January 1998

An idealized MetaML: simpler, and more expressive. (includes proofs)

Eugenio Moggi
Walid Taha
Zine El-Abidine Benaissa
Tim Sheard

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
http://digitalcommons.ohsu.edu/csetech/110

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
An Idealized MetaML:  
Simpler, and More Expressive*.  
(Includes Proofs)

Eugenio Moggi\textsuperscript{1}, Walid Taha\textsuperscript{2}, Zine El-Abidine Benaissa\textsuperscript{2}, and Tim Sheard\textsuperscript{2}

\textsuperscript{1} DISI, Univ di Genova  
Genova, Italy  
moggi@disi.unige.it

\textsuperscript{2} Oregon Graduate Institute  
Portland, OR, USA  
\{walidt,benaisa,sheard\}@cse.ogi.edu

Abstract. MetaML is a multi-stage functional programming language featuring three constructs that can be viewed as statically-typed refinements of the back-quote, comma, and eval of Scheme. Thus it provides special support for writing code generators and serves as a semantically-sound basis for systems involving multiple interdependent computational stages. In previous work, we reported on an implementation of MetaML, and on a small-step semantics and type-system for MetaML. In this paper, we present An Idealized MetaML (AIM) that is the result of our study of a categorical model for MetaML. An important outstanding problem is finding a type system that provides the user with a means for manipulating both open and closed code. This problem has eluded efforts by us and other researchers for over three years. AIM solves the issue by providing two type constructors, one classifies closed code and the other open code, and exploiting the way they interact.

1 Introduction  

"If thought corrupts language, language can also corrupt thought"\textsuperscript{1}. Staging computation into multiple steps is a well-known optimization technique used in many important algorithms, such as high-level program generation, compiled program execution, and partial evaluation. Yet few typed programming languages allow us to express staging in a natural and concise manner. MetaML was designed to fill this gap. Intuitively, MetaML has a special type for code that combines some
features of both open code, that is, code that can contain free variables, and closed code, that is, code that contains no free variables. In a statically typed setting, open code and closed code have different properties, which we explain in the following section.

Open and Closed Code A number of typed languages for manipulating code fragments have been proposed in the literature. Some have types for open code [9, 6, 3, 12], and others have types for closed code [4, 13]. On one hand, languages with open code types play an important role in the study of partial evaluation. Typically, they provide two constructs, one for building a code fragment with free variables, and one for combining such fragments. Being able to construct open fragments allows the user to force computations “under a lambda”. Generally, it has been hard for such languages to include constructs for executing such code fragments, because they can contain “not-yet-bound identifiers”. On the other hand, languages with closed code types play an important role in the study of run-time (machine) code generation. Typically, they include constructs for building closed code, and for executing them. Generally, in such languages there is no mechanism for forcing computations “under a lambda”.

The importance of having a way to execute code within a language is best illustrated by considering the eval of Scheme. In particular, Efficient implementations of Domain-Specific or “little” languages can be developed as follows: First, build a translator from the source language to Scheme, and then use eval to execute the generated Scheme code. For many languages, such an implementation would be almost as simple as an interpreter implementation (especially if back-quote and comma are utilized), but would incur almost none of the overhead associated with an interpretive implementation.

MetaML [12, 11] provides constructs for manipulating open code and executing it, but does not distinguish between open and closed code types. But open code cannot be executed because it may contain free variables that have not been bound yet. This means that in MetaML type information is not enough to decide whether or not we can safely execute a code fragment. In what follows, we introduce MetaML, explain what it allows us to express, and where it falls short.

MetaML MetaML has three staging annotations: Brackets ⟨⟩, Escape “~” and Run run . An expression ⟨ε⟩ defers the computation of ε; “~ε splices the deferred expression obtained by evaluating ε into the body of a surrounding Bracketed expression; and run ε evaluates ε to obtain a deferred expression, and then evaluates it. Note that “~ε is only legal within lexically enclosing Brackets. Finally, Brackets in types such as <int> are read “Code of int”. To illustrate, consider the following interactive session:

```
- | val rec exp = fn n => fn x =>
  if n=0 then <1> else "x * ~(exp (n-1) x) >;
val exp = fn : int -> <int> -> <int>
```
The function `exp` returns a code fragment representing an exponent, given an integer power `n` and a code fragment representing a base `x`. The function `exponent` is very similar, but takes only a power and returns a code fragment representing a function that takes a base and returns the exponent. The code fragment `cube` is the specialization of `exponent` to the power `3`. Next, we construct the code fragment `program` which is an application of the code of `cube` to the base `2`. Finally, the last declaration executes this code fragment.

Unfortunately, there is a problem with the above example. In particular, the very last declaration is not typable with the basic type system of MetaML [11]. Intuitively, the type system for MetaML must keep track of free variables in a code fragment, so as to ensure that programs don’t get stuck. But there is no way for the type system to know that `program` is closed, hence, a conservative approximation is made, and the term is rejected by the type system.

**Contribution and Organization of this Paper** In previous work [12], we reported on the implementation and applications of MetaML, and later [11] presented an axiomatic semantics and a type system for MetaML and proved type-safety. However, there were still a number of drawbacks:

1. As discussed above, there is a typing problem with executing a separately-declared code fragment. While this problem is addressed in the implementation using a sound rule for top-level declarations, this solution is ad hoc.
2. Only a call-by-value semantics could be defined for MetaML, because substitution was a partial function, only defined when variables are substituted with values.
3. The type judgment used two indices. Moreover, it has been criticized for not being based on a standard logical system [13].

This paper describes the type system and operational semantics of An Idealized MetaML (AIM), whose design is inspired by a categorical model for MetaML (such a model will be the subject of another paper). AIM is strictly more expressive than any known typed multi-level language, and features:
1. An open code type \( \langle t \rangle \), which corresponds to \( \bigcirc t \) of \( \lambda^\square \) [3] and \( \langle t \rangle \) of MetaML;
2. A closed code type \([t]\), which corresponds to \( \Box t \) of \( \lambda^\square \) [4];
3. Cross-stage persistence of MetaML;
4. A Run-with construct, generalizing Run of MetaML.

This work is the first to achieve a \textit{semantically sound} integration of Davies and Pfenning’s \( \lambda^\square \) [4] and Davies’ \( \lambda^\circ \) [3], and to identify useful interactions between them. Moreover, we present important simplifications over MetaML [11], which overcome the problems mentioned above:

1. The type system uses only one level annotation, like the \( \lambda^\circ \) type system [3];
2. The level Promotion and level Demotion lemmas, and the Substitution lemma, are proven in full generality and not just for the cases restricted to values. This development is crucial for a call-by-name semantics. Such a semantics seems to play an important role in the formal theory of Normalization by Evaluation [1] and Type Directed Partial Evaluation [2];
3. The big-step semantics is defined in the style in which \( \lambda^\circ \) was defined [3], and does not make explicit use of a stateful renaming function;
4. Terms have no explicit level annotations.

Finally, it is straightforward to extend AIM with new base types and constants, therefore it provides a general setting for investigating \textit{staging combinators}.

In the rest of the paper, we present the type system and establish several syntactic properties. We give a big-step semantics of AIM, including a call-by-name variant, and prove type-safety. We present embeddings of \( \lambda^\circ \), MetaML and \( \lambda^\square \) into AIM. Finally, we discuss related works.

## 2 AIM: An Idealized MetaML

The definition of AIM’s types \( t \in T \) and terms \( e \in E \) is parameterized with respect to a signature consisting of a set of \textbf{base types} \( b \) and \textbf{constants} \( c \):

\[
\begin{align*}
t \in T : = & \ b \mid t_1 \to t_2 \mid \langle t \rangle \mid [t] \\
e \in E : = & \ c \mid x \mid e_1 \ e_2 \mid \lambda x . e \mid \langle e \rangle \mid \text{run } e \text{ with } \{ x_i = e_i \mid i \in m \} \\
& \text{box } e \text{ with } \{ x_i = e_i \mid i \in m \} \mid \text{unbox } e
\end{align*}
\]

Where \( m \) is a natural number, and it is identified with the set of its predecessors.

The first four constructs are the standard ones in a call-by-value \( \lambda \)-calculus with constants. Bracket and Escape are the same as in MetaML [12, 11]. Run-With generalizes Run of MetaML, in that allows the use of additional variables \( x_i \) in the body of \( e \) if they satisfy certain typing requirements that are made explicit in the next section. Box-With and unbox are not in MetaML, but are motivated by \( \lambda^\square \) of Davies and Pfenning [4]. We use some abbreviated forms:

\[
\begin{align*}
\text{run } e \ & \text{for run } e \text{ with } \emptyset \\
\text{box } e \ & \text{for box } e \text{ with } \emptyset \\
\text{run } e \text{ with } x_i = e_i \ & \text{for run } e \text{ with } \{ x_i = e_i \mid i \in m \} \\
\text{box } e \text{ with } x_i = e_i \ & \text{for box } e \text{ with } \{ x_i = e_i \mid i \in m \}
\end{align*}
\]
\[
\begin{align*}
\Gamma &\vdash e : t^n & &\text{if } \Gamma \vdash x : t^m \text{ and } m \leq n \\
\Gamma &\vdash x : t^n & &\text{if } \Gamma \vdash \lambda x : t^m \vdash t^n \\
\Gamma &\vdash e_1 : (t_1 \rightarrow t_2)^n & &\text{if } \Gamma \vdash e_2 : t_1^n \\
\Gamma &\vdash e_1 ; e_2 : t_2^n & &\text{if } \Gamma \vdash e : t^{n+1} \\
\Gamma &\vdash e : (t)^n & &\text{if } \Gamma \vdash \langle e \rangle : t^n \\
\Gamma &\vdash \langle e \rangle : t^{n+1} & &\text{if } \Gamma \vdash "e" : t^{n+1} \\
\Gamma &\vdash e : [t]^n & &\text{if } \Gamma \vdash \{ x_i : [t_i]^n \mid i \in m \} \vdash e : t^0 \\
\Gamma &\vdash \text{run } e \text{ with } x_i = e_i : t^n & &\text{if } \Gamma \vdash e : [t]^n \\
\Gamma &\vdash \text{box } e \text{ with } x_i = e_i : [t]^n & &\text{if } \Gamma \vdash e : [t]^n \\
\Gamma &\vdash \text{unbox } e : t^n & &\text{if } \Gamma \vdash e : [t]^n
\end{align*}
\]

