EPiC Series in Computing
Volume 39, 2016, Pages 1–13
SCSS 2016. 7th International Symposium on Symbolic Computation in Software Science

Automatic Staging via Partial Evaluation Techniques

Kenichi Asai[1] and Yukiyoshi Kameyama[2]

[1] Ochanomizu University, Japan
asai@is.ocha.ac.jp
[2] University of Tsukuba, Japan
kameyama@acm.org

Abstract

Partial evaluation and staging are two of the well-known symbolic manipulation techniques of programs which generate efficient specialized code. On one hand, partial evaluation provides us with an automatic means to separate a program into two (or more) stages but its behavior is perceived as hard to control. On the other hand, staging (or staged calculus) requires us to manually separate a program but with full control over its behavior. In the previous work, the first author introduced a framework to relate the two techniques, giving a unified view to the two techniques. In this paper, we extend the framework to handle the cross-stage persistence (CSP) and show that the 2-level staging annotation obtained by the automatic separation is the best staging annotation in a system where CSP is allowed for base-type values only. In the presence of CSP for higher-type values, on the other hand, there is no single annotation that is better than all the other annotations.

1 Introduction

Programming languages for *staged computation* prevail; they allow programmers to write code generators that exploits domain-specific knowledge and optimizations, thus allowing generation of efficient code. They are used in a wide range of application domains including numeric computation [8, 10], image processing [11], and other high-performance computing applications [2, 15]. Staged languages including MetaOCaml [7, 14] let programmers control the behavior of code generators, by manually inserting *staging annotations*.

Although their manual insertion allows the fine control which is often necessary for expressing domain-specific optimizations, it soon becomes cumbersome for increasingly large programs, especially when the process is mechanical. This situation is in contrast to partial evaluation [6] where binding-time analysis automatically separates a program into multiple stages and thus no manual intervention is needed. Even though binding-time analysis sometimes requires careful modification of the source program to produce satisfactory results, it can certainly save the programmers' efforts for manual separation.

To benefit from both approaches, our previous work [1] introduced a unified framework that relates the staged λ-calculus (the base theory for staged languages) with the 2-level λ-calculus (the base theory for partial evaluation), and showed that they are essentially the same.

In this paper, we extend the previous framework to handle cross-stage persistence (CSP) which appears in many staged languages. We are particularly interested in the problem of finding the *best* staging annotation where the best staging annotation informally means that a program with it generates the most evaluated (specialized) code. The problem turns out to be non-trivial in the presence of CSP; defining the best annotation is already problematic in staged languages.

Our results show that, for a calculus with CSP restricted to base-type values, we can define the best staging annotation naturally, and an algorithm to obtain the best staging annotation by transforming the result of binding-time analysis to staging annotations.

Interestingly, if we allow CSP of higher-type values, things change drastically. We show that in the presence of CSP of higher-type values, there are distinct staging annotations neither of which is better than the other, which means that there is no best staging annotation in this setting. Such discussion becomes possible because we can define a natural notion of the best staging annotation.

The contributions of this paper are summarized as follows.

- We define the best staging annotation in terms of binding times of terms and types.
- We show an algorithm to obtain the best staging annotation for the staged λ-calculus with CSP of base-type values.
- We show that in the presence of CSP of higher-type values, there does not necessarily exist the best staging annotation.

Glossary. *Staged computation* separates a program into multiple stages. At runtime, earlier-stage (compile-time) computation is first performed, leaving later-stage (runtime) computation as code values, effectively achieving code generation. In this paper, we focus on a certain kind of staged computation where we write staging annotations on terms explicitly. The staging annotations usually include *bracket* to create a piece of code and *escape* to perform earlier-stage computation within a code segment. In addition, it is often useful if we can use earlier-stage values directly in the later-stage computation, which is called *cross-stage persistence* or CSP. The original CSP allows any values, including *higher-type* values (functions), to be lifted to later stages. Some staged languages restrict the use of CSP to *base-type* values (non-function values such as integers and strings) and higher-type primitives (such as the predefined + operation).

Partial evaluation, on the other hand, is a program transformation technique that, given a general program and some of its input, produces a specialized efficient version of the program using the given input. The *offline* variant of partial evaluation first separates a program into multiple stages by assigning each program part a *binding time* indicating if it can be executed at partial evaluation time. The two-stage binding times consist of *static* (known at partial evaluation time) and *dynamic* (deferred to runtime). Partial evaluation then executes all the static parts to produce a specialized program.

Organization. The paper is organized as follows. We introduce the staged λ-calculus in Section 2 and discuss various forms of CSP in Section 3. We then introduce the 2-level λ-calculus and discuss the best binding time in Section 4. In Section 5, we relate the two calculi and show how to obtain the best staging annotation from the result of binding-time analysis. Related work is in Section 6 and the paper concludes in Section 7.

2 Staged λ-calculus

The base calculus we consider is a simply-typed λ-calculus extended with integers and additions. We assume that the input term is already well typed according to the standard typing rules as

$$
\begin{array}{lcl}
\text{raw types:} & t & := \ \mathsf{int} \mid t_1 \to t_2 \\
\text{raw terms:} & e & := \ i \mid x \mid e_1 \mathbin{@} e_2 \mid \lambda x.\, e_1 \mid e_1 + e_2 \\
\text{typing rules:} & &
\end{array}
$$

$$
\frac{}{\Gamma \vdash i : \mathsf{int}}\ (\mathsf{TInt}_1) \quad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ (\mathsf{TVar}_1) \quad \frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \mathbin{@} e_2 : t_1}\ (\mathsf{TApp}_1)
$$

$$
\frac{\Gamma, x : t_2 \vdash e_1 : t_1}{\Gamma \vdash \lambda x.\, e_1 : t_2 \to t_1}\ (\mathsf{TLam}_1) \quad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ (\mathsf{TSum}_1)
$$

Figure 1: Simply-typed λ-calculus

shown in Figure 1.

