to escapes and 1 to brackets as shown in Fig. 3. For the latter component of w, we simply sum up the weights of all child nodes. It might be the simplest form of weight functions but works fine for our small experiments.

3.2 Solving the Staging Problem Efficiently

The optimization theorem [7] for MMP eliminates the computational explosion caused by the naive algorithm. It gives a transformation from a naive algorithm to a linear-time algorithm provided p and w satisfy the following conditions:

1. the number of marks is finite

2. p is in the form of catamorphism
3. domain of p is finite
4. w is in the form of catamorphism

where catamorphism is a generalized form of bottom-up recursive functions on recursive data structures [10]. We represent programs as syntax trees with four kinds of nodes, therefore catamorphism on the syntax trees is defined as follows:

$$
\begin{aligned}
&(\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ (V\ e) = \phi_1\ e \\
&(\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ (L\ e\ r_1\ r_2) \\
&\qquad = \phi_2\ e\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_1)\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_2) \\
&(\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ (A\ e\ r_1\ r_2) \\
&\qquad = \phi_3\ e\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_1)\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_2) \\
&(\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ (I\ e\ r_1\ r_2\ r_3) \\
&\qquad = \phi_4\ e\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_1)\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_2)\ ((\![\phi_1, \phi_2, \phi_3, \phi_4]\!)\ r_3)
\end{aligned}
$$

Note that the data constructors V, L, A, and I correspond variables, lambda abstractions, applications, and conditionals (if-then-else) respectively.

In our algorithm described in the previous section marks are bracket and escape. Thus the number of marks is 2, which is of course finite and satisfies the condition for the optimization theorem.

Type judgment is performed in a recursive way from subexpressions to larger pieces of expressions. It is exactly the form of catamorphism which is required by the optimization theorem. Moreover, the domain of type judgment is the type of subexpressions. It is finite for any programs and usually small enough. This fact enables the application of optimization theorem.

The weight function we employ is also a catamorphism because of the simple recursive form of summation. In future we can precompute some information on each node at the preprocessing phase and improve the weight function by making use of the additional information.

As a result, our choice of marks, a predicate and a weight function satisfies the conditions required by the optimization theorem so that we are able to obtain an efficient algorithm for staging.

4 Implementation

The optimization theorem transforms a naive algorithm to the efficient one. Although the program transformation is formally defined, it is somewhat complicated step to derive the efficient algorithm. We derived the efficient algorithm for staging from the naive one and implement it in a tiny system using the language Haskell [11]. The system accepts the language defined in Section 2.1 and applies our new algorithm to obtain the staged expression. For the sake of simplicity current system is implemented rather naively though the algorithm is not naive.

In this section we briefly describe the algorithm we implement. Section 4.1 and Section 4.2 formalize the type checking algorithm and the weight function

by means of catamorphism [10]. Section 4.3 shows the core algorithm which fuses the marking generation, the type checking, and the weight function into one catamorphism.

4.1 Type Judgment

The type judgment, which is the predicate p in our algorithm, is formalized as a catamorphism of a series of functions ϕ_i. The catamorphism scans the syntax tree in bottom-up way, applies ϕ_i depending on the type of nodes, and yields the final result (the type of the whole program or a type error). A staged type is represented as a pair of a type and a stage in the implementation.

For instance, ϕ_3 judges the application case

$$\frac{\Gamma \vdash_{s_1} e_1 : t_1 \quad \Gamma \vdash_{s_2} e_2 : t_2}{\Gamma \vdash_s e_1\, e_2 : t} A$$

where the subscripts represent the stages of (sub)expressions. The return value of

$$\phi_3\ e^*\ (t_1, s_1)\ (t_2, s_2)\ s$$

is (t, s) if the judgment holds, or $\perp$ (denoting type errors) if it does not hold. Note that the arguments (t_1, s_1) and (t_2, s_2) are results of type judgments of subexpressions e_1 and e_2, which are recursively performed by the catamorphism.