Fig. 1. Typing Rules

\section{2.1 Type System}

An AIM typing judgment has the form \( \Gamma \vdash e : t^n \), where \( t \in T \), \( n \in N \) and \( \Gamma \) is a type assignment, that is, a finite set \( \{ x_i : t_i^n \mid i \in m \} \) with the \( x_i \) distinct. The reading of \( \Gamma \vdash e : t^n \) is "term \( e \) has type \( t \) at level \( n \) in the type assignment \( \Gamma \)". We say that \( \Gamma \vdash x : t^n \) if \( x : t^n \) is in \( \Gamma \). Furthermore, we write \( \Gamma^{+r} \) for the type assignment obtained by incrementing the level annotations in \( \Gamma \) by \( r \), that is, \( \Gamma^{+r} x = t^{n+r} \) if and only if \( \Gamma x = t^n \). Figure 1 gives the typing rules for AIM.

The Constant rule says that a constant \( e \) of type \( t_c \), which has to be given in the signature, can be used at any level \( n \). The Variable rule incorporates cross-stage persistence, therefore if \( x \) is introduced at level \( m \) it can be used later, that is, at level \( n \geq m \), but not before. The Abstraction and Application rules are standard. The Bracket and Escape rules establish an isomorphism between \( t^{n+1} \) and \( \langle t \rangle^n \). Typing Run in MetaML [11] introduces an extra index-annotation on types for counting the number of Runs surrounding an expression (see Figure 3). We avoid this extra annotation by incrementing the level of all variables in \( \Gamma \).

In particular, the Run rule of MetaML becomes

\[
\begin{align*}
\Gamma^{+1} &\vdash e : \langle t \rangle^n \\
\Gamma &\vdash \text{run } e : t^n
\end{align*}
\]

The Box rule ensures that there are no "late" free variables in the term being Boxed. This ensures that when a Boxed term is evaluated in a type-safe context, the resulting value is a closed term. The Box rule ensures that only the variables explicitly bound in the Box statement can occur free in the term \( e \). At the same time, it ensures that no "late" free variable can infiltrate the body of a Box using one of these variables. This is accomplished by forcing the With-bound variables themselves to have a Boxed type. Note that in run \( e \) with \( x_i = e_i \) the term \( e \) may contain other free variables besides the \( x_i \).
2.2 Properties of the Type System

The following level Promotion, level Demotion and Substitution lemmas are needed for proving Type Preservation.

**Lemma 1 (Promotion).** If \( \Gamma_1, \Gamma_2 \vdash e : t^n \) then \( \Gamma_1, \Gamma_2^{+1} \vdash e : t^{n+1} \).

Meaning that if we increment the level of a well-formed term \( e \) it remains well-formed. Furthermore, we can simultaneously increment the level of an arbitrary subset of the variables in the environment.

Demotion on \( e \) at \( n \), written \( e \downarrow_n \), lowers the level of \( e \) from level \( n + 1 \) down to level \( n \), and is well-defined on all terms, unlike demotion for MetaML [11].

**Definition 1 (Demotion).** \( e \downarrow_n \) is defined by induction on \( e \):

\[
\begin{align*}
  e \downarrow_0 & = e \\
  x \downarrow_n & = x \\
  (e_1 e_2) \downarrow_n & = e_1 \downarrow_n e_2 \downarrow_n \\
  (\lambda x.e) \downarrow_n & = \lambda x.e \downarrow_{n+1} \\
  \langle e \rangle \downarrow_n & = (e \downarrow_{n+1}) \\
  "e" \downarrow_n & = \text{run } e \\
  ("e") \downarrow_{n+1} & = ("e" \downarrow_n) \\
  (\text{run } e \text{ with } x_i = e_i) \downarrow_n & = \text{run } e \downarrow_n \text{ with } x_i = e_i \downarrow_n \\
  (\text{box } e \text{ with } x_i = e_i) \downarrow_n & = \text{box } e \text{ with } x_i = e_i \downarrow_n \\
  (\text{unbox } e) \downarrow_n & = \text{unbox } e \downarrow_n
\end{align*}
\]

The key for making demotion total on all terms is handling the case for Escape "\( e \downarrow_0 \): Escape is simply replaced by Run. It should also be noted that demotion does not go into the body of Box.

**Lemma 2 (Demotion).** If \( \Gamma^{+1} \vdash e : t^{n+1} \) then \( \Gamma \vdash e \downarrow_n : t^n \).

Meaning that demotion of a well-formed term \( e \) is well-formed, provided the level of all free variables is decremented.

Details of all proofs can be found in the Appendix.

**Lemma 3 (Weakening).** If \( \Gamma_1, \Gamma_2 \vdash e_2 : t_3^n \) and \( x \) is fresh, then \( \Gamma_1, x : t_1^n, \Gamma_2 \vdash e_2 : t_3^n \).

**Lemma 4 (Substitution).** If \( \Gamma_1 \vdash e_1 : t_1^n \) and \( \Gamma_1, x : t_1^n, \Gamma_2 \vdash e_2 : t_2^n \) then \( \Gamma_1, \Gamma_2 \vdash e_2[x := e_1] : t_2^n \).

This is the expected substitution property, that is, a variable \( x \) can be replaced by a term \( e_1 \), provided \( e_1 \) meets the type requirements on \( x \).
Evaluation.

\[
\begin{align*}
\frac{e_1 \downarrow 0 \lambda x. e \quad e_2 \downarrow 0 \quad e[x := v_1] \downarrow 0 \quad v_2}{e_1 \downarrow e_2 \downarrow v_2} & \quad \frac{\lambda x. e \downarrow 0 \lambda x. e}{\lambda x. e \downarrow 0 \lambda x. e} \\
\frac{e_i \downarrow 0 \quad v_i \quad e[x_i := v_i] \downarrow 0 \quad (e') \downarrow 0 \quad e' \downarrow 0 \quad v}{\text{run } e \text{ with } x_i = e_i \downarrow v} & \quad \frac{e \downarrow 0 \quad (v)}{e \downarrow 0 \quad (v)} \\
\frac{e_i \downarrow 0 \quad v_i}{\text{box } e \text{ with } x_i = e_i \downarrow \text{box } e[x_i := v_i]} & \quad \frac{e \downarrow 0 \quad \text{box } e' \quad e' \downarrow 0 \quad v}{\text{unbox } e \downarrow 0 \quad v}
\end{align*}
\]

Building.

\[
\begin{align*}
\frac{e \downarrow n+1 \quad v}{\text{unbox } e \downarrow n+1 \quad \text{unbox } v} & \quad \frac{\lambda x. e \downarrow n+1 \quad \lambda x. v}{x \downarrow n+1 \quad x} \\
\frac{e \downarrow n+1 \quad v \quad e_i \downarrow n+1 \quad v_i}{\text{run } e \text{ with } x_i = e_i \downarrow \text{run } v \text{ with } x_i = v_i} & \quad \frac{e \downarrow n+1 \quad v \quad e \downarrow n+1 \quad v}{e \downarrow n+1 \quad v} \\
\frac{e_i \downarrow n+1 \quad v_i}{\text{box } e \text{ with } x_i = e_i \downarrow \text{box } e \text{ with } x_i = v_i} & \quad \frac{e_1 \downarrow n+1 \quad v_1 \quad e_2 \downarrow n+1 \quad v_2}{c \downarrow n+1 \quad c}
\end{align*}
\]

Stuck.

\[
\begin{align*}
\frac{e \downarrow 0 \quad v \not\equiv \text{box } e'}{\text{unbox } e \downarrow 0 \quad \text{err}} & \quad \frac{e_1 \downarrow 0 \quad v \not\equiv \lambda x. e \quad e_1 \downarrow e_2 \downarrow \text{err}}{x \downarrow 0 \quad \text{err}} \\
\frac{e_i \downarrow 0 \quad v_i \quad e[x_i := v_i] \downarrow 0 \quad v \not\equiv (e')}{\text{run } e \text{ with } x_i = e_i \downarrow \text{err}} & \quad \frac{e \downarrow 0 \quad v \not\equiv (e')}{e \downarrow 0 \quad \text{err}}
\end{align*}
\]

Fig. 2. Big-Step Semantics

3 Big-Step Semantics

The big-step semantics for MetaML [12] reflects the existing implementation: it is complex, and hence not very suitable for formal reasoning. Figure 2 presents a concise big-step semantics for AIM, which is presented at the same level of abstraction as that for \(\lambda^0\) [3]. We avoid the explicit use of a \textit{gensym} or \textit{newname} for renaming bound variables: this is implicitly done by substitution.
Definition 2 (Values).

\[ v^0 \in V^0 \quad ::= \quad \lambda x.e \mid \langle v^1 \rangle \mid \text{box } e \]

\[ v^1 \in V^1 \quad ::= \quad e \mid x \mid v^1 \mid v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \text{run } v^1 \text{ with } x_i = v_i^1 \mid \text{box } e \text{ with } x_i = v_i^1 \mid \text{unbox } v^1 \]

\[ v^{n+2} \in V^{n+2} \quad ::= \quad e \mid x \mid v^{n+2} \mid v^{n+2} \mid \lambda x.v^{n+2} \mid \langle v^{n+3} \rangle \mid v^{n+1} \mid \text{run } v^{n+2} \text{ with } x_i = v_i^{n+2} \mid \text{box } e \text{ with } x_i = v_i^{n+2} \mid \text{unbox } v^{n+2} \]

Values have three important properties: First, a value at level 1 can be a Bracketed or a Boxed expression, reflecting the fact that terms representing open and closed code are both considered acceptable results from a computation. Second, values at level \( n+1 \) can contain Applications such as \( \langle (\lambda y.y) \,(\lambda x.x) \rangle \), reflecting the fact that Brackets defer computations. Finally, there are no level 1 Escapes in values, reflecting the fact that having such an Escape in a term would mean that evaluating the term has not yet been completed. This is true, for example, in terms like \( \langle "f" \,x \rangle \).