On top of the simply-typed λ-calculus, the staged λ-calculus introduces three new constructs (*staging annotations*) to handle code values (Figure 2). First, a bracket expression $\langle e \rangle$ builds a piece of code representing e. For example, $\langle (\lambda x.\, x) \mathbin{@} (1 + 2) \rangle$ is a piece of code that applies an identity function to the result of $1 + 2$ when executed. In the MetaOCaml syntax, it is written as:

```
# .<(fun x -> x) (1 + 2)>. ;;
- : int code = .<(fun x_1  -> x_1) (1 + 2)>.
```

When the type of e is t, the type of $\langle e \rangle$ becomes $\langle t \rangle$ (t `code` in the MetaOCaml syntax). To build a larger code value from a smaller one, we use an escape expression $\sim e$. When $\sim e$ appears within $\langle \ldots \rangle$, the result of evaluating e, which must be a code value, is spliced in to the surrounding code $\langle \ldots \rangle$. For example, $\langle 3 + \sim ((\lambda x.\, x) \mathbin{@} \langle 1 + 2 \rangle) \rangle$ evaluates to $\langle 3 + (1 + 2) \rangle$, where the function application has been reduced and its result is spliced in to the surrounding code.

```
# .<3 + .~((fun x -> x) .<1 + 2>.)>. ;;
- : int code = .<3 + (1 + 2)>.
```

Note that we cannot (yet) reduce $1+2$ in this program as in $\langle 3 + \sim ((\lambda x.\, x) \mathbin{@} (1 + 2)) \rangle$, because the result of evaluating $(\lambda x.\, x) \mathbin{@} (1 + 2)$ is not a code value $\langle 3 \rangle$ but a present-stage value 3.

```
# .<3 + .~((fun x -> x) (1 + 2))>. ;;
Error: This expression has type int but an expression was expected of type
         int code
```

To reduce $1 + 2$, we need to turn the integer 3 to a code value $\langle 3 \rangle$. The % expression is used for this purpose. By writing $\%x$, we can use a base-type variable x holding a present-stage value (i.e., a non-code value) at the future stage (i.e., within $\langle \ldots \rangle$). For example, $\langle 3 + \sim ((\lambda x.\, \langle \%x \rangle) \mathbin{@} (1 + 2)) \rangle$ evaluates to $\langle 3 + 3 \rangle$. The present-stage value 3 in x is promoted to a code value at $\langle \%x \rangle$ and is spliced in to the surrounding code.[1]

```
# .<3 + .~((fun x -> .<x>.) (1 + 2))>. ;;
- : int code = .<3 + (* CSP x *) Obj.magic 3>.
```

Another way to reduce $1 + 2$ is to insert % in front of $1 + 2$ as in $\langle 3 + \sim ((\lambda x.\, x) \mathbin{@} \%(1 + 2)) \rangle$. The result of $1 + 2$ (i.e., 3) is promoted to a code value $\langle 3 \rangle$, and passed to the identity function

[1] In MetaOCaml, this promotion of variables is done automatically. We do not have to write it explicitly.

staged types: $t \;:=\; \mathsf{int} \mid t_1 \to t_2 \mid \langle t_1 \rangle$
staged terms: $e \;:=\; i \mid x \mid e_1 \,@\, e_2 \mid \lambda x.\, e_1 \mid e_1 + e_2 \mid \langle e_1 \rangle \mid \;\sim e_1 \mid \%e_1$
typing rules:

$$\frac{}{\Gamma \vdash_b i : \mathsf{int}}\ (\mathsf{TInt}_2) \qquad \frac{\Gamma(x^b) = t}{\Gamma \vdash_b x : t}\ (\mathsf{TVar}_2) \qquad \frac{\Gamma \vdash_b e_1 : t_2 \to t_1 \quad \Gamma \vdash_b e_2 : t_2}{\Gamma \vdash_b e_1 \,@\, e_2 : t_1}\ (\mathsf{TApp}_2)$$

$$\frac{\Gamma, x^b : t_2 \vdash_b e_1 : t_1}{\Gamma \vdash_b \lambda x.\, e_1 : t_2 \to t_1}\ (\mathsf{TLam}_2) \qquad \frac{\Gamma \vdash_b e_1 : \mathsf{int} \quad \Gamma \vdash_b e_2 : \mathsf{int}}{\Gamma \vdash_b e_1 + e_2 : \mathsf{int}}\ (\mathsf{TSum}_2)$$

$$\frac{\Gamma \vdash_1 e_1 : t_1}{\Gamma \vdash_0 \langle e_1 \rangle : \langle t_1 \rangle}\ (\mathsf{TCod}_2) \qquad \frac{\Gamma \vdash_0 e_1 : \langle t_1 \rangle}{\Gamma \vdash_1 \sim e_1 : t_1}\ (\mathsf{TEsc}_2) \qquad \frac{\Gamma \vdash_0 e_1 : \mathsf{int}}{\Gamma \vdash_1 \%e_1 : \mathsf{int}}\ (\mathsf{TCsp}_2)$$

Figure 2: Staged λ-calculus

$\lambda x.\, x$. The final result becomes $\langle 3 + 3 \rangle$. (We cannot show this example in MetaOCaml, since MetaOCaml does not support promotion of an expression other than a variable — but we can emulate it; see Section 3.1). The promotion of present-stage values to the future stage is called *cross-stage persistence* or CSP. There are several forms of CSP and we will discuss them in the next section.

The typing rules for the staged λ-calculus are shown in Figure 2. The typing judgement has the form $\Gamma \vdash_b e : t$, which reads: under a typing environment Γ, a term e has type t at stage b. The stage b indicates within how many brackets $\langle \ldots \rangle$ the term e is located. For example, $\lambda x.\, x$ in $\langle (\lambda x.\, x) \,@\, (1 + 2) \rangle$ is at stage 1. Within a bracket, an escaped expression is at stage 0 because it is evaluated at the present stage ((TCod_2) and (TEsc_2)). For example, $\lambda x.\, x$ in $\langle 3 + \sim ((\lambda x.\, x) \,@\, \langle 1 + 2 \rangle) \rangle$ is at stage 0. When % is used, a present-stage value is promoted to the later stage (TCsp_2). The stage changes only when $\langle e \rangle$, $\sim e$, or $\%e$ is used. The stage stays the same for the other standard constructs.

