Toward Introducing Binding-Time Analysis to MetaOCaml

Kenichi Asai *
Ochanomizu University, Japan
asai@is.ocha.ac.jp

Abstract

This paper relates 2-level λ-calculus and staged λ-calculus (restricted to 2 stages) to obtain monovariant binding-time analysis for λ-calculus that produces the output in the form of staging annotations. The relationship between the two λ-calculi provides us with a precise and easy instruction on how to implement binding-time analysis to be incorporated in the staged λ-calculus. It forms a basis for introducing binding-time analysis to full-fledged staged languages such as MetaOCaml.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Partial evaluation; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems

General Terms Languages

Keywords Partial evaluation, binding-time analysis, staging, 2-level lambda-calculus, MetaOCaml

1. Introduction

Although partial evaluation [5] has been studied for a long time, it is still difficult to use it in everyday programming and its application has been limited. Two of the main reasons would be:

- it is said to be difficult for non-experts to control the behavior of partial evaluation, and
- it is difficult to use the result of partial evaluation in the current environment, because partial evaluation is a source-to-source transformation.

To resolve these problems, staged programming is introduced. See [9] for nice introduction and reference therein. By manually adding staging annotations, one can precisely control how code is generated. One can even run the resulting code in the current environment, achieving runtime code generation. As the staged programming becomes popular, partly due to its ease of use, a full-fledged language like MetaOCaml is maintained [6] and widely used.

However, staged programming does not necessarily inherit all the benefit of partial evaluation. While offline partial evaluation typically incorporates binding-time analysis to stage programs automatically, in staged programming, one always has to stage programs manually. Although sophisticated code generation would require careful manual annotations, simple code generation would not require manual intervention but automatic means would suffice. Furthermore, there are situations where one cannot stage manually: when one wants to stage a program that is generated at runtime [1].

The goal of our project is to introduce binding-time analysis to MetaOCaml, so that simple staging can be done automatically. Toward this goal, this paper makes the following contributions:

- We relate (in Section 5) the 2-level λ-calculus (Section 2) and the staged λ-calculus (restricted to 2 stages, Section 3) via an intermediate staged 2-level λ-calculus (Section 4). It enables us to obtain the result of binding-time analysis as a staged term. In contrast to the previous work [7, 8], it does not require backtracking or complex constraint solving.
- As a proof of concept, we have implemented (in Section 6) the binding-time analysis (for λ-calculus terms) in MetaOCaml with lifting (Section 7). An example execution is found in Section 8. It shows that incorporating (monovariant) binding-time analysis to MetaOCaml is not very difficult.

The proofs not shown in the paper are available in the auxiliary material (Section A).

2. 2-level λ-calculus

The input language we consider is a simply-typed λ-calculus with integers as shown in Figure 1. We assume that input terms are already type-checked according to the standard typing rules, also shown in Figure 1.

Given a raw term in the simply-typed λ-calculus, binding-time analysis transforms it into a term in the 2-level λ-calculus [3]. We use the multi-level formulation [2] restricted to 2 levels. See Figure 2. In the 2-level λ-calculus, types and terms are annotated[1] with *binding times*. Since we consider only two stages in this paper, the binding times are either 0 (static) or 1 (dynamic). Binding times are attached to all the sub parts of types and terms. For example, ${t_1}^{b_1} \to^{b} {t_2}^{b_2}$ represents a function type at stage b, whose argument type is t_1 at stage b_1 and result type is t_2 at stage b_2. We say b is the *toplevel* binding time of ${t_1}^{b_1} \to^{b} {t_2}^{b_2}$ or simply the binding time of ${t_1}^{b_1} \to^{b} {t_2}^{b_2}$. When the toplevel binding time is 0, we say the type is static, and otherwise dynamic. Not all the combinations of b, b_1, and b_2 are meaningful. We require that if the function type itself is dynamic, its argument and result types be also dynamic. This requirement is enforced by the well-formedness condition on types shown in Figure 2.

Binding times are attached to terms in a similar way. We call each b in the production rule for annotated terms in Figure 2 the

* This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 25280020.

PEPM '16, January 18-19, 2016, St. Petersburg, FL, USA.

http://dx.doi.org/10.1145/2847538.2847547

[1] In this paper, we use the verb 'annotate' for adding binding times to terms or types, not for adding staging annotations ($\langle . \rangle$ and ˜).

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

$$\frac{}{\Gamma \vdash i : \mathsf{int}}\ (\mathsf{TInt}_1) \qquad \frac{\Gamma, x : t_2 \vdash e_1 : t_1}{\Gamma \vdash \lambda x.\, e_1 : t_2 \to t_1}\ (\mathsf{TLam}_1)$$

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

Figure 1. Simply-typed λ-calculus

(toplevel) binding time of the corresponding term. When it is 0, the term is static, otherwise dynamic. Typing rules for annotated terms are also shown in Figure 2. Static and dynamic integers have static and dynamic integer types, respectively. We will discuss lifting of base-type values in Section 7.

A variable is static (dynamic) when it is bound by a static (dynamic, resp.) abstraction. To maintain when a variable is bound, we annotate each variable in the type environment with its binding time. The binding time of a variable indicates whether the variable can be dereferenced at specialization time: if it is dynamic, the variable is a piece of code carrying no value; if it is static, the variable carries either a static or dynamic value. The *type* of a variable has the information on the binding time of the carried value. When a variable is dynamic, its type is always dynamic. When a variable is static, its type can be either static or dynamic. We will later see that whenever Γ is well-formed (to be defined soon), we have $b \leq b_1$ in TVar_2. It is important to observe that the binding time of a variable and the binding time of its type can be different. The same holds for applications. For integers and abstractions, they are always the same.

A static abstraction can receive either a static value or a dynamic value and return either a static value or a dynamic value. The premise ${t_2}^{b_2}\ \mathsf{wft}$ assures that the newly-introduced binding times of the argument type (which can be a function type) are properly maintained. A dynamic abstraction, on the other hand, receives a dynamic value and returns a dynamic value. This constraint is expressed as the two premises, $b \leq b_1$ and $b \leq b_2$.