Lemma 5 (Orthogonality). If \( v \in V^0 \) and \( \Gamma \vdash v;[t]^0 \) then \( \emptyset \vdash v;[t]^0 \).

Theorem 1 (Type Preservation). If \( \Gamma^{+1} \vdash e:t^n \) and \( e \overset{\varepsilon}{\rightarrow} v \) then \( v \in V^n \) and \( \Gamma^{+1} \vdash v;t^n \).

Note that in AIM (unlike ordinary programming languages) we cannot restrict the evaluation rules to closed terms, because at levels above 0 evaluation is symbolic and can go inside the body of binders. On the other hand, evaluation of a variable at level 0 is an error! The above theorem strikes the right balance, namely it allows open terms provided their free variables are at level above 0 (this is reflected by the use of \( \Gamma^{+1} \) in the typing judgment).

Having no level 1 escapes ensures that demotion is the identity on \( V^{n+1} \) as shown in following lemma. Thus, we don’t need to perform demotion in the evaluation rule for Run when evaluating a well-formed term.

Lemma 6 (Value Demotion). If \( v \in V^{n+1} \) then \( v \downarrow_{\varepsilon} \equiv v \).

A good property for multi-level languages is the existence of a bijection between programs \( \emptyset \vdash e:t^n \) and program representations \( \emptyset \vdash \langle v; \langle t \rangle^0 \). This property holds for AIM, in fact it is a consequence of the following result:

Proposition 1 (Reflection). If \( \Gamma \vdash e:t^n \), then \( \Gamma^{+1} \vdash e:t^{n+1} \) and \( e \in V^{n+1} \).

Conversely, if \( v \in V^{n+1} \) and \( \Gamma^{+1} \vdash v;t^{n+1} \), then \( \Gamma \vdash v;t^n \).

3.1 Call-by-Name

The difference between the call-by-name semantics and the call-by-value semantics for AIM is only in the evaluation rule for Application at level 0. For call-by-name, this rule becomes

\[
\frac{e_1 \overset{\emptyset}{\rightarrow} \lambda x.e \quad e[x := e_2] \overset{\emptyset}{\rightarrow} v}{e_1 \, e_2 \overset{\emptyset}{\rightarrow} v}
\]
The Type Preservation proof must be changed for this case. However, this not problematic, since the Substitution Lemma for the AIM’s type system has no value restriction.

**Theorem 2 (CBN Type Preservation).** If \( \Gamma^{+1} \vdash e : t^n \) and \( e \rightarrow^* v \) then \( v \in V^n \) and \( \Gamma^{+1} \vdash v : t^n \).

### 3.2 Expressiveness

MetaML’s type system has one Code type constructor, which tries to combine the features of the Box and Circle type constructors of Davies and Pfenning. However, this combination leads to the typing problem discussed in the introduction. In contrast, AIM’s type system incorporates both Box and Circle type constructors, thereby providing correct semantics for the following functions:

1. **unbox**: \([t] \rightarrow t\). This function executes closed code. AIM has no function of type \( t \rightarrow [t] \), thus we avoid the “collapse” of types in the recent work of Wickline, Lee, and Pfenning [13]. Such a function does not exist in MetaML.

2. **up**: \( t \rightarrow \langle t \rangle \). This function corresponds to cross-stage persistence [12], in fact it embeds any value into an open fragment, including values of functional type. Such a function does not exist in \( \lambda^O \). At the same time, AIM has no function of type \( \langle t \rangle \rightarrow t \), reflecting the fact that open code cannot be executed. up is expressible as \( \lambda x.(x) \).

3. **weaken**: \([t] \rightarrow \langle t \rangle \). The composite of the two functions above. weaken reflects the fact that closed code can always be viewed as open code. AIM has no function of type \( \langle t \rangle \rightarrow [t] \).

4. **execute**: \( [\langle t \rangle] \rightarrow t \). This function executes closed code, and it can be defined in AIM as \( \lambda x.\text{run } x \) with \( x = x \).

5. **build**: \( [\langle t \rangle] \rightarrow [\langle t \rangle] \) This function forces the building of an open fragment known to be closed. build is not expressible in the language, but it can be added as a new combinator with the following semantics:

\[
\begin{align*}
\varepsilon & \leftrightarrow \text{box } \varepsilon' \\
\varepsilon' & \leftrightarrow \langle v \rangle \\
\text{build } \varepsilon & \leftrightarrow \text{box } \langle v \rangle
\end{align*}
\]

Type Preservation is still valid with such an extension.

Now, the MetaML example presented in the Introduction can be expressed in AIM as follows:

```ml
- | val rec exp = box (fn n => fn x =>
    if n=0 then <1> else <"x * "((unbox exp) (n-1) x) >)
    with {exp=exp};
val exp = [fn] : [int -> <int> -> <int>]
```

```ml
- | val exponent = box (fn n =>
```
In AIM, asserting that a code fragment is closed (using Box) has become part of the responsibilities of the programmer. Furthermore, Build is needed to explicitly overcome the default lazy behavior of Box. If Build was not used in the above examples, the (Boxed code) values returned for cube and program would contain level 0 escapes. In general, it appears that the lazy behavior of Box is not needed when our primary concern is high-level program generation. Unfortunately, the syntax is verbose compared to that of MetaML. In future work, we hope to improve the syntax based on experience using AIM.

4 Embedding Results

This section shows that other languages for staging computations can be translated into AIM, and that the embedding respects the typing and evaluation. The languages we consider are $\lambda^\Lambda$ [3], MetaML [11], and $\lambda^\Sigma$ [4].

4.1 Embedding of $\lambda^\Lambda$

The embedding of $\lambda^\Lambda$ into AIM is straightforward. In essence, $\lambda^\Lambda$ corresponds to the Open fragment of AIM:

$t \in T_{open} ::= b \mid t_1 \to t_2 \mid \langle t \rangle$
$e \in E_{open} ::= e \mid x \mid e_1 e_2 \mid \lambda x.e \mid \langle e \rangle \mid "e"

The translation ($\subseteq$) between $\lambda^\Lambda$ and AIM is as follows: $(\iota t)^{\Lambda}=\langle t^{\Lambda} \rangle$, $(\text{next } e)^{\Lambda}=\langle e^{\Lambda} \rangle$, and $(\text{prev } e)^{\Lambda}="(e^{\Lambda})"$. With these identifications the typing and evaluation rules for $\lambda^\Lambda$ are those of AIM restricted to the relevant fragment. The only exception is the typing rule for variables, which in $\lambda^\Lambda$ is simply $\Gamma \vdash x:t''$ if $\Gamma \vdash t'=t''$ (this reflects the fact that $\lambda^\Lambda$ has no cross-stage persistence).

We write $\Gamma \vdash^{\Lambda} e:t$ and $e \in_{\lambda^\Lambda} v$ for the typing and evaluation judgments of $\lambda^\Lambda$, so that they are not confused with the corresponding judgments of AIM.
Proposition 2 (Temporal Type Embedding). If $\Gamma \vdash \circ c : t^n$ is derivable in $\lambda^\Box$, then $\Gamma^\Box \vdash c^\Box : (t^\Box)^n$ is derivable in AIM.

Proposition 3 (Temporal Semantics Embedding). If $c^\Box \xrightarrow{\circ} v$ is derivable in $\lambda^\Box$, then $c^\Box \xrightarrow{\circ} v^\Box$ is derivable in AIM.

4.2 Embedding of MetaML

The difference between MetaML and AIM is in the type system. We show that while AIM’s typing judgments are simpler, what is typable in MetaML remains typable in AIM.

\[
\begin{align*}
\Gamma \vdash c : (t, r)^n & \quad \Gamma \vdash x : (t, r)^n \text{ if } \Gamma \vdash x = (t, p)^m \text{ and } m + r \leq n + p \\
\Gamma, x : (t_1, r)^n & \vdash c : (t_2, r)^n \quad \Gamma \vdash e_1 : (t_3 \to t_2, r)^n \quad \Gamma \vdash e_2 : (t_2, r)^n \\
\Gamma \vdash \lambda x. e : (t_1 \to t_2, r)^n & \quad \Gamma \vdash e_1 : (t_3 \to (t_2, r)^n) \\
\Gamma \vdash e : (t, r)^{n+1} & \quad \Gamma \vdash e : ((t), r)^n \\
\Gamma \vdash \langle e \rangle : ((t), r)^n & \quad \Gamma \vdash ~e : (t, r)^{n+1} & \quad \Gamma \vdash \text{run } e : (t, r)^n
\end{align*}
\]

![Fig. 3. MetaML Typing rules](image)

Definition 3 (Acceptable Judgment). We say that a MetaML typing judgment $\{ x_i : (t_i, r_i)^n \mid i \in m \} \vdash c : (t, r)^n$ is acceptable if and only if $\forall i \in m, r_i \leq r$.