If an expression is surrounded by more than one bracket, the stage can be bigger than 1. For example, $1 + 2$ in $\langle\langle 1 + 2 \rangle\rangle$ is at stage 2. In this paper, however, we restrict ourselves to two stages only, and thus the stage b is either 0 or 1.

3 Variations of Cross-Stage Persistence

In the previous section, we introduced a staged λ-calculus that allows CSP for arbitrary expressions, but restricted to base types (the rule TCsp_2 in Figure 2). This differs from the current MetaOCaml which allows CSP for variables of arbitrary type.

In this section, we discuss the difference and justify our choice.

3.1 CSP of Arbitrary Expressions vs. CSP of Variables

The rule TCsp_2 in Figure 2 allows us to promote an arbitrary expression e_1, whereas MetaOCaml allows the promotion of variables only. This does not bring an essential difference; instead of writing the expression $\%e$, we can write $(\lambda x.\, \langle \%x \rangle) \,@\, e$ which is typable if and only if $\%e$ is typable, and the two terms have the same operational behavior. (We elide the operational semantics of the calculi, as it is fairly standard. See [14] for instance.)

Although this paper employs a calculus that supports CSP of arbitrary expressions, the results in this paper carry over to a calculus with CSP of variables only via the above encoding.

3.2 CSP of Base-Type Values vs. CSP of Higher-Type Values

The rule TCsp_2 imposes a restriction that an expression of only base type can be promoted. We could obtain a calculus closer to the current MetaOCaml by allowing CSP of higher-type expressions. However, as we will see below, we would then lose an important property that the best staging annotation always exists.

Consider the term $\lambda p.\,(\lambda f.\,p \mathbin{@} f \mathbin{@} (f \mathbin{@} 1)) \mathbin{@} (\lambda x.\,(\lambda y.\,y) \mathbin{@} x)$. In this term, the function $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ is bound to f and used twice. By regarding p as a pairing function, we can think of the term as returning a pair of f and $f \mathbin{@} 1$.

Suppose we are given the above term and the type $\langle((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})) \to \mathsf{int}\rangle$, and are asked to add staging annotations to the term such that the resulting staged term has the type above. What would be the *best* annotations? By the best annotation, we informally mean the one that evaluates to the most reduced form. (The precise definition of the best annotation is non-trivial, and will be given in the next section.)

The point here is that the function $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ bound to f is used in two different ways, but we can annotate it in only one way (i.e., monovariantly). Since the value of p is not available yet, we have to leave the application of p in the resulting code.

The best annotation that does not use CSP is given as follows:

$$\langle\lambda p.\sim((\lambda f.\,\langle p \mathbin{@} \sim f \mathbin{@} (\sim f \mathbin{@} 1)\rangle) \mathbin{@} \langle(\lambda x.\sim((\lambda y.\,y) \mathbin{@} \langle x\rangle))\rangle)\rangle$$

which evaluates to $\langle\lambda p.\,p \mathbin{@} (\lambda x.\,x) \mathbin{@} ((\lambda x.\,x) \mathbin{@} 1)\rangle$. Although $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ is reduced to $\lambda x.\,x$ which is inlined in place of f, the application of $\lambda x.\,x$ to 1 is not reduced, because $\lambda x.\,x$ is a piece of code and cannot be applied. In the presence of CSP of higher-type values, on the other hand, we can annotate the above term as follows:

$$\langle\lambda p.\sim((\lambda f.\,\langle p \mathbin{@} \%f \mathbin{@} \sim(f \mathbin{@} \langle 1\rangle)\rangle) \mathbin{@} (\lambda x.\,(\lambda y.\,y) \mathbin{@} x))\rangle$$

which evaluates to $\langle\lambda p.\,p \mathbin{@} f \mathbin{@} 1\rangle$ where f is a pointer to the runtime value representing the closure $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$. Here, $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ is evaluated to a closure and is used to reduce $f \mathbin{@} 1$. However, since f is bound to a runtime closure rather than a piece of code, the result of evaluation contains a pointer to the runtime value (closure) and cannot be represented as a program text. It may not be a problem, if we only have to execute the resulting code and do not have to obtain the result as a printable program text. A bigger problem is that $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ evaluates to a closure under call-by-value and its body is not reduced. Thus, the result behaves in the same way as $\langle\lambda p.\,p \mathbin{@} (\lambda x.\,(\lambda y.\,y) \mathbin{@} x) \mathbin{@} 1\rangle$, which is clearly unsatisfactory, because $(\lambda y.\,y) \mathbin{@} x$ is not reduced.

Without CSP of higher-type values, we could reduce the body of $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ but not $f \mathbin{@} 1$. With CSP of higher-type values, the situation is the other way around. Thus, neither solution is better than the other. To reduce $f \mathbin{@} 1$, we need to classify $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ as a present-stage function whereas to reduce its body, we need to classify it as a piece of code. Put differently, we need polyvariant annotation: we need to annotate $\lambda x.\,(\lambda y.\,y) \mathbin{@} x$ in two different ways.

CSP of higher-type values is thus problematic. One simple case where CSP of higher-type values is useful is CSP of primitive functions such as + or functions defined in other modules. Since they are already in the reduced form, we do not encounter the above problem; we only have to treat the references to them as runtime values, and use CSP to embed them into code values.

4 The Best Binding Time

Given an unannotated well-typed term and its staged type, we want to add staging annotations to the term to obtain a staged term which has the specified type. For this problem, there are many possible staging annotations in general. In this section, we consider the best staging annotation among them. Intuitively, the best staging annotation is the one that evaluates as many expressions and as early as possible, which would result in the most reduced form. However, since staging annotations express only the stage of each expression and not whether the expression is a code value or not, we cannot immediately judge which staging annotation is better. Furthermore, there exist distinct but equally good staging annotations.