Finally, an application is static (dynamic) when the *type of the function part* is static (dynamic, resp.). In other words, the application is static when it can be reduced. Note that the binding time of an application b and the binding time of the result of the application b_1 are different. Even if the whole application ${e_1}^{b'} \mathbin{@}^b {e_2}^{b'_2}$ can be reduced ($b = 0$), the result of the application (of type ${t_1}^{b_1}$) can be dynamic ($b_1 = 1$), if the function part returns a dynamic value. Likewise, even if the type ${t_2}^{b_2} \to^b {t_1}^{b_1}$ of the function part ${e_1}^{b'}$ is dynamic ($b = 1$), it does not mean that ${e_1}^{b'}$ is dynamic, because e_1 can be a static application that returns a dynamic value. The distinction between the binding time of a term and the binding time of its type is the important step toward bridging the gap between the 2-level λ-calculus and the staged λ-calculus to be introduced in the next section.

In the 2-level λ-calculus, we maintain the following well-formed condition for type environments.

DEFINITION 1. *A type environment Γ is well-formed if, for all x^b in the domain of Γ, if $\Gamma(x^b) = t^{b'}$, then $b \leq b'$ and $t^{b'}$ wft.*

We can easily show that types that appear in a derivation of the 2-level λ-calculus are always well formed.

PROPOSITION 2. *Assume that Γ is a well-formed environment. If $\Gamma \vdash e^{b'} : t^b$, then t^b wft.*

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: $e^b ::= i^b \mid x^b \mid \lambda^b x.\, {e_1}^{b_1} \mid {e_1}^{b_1} \mathbin{@}^b {e_2}^{b_2}$
well-formed types:

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

typing rules:

$$\frac{}{\Gamma \vdash i^b : \mathsf{int}^b}\ (\mathsf{TInt}_2) \qquad \frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_2)$$

$$\frac{\Gamma, x^b : {t_2}^{b_2} \vdash {e_1}^{b'_1} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \leq b_1 \quad b \leq b_2}{\Gamma \vdash \lambda^b x.\, {e_1}^{b'_1} : {t_2}^{b_2} \to^b {t_1}^{b_1}}\ (\mathsf{TLam}_2)$$

$$\frac{\Gamma \vdash {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash {e_2}^{b'_2} : {t_2}^{b_2}}{\Gamma \vdash {e_1}^{b'} \mathbin{@}^b {e_2}^{b'_2} : {t_1}^{b_1}}\ (\mathsf{TApp}_2)$$

Figure 2. 2-level λ-calculus (0: static, 1: dynamic)

Given a well-typed term (with its typing derivation) in the simply-typed λ-calculus, the task of binding-time analysis is to transform it into a 2-level λ-calculus term by annotating the term and its type with binding times. Although the typing rules for the 2-level λ-calculus are not strictly syntax directed because we have effectively two rules (for the binding times, 0 and 1) for each construct, annotating binding times to a term and its type is actually easy. There is an efficient such algorithm [4], which can be further simplified in case the input term is already well typed. Since we want to obtain a term as static as possible, we first assume that all the binding times are 0. The typing derivation for the simply-typed λ-calculus then becomes a valid typing derivation for the 2-level λ-calculus, because all the well-formedness conditions are trivially satisfied if all the binding times are 0. When we raise a certain binding time to 1 (because of the initial binding-time division given by the user), we propagate this information through binding-time constraints: whenever a binding time b is raised to 1 and we have a binding-time constraint $b \leq b_1$, we raise b_1 to 1. We continue this process until all the binding-time constraints are satisfied. To realize this process, we maintain for each binding time a list of binding times that have to be raised. The process is efficient. The number of binding times is proportional to the size of types appearing in judgements and type environments (since binding times of terms can be easily inferred from them). Because each binding time is raised at most once, the process will finish when all the binding times are raised in the worst case. We do not need any backtracking or complex constraint solving.

3. Staged λ-calculus

Given an annotated term in the 2-level λ-calculus, we want to execute it in a staged language such as MetaOCaml. However, the theory MetaOCaml is based on is not the 2-level λ-calculus but a *staged* λ-calculus. To relate the two λ-calculi, we review in this section the staged λ-calculus as presented in [8]. See Figure 3.

Terms in the staged λ-calculus consist of terms in the simply-typed λ-calculus extended with code $\langle e \rangle$ and escape $\tilde{}e$. The code $\langle e \rangle$ constructs a code value that represents e, without evaluating e. Within $\langle e \rangle$, however, one can write $\tilde{}e$, which is evaluated to a code value and spliced in to the surrounding code where $\tilde{}e$ was originally placed. In the 2-level λ-calculus terminology, $\langle e \rangle$ is a dynamic term and $\tilde{}e$ is a static term that produces a dynamic term.

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

$$\frac{}{\Gamma \vdash_b i : \mathsf{int}}\ (\mathsf{TInt}_3) \qquad \frac{\Gamma(x^b) = t}{\Gamma \vdash_b x : t}\ (\mathsf{TVar}_3)$$

$$\frac{\Gamma, x^b : t_2 \vdash_b e_1 : t_1}{\Gamma \vdash_b \lambda x.\, e_1 : t_2 \to t_1}\ (\mathsf{TLam}_3)$$

$$\frac{\Gamma \vdash_b e_1 : t_2 \to t_1 \quad \Gamma \vdash_b e_2 : t_2}{\Gamma \vdash_b e_1 \mathbin{@} e_2 : t_1}\ (\mathsf{TApp}_3)$$

$$\frac{\Gamma \vdash_1 e_1 : t_1}{\Gamma \vdash_0 \langle e_1 \rangle : \langle t_1 \rangle}\ (\mathsf{TCod}_3) \qquad \frac{\Gamma \vdash_0 e_1 : \langle t_1 \rangle}{\Gamma \vdash_1 \tilde{}\, e_1 : t_1}\ (\mathsf{TEsc}_3)$$

Figure 3. Staged λ-calculus

Since the staged λ-calculus handles code as a first-class object, it has a type for code objects: $\langle t \rangle$ represents a type of code of type t. That is, when e has type t, $\langle e \rangle$ has type $\langle t \rangle$.

The typing rules in the staged λ-calculus have the form $\Gamma \vdash_b e : t$, which reads: under a type environment Γ, a term e has type t at stage b. The stage b in the judgement represents the context in which the term e is placed: $b = 0$ means that e appears in the static context (intuitively outside $\langle\cdot\rangle$ or within escape), while $b = 1$ means that e appears in the dynamic context (within $\langle\cdot\rangle$ without being escaped).