4.2 Weight Function

Similar to the type judgment in the previous section, the weight function w is also defined by catamorphism as

$$\begin{aligned} w = (\!|\mu_1, \mu_2, \mu_3, \mu_4|\!) \circ map\ f \\ \text{where } \mu_1\ e &= e \\ \mu_2\ e\ r_1\ r_2 &= e \oplus r_1 \oplus r_2 \\ \mu_3\ e\ r_1\ r_2 &= e \oplus r_1 \oplus r_2 \\ \mu_4\ e\ r_1\ r_2\ r_3 &= e \oplus r_1 \oplus r_2 \oplus r_3 \end{aligned}$$

The function f computes weights for each marked node of syntax tree, and then the catamorphism accumulates the tree of weights. The binary operator $\oplus$ characterizes how to combine the weights of subexpressions into one value. In our system we currently use $+$ for $\oplus$.

4.3 Derived Algorithm

Applying the optimization theorem to the naive algorithm composed of the type judgment (Section 4.1) and the weight function (Section 4.2) we derived an efficient algorithm for staging as in Fig. 4. The function *core* plays the central role in our algorithm. It takes two arguments: t_0 is the desired staged type and

$$
\begin{array}{l}
\mathit{core} :: \mathit{Type} \to E\ \alpha \to E\ \alpha^* \\
\mathit{core}\ t_0\ e = \\
\quad \text{let } \mathit{opts} = (\!|\psi_1, \psi_2, \psi_3, \psi_4|\!)\ e \\
\quad \text{in } \mathit{snd}\ (\mathit{reduce}\ (\uparrow_{\mathit{fst}})\ [\ (w, r^*)\ |\ (((t, s), \_), w, r^*)\ \leftarrow \mathit{opts}, \mathit{accept}\ (t, s)\ ]) \\
\quad \text{where } \psi_3\ e\ \mathit{cand}_1\ \mathit{cand}_2 = \\
\qquad \mathit{eachmax}\ [\ (\ (\phi_3\ e^*\ (t_1, s_1)\ (t_2, s_2)\ s, s),\ f\ e^* \oplus w_1 \oplus w_2,\ A\ e^*\ r_1^*\ r_2^*\ ) \\
\qquad\qquad |\ e^* \leftarrow \mathit{marks}\ e, \quad s \leftarrow \mathit{Stage}, \\
\qquad\qquad\ (((t_1, s_1), s_1), w_1, r_1^*) \leftarrow \mathit{filter}\ (\mathit{stageEq}\ (\delta\ e^*\ s))\ \mathit{cand}_1, \\
\qquad\qquad\ (((t_2, s_2), s_2), w_2, r_2^*) \leftarrow \mathit{filter}\ (\mathit{stageEq}\ (\delta\ e^*\ s))\ \mathit{cand}_2\ ] \\
\qquad \mathit{marks}\ e = [\ (e, m)\ |\ m \leftarrow \mathit{Mark}\ ] \\
\qquad \mathit{stageEq}\ s\ (((\_, \_), s'), \_, \_) = s == s' \\
\qquad \delta\ e^*\ s = s \\
\qquad \mathit{accept}\ (t, s) = t == t_0\ \&\&\ s == 0 \\
\qquad \phi_3\ e^*\ (t_1, s_1)\ (t_2, s_2)\ s = \text{type judgment in Section 4.1} \\
\qquad (f, \oplus) = \text{a part of weight function in Section 4.2}
\end{array}
$$

Fig. 4. Core algorithm of staging

e is the expression to be staged. The return value is the staged version of e. This corresponds the concrete program

```
stage e0 = e at t0;
```

in our system.

In the algorithm the three steps in the naive algorithm namely

1. generation of all possible markings,
2. staged type checking, and
3. selection of the optimal solution

are fused into one large loop in the form of catamorphism. When ψ_i is applied to each node by the catamorphism, it performs three steps above. In the body of ψ_3 in Fig. 4 the list comprehensions ($e^* \leftarrow \mathit{maks}\ e$ and $s \leftarrow \mathit{Stage}$) generate all possible staged types as step 1. The generated pairs are immediately compared with the recursive parts (cand_i) and only valid pairs are filtered and processed further.