For example, consider the following three staged terms of the same type that we saw earlier:

$$\begin{array}{l}\langle 3+\sim((\lambda x.\, x)\,@\,\langle 1+2\rangle)\rangle\\ \langle 3+\sim((\lambda x.\, \langle \% x\rangle)\,@\,(1+2))\rangle\\ \langle 3+\sim((\lambda x.\, x)\,@\,\langle \%(1+2)\rangle)\rangle\end{array}$$

In the first annotation, $1+2$ is at stage 1 but x is at stage 0. In the second one, $1+2$ is at stage 0 but x occurs in the stage 1 context. Thus, the stages of expressions do not apparently tell us which annotation is better. Furthermore, although the second and the third ones have different staging annotations, they both reduce to the same result. These examples show that it is difficult to find the best staging annotation by simply looking at the staging annotations.

What we really want to know is the stages of the *values* of expressions (rather than the stages of the expressions). In the first annotation, both $1+2$ and x are code values. In the second one, they are both integers. In the third one, $1+2$ is an integer but x holds a code value. Because we obtain more reduced form if we know values of expressions earlier, we can conclude that the second one is a better annotation than the first one. Even for equally good staging annotations (that reduce to the same result), we can introduce a sensible order: the second one is better than the third one, because the value of x is known earlier; it would have been in fact the better annotation if x were used within the body of the lambda abstraction.

How can we obtain the stages of the values of expressions? The answer is well known in the partial-evaluation community — by binding-time analysis.

4.1 2-level λ-calculus

Given a well-typed term, binding-time analysis assigns a binding time to each of its subexpressions and types, producing a term in the so-called 2-level λ-calculus [4]. Figure 3 shows a variant of the 2-level λ-calculus formulated as a multi-level λ-calculus [3] restricted to 2 levels.

The binding time takes a value of either 0 (static; meaning the expression is at stage 0) or 1 (dynamic; meaning the expression is at stage 1, producing a piece of code). A binding time of a type represents when the *value* of an expression of that type is available. The base types int^0 and int^1 represent that an integer is available at stage 0 and 1, respectively. For the function type, not only the toplevel type but also its argument and result types come with binding times. When the toplevel binding time is static, it means that the function is available at stage 0: it can be applied to either a static or dynamic argument and produce either a static or dynamic result. If the toplevel binding time of a function is dynamic, on the other hand, the function is a piece of code. In this case, both the argument and result types must also be dynamic. This constraint is formalized as the well-formed condition on types shown in Figure 3.

In contrast to the binding time of a type, a binding time of a term represents when the term can be evaluated. A present-stage integer has the binding time 0, while integer code has

binding times:	b	$:=$	$0 \mid 1$
annotated types:	t^b	$:=$	$\mathsf{int}^b \mid {t_1}^{b_1} \to^b {t_2}^{b_2}$
annotated terms:	e^b	$:=$	$i^b \mid x^b \mid {e_1}^{b_1} \mathbin{@^b} {e_2}^{b_2} \mid \lambda^b x.\, {e_1}^{b_1} \mid {e_1}^{b_1} +^b {e_2}^{b_2} \mid \%^1 e^0$

well-formed types:

$$\frac{}{\mathsf{int}^b\ \mathsf{wft}} \qquad \frac{{t_1}^{b_1}\ \mathsf{wft} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \le b_1 \quad b \le b_2}{({t_1}^{b_1} \to^b {t_2}^{b_2})\ \mathsf{wft}}$$

typing rules:

$$\frac{}{\Gamma \vdash i^b : \mathsf{int}^b}\ (\mathsf{TInt}_3) \qquad \frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_3) \qquad \frac{\Gamma \vdash {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash {e_2}^{b'_2} : {t_2}^{b_2}}{\Gamma \vdash {e_1}^{b'} \mathbin{@^b} {e_2}^{b'_2} : {t_1}^{b_1}}\ (\mathsf{TApp}_3)$$

$$\frac{\Gamma, x^b : {t_2}^{b_2} \vdash {e_1}^{b'_1} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \le b_1 \quad b \le b_2}{\Gamma \vdash \lambda^b x.\, {e_1}^{b'_1} : {t_2}^{b_2} \to^b {t_1}^{b_1}}\ (\mathsf{TLam}_3)$$

$$\frac{\Gamma \vdash {e_1}^{b_1} : \mathsf{int}^b \quad \Gamma \vdash {e_2}^{b_2} : \mathsf{int}^b}{\Gamma \vdash {e_1}^{b_1} +^b {e_2}^{b_2} : \mathsf{int}^b}\ (\mathsf{TSum}_3) \qquad \frac{\Gamma \vdash e^0 : \mathsf{int}^0}{\Gamma \vdash \%^1 e^0 : \mathsf{int}^1}\ (\mathsf{TCsp}_3)$$

Figure 3: 2-level λ-calculus with lifting (0: static, 1: dynamic)

the binding time 1. The binding time of a variable is determined by the time the variable is introduced. A static variable can hold either a static or dynamic value, while a dynamic variable is a code value representing the variable itself. An application is static when the function part is static and can be applied. In that case, the result of application can be either static or dynamic depending on the result type of the function. Note that the binding time of an application and the binding time of the type of the application may be different. The former represents whether the application can be reduced, while the latter represents what we obtain as the result of application. A static and dynamic abstraction introduces a static and dynamic variable, respectively. The condition on binding times ensures that the introduced function type is well formed. Addition can be done statically only when both of its arguments are available. This constraint is expressed by having the same binding time b for the addition and the types of its arguments. Finally, $\%^1 e^0$ represents CSP (or *lifting* in the partial-evaluation terminology) of a base-type expression.

4.2 Binding-Time Analysis

Given a well-typed term, the binding-time analysis proceeds by attaching binding times to the term and its type according to the typing rules in Figure 3. However, although the typing rules are mostly syntax directed, the rule TCsp_3 is not: we cannot decide syntactically where to insert TCsp_3. To decide where to insert TCsp_3, we introduce the following slightly modified typing rule:

$$\frac{\Gamma \vdash e^{b_0} : \mathsf{int}^{b'_0} \quad b'_0 \le b_1}{\Gamma \vdash \%^{b_1} e^{b_0} : \mathsf{int}^{b_1}}\ (\mathsf{TCsp}'_3)$$

and insert % to all the subexpressions of type int. After all the binding-time variables obtain concrete values, we check if the inequality $b'_0 < b_1$ in TCsp'_3 holds. If it does, we leave % intact; we delete it otherwise.