The first four typing rules in Figure 3 are exactly the same as those for the simply-typed λ-calculus in Figure 1: the stage b is simply placed uniformly to all the judgements. It means that these four rules apply uniformly for all stages. The stage changes when the last two typing rules are used. The first rule TCod_3 states that to check $\langle e_1 \rangle$ has type $\langle t_1 \rangle$ at stage 0, we have to check that e_1 has type t_1 at stage 1. Using the stage, we remember whether e_1 is in $\langle\cdot\rangle$ or not. When in $\langle\cdot\rangle$ (i.e., at stage 1), we can splice in a static term e_1 using TEsc_3. The spliced term e_1 is type-checked at stage 0 and it must have a code type $\langle t_1 \rangle$ for some t_1 so that it can be spliced in to the surrounding code.

In the 2-level λ-calculus, the binding time of a term and the binding time of its type can be different. In the staged λ-calculus, only a stage b is maintained, which roughly corresponds to the binding time of a term. The information on the binding time of a type is recovered by the two additional rules, TCod_3 and TEsc_3.

4. Staged 2-level λ-calculus

To relate the 2-level λ-calculus and the staged λ-calculus, we introduce in this section an intermediate λ-calculus, called *staged 2-level λ-calculus*, that bridges the gap between the two. It is similar to the 2-level λ-calculus, but keeps track of the stage b of each judgement. The typing rules of the staged 2-level λ-calculus are shown in Figure 4.

Since the stage roughly corresponds to the binding time of a term, the first four rules are obtained from the rules for the 2-level λ-calculus (Figure 2) by simply adding the binding time as a stage to all the judgements: for each $\Gamma \vdash e^b : t^{b'}$ in Figure 2, we have $\Gamma \vdash_b e^b : t^{b'}$ in Figure 4. This is not enough, however, because in the premises of these rules, binding times of terms are different from b. To adjust them, the staged 2-level λ-calculus includes two additional rules, TCod_4 and TEsc_4, that change the stage without changing the term and its type.

It is easy to see that we can always transform a derivation for the staged 2-level λ-calculus back into the one for the 2-level λ-calculus. By simply removing the stage and collapsing TCod_4 and TEsc_4, we obtain a derivation for the 2-level λ-calculus.

typing rules:

$$\frac{}{\Gamma \vdash_b i^b : \mathsf{int}^b}\ (\mathsf{TInt}_4) \qquad \frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash_b x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_4)$$

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

$$\frac{\Gamma \vdash_b {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash_b {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash_b {e_1}^{b'} \mathbin{@}^b {e_2}^{b_2'} : {t_1}^{b_1}}\ (\mathsf{TApp}_4)$$

$$\frac{\Gamma \vdash_1 e^1 : t^1}{\Gamma \vdash_0 e^1 : t^1}\ (\mathsf{TCod}_4) \qquad \frac{\Gamma \vdash_0 e^0 : t^1}{\Gamma \vdash_1 e^0 : t^1}\ (\mathsf{TEsc}_4)$$

Figure 4. Staged 2-level λ-calculus (Definitions for binding times, annotated types and terms, and well-formed types are the same as Figure 2.)

In the next theorem, we will see that the converse is also true: given a derivation for the 2-level λ-calculus, we can construct a derivation for the staged 2-level λ-calculus. Although TCod_4 is applicable only for dynamic terms and dynamic types and TEsc_4 is applicable only for static terms and dynamic types, the theorem states that they are enough to recover the derivation of the 2-level λ-calculus.

THEOREM 3. *Assume that Γ is a well-formed environment. If we have $\Gamma \vdash e^{b'} : t^b$ in the 2-level λ-calculus, then for all b'' such that $b'' \le b$, we have $\Gamma \vdash_{b''} e^{b'} : t^b$ in the staged 2-level λ-calculus.*

Because the proof of this theorem forms an algorithm to convert a 2-level λ-calculus term into a staged 2-level λ-calculus term, we show the entire proof below. It is done by induction on the derivation of $\Gamma \vdash e^{b'} : t^b$. For each of the typing rules in Figure 2, we show that the same derivation can be obtained by the corresponding rule in Figure 4, using TCod_4 and TEsc_4 to adjust stages.

Proof (of THEOREM 3). By induction on the derivation of $\Gamma \vdash e^{b'} : t^b$.

(case TInt_2) From assumption, we have:

$$\frac{}{\Gamma \vdash i^b : \mathsf{int}^b}\ (\mathsf{TInt}_2)$$

If $b = 0$, we have:

$$\frac{}{\Gamma \vdash_0 i^0 : \mathsf{int}^0}\ (\mathsf{TInt}_4)$$

If $b = 1$, we have:

$$\frac{}{\Gamma \vdash_1 i^1 : \mathsf{int}^1}\ (\mathsf{TInt}_4) \quad \text{and} \quad \frac{\dfrac{}{\Gamma \vdash_1 i^1 : \mathsf{int}^1}\ (\mathsf{TInt}_4)}{\Gamma \vdash_0 i^1 : \mathsf{int}^1}\ (\mathsf{TCod}_4)$$

(case TVar_2) From assumption, we have:

$$\frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_2)$$

If $b_1 = 0$, we have $b = 0$ since $b \le b_1$ from the well-formedness of Γ. We then have:

$$\frac{\Gamma(x^0) = {t_1}^0}{\Gamma \vdash_0 x^0 : {t_1}^0}\ (\mathsf{TVar}_4)$$

If $b_1 = 1$, we need to consider two cases for b. When $b = 1$, we have:

$$\frac{\Gamma(x^1) = {t_1}^1}{\Gamma \vdash_1 x^1 : {t_1}^1}\ (\mathsf{TVar}_4) \quad \text{and} \quad \frac{\dfrac{\Gamma(x^1) = {t_1}^1}{\Gamma \vdash_1 x^1 : {t_1}^1}\ (\mathsf{TVar}_4)}{\Gamma \vdash_0 x^1 : {t_1}^1}\ (\mathsf{TCod}_4)$$