Remark 1. A careful analysis of MetaML’s typing rules shows that typing judgments occurring in the derivation of a judgment $\emptyset \vdash c : (t, r)^n$ are acceptable: In MetaML typing rules are acceptable whenever its conclusion is acceptable, simply because the index $r$ never decreases when we go from the conclusion of a type rule to its premise, thus, we never get an environment binding with an $r$ higher than that of the judgment.

Proposition 4 (MetaML Type Embedding). If $\{ x_i : (t_i, r_i)^n \mid i \in m \} \vdash c : (t, r)^n$ is acceptable, then it is derivable in MetaML if and only if $\{ x_i : (t^\Box_i)^{n+1} \mid i \in m \} \vdash c : t^\Box$ is derivable in AIM.

4.3 Embedding of $\lambda^\Box$

Figure 4 summarizes the language $\lambda^\Box$ [4]. We translate $\lambda^\Box$ into the Closed
Syntax

Types \( t \in T \ ::= \ b \mid t_1 \to t_2 \mid \square t \)

Expressions \( e \in E \ ::= \ x \mid \lambda x. e \mid e_1 \ e_2 \mid \text{box } e \mid \text{let } \text{box } x = e_1 \text{ in } e_2 \)

Type assignments \( \Gamma, \Delta \ ::= \ {x_i : t_i \mid i \in m} \)

Type System

\[
\begin{align*}
\Delta; \Gamma \vdash \square x : t & \text{ if } \Delta x = t & \Delta; \Gamma \vdash \square x : t & \text{ if } \Gamma x = t \\
\Delta; \Gamma \vdash \lambda x. e : t' \to t & \Delta; \Gamma \vdash \lambda x. e : t' \to t \\
\Delta; \Gamma \vdash e_1 : t' \to t & \Delta; \Gamma \vdash e_2 : t' & \Delta; \Gamma \vdash e_1 : \square t' & \Delta; \Gamma \vdash e_2 : \square t \\
\Delta; \Gamma \vdash \text{let box } x = e_1 \text{ in } e_2 : t & \Delta; \Gamma \vdash \text{let box } e = e_1 \text{ in } e_2 : \square t
\end{align*}
\]

Big-Step Semantics

\[
\begin{align*}
e_1 \mapsto \square \lambda x. e & \quad e_2 \mapsto \square v & \quad e[x = v'] \mapsto \square v & \quad \lambda x. e \mapsto \square \lambda x. e \\
e_1 \mapsto \square \text{box } e & \quad e_2[x = e] \mapsto \square v & \quad \text{let box } x = e_1 \text{ in } e_2 \mapsto \square v & \quad \text{box } e \mapsto \square \text{box } e
\end{align*}
\]

Fig. 4. Description of \( \lambda^\square \)

fragment of AIM:

\[
t \in T_{\text{Closed}} \ ::= \ b \mid t_1 \to t_2 \mid [t]
\]
\[
e \in E_{\text{Closed}} \ ::= \ c \mid x \mid e_1 \ e_2 \mid \lambda x. e \mid \text{box } e \text{ with } x_i = e_i \mid \text{unbox } e
\]

Furthermore, we consider only typing judgments of the form \( \{x_i : t_i \mid i \in m\} \vdash e : t^0 \) and evaluation judgments of the form \( e \mapsto v \). These restrictions are possible for two reasons. If the conclusion of a typing rule is of the form \( \{x_i : t_i \mid i \in m\} \vdash e : t^0 \) with types and terms in the Closed fragment, then also the premises of the typing rule enjoy such properties. When \( e \) is a closed term in the Closed fragment, the only judgments \( e^n \mapsto v \) that can occur in the derivation of \( e \mapsto v \) are such that \( n = 0 \) and \( v' \) term \( v' \) are closed terms in the Closed fragment.

Definition 4 (Modal Type Translation). The translation of \( \lambda^\square \) types is given by

\[
\begin{align*}
b^\square & = b \\
(t_1 \to t_2)^\square & = t_1^\square \to t_2^\square \\
(\square t)^\square & = [t]^\square
\end{align*}
\]
The translation of $\lambda^\Box$ terms depends on a set $X$ of variables, namely those declared in the modal context $\Delta$.

\[
x^\Box X = \text{unbox } x \quad \text{if } x \in X
\]
\[
x^\Box X = x \quad \text{if } x \notin X
\]
\[
(\text{box } e)^\Box X = \text{box } e^\Box X \text{ with } \{ x = x \mid x \in \text{FV}(e) \cap X \}
\]
\[
(\lambda x.e)^\Box X = \lambda x.e^\Box X
\]
\[
(e_1 e_2)^\Box X = e_1^\Box X e_2^\Box X
\]

**Proposition 5 (Modal Type Embedding).** If $\Delta; \Gamma \vdash e : t$ is derivable in $\lambda^\Box$, then $[\Delta^\Box], \Gamma^\Box \vdash e^\Box : t^\Box$ is derivable in AIM's Closed fragment, where $X$ is the set of variables declared in $\Delta$, $\{ x_i : t_i \mid i \in m \}^\Box$ is $\{ x_i^\Box : t_i^\Box \mid i \in m \}$, and $[[ x_i : t_i^\Box \mid i \in m ]] = \{ x_i : t_i^\Box \mid i \in m \}$.

The translation of $\lambda^\Box$ into the AIM's Closed fragment does not preserve evaluation on the nose (that is, up to syntactic equality). Therefore, we need to consider an administrative reduction.

**Definition 5 (Box-Reduction).** The $\to_{\text{box}}$ reduction is given by the rewrite rules

- **unbox** ($\text{box } e$) $\rightarrow e$
- box $e'$ with $x_i = e_i, x_j = e_j$ $\rightarrow$ box $e'[x := \text{box } e]$ with $x_i = e_i, x_j = e_j$

where $e$ is a closed term of the Closed fragment.

**Lemma 7 (Properties of Box-Reduction).** The $\to_{\text{box}}$ reduction on the Closed fragment satisfies the following properties:

- **Subject Reduction**, that is, $\Gamma \vdash e : t$ and $e \rightarrow e'$ imply $\Gamma \vdash e' : t$
- **Confluence and Strong Normalization**
- **Compatibility with Evaluation on closed terms**, that is, $e_1 \rightarrow v_1$ and $e_1 \rightarrow_{\text{box}} e_2$ imply that exists $v_2$ s.t. $v_1 \rightarrow_{\text{box}} v_2$ and $e_2 \rightarrow v_2$.

**Lemma 8 (Substitutivity).** Given a closed term $e_0 \in E^\Box$, the following properties hold:

- $e^\Box X[y := e_0^\Box] \equiv (e[y := e_0]^\Box)^X$, provided $y \notin X$
- $e^\Box X[u(x)] \equiv \text{box } e_0^\Box[x := \text{box } e_0^\Box]^\Box X$

**Proposition 6 (Modal Semantics Embedding).** If $e \in E^\Box$ is closed and $e \rightarrow_{\Box} v$ is derivable in $\lambda^\Box$, then there exists $v'$ such that $e^\Box v \rightarrow v'$ and $v' \rightarrow_{\text{box}} v^\Box v'$. 
5 Related Work

Multi-stage programming techniques have been used in a wide variety of settings [12], including run-time specialization of C programs [10].

Nielsen and Nielsen present a seminal detailed study into a two-level functional programming language [9]. This language was developed for studying code generation. Davies and Pfenning show that a generalization of this language to a multi-level language called $\lambda^2$ gives rise to a type system related to a modal logic, and that this type system is equivalent to the binding-time analysis of Nielsen and Nielsen [4]. Intuitively, $\lambda^2$ provides a natural framework where Scheme’s back-quote and eval can be present in a language. The semantics of our Box and Unbox correspond closely to those of back-quote and eval, respectively.

Gomard and Jones [6] use a statically-typed two-level language for partial evaluation of the untyped $\lambda$-calculus. This language is the basis for many binding-time analyses.

Glick and Jørgensen study partial evaluation in the generalized context where inputs can arrive at an arbitrary number of times rather than just two (namely, specialization-time and run-time) [5], and demonstrate that binding-time analysis in a multi-level setting can be done with efficiency comparable to that of two-level binding time analysis.

Davies extends the Curry-Howard isomorphism to a relation between temporal logic and the type system for a multi-level language [3]. Intuitively, $\lambda^2$ provides a good framework for formalizing the presence of back-quote and comma in a statically typed language. The semantics of our Bracket and Escape correspond closely to those of back-quote and comma, respectively.

Moggi [8] advocates a categorical approach to two-level languages based on indexed categories, and stresses formal analogies with a categorical account of phase distinction and module languages.

References


A Type preservation

Convention 6 (Notation for Proofs on Derivations) Whenever convenient, proofs will be laid out in 2-dimensions to reflect that we are relating one derivation tree to another. Symbols such as \( \stackrel{\lor}{\mapsto} \), \( \Leftrightarrow \), and \( \stackrel{\land}{\mapsto} \) will be used for (one- or two-way) implications, where \( R \) is the list of rule(s) used to achieve this implication.

Proofs start from the left and proceed either up, down, or right. We go up or down depending on the normal orientation of the particular rule that is being used in an implication. Horizontal implications are aligned with their precedents and antecedents.

Lemma 1 If \( \Gamma_1, \Gamma_2 \vdash e : t^n \) then \( \Gamma_1, \Gamma_2^{n+1} \vdash e : t^{n+1} \).

Proof: By structural induction over the first derivation.