For example, the term $\lambda s.\, \lambda d.\, (\lambda x.\, x + s) \mathbin{@} d$ of type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ is converted to $\lambda s.\, \lambda d.\, \%((\lambda x.\, \%(\%x + \%s)) \mathbin{@} \%d)$. We can then construct a typing derivation in a syntax-directed way, introducing binding-time variables to each subterm and type and collecting constraints on them. The typing derivation for $\lambda s.\, \lambda d.\, \%((\lambda x.\, \%(\%x + \%s)) \mathbin{@} \%d)$ is shown as follows. In this derivation, unification of binding-time variables are already resolved; alternatively, we can express unification as two symmetric inequality constraints.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{
              \dfrac{
                \dfrac{}{\Gamma \vdash x^{b_4} : \mathsf{int}^{b_7}}\;(\mathsf{TVar}_3)
              }{\Gamma' \vdash \%^{b_6} x^{b_4} : \mathsf{int}^{b_6}}\;(\mathsf{TCsp}'_3)
              \quad
              \dfrac{
                \dfrac{}{\Gamma \vdash s^{b_1} : \mathsf{int}^{b_8}}\;(\mathsf{TVar}_3)
              }{\Gamma' \vdash \%^{b_6} s^{b_1} : \mathsf{int}^{b_6}}\;(\mathsf{TCsp}'_3)
            }{\Gamma' \vdash \%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1} : \mathsf{int}^{b_6}}\;(\mathsf{TSum}_3)
          }{\Gamma, x^{b_4} : \mathsf{int}^{b_7} \vdash \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1}) : \mathsf{int}^{b_5}}\;(\mathsf{TCsp}'_3)
        }{\Gamma \vdash \lambda^{b_4} x.\, \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1}) : \mathsf{int}^{b_7} \to^{b_4} \mathsf{int}^{b_5}}\;(\mathsf{TLam}_3)
        \quad
        \dfrac{
          \dfrac{}{\Gamma \vdash d^{b_2} : \mathsf{int}^{b_9}}\;(\mathsf{TVar}_3)
        }{\Gamma \vdash \%^{b_7} d^{b_2} : \mathsf{int}^{b_7}}\;(\mathsf{TCsp}'_3)
      }{\Gamma \vdash (\lambda^{b_4} x.\, \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1})) \mathbin{@}^{b_4} \%^{b_7} d^{b_2} : \mathsf{int}^{b_5}}\;(\mathsf{TApp}_3)
    }{s^{b_1} : \mathsf{int}^{b_8}, d^{b_2} : \mathsf{int}^{b_9} \vdash \%^{b_3}((\lambda^{b_4} x.\, \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1})) \mathbin{@}^{b_4} \%^{b_7} d^{b_2}) : \mathsf{int}^{b_3}}\;(\mathsf{TCsp}'_3)
  }{s^{b_1} : \mathsf{int}^{b_8} \vdash \lambda^{b_2} d.\, \%^{b_3}((\lambda^{b_4} x.\, \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1})) \mathbin{@}^{b_4} \%^{b_7} d^{b_2}) : \mathsf{int}^{b_9} \to^{b_2} \mathsf{int}^{b_3}}\;(\mathsf{TLam}_3)
}{\vdash \lambda^{b_1} s.\, \lambda^{b_2} d.\, \%^{b_3}((\lambda^{b_4} x.\, \%^{b_5}(\%^{b_6} x^{b_4} +^{b_6} \%^{b_6} s^{b_1})) \mathbin{@}^{b_4} \%^{b_7} d^{b_2}) : \mathsf{int}^{b_8} \to^{b_1} \mathsf{int}^{b_9} \to^{b_2} \mathsf{int}^{b_3}}\;(\mathsf{TLam}_3)
$$

where $\Gamma = s^{b_1} : \mathsf{int}^{b_8}, d^{b_2} : \mathsf{int}^{b_9}$ and $\Gamma' = \Gamma, x^{b_4} : \mathsf{int}^{b_7}$. The collected constraints are as follows:

$$
\begin{array}{llllllll}
b_1 \le b_2 & b_1 \le b_8 & b_2 \le b_3 & b_2 \le b_9 & b_4 \le b_5 & b_4 \le b_7 & \text{for } (\mathsf{TLam}_3) \\
b_5 \le b_3 & b_6 \le b_5 & b_7 \le b_6 & b_8 \le b_6 & b_9 \le b_7 & & \text{for } (\mathsf{TCsp}'_3)
\end{array}
$$

After collecting constraints on binding-time variables, we assign concrete binding times to all the binding-time variables that satisfy the constraints and *the initial constraint* given by the user. The initial binding-time constraint specifies variables that are available only at stage 1. For example, if we want to obtain the binding times where the variable s is known early at stage 0, but d is known only at stage 1, we specify $b_2 = 1$ as the initial constraint. Without the initial constraint, we can always obtain the best solution trivially: i.e., $b_i = 0$ for all i.

There is an efficient algorithm to obtain the best solution of the constraints [5]. We first set $b_i = 0$ for all i. We then raise the value of a binding-time variable to 1 if it is specified as 1 in the initial constraint. Whenever the value of a binding-time variable b_i is raised to 1, we search for constraints of the form $b_i \le b_j$ for some j. When such j is found, b_j is also raised to 1 (if it is not raised to 1 yet). We continue this process until values of binding-time variables are no longer raised and become stable.