When $b = 0$, we have

$$\frac{\Gamma(x^0) = {t_1}^1}{\Gamma \vdash_0 x^0 : {t_1}^1}\ (\mathsf{TVar}_4) \quad \text{and} \quad \frac{\dfrac{\Gamma(x^0) = {t_1}^1}{\Gamma \vdash_0 x^0 : {t_1}^1}\ (\mathsf{TVar}_4)}{\Gamma \vdash_1 x^0 : {t_1}^1}\ (\mathsf{TEsc}_4)$$

(case TLam_2) From assumption, we have:

$$\frac{\Gamma, x^b : {t_2}^{b_2} \vdash {e_1}^{b_1'} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \le b_1 \quad b \le b_2}{\Gamma \vdash \lambda^b x.\, {e_1}^{b_1'} : {t_2}^{b_2} \to^b {t_1}^{b_1}}\ (\mathsf{TLam}_2)$$

Because $b \le b_2$ and ${t_2}^{b_2}\ \mathsf{wft}$, we have that $\Gamma, x^b : {t_2}^{b_2}$ is a well-formed environment. From the induction hypothesis, for all b_1'' such that $b_1'' \le b_1$, we have $\Gamma, x^b : {t_2}^{b_2} \vdash_{b_1''} {e_1}^{b_1'} : {t_1}^{b_1}$. If $b = 0$, by setting $b_1'' = 0$, we have $b_1'' \le b_1$, and thus:

$$\frac{\Gamma, x^0 : {t_2}^{b_2} \vdash_0 {e_1}^{b_1'} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad 0 \le b_1 \quad 0 \le b_2}{\Gamma \vdash_0 \lambda^0 x.\, {e_1}^{b_1'} : {t_2}^{b_2} \to^0 {t_1}^{b_1}}\ (\mathsf{TLam}_4)$$

If $b = 1$, we know $b_1 = b_2 = 1$ from $b \le b_1$ and $b \le b_2$. By setting $b_1'' = 1$, we have $b_1'' \le b_1$ (since $b_1 = 1$), and thus:

$$\frac{\Gamma, x^1 : {t_2}^1 \vdash_1 {e_1}^{b_1'} : {t_1}^1 \quad {t_2}^1\ \mathsf{wft} \quad 1 \le 1 \quad 1 \le 1}{\Gamma \vdash_1 \lambda^1 x.\, {e_1}^{b_1'} : {t_2}^1 \to^1 {t_1}^1}\ (\mathsf{TLam}_4)$$

and

$$\frac{\dfrac{\Gamma, x^1 : {t_2}^1 \vdash_1 {e_1}^{b_1'} : {t_1}^1 \quad {t_2}^1\ \mathsf{wft} \quad 1 \le 1 \quad 1 \le 1}{\Gamma \vdash_1 \lambda^1 x.\, {e_1}^{b_1'} : {t_2}^1 \to^1 {t_1}^1}\ (\mathsf{TLam}_4)}{\Gamma \vdash_0 \lambda^1 x.\, {e_1}^{b_1'} : {t_2}^1 \to^1 {t_1}^1}\ (\mathsf{TCod}_4)$$

(case TApp_2) From assumption, we have:

$$\frac{\Gamma \vdash {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash {e_1}^{b'} @^b {e_2}^{b_2'} : {t_1}^{b_1}}\ (\mathsf{TApp}_2)$$

From the induction hypotheses, we have:

- for all b'' such that $b'' \le b$, $\Gamma \vdash_{b''} {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1}$
- for all b_2'' such that $b_2'' \le b_2$, $\Gamma \vdash_{b_2''} {e_2}^{b_2'} : {t_2}^{b_2}$

From PROPOSITION 2, we have $({t_2}^{b_2} \to^b {t_1}^{b_1})\ \mathsf{wft}$, i.e., $b \le b_1$ and $b \le b_2$. If $b_1 = 0$, we must have $b = 0$ (from $b \le b_1$). In this case, by setting $b'' = 0$ and $b_2'' = 0$, we have $b'' \le b$ and $b_2'' \le b_2$, and thus:

$$\frac{\Gamma \vdash_0 {e_1}^{b'} : {t_2}^{b_2} \to^0 {t_1}^0 \quad \Gamma \vdash_0 {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash_0 {e_1}^{b'} @^0 {e_2}^{b_2'} : {t_1}^0}\ (\mathsf{TApp}_4)$$

If $b_1 = 1$, we need to consider two cases for b. When $b = 1$, we have $b_2 = 1$ from $b \le b_2$. In this case, by setting $b'' = 1$ and $b_2'' = 1$, we have $b'' \le b$ and $b_2'' \le b_2$, and thus:

$$\frac{\Gamma \vdash_1 {e_1}^{b'} : {t_2}^1 \to^1 {t_1}^1 \quad \Gamma \vdash_1 {e_2}^{b_2'} : {t_2}^1}{\Gamma \vdash_1 {e_1}^{b'} @^1 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TApp}_4)$$

and

$$\frac{\dfrac{\Gamma \vdash_1 {e_1}^{b'} : {t_2}^1 \to^1 {t_1}^1 \quad \Gamma \vdash_1 {e_2}^{b_2'} : {t_2}^1}{\Gamma \vdash_1 {e_1}^{b'} @^1 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TApp}_4)}{\Gamma \vdash_0 {e_1}^{b'} @^1 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TCod}_4)$$

When $b = 0$, by setting $b'' = 0$ and $b_2'' = 0$, we have $b'' \le b$ and $b_2'' \le b_2$, and thus:

$$\frac{\Gamma \vdash_0 {e_1}^{b'} : {t_2}^{b_2} \to^0 {t_1}^1 \quad \Gamma \vdash_0 {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash_0 {e_1}^{b'} @^0 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TApp}_4)$$

and

$$\frac{\dfrac{\Gamma \vdash_0 {e_1}^{b'} : {t_2}^{b_2} \to^0 {t_1}^1 \quad \Gamma \vdash_0 {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash_0 {e_1}^{b'} @^0 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TApp}_4)}{\Gamma \vdash_1 {e_1}^{b'} @^0 {e_2}^{b_2'} : {t_1}^1}\ (\mathsf{TEsc}_4)$$

□

5. Adding Staging Annotations

In the previous section, we saw that a derivation in the 2-level λ-calculus can be converted to a derivation in the staged 2-level λ-calculus. We now convert the latter into a derivation in the staged λ-calculus. In other words, we introduce staging annotations ($\langle\cdot\rangle$ and $\tilde{}$).