- Lambda abstraction:
  \[
  \begin{align*}
    & \Gamma_1, (\Gamma_2, x : t^1_1) \vdash e : t^n_1 \\
    & \Rightarrow \quad \Gamma_1 \vdash x : t^n \\
    & \Leftrightarrow \quad \Gamma_1 \vdash x : t^{n+1} \\
  \end{align*}
  \]

- Variables (I):
  \[
  \begin{align*}
    & \Gamma_1 \vdash x = (t, n') \\
    & \Rightarrow \quad \Gamma_1 \vdash x = (t, n') \\
    & \Leftrightarrow \quad \Gamma_1 \vdash x = (t, n' + 1) \\
  \end{align*}
  \]

- Variables (II):
  \[
  \begin{align*}
    & \Gamma_1 \vdash x = (t, n') \\
    & \Rightarrow \quad \Gamma_1 \vdash x = (t, n') \\
    & \Leftrightarrow \quad \Gamma_1 \vdash x = (t, n' + 1) \\
  \end{align*}
  \]

The rest of the cases are completely structural.

- Applications:
  \[
  \begin{align*}
    & \Gamma_1, \Gamma_2 \vdash e_1 : t_1 \Rightarrow t^n_2 \\
    & \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash e_1 : t_1 \Rightarrow t^{n+1} \\
    & \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash e_1 : t_1 \Rightarrow e_2 : t_2^{n+1} \\
  \end{align*}
  \]

- Run:
  \[
  \begin{align*}
    & \Gamma_1, \Gamma_2 \vdash \text{run } e \text{ with } x_i = e_i : t^n \\
    & \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash \text{run } e \text{ with } x_i = e_i : t^{n+1} \\
  \end{align*}
  \]
Lemma 2 If $\Gamma \vdash e : t^{n+1}$ then $\Gamma \vdash e \downarrow : t^n$.

Proof: By structural induction over the first derivation.

- Lambda abstraction:

$\Gamma \vdash e : t_1^{n+1} \Rightarrow \Gamma \vdash \lambda x. e : t_1^{n+1} \Rightarrow \Gamma \vdash \lambda x. e : t_1^{n+1}$

- Variables:

$\Gamma \vdash x = (t, n') \Rightarrow \Gamma \vdash x = (t, n')$

- Applications:

$\Gamma \vdash e_1 : t_1 \Rightarrow \Gamma \vdash e_2 : t_2^{n+1} \Rightarrow \Gamma \vdash e_1 e_2 : t_2^n$
- Escape (I): Note that here $^\rightarrow e \downarrow = \text{run } e$.

$$
\Gamma \vdash e : t_1 \quad \Rightarrow \quad \Gamma \vdash \lambda e : t_1 \downarrow_1
$$

- Escape (II):

$$
\Gamma \vdash e : (t_1)^n+1 \quad \Rightarrow \quad \Gamma \vdash \lambda e : (t_1)^n \downarrow_1
$$

- Bracket:

$$
\Gamma \vdash e : t_1^n+1 \quad \Rightarrow \quad \Gamma \vdash \lambda e : (t_1)^n \downarrow_1
$$

- Box:

$$
\Gamma \vdash e : t_1^n+1 \quad \Rightarrow \quad \Gamma \vdash \lambda e : (t_1)^n \downarrow_1
$$

- Unbox:

$$
\Gamma \vdash e : [t_1]^n+1 \quad \Rightarrow \quad \Gamma \vdash \lambda e : [t_1]^n \downarrow_1
$$

Lemma 4 If $\Gamma_1 \vdash e_1 : t_1^n$ and $\Gamma_1, x : t_1^n, \Gamma_2 \vdash e_2 : t_2^n$ then $\Gamma_1, \Gamma_2 \vdash e_2[x \leftarrow e_1] : t_2^n$.

Proof: By structural induction over the second derivation.

- Lambda abstraction: By Barendregt’s convention, $y \neq x$.

$$
\Gamma_1, x : t_1^n, \Gamma_2, y : t_3^n \vdash e : t_4^n \quad \Rightarrow \quad \Gamma_1, \Gamma_2, \lambda y : t_3^n \vdash e[x \leftarrow e_1] : t_4^n
$$

- Variables (I): If $e_2 = x$, we already know $\Gamma_1 \vdash e_1 : t_1^n$. By weakening we have $\Gamma_1, \Gamma_2 \vdash e_1 : t_1^n$, and by $n - n'$ uses of the promotion lemma we have $\Gamma_1, \Gamma_2 \vdash e_1 : t_1^n$.

- Variables (II): If $e_2 \neq x$, we already know $\Gamma_1, x : t_1^n, \Gamma_2 \vdash e_2 : t_2^n$ and by weakening we have $\Gamma_1, \Gamma_2 \vdash e_2 : t_2^n$. The rest of the cases are completely structural.

- Applications:

$$
\Gamma_1, x : t_1^n, \Gamma_2 \vdash e_1 : t_1 \rightarrow t_2^n \quad \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash e_1 \downarrow_1
$$

$$
\Gamma_1, x : t_1^n, \Gamma_2 \vdash e_2 : t_2^n \quad \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash e_2 \downarrow_1
$$
- Run: By applying promotion lemma to the premise $I_1 \vdash \epsilon_1 : t_1^n$, we get $I_1^{+1} \vdash \epsilon_1 : t_1^{n+1}$. Now, we use these two judgement to apply induction hypothesis.

\[
\frac{I_1, x : t_1^n, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1^{+1}, x : t_1^{n+1}, I_2^{+1}, \{x_1 : t_1^n\} \vdash \epsilon_1 : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1^{+1}, I_2^{+1}, \{x_1 : t_1^n\} \vdash \epsilon_1 : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : t^n} \\
\]

- Escape:

\[
\frac{I_1, x : t_1^n, I_2 \vdash \epsilon : \langle t \rangle^n}{I_1, x : t_1^n, I_2 \vdash \epsilon : t^{n+1}} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash \epsilon_1 : t^{n+1}} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : t^n} \\
\]

- Bracket:

\[
\frac{I_1, x : t_1^n, I_2 \vdash \epsilon : t^{n+1}}{I_1, x : t_1^n, I_2 \vdash \langle \epsilon \rangle : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash \langle \epsilon \rangle \mid x_i = \epsilon_i \mid x_i = \epsilon_i] : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : t^n} \\
\]

- Box:

\[
\frac{I_1, x : t_1^n, I_2 \vdash \epsilon : t^n}{I_1, x : t_1^n, I_2 \vdash \epsilon_i \mid x_i : t_i^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 : \langle t \rangle^n}{I_1, I_2 \vdash \epsilon_1 \mid x_i : t_i^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 \mid x_i : t_i^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 \mid x_i : t_i^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : t^n} \\
\]

- Unbox:

\[
\frac{I_1, x : t_1^n, I_2 \vdash \epsilon : t^n}{I_1, x : t_1^n, I_2 \vdash \text{unbox } \epsilon : t_i^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 \mid x_i : t_i^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : \langle t \rangle^n} \quad \frac{I_1, I_2 \vdash \epsilon_1 \mid x_i : t_i^n}{I_1, I_2 \vdash (\text{run } \epsilon \text{ with } x_i = \epsilon_i) \mid x_i = \epsilon_i] : t^n} \\
\]

\[\boxed{}\]

**Lemma 5** If $v \in V^n$ and $I \vdash v : [t]_0$ then $\emptyset \vdash v : [t]_0$.

**Proof.** Since $v \in V^n$ and $I \vdash v : [t]_0$, then $v = \text{box } \epsilon$ for some $\epsilon$. By box rule, the expression $\epsilon$ must have type $\emptyset \vdash \epsilon : t_0$. Thus

\[
\frac{\emptyset \vdash \epsilon : t_0}{\emptyset \vdash \text{box } \epsilon : [t]_0}
\]

**Proposition 1** If $I^{+1} \vdash \epsilon : t^n$ and $\epsilon \xrightarrow{\text{run}} v$ then $v \in V^n$ and $I^{+1} \vdash v : t^n$.

**Proof:** By structural induction over the derivation of $\epsilon$.

- Bottom: $\bot \xrightarrow{\text{run}} \bot$ and $\bot \in V^{n+1}$ and $I^{+1} \vdash \bot : t^{n+1}$.
- Lambda-abstraction: Interestingly, no induction is needed here. In particular, $\lambda x. \epsilon \xrightarrow{\text{run}} \lambda x. \epsilon$. By definition, $\lambda x. \epsilon \in V_0$, and from the premise $I^{+1} \vdash \lambda x. \epsilon : t_0$. 


- Lambda-abstraction II: Straight forward induction.

\[
\begin{align*}
\Gamma^{+1} \vdash x : t_{i}^{n+1} \vdash e : t^{n+1} & \quad \Rightarrow \quad \frac{\Gamma^{+1} \vdash \lambda x.e : t_{1}^{n+1} \rightarrow t_{1}^{n+1}}{\Gamma^{+1} \vdash \lambda x.e \rightarrow \lambda x.e'} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{e \in V^{n+1}}{e' \in V^{n+1}} \quad \Rightarrow \quad \frac{\Gamma^{+1} \vdash \lambda x.e' : t_{1}^{n+1} \rightarrow t_{1}^{n+1}}{\Gamma^{+1} \vdash \lambda x.e' : t_{1}^{n+1} \rightarrow t_{1}^{n+1}} \quad \overset{\text{IH}}{\Rightarrow}
\end{align*}
\]

- Variables I: \( n = 0 \) is vacuous, because we can only derive \( \epsilon \).
- Variables II: \( x \vdash x^{n+1} \) and \( x \in V^{n+1} \) and \( \Gamma^{+1} \vdash x : t^{n+1} \).
- Applications I: First, we use (twice) the induction hypothesis, which gives us a result that we can use with the substitution lemma (\( = : \)):

\[
\begin{align*}
\Gamma^{+1} \vdash e_{1} : t_{1} \rightarrow t^{0} & \quad \Rightarrow \quad \frac{\Gamma^{+1} \vdash e_{2} : t_{2}^{0}}{\Gamma^{+1} \vdash e_{1} e_{2} : t_{1}^{0} \rightarrow t_{1}^{0}} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \epsilon_{1} \rightarrow \lambda x.e}{\epsilon_{1} \rightarrow \lambda x.e} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash e_{2} : t_{2}^{0}}{\Gamma^{+1} \vdash e_{1} e_{2} : t_{1}^{0} \rightarrow t_{1}^{0}} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \epsilon_{1} \rightarrow \lambda x.e}{\epsilon_{1} \rightarrow \lambda x.e} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \epsilon_{2} : t_{2}^{0}}{\Gamma^{+1} \vdash \epsilon_{1} \epsilon_{2} : t_{1}^{0} \rightarrow t_{1}^{0}} \quad \overset{\text{IH}}{\Rightarrow}
\end{align*}
\]

Note that we also use the judgment \( \Gamma^{+1} \vdash \epsilon_{1} : t_{1}^{0} \vdash \epsilon : t^{0} \) when we apply the substitution lemma. Then, based on this information about \( \epsilon_{1} \vdash \epsilon_{2} \) we apply the induction hypothesis for the third time to get \( \epsilon_{1} \in V^{0} \) and \( \Gamma^{+1} \vdash \epsilon_{1} : t^{0} \).