For example, from the initial constraint $b_2 = 1$, and the constraints $b_2 \le b_3$ and $b_2 \le b_9$, we set $b_3 = b_9 = 1$. From $b_9 = 1$ and $b_9 \le b_7$, we need to have $b_7 = 1$. From $b_7 \le b_6$ and $b_6 \le b_5$, we further have $b_6 = b_5 = 1$. We also have $b_5 \le b_3$, but since b_3 is already set to 1, we are done. The obtained solution is:

$$b_1 = b_4 = b_8 = 0 \quad b_2 = b_3 = b_5 = b_6 = b_7 = b_9 = 1$$

Among the constraints for TCsp'_3, we find that the inequality holds only for $b_8 < b_6$. Thus, we keep $\%$ for s only. The resulting 2-level term becomes $\lambda^0 s.\, \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \mathbin{@}^0 d^1$. We see that β-reduction can be performed at stage 0, but addition is deferred to stage 1. Observe that although x is at stage 0 ($b_4 = 0$), it holds a stage 1 value ($b_7 = 1$).

4.3 The Best Binding Time

Let e_0 be a term in the simply-typed λ-calculus, and t_1 be a type in the 2-level λ-calculus. Then a *solution* for these data is a term e_1 in the 2-level λ-calculus such that (1) e_0 and e_1 are the same except for the binding times and insertion of %, and (2) $\vdash e_1 : t_1$ is derivable.

Definition 1 (The best binding time)**.** *Let e_0 and t_1 be the same as above, and e_1 and e_2 be solutions for these data. We define that e_1 has a better binding time than e_2 does if, for all the subexpressions of e_1 other than $\%e$, the binding time of its type is equal to or smaller than the corresponding binding time of e_2. If e_1 has a better binding time than any other solutions for e_0 and t_1, e_1 has the best binding time for e_0 and t_1.*

It is easy to show the result of binding-time analysis produces the best binding time.

Theorem 2. *The result of the binding-time analysis presented in the previous section produces the best binding time that is consistent with the initial constraint, if there is any solutions consistent with the initial constraint.*

Proof sketch. We show that the result of the binding-time analysis e_1 is better than any other term e_2 with different binding times for the same term. Suppose that there is a subexpression where the binding time b of its type is 1 in e_1, but 0 in e_2. Since the binding-time analysis raises the value of a binding-time variable b only when (1) it is set to 1 in the initial constraint, or (2) there is a path of constraints that transitively requires that b must be 1. Setting $b = 0$ violates one of these two cases. Thus, e_2 cannot be a valid 2-level λ-calculus term. □

5 The Best Staging Annotation

In the previous section, we have seen that we can obtain the best binding time via the binding-time analysis. In this section, we transform the result of the binding-time analysis into a staging annotation. In the previous work [1], we have established the relationship between the 2-level λ-calculus and the staged λ-calculus. Here, we review it and extend it to support CSP. Previously, CSP of expressions were not supported and CSP of variables were supported only in an ad-hoc way.

5.1 Relating Binding Times and Staging Annotations

The key difference between the two calculi is that the 2-level λ-calculus maintains stages as the binding times of types (and terms) while the staged λ-calculus maintains a stage in a judgement, which is close to the binding times of terms. To relate these two calculi, we introduce an intermediate calculus called the staged 2-level λ-calculus that has both features; see Figure 4.

The types and terms of the staged 2-level λ-calculus are based on those of the 2-level λ-calculus: every node has its own binding time. In addition, we have added a code type and staged expressions to indicate where we should place these staging annotations. The typing rules are also similar to the corresponding rules in the 2-level λ-calculus. The first five rules are obtained simply by attaching a stage b to judgements. The stage b roughly corresponds to the binding time of terms. In fact, they coincide in the conclusion of the first five rules. However, they deviate in the premises of these rules (except for (TSum_4)). For example, in the first premise of (TApp_4), the binding time of e_1 is b' rather than b: even though the application is at stage 1, it does not necessarily mean that the function part is a piece of code; e_1 can be

binding times:	b	$:=$	$0 \mid 1$
annotated types:	t^b	$:=$	$\mathsf{int}^b \mid {t_1}^{b_1} \to^b {t_2}^{b_2} \mid \langle t^b \rangle$
annotated terms:	e^b	$:=$	$i^b \mid x^b \mid {e_1}^{b_1} \,@^b\, {e_2}^{b_2} \mid \lambda^b x.\, {e_1}^{b_1} \mid {e_1}^{b_1} +^b {e_2}^{b_2} \mid \langle e^b \rangle \mid\ \sim e^b \mid \%^1 e^0$

typing rules:

$$\frac{}{\Gamma \vdash_b i^b : \mathsf{int}^b}\ (\mathsf{TInt}_4) \qquad \frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash_b x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_4) \qquad \frac{\Gamma \vdash_b {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash_b {e_2}^{b'_2} : {t_2}^{b_2}}{\Gamma \vdash_b {e_1}^{b'} \,@^b\, {e_2}^{b'_2} : {t_1}^{b_1}}\ (\mathsf{TApp}_4)$$

$$\frac{\Gamma, x^b : {t_2}^{b_2} \vdash_b {e_1}^{b'_1} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \le b_1 \quad b \le b_2}{\Gamma \vdash_b \lambda^b x.\, {e_1}^{b'_1} : {t_2}^{b_2} \to^b {t_1}^{b_1}}\ (\mathsf{TLam}_4)$$

$$\frac{\Gamma \vdash_b {e_1}^{b_1} : \mathsf{int}^b \quad \Gamma \vdash_b {e_2}^{b_2} : \mathsf{int}^b}{\Gamma \vdash_b {e_1}^{b_1} +^b {e_2}^{b_2} : \mathsf{int}^b}\ (\mathsf{TSum}_4)$$

$$\frac{\Gamma \vdash_1 e^1 : t^1}{\Gamma \vdash_0 \langle e^1 \rangle : \langle t^1 \rangle}\ (\mathsf{TCod}_4) \qquad \frac{\Gamma \vdash_0 e^0 : \langle t^1 \rangle}{\Gamma \vdash_1 \sim e^0 : t^1}\ (\mathsf{TEsc}_4) \qquad \frac{\Gamma \vdash_0 e^0 : \mathsf{int}^0}{\Gamma \vdash_1 \%^1 e^0 : \mathsf{int}^1}\ (\mathsf{TCsp}_4)$$

Figure 4: Staged 2-level λ-calculus (Well-formed types are defined in Figure 3.)

a function at stage 0 that produces a code value. To adjust the discrepancy between the stage of the judgement and the binding time of the term, (TCod_4) and (TEsc_4) are used.