We first relate annotated types and staged types. While an annotated type has its own stage information in its binding time, a staged type is context sensitive: int is static if it appears alone but int in $\langle\mathsf{int}\rangle$ is dynamic. To relate the two kinds of types, we define a conversion function $A_b[\cdot]$ from annotated types to staged types at stage b.

$$\begin{array}{rcl} A_b[\mathsf{int}^b] & = & \mathsf{int} \\ A_0[\mathsf{int}^1] & = & \langle\mathsf{int}\rangle \\ A_b[{e_1}^{b_1} \to^b {e_2}^{b_2}] & = & A_b[{e_1}^{b_1}] \to A_b[{e_2}^{b_2}] \\ A_0[{e_1}^1 \to^1 {e_2}^1] & = & \langle A_1[{e_1}^1] \to A_1[{e_2}^1]\rangle \end{array}$$

When the binding time of a type is the same as the stage b, $A_b[\cdot]$ converts the type homomorphically. When the type is more dynamic than the stage, $\langle\cdot\rangle$ is introduced. Note that $A_1[\cdot]$ is not defined for static types; static types appear only at stage 0. We can easily show the following proposition by case analysis on t^1.

PROPOSITION 4. $A_0[t^1] = \langle A_1[t^1]\rangle$

The conversion function is naturally extended to type environments as follows:

$$A[x^{b'} : t^b, \ldots] \quad = \quad x^{b'} : A_{b'}[t^b], \ldots$$

For each variable at stage b', we use $A_{b'}[\cdot]$ to convert its type.

Now, we relate annotated terms and staged terms.

$$\begin{array}{rcl} A_b[i^b] & = & i \\ A_b[x^b] & = & x \\ A_b[\lambda^b x.\, {e_1}^{b_1}] & = & \lambda x.\, A_b[{e_1}^{b_1}] \\ A_b[{e_1}^{b_1} @^b {e_2}^{b_2}] & = & A_b[{e_1}^{b_1}] \mathbin{@} A_b[{e_2}^{b_2}] \end{array}$$

$$\begin{array}{rcl} A_0[e^1] & = & \langle A_1[e^1]\rangle \\ A_1[e^0] & = & \tilde{}A_0[e^0] \end{array}$$

Similarly to the type conversion, $A_b[\cdot]$ simply removes binding times from terms when binding times are equal to the current stage. Otherwise, it introduces either $\langle\cdot\rangle$ or $\tilde{}$.

We can now relate a derivation in the staged 2-level λ-calculus and a derivation in the staged λ-calculus.

THEOREM 5. *If we have $\Gamma \vdash_{b''} e^{b'} : t^b$ in the staged 2-level λ-calculus, then we have $A[\Gamma] \vdash_{b''} A_{b''}[e^{b'}] : A_{b''}[t^b]$ in the staged λ-calculus.*

The proof goes by induction on the derivation of $\Gamma \vdash_{b''} e^{b'} : t^b$. We show that there is one-to-one correspondence between the rules in Figure 4 and the ones in Figure 3.

$$\frac{}{\Gamma \vdash_0 i : 0 \rightsquigarrow i} \qquad \frac{x^0 \in \Gamma}{\Gamma \vdash_0 x : b \rightsquigarrow x} \qquad \frac{\Gamma, x^0 \vdash_0 e_1 : b_1 \rightsquigarrow e_1'}{\Gamma \vdash_0 \lambda x.\, e_1 : 0 \rightsquigarrow \lambda x.\, e_1'} \qquad \frac{\Gamma \vdash_0 e_1 : 0 \rightsquigarrow e_1' \quad \Gamma \vdash_0 e_2 : b_2 \rightsquigarrow e_2'}{\Gamma \vdash_0 e_1 \mathbin{@} e_2 : b \rightsquigarrow e_1' \mathbin{@} e_2'}$$

$$\frac{}{\Gamma \vdash_0 i : 1 \rightsquigarrow \langle i \rangle} \qquad \frac{x^1 \in \Gamma}{\Gamma \vdash_0 x : 1 \rightsquigarrow \langle x \rangle} \qquad \frac{\Gamma, x^1 \vdash_1 e_1 : 1 \rightsquigarrow e_1'}{\Gamma \vdash_0 \lambda x.\, e_1 : 1 \rightsquigarrow \langle \lambda x.\, e_1' \rangle} \qquad \frac{\Gamma \vdash_1 e_1 : 1 \rightsquigarrow e_1' \quad \Gamma \vdash_1 e_2 : 1 \rightsquigarrow e_2'}{\Gamma \vdash_0 e_1 \mathbin{@} e_2 : 1 \rightsquigarrow \langle e_1' \mathbin{@} e_2' \rangle}$$

$$\frac{}{\Gamma \vdash_1 i : 1 \rightsquigarrow i} \qquad \frac{x^1 \in \Gamma}{\Gamma \vdash_1 x : 1 \rightsquigarrow x} \qquad \frac{\Gamma, x^1 \vdash_1 e_1 : 1 \rightsquigarrow e_1'}{\Gamma \vdash_1 \lambda x.\, e_1 : 1 \rightsquigarrow \lambda x.\, e_1'} \qquad \frac{\Gamma \vdash_1 e_1 : 1 \rightsquigarrow e_1' \quad \Gamma \vdash_1 e_2 : 1 \rightsquigarrow e_2'}{\Gamma \vdash_1 e_1 \mathbin{@} e_2 : 1 \rightsquigarrow e_1' \mathbin{@} e_2'}$$

$$\frac{x^0 \in \Gamma}{\Gamma \vdash_1 x : 1 \rightsquigarrow \tilde{} x} \qquad \frac{\Gamma \vdash_0 e_1 : 0 \rightsquigarrow e_1' \quad \Gamma \vdash_0 e_2 : b_2 \rightsquigarrow e_2'}{\Gamma \vdash_1 e_1 \mathbin{@} e_2 : 1 \rightsquigarrow \tilde{} (e_1' \mathbin{@} e_2')}$$

Figure 5. Algorithm for adding staging annotations $\langle \cdot \rangle$ and $\tilde{}$

Proof (of THEOREM 5). By induction on the derivation of $\Gamma \vdash_{b''} e^{b'} : t^b$. The first four cases proceed without surprise. We show the TLam_4 case below. For the last two cases, we show the TEsc_4 case. The TCod_4 case is similar.