The elements of the list comprehension have their staged types, weights and marked trees itself. There exists the type judgment ($\phi_3\ e^*\ (t_1, s_1)\ (t_2, s_2)\ s$) described in Section 4.1 that checks the type as soon as possible. The second component of the triplet ($f\ e^* \oplus w_1 \oplus w_2$) makes use of f and $\oplus$ which are parts of the weight function in Section 4.2. The last component of the triplet records the marked subtree under current node to return as result.

The function *eachmax* aggregates the list to reduce intermediate results. If there are plural elements of the same staged type, *eachmax* returns the one having the maximum weight. Thus the length of the list returned by ψ_i is at most $|\mathit{Type}|$, though it is usually much smaller.

5 Conclusion

We have developed an efficient algorithm to solve staging problems as maximum marking problems. The optimization theorem has played an important role to derive efficient solutions. The complexity of running time is certainly reduced though the constant factor is rather big. We should also reduce the constant by improving both algorithm and implementation.

Due to the optimization theorem, our algorithm runs in

$$\mathrm{O}\left((|\mathit{Type}| \times |\mathit{Stage}|)^3 \times |\mathit{Mark}| \times n\right).$$

Here $|\mathit{Stage}|$ is a small number (2 for static and dynamic case) and $|\mathit{Mark}|$ is 2 (bracket and escape). $|\mathit{Type}|$ is the maximum arity of all functions within the analyzed program. It is naturally finite for any programs and usually small number which seldom exceeds 8.

Besides the reduced running time our algorithm can be extended in some directions. We already have extended *Stage* from 2 stages to 3 stages. Thanks to the optimization theorem, we just needed to rewrite $\mathit{stages} = [0..1]$ to $\mathit{stages} = [0..2]$ in our Haskell program to obtain the 3-stage algorithm. In this case each node cannot have more than one mark, so $\langle\langle x \rangle\rangle$ is not allowed. It can be addressed by adding the new mark `<<>>` to *Mark*. Another opportunity of extending *Mark* is to support more syntaxes on stages. Our system actually supports the mark `lift` that corresponds the keyword "lift" in MetaML. All the extensions are in the framework of MMP so that the optimization theorem is applicable.

The last point is the weight function. Because of the modularity of the algorithm we can easily redefine the weight function and apply the optimization theorem to get an improved staging algorithm. The optimization theorem guarantees that the derived algorithm always computes the optimal solution with respect to the given weight function, hence the refinement of the weight function is the key of our algorithm to get more accurate staging algorithms. Although current system employs rather simple weight function, we are trying out other measurements of optimality. For example, we can consider another weight function which takes account of the number of child nodes.

References

1. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
2. Jones, N.D.: An Introduction to Partial Evaluation. ACM Computing Surveys **28** (1996) 480–503
3. Consel, C.: Binding Time Analysis for Higher Order Untyped Functional Languages. In: Proceedings of the 1990 ACM conference on LISP and functional programming, ACM Press (1990) 264–272
4. Henglein, F.: Efficient Type Inference for Higher-Order Binding-Time Analysis. In: Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture. Volume 523 of Lecture Notes in Computer Science., Cambridge, USA, Springer-Verlag (1991) 448–472 Lecture Notes in Computer Science, Vol. 523.

5. Glück, R., Jørgensen, J.: Fast Binding-Time Analysis for Multi-Level Specialization. In: Proceedings of Perspectives of System Informatics. (1996) 261–272
6. 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, Aizu, Japan (2002) 20–31
7. Sasano, I., Hu, Z., Takeichi, M., Ogawa, M.: Make it Practical: A Generic Linear-Time Algorithm for Solving Maximum-Weightsum Problems. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, Montreal, Canada (2000) 137–149
8. Sasano, I., Hu, Z., Takeichi, M.: Generation of Efficient Programs for Solving Maximum Multi-Marking Problems. In: Workshop on the Semantics, Applications, and Implementation of Program Generation. Volume 2196 of Lecture Notes in Computer Science. Springer-Verlag, Firenze, Italy (2001) 72–91
9. Sheard, T.: Using MetaML: A Staged Programming Language. In: Advanced Functional Programming. Volume 1129 of Lecture Notes in Computer Science., Springer-Verlag (1998) 207–239
10. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1997)
11. Bird, R.: Introduction to Functional Programming using Haskell (2nd edition). Prentice Hall (1998)