Given a typing derivation of the 2-level λ-calculus, we can mechanically transform it to the typing derivation of the staged 2-level λ-calculus by attaching stages to judgements. We start from the root of the tree and assign stage 0 to it. We then build a tree using the corresponding typing rules of the staged 2-level λ-calculus. Whenever the stage of the judgement and the binding time of the term differ, (TCod_4) or (TEsc_4) is used. For example, we have the following typing derivation for the example term in Section 4.2 after removing unnecessary %.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash x^0 : \mathsf{int}^1}\ (\mathsf{TVar}_3) \quad \dfrac{\dfrac{}{\Gamma \vdash s^0 : \mathsf{int}^0}\ (\mathsf{TVar}_3)}{\Gamma' \vdash \%^1 s^0 : \mathsf{int}^1}\ (\mathsf{TCsp}_3)}{\Gamma, x^0 : \mathsf{int}^1 \vdash x^0 +^1 \%^1 s^0 : \mathsf{int}^1}\ (\mathsf{TSum}_3)}{\Gamma \vdash \lambda^0 x.\, x^0 +^1 \%^1 s^0 : \mathsf{int}^1 \to^0 \mathsf{int}^1}\ (\mathsf{TLam}_3) \quad \dfrac{}{\Gamma \vdash d^1 : \mathsf{int}^1}\ (\mathsf{TVar}_3)}{s^0 : \mathsf{int}^0, d^1 : \mathsf{int}^1 \vdash (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \,@^0\, d^1 : \mathsf{int}^1}\ (\mathsf{TApp}_3)}{s^0 : \mathsf{int}^0 \vdash \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \,@^0\, d^1 : \mathsf{int}^1 \to^1 \mathsf{int}^1}\ (\mathsf{TLam}_3)}{\vdash \lambda^0 s.\, \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \,@^0\, d^1 : \mathsf{int}^0 \to^0 \mathsf{int}^1 \to^1 \mathsf{int}^1}\ (\mathsf{TLam}_3)$$

To transform this derivation to the one for the staged 2-level λ-calculus, we first assign stage 0 to the root judgement and apply the corresponding rule to it:

$$\dfrac{\dfrac{\vdots}{s^0 : \mathsf{int}^0 \vdash_0 \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \,@^0\, d^1 : \mathsf{int}^1 \to^1 \mathsf{int}^1}}{\vdash_0 \lambda^0 s.\, \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0) \,@^0\, d^1 : \mathsf{int}^0 \to^0 \mathsf{int}^1 \to^1 \mathsf{int}^1}\ (\mathsf{TLam}_4)$$

At this point, however, the binding time of the term is 1 which is different from the stage of the judgement. To adjust the stage, we insert (TCod_4) here:

$$\dfrac{\dfrac{\dfrac{\vdots}{s^0 : \mathsf{int}^0 \vdash_1 \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0)\, @^0\, d^1 : \mathsf{int}^1 \to^1 \mathsf{int}^1}}{s^0 : \mathsf{int}^0 \vdash_0 \langle \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0)\, @^0\, d^1 \rangle : \langle \mathsf{int}^1 \to^1 \mathsf{int}^1 \rangle}\ (\mathsf{TCod}_4)}{\vdash_0 \lambda^0 s.\, \langle \lambda^1 d.\, (\lambda^0 x.\, x^0 +^1 \%^1 s^0)\, @^0\, d^1 \rangle : \mathsf{int}^0 \to^0 \langle \mathsf{int}^1 \to^1 \mathsf{int}^1 \rangle}\ (\mathsf{TLam}_4)$$

Notice that to insert (TCod_4), we also need to insert staging annotations to the term and type: the conclusion of (TCod_4) says that they must be a piece of code. By continuing this process, we obtain staged 2-level λ-calculus term. It can be shown [1] that (TCod_4) and (TEsc_4) are sufficient to recover the harmony between the stage of the judgement and the binding time of the term. The final result for the above example becomes as follows, where $\Gamma = s^0 : \mathsf{int}^0, d^1 : \mathsf{int}^1$ and $\Gamma' = \Gamma, x^0 : \langle \mathsf{int}^1 \rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash_0 x^0 : \langle \mathsf{int}^1 \rangle}\ (\mathsf{TVar}_4)}{\Gamma' \vdash_1 \sim x^0 : \mathsf{int}^1}\ \boxed{(\mathsf{TEsc}_4)} \quad \dfrac{\dfrac{}{\Gamma' \vdash_0 s^0 : \mathsf{int}^0}\ (\mathsf{TVar}_4)}{\Gamma' \vdash_1 \%^1 s^0 : \mathsf{int}^1}\ (\mathsf{TCsp}_4)}{\Gamma' \vdash_1 \sim x^0 +^1 \%^1 s^0 : \mathsf{int}^1}\ (\mathsf{TSum}_4)}{\Gamma, x^0 : \langle \mathsf{int}^1 \rangle \vdash_0 \langle \sim x^0 +^1 \%^1 s^0 \rangle : \langle \mathsf{int}^1 \rangle}\ \boxed{(\mathsf{TCod}_4)}}{\Gamma \vdash_0 \lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle : \langle \mathsf{int}^1 \rangle \to^0 \langle \mathsf{int}^1 \rangle}\ (\mathsf{TLam}_4) \quad \dfrac{\dfrac{}{\Gamma \vdash_1 d^1 : \mathsf{int}^1}\ (\mathsf{TVar}_4)}{\Gamma \vdash_0 \langle d^1 \rangle : \langle \mathsf{int}^1 \rangle}\ \boxed{(\mathsf{TCod}_4)}}{s^0 : \mathsf{int}^0, d^1 : \mathsf{int}^1 \vdash_0 (\lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle)\, @^0\, \langle d^1 \rangle : \langle \mathsf{int}^1 \rangle}\ (\mathsf{TApp}_4)}{s^0 : \mathsf{int}^0, d^1 : \mathsf{int}^1 \vdash_1 \sim ((\lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle)\, @^0\, \langle d^1 \rangle) : \mathsf{int}^1}\ \boxed{(\mathsf{TEsc}_4)}}{s^0 : \mathsf{int}^0 \vdash_1 \lambda^1 d.\sim ((\lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle)\, @^0\, \langle d^1 \rangle) : \mathsf{int}^1 \to^1 \mathsf{int}^1}\ (\mathsf{TLam}_4)}{s^0 : \mathsf{int}^0 \vdash_0 \langle \lambda^1 d.\sim ((\lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle)\, @^0\, \langle d^1 \rangle) \rangle : \langle \mathsf{int}^1 \to^1 \mathsf{int}^1 \rangle}\ \boxed{(\mathsf{TCod}_4)}}{\vdash_0 \lambda^0 s.\, \langle \lambda^1 d.\sim ((\lambda^0 x.\, \langle \sim x^0 +^1 \%^1 s^0 \rangle)\, @^0\, \langle d^1 \rangle) \rangle : \mathsf{int}^0 \to^0 \langle \mathsf{int}^1 \to^1 \mathsf{int}^1 \rangle}\ (\mathsf{TLam}_4)$$