(case TLam_4) From assumption, we have:

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

Since $A_b[\lambda^b x.\, {e_1}^{b_1'}] = \lambda x.\, A_b[{e_1}^{b_1'}]$ and $A_b[{t_2}^{b_2} \to^b {t_1}^{b_1}] = A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}]$, we obtain the conclusion using the induction hypothesis by:

$$\frac{A[\Gamma], x^b : A_b[{t_2}^{b_2}] \vdash_b A_b[{e_1}^{b_1'}] : A_b[{t_1}^{b_1}]}{A[\Gamma] \vdash_b \lambda x.\, A_b[{e_1}^{b_1'}] : A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}]}\ (\mathsf{TLam}_3)$$

(case TEsc_4) From assumption, we have:

$$\frac{\Gamma \vdash_0 e^0 : t^1}{\Gamma \vdash_1 e^0 : t^1}\ (\mathsf{TEsc}_4)$$

Since $A_1[e^0] = \tilde{} A_0[e^0]$ from the definition and $A_0[t^1] = \langle A_1[t^1] \rangle$ from PROPOSITION 4, we obtain the conclusion using the induction hypothesis by:

$$\frac{A[\Gamma] \vdash_0 A_0[e^0] : \langle A_1[t^1] \rangle}{A[\Gamma] \vdash_1 \tilde{} A_0[e^0] : A_1[t^1]}\ (\mathsf{TEsc}_3) \qquad \square$$

6. Implementation

By combining the two theorems, we can transform a term in the 2-level λ-calculus into a term in the staged λ-calculus. Since the proof of THEOREM 5 states that there is one-to-one correspondence between the staged 2-level λ-calculus and the staged λ-calculus, the key to the transformation is THEOREM 3: its proof provides us with a transformation algorithm. For example, the case for TApp_2 in the proof of THEOREM 3 examines five different subcases. Three of them use TApp_4 only, but the other two additionally use TCod_4 or TEsc_4. By classifying them according to the stage and the toplevel binding time of their type, we obtain a transformation algorithm shown in Figure 5.

The judgement in Figure 5 has the form $\Gamma \vdash_{b'} e : b \rightsquigarrow e'$ which reads: under type environment Γ (holding stages of variables only) and the current stage b', a 2-level λ-calculus term e (ignoring its binding time) whose type has the toplevel binding time b is translated to a staged λ-calculus term e'. Note that b is the toplevel binding time of the *type* of e, not the toplevel binding time of e itself. Figure 5 is organized as follows. One column is used for each syntactic construct of the 2-level λ-calculus. The top two lines are for stage 0 and the bottom one or two lines are for stage 1.

Figure 5 is used as follows. Given a derivation $\Gamma \vdash e^{b'} : t^b$ in the 2-level λ-calculus, we select a rule $\Gamma \vdash_{b''} e : b \rightsquigarrow e'$ in Figure 5 for each step of the derivation using e and b'', starting with $b'' = 0$. When there are two rules, we use other information to disambiguate. When $e = i$, we use b to select one of the two rules. The same for $e = \lambda x.\, e_1$. For $e = x$, we use the stage of x maintained in Γ. For $e = e_1 \mathbin{@} e_2$, we use the binding time of the type of e_1. The algorithm is somewhat non-standard in that it dispatches according to not only the type of the current term but also the type of its subterm.

We have incorporated the binding-time analysis presented so far in MetaOCaml as follows. We provide a single external function called `stage` of type `('a -> 'b -> 'c) code -> ('a -> ('b -> 'c) code) code`, which turns a given two-argument function[2] into a staged function whose first argument is static and second dynamic. By running the staged function and passing a static argument, we can obtain a specialized function with respect to the static argument. An example execution is found in Section 8.

Since the function `stage` is provided as a standard function, its code argument is type-checked by MetaOCaml before passed to `stage`. Given a well-typed code, `stage` annotates it with binding times (as attributes of internal typed AST introduced from OCaml 4.02) and performs binding-time analysis as outlined in Section 2. It then removes binding times and adds staging annotations according to Figure 5. As a final bit, it checks whether the binding time of the first argument is inferred as static. If it's not, it raises an error. Although the output in the staged λ-calculus is guaranteed to be well typed (because the two theorems produce typing derivation), the first static argument can be classified as dynamic if it appears in the dynamic context (i.e., if it is residualized in the final result). In this case, the output term would have type `('a code -> ('b -> 'c) code) code` rather than `('a -> ('b -> 'c) code) code`.

The implementation is light-weight. Since staging annotations of MetaOCaml are implemented as attributes, we do not have to modify the structure of the internal typed AST. We only need to maintain stages of variables in an environment, use attributes to attach a binding time to each node of the AST, and replace the attributes with staging annotations. The function `stage` is provided as an external library function, like `!.` (run) in MetaOCaml. Recompilation of MetaOCaml itself is not required to incorporate such external library functions [6].

7. Lifting

In addition to $\langle \cdot \rangle$ and $\tilde{}$ (and *run*), staged λ-calculus allows us to use a static variable in a dynamic context, so-called *cross-stage*

[2] We assume that the given function is a one-stage function. That is, `'a`, `'b`, and `'c` do not contain any `code` type.

persistence or CSP:

$$\frac{\Gamma(x^0) = t}{\Gamma \vdash_1 x : t}\ (\mathsf{TVar}'_3)$$

Using CSP, we can support lifting of base-type values by inserting an identity function. We add the following rule for promoting the binding time of integers:

$$\frac{b_2 \leq b_1}{\Gamma \vdash \lambda^0 a.\, a^0 : \mathsf{int}^{b_2} \to^0 \mathsf{int}^{b_1}}\ (\mathsf{TLift}_2)$$

(and similarly for TLift_4 with stage 0). The case for $b_2 < b_1$ (i.e, $b_2 = 0$ and $b_1 = 1$) can be translated to the staged λ-calculus as follows:

$$\frac{\dfrac{\dfrac{}{\Gamma, a^0 : \mathsf{int} \vdash_1 a : \mathsf{int}}\ (\mathsf{TVar}'_3)}{\Gamma, a^0 : \mathsf{int} \vdash_0 \langle a \rangle : \langle \mathsf{int} \rangle}\ (\mathsf{TCod}_3)}{\Gamma \vdash_0 \lambda a.\, \langle a \rangle : \mathsf{int} \to \langle \mathsf{int} \rangle}\ (\mathsf{TLam}_3)$$

Whenever a term e has a base type, we can insert an identity function as in $(\lambda a.\, a)\,@\,e$. If lifting is required, binding-time analysis turns the identity function to a staged one: $\lambda a.\, \langle a \rangle$. Although our current implementation asks programmers to insert the identity function manually (because it does not currently modify the structure of the internal AST for ease of implementation), it should be easy to automatically add it after the initial type checking.