- Applications II:

\[
\begin{align*}
\Gamma^{+1} \vdash e_{1} : t_{1} \rightarrow t^{n+1} & \quad \Rightarrow \quad \frac{\Gamma^{+1} \vdash e_{2} : t_{2}^{n+1}}{\Gamma^{+1} \vdash e_{1} e_{2} : t_{1}^{n+1} \rightarrow t_{1}^{n+1}} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \epsilon_{1} \rightarrow \lambda x.e}{\epsilon_{1} \rightarrow \lambda x.e} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash e_{2} : t_{2}^{n+1}}{\Gamma^{+1} \vdash e_{1} e_{2} : t_{1}^{n+1} \rightarrow t_{1}^{n+1}} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \epsilon_{1} \rightarrow \lambda x.e}{\epsilon_{1} \rightarrow \lambda x.e} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash e_{2} : t_{2}^{n+1}}{\Gamma^{+1} \vdash e_{1} e_{2} : t_{1}^{n+1} \rightarrow t_{1}^{n+1}} \quad \overset{\text{IH}}{\Rightarrow}
\end{align*}
\]

- Run I: First, we apply the induction hypothesis once, then reconstruct the type judgment of the result:

\[
\begin{align*}
\Gamma^{+1} \vdash e_{1} : [t_{i}^{0}] & \quad \Rightarrow \quad \frac{\Gamma^{+2} \vdash \{ x_{i} : [t_{i}^{0}] \} \vdash \epsilon : \langle t \rangle \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \text{box } e_{i}^{0}}{\Gamma^{+1} \vdash \text{box } e_{i}^{0} : [t_{i}^{0}] \quad \overset{\text{IH}}{\Rightarrow} \quad \Gamma^{+1} \vdash \langle \epsilon \rangle \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \langle \epsilon \rangle}{\Gamma^{+1} \vdash \langle \epsilon \rangle} \quad \overset{\text{IH}}{\Rightarrow} \quad \frac{\Gamma^{+1} \vdash \langle \epsilon \rangle}{\Gamma^{+1} \vdash \langle \epsilon \rangle}}{\Gamma^{+1} \vdash \langle \epsilon \rangle \quad \overset{\text{IH}}{\Rightarrow}} \quad \frac{\Gamma^{+1} \vdash \langle \epsilon \rangle}{\Gamma^{+1} \vdash \langle \epsilon \rangle} \quad \overset{\text{IH}}{\Rightarrow}
\end{align*}
\]

By Orthogonality lemma applied to \( \Gamma^{+1} \vdash \text{box } e_{i}^{0} : [t_{i}^{0}] \), we get \( \Gamma^{+2} \vdash \text{box } e_{i}^{0} : [t_{i}^{0}] \).
By Substitution lemma applied to \( \Gamma^{+2} \vdash \text{box } e_{i}^{0} : [t_{i}^{0}] \) and \( \Gamma^{+2} \vdash \{ x_{i} : [t_{i}^{0}] \} \vdash \epsilon : \langle t \rangle \), we get \( \Gamma^{+2} \vdash \{ x_{i} : = \text{box } e_{i}^{0} : \langle t \rangle \} \).
By IH applied to \( \Gamma^{+2} \vdash \{ x_{i} : = \text{box } e_{i}^{0} : \langle t \rangle \}^{0} \) and \( \epsilon_{1} : = \text{box } e_{i}^{0} : \langle t \rangle \), we get \( \Gamma^{+2} \vdash \{ x_{i} : = \text{box } e_{i}^{0} : \langle t \rangle \}^{0} \).
By Bracket rule applied to \( \Gamma^{+2} \vdash \langle \epsilon \rangle : \langle t \rangle \), we get \( \Gamma^{+2} \vdash \epsilon : t^{1} \).
By Demotion lemma applied to \( \Gamma^{+2} \vdash \epsilon : t^{1} \), we get \( \Gamma^{+1} \vdash \epsilon : t^{0} \).
- Run II: For space reason, we assume $R = \text{run } \varepsilon$ with $x_i = \varepsilon_1$ and $R' = \text{run } \varepsilon'$ with $x_i = \varepsilon'_1$.

\[ \frac{I^{n+1} + \varepsilon_i: [t_i]^0}{I^{n+2}, \{x_i:[t_i]^0\} + \varepsilon: \langle t \rangle^{n+1}} \quad \frac{R \quad R'}{I^{n+1} + \varepsilon'_1: [t_i]^0 + \varepsilon: \langle t \rangle^{n+1}} \]

- Escape I:

\[ \frac{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}}{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}} \quad \frac{\varepsilon \mapsto \varepsilon'}{\varepsilon \mapsto \varepsilon''} \quad \frac{\varepsilon'' \in V^{n+1}}{\varepsilon' \in V^{n+1}} \quad \frac{I^{n+1} + \varepsilon' : \langle t \rangle^{n+1}}{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}} \]

- Escape II:

\[ \frac{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}}{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}} \quad \frac{\varepsilon \mapsto \varepsilon'}{\varepsilon \mapsto \varepsilon''} \quad \frac{\varepsilon'' \in V^{n+1}}{\varepsilon' \in V^{n+1}} \quad \frac{I^{n+1} + \varepsilon' : \langle t \rangle^{n+1}}{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}} \]

- Bracket:

\[ \frac{I^{n+1} + \varepsilon: \langle t \rangle^{n+1}}{I^{n+1} + \langle t \rangle^{n+1}} \quad \frac{\varepsilon \mapsto \varepsilon'}{\varepsilon \mapsto \varepsilon''} \quad \frac{\varepsilon'' \in V^{n+1}}{\varepsilon' \in V^{n+1}} \quad \frac{I^{n+1} + \varepsilon' : \langle t \rangle^{n+1}}{I^{n+1} + \langle t \rangle^{n+1}} \]

- Box I: First, we apply the induction hypothesis once, then reconstruct the type judgment of the result:

\[ \frac{I^{n+1} + \varepsilon_1: [t_i]^0 \quad \varepsilon_i \mapsto \text{box } \varepsilon'_i}{I^{n+1} + \text{box } \varepsilon'_i: [t_i]^0} \]

By Orthogonality lemma applied to $I^{n+1} + \text{box } \varepsilon'_i: [t_i]^0$, we get $\{\} \vdash \text{box } \varepsilon'_i: [t_i]^0$.

By Substitution lemma applied to $\{\} \vdash \text{box } \varepsilon'_i: [t_i]^0$ and $\{x_i:[t_i]^0\} \vdash \varepsilon: t^0$, we get $\{\} \vdash \varepsilon[x_i := \text{box } \varepsilon'_i]: t^0$.

By Box rule applied to $\{\} \vdash \varepsilon[x_i := \text{box } \varepsilon'_i]: t^0$, we get $I^{n+1} \vdash \text{box } \varepsilon[x_i := \text{box } \varepsilon'_i]: t^0$.

- Box II: For space reason, we assume $B = \text{box } \varepsilon$ with $x_i = \varepsilon_1$ and $B' = \text{box } \varepsilon'$ with $x_i = \varepsilon'_1$.

\[ \frac{I^{n+1} + \varepsilon_i: [t_i]^0 \quad \varepsilon_i \mapsto \text{box } \varepsilon'_i}{I^{n+1} + \text{box } \varepsilon'_i: [t_i]^0} \frac{R \quad R'}{I^{n+1} + \varepsilon'_1: [t_i]^0 + \varepsilon: \langle t \rangle^{n+1}} \]
Unbox I:

\[ \Gamma^+ \vdash e : [t]^0 \quad \xRightarrow{\uparrow} \quad \boxempty e \vdash e' \quad \xRightarrow{\uparrow} \quad \Gamma^+ \vdash \boxempty e' : t^n \]

\[ \xRightarrow{\uparrow} \quad \Gamma^+ \vdash \text{unbox } e : t^n \quad \xRightarrow{\uparrow} \quad \Gamma^+ \vdash e' \rightarrow e'' \]

By Weakening lemma applied to \( \{ \} \vdash e' : t^0 \), we get \( \Gamma^+ \vdash e' : t^0 \). By IH applied to \( \Gamma^+ \vdash e' : t^0 \) and \( e' \rightarrow e'' \), we get \( \Gamma^+ \vdash e'' : t^0 \).

Unbox II:

\[ \xRightarrow{\uparrow} \quad \Gamma^+ \vdash e : [t]^0 \quad \xRightarrow{\uparrow} \quad \Gamma^+ \vdash \text{unbox } e : t^n \]

\[ \xRightarrow{\uparrow} \quad \Gamma^+ \vdash e' \rightarrow e'' \quad \xRightarrow{\uparrow} \quad \Gamma^+ \vdash e' \rightarrow e'' \]

Lemma 4 The judgment \( \Gamma \vdash^n e : c \rightarrow r \) is derivable if and only if the judgment \( (\pi\Gamma)^{+r} \vdash e : t^n \) is also derivable.