Once we obtain a typing derivation for a term in the staged 2-level λ-calculus, we can obtain the final term in the staged λ-calculus by simply removing all the binding times. For the above example, the final result becomes $\lambda s.\, \langle \lambda d. \sim ((\lambda x.\, \langle \sim x + \%s \rangle)\, @\, \langle d \rangle) \rangle$.

5.2 The Best Staging Annotation

It is now straightforward to formally define the best staging annotation.

Let e_0 be a term in the simply-typed λ-calculus, and t_1 be a type in the staged λ-calculus.

Definition 3 (The best staging annotation)**.** *A term e_1 in the staged λ-calculus has the best staging annotation for e_0 and t_1, if the 2-level λ-calculus term corresponding to e_1 has the best binding time for the data e_0 and t_2 where t_2 is the type in the 2-level λ-calculus which corresponds to t_1.*

Note that since there is a one-to-one correspondence between the staged 2-level λ-calculus and the staged λ-calculus, we can always recover the binding times of e_1 and its type in the staged 2-level λ-calculus, from which t_2 is obtained by removing staging annotations.

Theorem 4. *For any well-typed term e in the simply-typed λ-calculus and a type t in the staged λ-calculus, there exists a term in the staged λ-calculus that has the best staging annotation for e and t, if there is any solution for e and t.*

Proof. For any well-typed term e, we can perform the binding-time analysis to obtain its best binding time (by **Theorem 2**). By removing binding times from the result, we obtain a term in the staged λ-calculus that has the best staging annotation. $\square$

This theorem is obvious, once we set up the necessary definition for the best staging annotation and its relationship to binding times. The simple but important idea presented in this paper is that we can define the best staging annotation in terms of binding times.

6 Related Work

Sheard and Linger [12] showed a search-based binding-time analysis for MetaML. Rather than selecting the best staging annotation, they search for all the possible staging annotations with various heuristics to obtain 'good' staging annotations earlier. Later, Linger and Sheard [9] presented a constraint-based binding-time analysis for MetaML. However, they only show the constraints to be satisfied and do not provide the definition for the best staging annotation. Shimizu [13] extended the latter work to support CSP and showed an algorithm to obtain a possible solution. However, it is not guaranteed to be the best staging annotation, partly because he did not define the best staging annotation and thus there are multiple staging annotations that are equally good. In our previous work [1], we related the staged λ-calculus and the 2-level λ-calculus and showed that the result of binding-time analysis can be transformed to staging annotations. In this paper, we have extended the framework to handle CSP. Also, we have formally defined the best staging annotation and proved the existence of the best staging annotation if there are any annotations.

7 Conclusion

In this paper, we have defined the best staging annotation in terms of the binding times of terms and types. We then showed an algorithm to obtain the best staging annotation in the presence of CSP of base-type values. We also discussed that in the presence of CSP of higher-type values, there is no best staging annotation in general. In the future, we plan to extend the framework to multiple stages. We also plan to support more realistic language constructs and ultimately to implement the binding-time analysis in MetaOCaml.

Acknowledgements. We thank the anonymous reviewers for constructive comments. This work is partly supported by JSPS Grant-in-Aid for Scientific Research (B) 25280020.

References

[1] Kenichi Asai. Toward introducing binding-time analysis to MetaOCaml. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA*, pages 97–102, 2016.

[2] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA*, pages 105–116, 2013.

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

[4] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Trans. Program. Lang. Syst.*, 14(2):147–172, April 1992.

[5] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA, USA*, pages 448–472, 1991.

[6] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.

[7] Oleg Kiselyov. The design and implementation of BER metaocaml - system description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan*, pages 86–102, 2014.

[8] Oleg Kiselyov and Walid Taha. Relating FFTW and split-radix. In *Embedded Software and Systems, First International Conference, ICESS 2004, Hangzhou, China*, pages 488–493, 2004.

[9] Nathan Linger and Tim Sheard. Binding-time analysis for metaml via type inference and constraint solving. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Barcelona, Spain*, pages 266–279, 2004.

[10] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA*, pages 125–134, 2013.

[11] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA*, pages 519–530, 2013.

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

[13] Haruki Shimizu. Automatic generation of optimal code generators in staged languages (in japanese). Master's thesis, Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, 2014.

[14] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, Revised Papers*, pages 30–50, 2003.

[15] Naoki Takashima, Hiroki Sakamoto, and Yukiyoshi Kameyama. Generate and offshore: type-safe and modular code generation for low-level optimization. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing, FHPC@ICFP 2015, Vancouver, BC, Canada*, pages 45–53, 2015.