Restricting lifting to base types precludes primitive operations (such as arithmetic operations) from appearing in a dynamic context. To support primitive operations, we assign a single binding time to their types, assuming that primitive operations would permit useful computation only when they are fully applied. For example, `+` is given a type $\mathsf{int}^b \to^b \mathsf{int}^b \to^b \mathsf{int}^b$. Thus, `+` is either completely static or completely dynamic. The latter case effectively works as CSP of primitive operations.

Lifting a static variable of higher type in general requires deeper consideration. Suppose a static variable f is bound to an abstraction $\lambda x.\, e$, as in $(\lambda f.\, e')\,@\,(\lambda x.\, e)$. What should happen if f appears three times in e', once applied to a static value, once applied to a dynamic value, and once residualized in the result? In the standard monomorphic binding-time analysis, $\lambda x.\, e$ is classified as completely dynamic, because it is residualized in the final result. In a monomorphic binding-time analysis with CSP for higher types, it seems we need to classify $\lambda x.\, e$ as a static function that receives a dynamic argument for the first two uses of f and use CSP to residualize it. In other words, we need to keep track of the completely dynamic case for CSP in addition to other standard cases. It appears difficult to achieve it without considering more complex polymorphic binding-time analysis — future work.

8. Example

To stage a program, we apply `stage` to code of a two-argument function whose first and second arguments we want to classify as static and dynamic, respectively. The following example highlights various aspects of the binding-time analysis presented in this paper.[3]

```
# let a = stage
    .<fun s d -> (fun g -> g d) (fun c ->
        (((+) c) ((fun a -> a) (((+) s) 3))))>. ;;
val a : (int -> (int -> int) code) code =
.<fun s -> .<fun d ->
  .~((fun g -> g .< d >.)
     (fun c ->
       .<((+) .~c) .~((fun a -> .< a >.)
                      (((+) s) 3)) >.)) >.>.
```

The outermost abstraction of the result is static whose body is dynamic. Within the dynamic abstraction, the application is done statically. When a dynamic variable `d` is used in a static context, it is surrounded by $\langle\cdot\rangle$. Conversely, when a static variable `c` is used in a dynamic context, it is escaped. Within a static context, the addition of `s` and `3` is done statically, whose result is lifted using the user-supplied identity function.

By running it and applying it to a static argument, we obtain a specialized version.

```
# let b = !. a 2 ;;
val b : (int -> int) code =
.<fun d -> ((+) d) (* CSP a *) Obj.magic 5>.
# !. b 1 ;;
- : int = 6
```

9. Related Work

The only work we are aware of on introducing binding-time analysis to a staged language is done by Sheard and Linger. They formulated binding-time analysis as a search problem for staging annotations that satisfy a given type specification [8]. They later reformulated it as constraint-based type analysis [7]. In contrast to their approaches built on top of the staged λ-calculus, we start from simple binding-time analysis for the 2-level λ-calculus where the 'best' binding times can be easily found, thus avoiding need for searching or complex constraint solving. On the other hand, they consider multiple stages as well as polymorphism and polyvariance in the same framework. We expect that our framework extends naturally to multiple stages, but it needs to be confirmed. As for polymorphism and polyvariance, it is not at all clear if our approach scales to support them; it is an interesting research topic for future work.

References

[1] Asai, K. "Compiling a Reflective Language using MetaOCaml," *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE '14)*, pp. 113–122 (September 2014).

[2] Glück, R., and J. Jørgensen "An Automatic Program Generator for Multi-Level Specialization," *Lisp and Symbolic Computation*, Vol. 10, No. 2, pp. 113–158, Kluwer Academic Publishers (July 1997).

[3] Gomard, C. K. "A Self-Applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 2, pp. 147–172 (April 1992).

[4] Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 448–472 (August 1991).

[5] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[6] Kiselyov, O. "The Design and Implementation of BER MetaOCaml, System Description," In M. Codish, and E. Sumii, editors, *Functional and Logic Programming (LNCS 8475)*, pp. 86–102 (June 2014).

[7] Linger, N., and T. Sheard "Binding-Time Analysis for MetaML via Type Inference and Constraint Solving," In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2988)*, pp. 266–279 (March 2004).

[8] Sheard, T., N. Linger "Search-Based Binding Time Analysis using Type-Directed Pruning," *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM'02)*, pp. 20–31 (September 2002).

[9] Taha, W. "A Gentle Introduction to Multi-stage Programming," In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation (LNCS 3016)*, pp. 30–50 (2004).

[3] In the result, we renamed variable names and adjusted spacing. In OCaml, `s + 3` is implemented more efficiently than `((+) s) 3`. The current implementation supports only the second form of application.

A. Proofs

The appendix presents all the proofs omitted in the main content of the paper.

PROPOSITION 2. *Assume that Γ is a well-formed environment. If $\Gamma \vdash {e^b}' : t^b$, then t^b wft.*

Proof. By induction on the derivation of $\Gamma \vdash {e^b}' : t^b$.

(case TInt_2) From assumption, we have:

$$\frac{}{\Gamma \vdash i^b : \mathsf{int}^b}\ (\mathsf{TInt}_2)$$

We have int^b wft.

(case TVar_2) From assumption, we have:

$$\frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_2)$$

Since Γ is well-formed, $\Gamma(x^b)$ is also well-formed. Thus, we have ${t_1}^{b_1}$ wft.