Remark 2 (Environment Restriction). Technically, we also assume \( \Gamma x = (t, n', r') \) imply \( r' \leq r \). The restriction “\( \Gamma x = (t, n', r') \) implies \( r' \leq r \)” is implicit in the original type system. In particular, \( r \) never decreases when we go from the conclusion of a type rule to its premise, thus, we never add a binding with an \( r \) higher than that of the judgement.

Proof: By structural induction over the first and the second derivations (to prove the implications in both directions). The different cases for each of the two derivations are one-to-one, and so we combine corresponding pairs and do the proof in both directions for each pair.

Lambda abstraction:

\[ v_\alpha \Downarrow \quad \Gamma, x : (t_1, n, r) \vdash^n e : t_2 \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash \lambda x . c : t_1 \rightarrow t_2 \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash c : t_2^n \]

Variables:

\[ \uparrow \quad \Gamma \vdash^n x : t, r \quad \xRightarrow{\alpha} \quad \Gamma \vdash^n x : t, r \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash x : t^n \]

Applications:

\[ \xRightarrow{\uparrow} \quad \Gamma \vdash^n e_1 : t_1 \rightarrow t_2, r \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash e_1 : t_1 \rightarrow t_2^n \]

\[ \xRightarrow{\uparrow} \quad \Gamma \vdash^n e_2 : t_1, r \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash e_2 : t_1^n \]

\[ \xRightarrow{\uparrow} \quad \Gamma \vdash^n e_1 e_2 : t_2, r \quad \xRightarrow{\alpha} \quad (\pi\Gamma)^{+r} \vdash e_1 e_2 : t_2^n \]
B Proofs of the Embedding of \( \lambda \)

B.1 Strong Normalisation

The proof follows Harper, Honsell, and Plotkin [7]. We present a translation into the simply-typed \( \lambda \)-calculus. This translation essentially ignores all details of the original expression, except for the cases when there is a Box-redex, in which case the translation produces a term with a \( \beta \)-redex. We translate raw terms of the closed fragment into terms of the simply typed \( \lambda \)-calculus with one base type \( o \) and additional constants

\[
\text{app} : o \rightarrow (o \rightarrow o) \\
\text{abs} : (o \rightarrow o) \rightarrow o
\]

The mapping for types is simply: \( t^* = o \). The mapping for terms is

\[
\begin{align*}
x^* &= x \\
(\epsilon_1 \, \epsilon_2)^* &= \text{app} \, \epsilon_1^* \, \epsilon_2^* \\
(\lambda x. \epsilon)^* &= \text{abs} \, (\lambda x : o. \epsilon^*) \\
(\text{unbox} \, \epsilon)^* &= (\lambda x : o. \epsilon)^* \\
(\text{box} \, \epsilon)^* &= \epsilon^* \\
(\text{box} \, e \text{ with } x_i = \epsilon_i)^* &= (\lambda x_1 : o \ldots (\lambda x_n : o. \epsilon)^* \epsilon_n^* \ldots) \, e_1^*
\end{align*}
\]

Next, we show that \( \epsilon_1 \rightarrow_{\text{box}} \epsilon_2 \) implies \( \epsilon_1^* \rightarrow_{\text{beta}} \epsilon_2^* \).

First, we make sure that the terms produced by the translation are indeed in the simply typed \( \lambda \)-calculus, in particular, that they are well-typed.

**Lemma 9 (Star Well-Typedness).** If \( \epsilon \in E_{\text{Closed}} \) and \( \text{FV}(\epsilon) \subseteq \{x_i\} \), then \( \{x_i : o\} \vdash \epsilon^* : o \).

**Proof.** By induction over the typing derivation. \( \square \)
Then, we prove the following distributivity property that will be needed to prove strong normalization.

Lemma 10 (Star Substitution). $e^*[x := e'] \equiv (e[x := e'])^*$.

Proof. By induction on the structure of $e$. One issue arising in this proof is the need for substitution to propagate into the body of Box expressions, which is not needed when we are dealing only with well-typed terms. □

Lemma 11 (Lockstep). If $e_1 \to_{Box} e_2$ then $e_1^* \to_{\beta} e_2^*$.

Proof. By induction on the derivation of $\to_{Box}$. Most cases are treated by an application of the induction hypothesis, with the exception of the two cases when a Box reduction takes place explicitly. The interesting case is when

$$e_1 \equiv \boxed{e'} \text{ with } x_i = e_i, x = \boxed{e}, x_j = e_j$$

and then

$$e_2 \equiv \boxed{e'[x := \boxed{e}]} \text{ with } x_i = e_i, x_j = e_j$$

we proceed as follows:

$$e_1^* = (\lambda x_j; a, \ldots (\lambda x_i; a, \lambda x_{i'}; a, e') e_{i'}^* \ldots) (\boxed{e})^* \ldots e_{j_n}^*$$

$$\to_{\beta} (\lambda x_j; a, \ldots (\lambda x_i; a, e'[x := (\boxed{e})^*]) e_{i'}^* [x := (\boxed{e})^*] \ldots \ldots) e_{j_n}^*$$

By Barendregt’s convention for variable names $x \notin \text{FV}(e_i)$

$$= (\lambda x_j; a, \ldots (\lambda x_{i'}; a, e'[x := (\boxed{e})^*]) e_{i'}^* \ldots \ldots) e_{j_n}^*$$

By Star Distribution lemma

$$= (\lambda x_j; a, \ldots (\lambda x_{i'}; a, (e'[x := \boxed{e}]^*)) e_{i'}^* \ldots \ldots) e_{j_n}^*$$

$$= \boxed{e'[x := \boxed{e}]} \text{ with } x_i = e_i, x_j = e_j$$

Finally, we can prove our main lemma

Lemma 12 (Strong Normalization). $\to_{Box}$ is strongly normalizing.

Proof. From Lockstep, and because the simply-typed $\lambda$-calculus is strongly normalizing, it follows that Box-reduction is likewise. □

B.2 Confluence

Lemma 13. The $\to_{Box}$ relation is sub-commutative.

Proof. The only critical pair is the second rule with itself. It easy to check it is convergent and close in one step.

Corollary 1. The $\to_{Box}$ relation is confluent.

Proof. Direct consequence of the previous lemma.
B.3 Compatibility w.r.t the semantics of AIM box fragment

We use the notion of parallel reduction to deal with duplication of redices caused by substitution. Roughly speaking, parallel reduction is defined by the simultaneous reduction of a set of redices in a term.

Definition 7 (Parallel Reduction). \( \Rightarrow_{\text{box}} \) is formally defined as follows:

\[
\begin{align*}
\text{unbox} & \quad \Rightarrow_{\text{box}} \\
\text{box} & \quad \Rightarrow_{\text{box}} \\
\end{align*}
\]

\[e \Rightarrow_{\text{box}} e' \]

\[a \Rightarrow_{\text{box}} a' \quad b \Rightarrow_{\text{box}} b' \quad c_i \Rightarrow_{\text{box}} c_i' \]

\[\text{box } a \text{ with } x_i = c_i, x = \text{box } b \Rightarrow_{\text{box}} \text{box } a'[x_i = \text{box } b'] \text{ with } x_i = c_i' \]

and the usual distribution rules for all constructs including the \text{unbox} and box constructs. Constants and variables are axioms.

Lemma 14 (Parallel Reduction and Substitution). If \( a \Rightarrow_{\text{box}} a' \) and \( b \Rightarrow_{\text{box}} b' \) then \( a[x_i = b] \Rightarrow_{\text{box}} a'[x_i = b'] \).

Proof. Induction on the structure of \( a \).

Lemma 15 (Soundness of \( \Rightarrow_{\text{box}} \)). If \( e_1 \Rightarrow_{\text{box}} e_2 \) and \( e_2 \Rightarrow_0 v_2 \) then there exists \( v_1 \) such that \( e_1 \Rightarrow_0 v_1 \) and \( v_1 \Rightarrow_{\text{box}} v_2 \).

Proof. Induction on the lexicographic composition of the derivation of \( e_2 \Rightarrow_0 v_2 \) and the derivation of \( e_1 \Rightarrow_{\text{box}} e_2 \).

If one of the redices of \( e_1 \) is at the root then two cases are possible:

1. If \( e_1 = \text{unbox } e \) then \( e_1 \Rightarrow_{\text{box}} e \) and \( e \Rightarrow_{\text{box}} e_2 \).

   By applying IH to \( e_2 \Rightarrow_0 v_2 \) and \( e \Rightarrow_{\text{box}} e_2 \), we derive \( e \Rightarrow_0 v_1 \) and \( v_1 \Rightarrow_{\text{box}} v_2 \). Thus, we have

\[
\begin{align*}
\text{box } e & \Rightarrow_0 \text{box } e \quad e \Rightarrow_0 v_1 \\
\text{unbox } e & \Rightarrow_0 v_1
\end{align*}
\]

2. If \( e_1 = \text{box } a \) with \( x_i = c_i, x = \text{box } b \) then \( e_2 = \text{box } a'[x_i = \text{box } b'] \) with \( x_i = c_i' \), where \( a \Rightarrow_{\text{box}} a' \), \( b \Rightarrow_{\text{box}} b' \), and \( c_i \Rightarrow_{\text{box}} c_i' \). By the semantics of box, we know that:

\[
\begin{align*}
\text{box } a'[x_i = \text{box } b'] \text{ with } x_i = c_i' \Rightarrow_0 \text{box } a'[x_i = \text{box } b'][x_i = v_i]
\end{align*}
\]

By applying the IH to \( c_i' \Rightarrow_0 v_i' \) and \( c_i \Rightarrow_{\text{box}} c_i' \), we derive \( c_i \Rightarrow_0 v_i \) and \( v_i \Rightarrow_{\text{box}} v_i' \). Thus
If all redices are subterms of \( \epsilon_1 \).