(case TLam_2) From assumption, we have:

$$\frac{\Gamma, x^b : {t_2}^{b_2} \vdash {e_1}^{b_1'} : {t_1}^{b_1} \quad {t_2}^{b_2}\ \mathsf{wft} \quad b \leq b_1 \quad b \leq b_2}{\Gamma \vdash \lambda^b x.\, {e_1}^{b_1'} : {t_2}^{b_2} \to^b {t_1}^{b_1}}\ (\mathsf{TLam}_2)$$

Since we have $b \leq b_2$ and ${t_2}^{b_2}$ wft, from the induction hypothesis, we also have ${t_1}^{b_1}$ wft. With $b \leq b_2$ and $b \leq b_1$, we have:

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

(case TApp_2) From assumption, we have:

$$\frac{\Gamma \vdash {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash {e_1}^{b'} \mathbin{@^b} {e_2}^{b_2'} : {t_1}^{b_1}}\ (\mathsf{TApp}_2)$$

From the induction hypothesis for the first premise, we have:

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

Thus, we have ${t_1}^{b_1}$ wft. □

PROPOSITION 4. $A_0[t^1] = \langle A_1[t^1] \rangle$

Proof. By case analysis on t^1.

(case $t^1 = \mathbf{int}^1$)

$$A_0[t^1] = \langle \mathsf{int} \rangle = \langle A_1[t^1] \rangle.$$

(case $t^1 = {e_1}^1 \to^1 {e_2}^1$ for some ${e_1}^1$ and ${e_2}^1$)

$$A_0[t^1] = \langle A_1[{e_1}^1] \to^1 A_1[{e_2}^1] \rangle = \langle A_1[t^1] \rangle.$$

□

THEOREM 5. *If we have $\Gamma \vdash_{b''} {e^b}' : t^b$ in the staged 2-level λ-calculus, then we have $A[\Gamma] \vdash_{b''} A_{b''}[{e^b}'] : A_{b''}[t^b]$ in the staged λ-calculus.*

Proof. By induction on the derivation of $\Gamma \vdash_{b''} {e^b}' : t^b$.

(case TInt_4) From assumption, we have:

$$\frac{}{\Gamma \vdash_b i^b : \mathsf{int}^b}\ (\mathsf{TInt}_4)$$

Since $A_b[i^b] = i$ and $A_b[\mathsf{int}^b] = \mathsf{int}$, we obtain the conclusion by:

$$\frac{}{A[\Gamma] \vdash_b i : \mathsf{int}}\ (\mathsf{TInt}_3)$$

(case TVar_4) From assumption, we have:

$$\frac{\Gamma(x^b) = {t_1}^{b_1}}{\Gamma \vdash_b x^b : {t_1}^{b_1}}\ (\mathsf{TVar}_4)$$

Since $A_b[x^b] = x$ and $A[\Gamma](x^b) = A_b[\Gamma(x^b)] = A_b[{t_1}^{b_1}]$, we obtain the conclusion by:

$$\frac{A[\Gamma](x^b) = A_b[{t_1}^{b_1}]}{A[\Gamma] \vdash_b x : A_b[{t_1}^{b_1}]}\ (\mathsf{TVar}_3)$$

(case TLam_4) From assumption, we have:

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

Since $A_b[\lambda^b x.\, {e_1}^{b_1'}] = \lambda x.\, A_b[{e_1}^{b_1'}]$ and $A_b[{t_2}^{b_2} \to^b {t_1}^{b_1}] = A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}]$, we obtain the conclusion using the induction hypothesis by:

$$\frac{A[\Gamma], x^b : A_b[{t_2}^{b_2}] \vdash_b A_b[{e_1}^{b_1'}] : A_b[{t_1}^{b_1}]}{A[\Gamma] \vdash_b \lambda x.\, A_b[{e_1}^{b_1'}] : A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}]}\ (\mathsf{TLam}_3)$$

(case TApp_4) From assumption, we have:

$$\frac{\Gamma \vdash_b {e_1}^{b'} : {t_2}^{b_2} \to^b {t_1}^{b_1} \quad \Gamma \vdash_b {e_2}^{b_2'} : {t_2}^{b_2}}{\Gamma \vdash_b {e_1}^{b'} \mathbin{@^b} {e_2}^{b_2'} : {t_1}^{b_1}}\ (\mathsf{TApp}_4)$$

Since $A_b[{e_1}^{b'} \mathbin{@^b} {e_2}^{b_2'}] = A_b[{e_1}^{b'}] \mathbin{@} A_b[{e_2}^{b_2'}]$ and $A_b[{t_2}^{b_2} \to^b {t_1}^{b_1}] = A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}]$, we obtain the conclusion using the induction hypotheses by:

$$\frac{A[\Gamma] \vdash_b A_b[{e_1}^{b'}] : A_b[{t_2}^{b_2}] \to A_b[{t_1}^{b_1}] \quad A[\Gamma] \vdash_b A_b[{e_2}^{b_2'}] : A_b[{t_2}^{b_2}]}{A[\Gamma] \vdash_b A_b[{e_1}^{b'}] \mathbin{@} A_b[{e_2}^{b_2'}] : A_b[{t_1}^{b_1}]}\ (\mathsf{TApp}_3)$$

(case TCod_4) From assumption, we have:

$$\frac{\Gamma \vdash_1 e^1 : t^1}{\Gamma \vdash_0 e^1 : t^1}\ (\mathsf{TCod}_4)$$

Since $A_0[e^1] = \langle A_1[e^1] \rangle$ from the definition and $A_0[t^1] = \langle A_1[t^1] \rangle$ from PROPOSITION 4, we obtain the conclusion using the induction hypothesis by:

$$\frac{A[\Gamma] \vdash_1 A_1[e^1] : A_1[t^1]}{A[\Gamma] \vdash_0 \langle A_1[e^1] \rangle : \langle A_1[t^1] \rangle}\ (\mathsf{TCod}_3)$$

(case TEsc_4) From assumption, we have:

$$\frac{\Gamma \vdash_0 e^0 : t^1}{\Gamma \vdash_1 e^0 : t^1}\ (\mathsf{TEsc}_4)$$

Since $A_1[e^0] = \tilde{} A_0[e^0]$ from the definition and $A_0[t^1] = \langle A_1[t^1] \rangle$ from PROPOSITION 4, we obtain the conclusion using the induction hypothesis by:

$$\frac{A[\Gamma] \vdash_0 A_0[e^0] : \langle A_1[t^1] \rangle}{A[\Gamma] \vdash_1 \tilde{} A_0[e^0] : A_1[t^1]}\ (\mathsf{TEsc}_3)$$

□