- If \( \epsilon_1 = \lambda x. e \) and \( \epsilon_2 = \lambda x. e' \) such that \( e \mapsto_{\text{box}} e' \). It is easy to check that \( v_1 = \lambda x.e \) and \( v_2 = \lambda x.e' \). Hence, \( v_1 \mapsto_{\text{box}} v_2 \).

- If \( \epsilon_1 = a b \) and \( \epsilon_2 = a' b' \) such that \( a \mapsto_{\text{box}} a' \) and \( b \mapsto_{\text{box}} b' \), we know that

\[
\frac{\epsilon' \mapsto \lambda x.e' \quad b' \mapsto v' \quad \epsilon'[x:=v'] \mapsto v_2}{a' \quad b' \mapsto v_2}
\]

By applying IH to \( \epsilon' \mapsto \lambda x.e' \) and \( a \mapsto_{\text{box}} a' \) and then to \( b' \mapsto v' \) and \( b \mapsto_{\text{box}} b' \), we derive \( a \mapsto \lambda x.e \) and \( \lambda x.e \mapsto_{\text{box}} \lambda x.e' \), and \( b \mapsto v \) and \( v \mapsto_{\text{box}} v' \).

By lemma 14, \( \epsilon[x:=v] \mapsto_{\text{box}} \epsilon'[x:=v'] \). Thus, we can apply the induction hypothesis to \( \epsilon'[x:=v] \mapsto_{\text{box}} v_2 \) and \( \epsilon[x:=v] \mapsto_{\text{box}} \epsilon'[x:=v] \) to derive \( \epsilon[x:=v] \mapsto_{\text{box}} v_1 \) and \( v_1 \mapsto_{\text{box}} v_2 \). Hence,

\[
\frac{a \mapsto \lambda x.e \quad b \mapsto v \quad \epsilon[x:=v'] \mapsto v_1}{a b \mapsto v_1}
\]

- If \( \epsilon_1 = \text{box } a \) with \( x_i = b_i \) and \( \epsilon_2 = \text{box } a' \) with \( x_i = b'_i \) such that \( a \mapsto_{\text{box}} a' \) and \( b_i \mapsto_{\text{box}} b'_i \). We know that,

\[
\frac{b'_i \mapsto v_i}{\epsilon'_i \mapsto \text{box } a'[x_i:=v'_i]}
\]

Notice that \( v_2 = \text{box } a'[x_i:=v'_i] \). By applying IH to \( b'_i \mapsto v_i \) and \( b_i \mapsto_{\text{box}} b'_i \), we derive \( b_i \mapsto_{\text{box}} v_i \) and \( v_i \mapsto_{\text{box}} v'_i \). By Lemma 14, we have \( a[x:=v_i] \mapsto_{\text{box}} a'[x:=v'_i] \) since \( a \mapsto_{\text{box}} a' \) and \( v_i \mapsto_{\text{box}} v'_i \). Thus, \( \text{box } a[x:=v_i] \mapsto_{\text{box}} \text{box } a'[x:=v'_i] \) and

\[
\frac{b_i \mapsto v_i}{\text{box } a \mapsto v_i}
\]

- \( \epsilon_1 = \unbox e \) and \( \epsilon_2 = \unbox e' \) such that \( e \mapsto_{\text{box}} e' \). We have

\[
\frac{e' \mapsto \text{box } a' \quad a' \mapsto v_2}{\unbox e' \mapsto v_2}
\]
by applying IH to $\epsilon' \leftrightarrow^0 \bbox a'$ and $\epsilon \leftarrow^\bot_{\boxdot x} \epsilon'$, we derive $\epsilon \leftrightarrow^0 \bbox a$ and $a \leftarrow^\bot_{\boxdot x} \bbox a'$. Hence, $a \leftarrow^\bot_{\boxdot x} \bbox a'$. Thus IH applies to $a' \leftrightarrow^0 v_2$ and $a \leftarrow^\bot_{\boxdot x} a'$, to derive $a \leftrightarrow v_1$ and $v_1 \leftarrow^\bot_{\boxdot x} v_2$. We conclude by

\[
\frac{\epsilon \leftrightarrow^0 \bbox a \quad a \leftrightarrow^0 v_2}{\bbox \epsilon \leftrightarrow^0 v_2}
\]

Corollary 2. If $\epsilon_1 \rightarrow^\bot_{\boxdot x} \epsilon_2$ and $\epsilon_2 \leftrightarrow^0 v_2$ then there exists $v_1$ such that $\epsilon_1 \leftrightarrow^0 v_1$ and $v_1 \leftarrow^\bot_{\boxdot x} v_2$.

*Proof.* Use the precedent lemma, and the facts that $\rightarrow^\bot_{\boxdot x} \subset \Rightarrow^\bot_{\boxdot x}$ for the hypothesis and $\Rightarrow^\bot_{\boxdot x} \subset \leftarrow^\bot$ for the conclusion of this corollary.

### B.4 Substitutivity

**Lemma 16 (Weakening).** If $\text{FV}(\epsilon) \cap S = \emptyset$ then $\epsilon^\square T = \epsilon^\square T \cup S$.

*Proof.* By induction on the structure of $\epsilon$. Box case requires some simple set logic. □

**Lemma 8** Given a closed term $\epsilon_0 \in E_{\square}$ the following properties hold:

- $\epsilon^\square X [y := e_0^\square] \equiv (\epsilon [y := e_0])^\square X$, provided $y \notin X$.
- $\epsilon^\square u(x) [x := e_0] \rightarrow_{\boxdot x} (\epsilon [x := e_0])^\square X$.

*Proof.* By induction over the structure of $\epsilon$, for both properties. The interesting case—in both proofs—is Box, which requires explicit reasoning about sets of free variables. Some Variable sub-cases require Weakening (Lemma 16). □

### B.5 Main Result: Embedding of $\lambda^\square$

**Proposition 6** If $\epsilon \in E_{\square}$ is closed and $\epsilon \leftrightarrow_{\square} v$ is derivable in $\lambda^\square$, then there exists $\epsilon'$ such that $\epsilon^\square v \leftrightarrow^0 \epsilon'$ and $\epsilon' \rightarrow_{\boxdot x} v^\square$.

*Proof.* Induction on the derivation of $\epsilon \leftrightarrow_{\square} v$.

- If $\epsilon = x$, vacuous.
- If $\epsilon = \bbox e_1$ then we have: $\bbox e_1 \leftrightarrow_{\square} \bbox e_1$ and $\bbox e^\square \bbox e^\square$, hence $\bbox e^\square \leftrightarrow^0 \bbox e^\square$.
- If $\epsilon = e_1 e_2$, we have

\[
\begin{align*}
e_1 & \leftrightarrow_{\square} \lambda x.\epsilon' \\
e_2 & \leftrightarrow_{\square} \epsilon' \\
e'[x := e'] & \leftrightarrow_{\square} v
\end{align*}
\]

\[
\frac{e_1 e_2 \leftrightarrow_{\square} v}{e_1 e_2 \leftrightarrow_{\square} v}
\]
By applying respectively (IH) to (1), (2) and (3), we derive $e_1^\Box \vdash \lambda x.e''$, $e_2^\Box \vdash v''$, and $e'[x := \nu]^\Box \vdash v_1$ such that $\lambda x.e'' \vdash_{b,ox} (\lambda x.e')^\Box$, $v'' \vdash_{b,ox} (v')^\Box$, and $v_1 \vdash_{b,ox} v^\Box$.

By lemma 8, we have $e'[x := v]^\Box \vdash (e')^\Box[v] \vdash (e')^\Box$. Since $e'' \vdash_{b,ox} (e')^\Box$ and $v'' \vdash_{b,ox} (v')^\Box$, we have $e''[x := v'] \vdash_{b,ox} (e')^\Box[v] \vdash (e')^\Box$.

Hence, we apply the compatibility lemma to derive: $e''[x := v'] \vdash v'_1$ and $v'_1 \vdash_{b,ox} v'$.

Thus, $v'_1 \vdash_{b,ox} v^\Box$ and

$$
\frac{e_1^\Box \vdash \lambda x.e'' \quad e_2^\Box \vdash v'' \quad e''[x := \nu]^\Box \vdash v_1}{(e_1 e_2)^\Box \vdash v'_1}
$$

- If $e = \text{let box } x = e_1 \text{ in } e_2$, we have

$$
e_1 \vdash_{\Box} \text{box } e_3 \quad \text{(1)}
$$
$$e_2[x := e_3] \vdash_{\Box} v' \quad \text{(2)}
$$

let box $x = e_1$ in $e_2 \vdash_{\Box} v$.

By applying respectively (IH) to (1) and (2), we derive: $e_1^\Box \vdash v_1$ and $e_2[x := e_3]^\Box \vdash v$ such that $v_1 \vdash_{b,ox} \text{box } e_3$ and $v' \vdash_{b,ox} v^\Box$.

Since $v_1 \in V^\Box$, rewrite to box $e_3^\Box$ then $v_1 \equiv \text{box } e_4$.

By hypothesis and typing rule we know $\{x; \Box t', \emptyset \vdash e_2; \Box x\}$. Hence, we can apply property 2 of Lemma 8, we have:

$$e_2[x := \text{box } e_4] \vdash_{b,ox} e_2[x := \text{box } e_3]^\Box \vdash_{b,ox} e_2[x := e_3]^\Box$$

By compatibility of $\vdash_{b,ox}$, we have: $e_2^\Box[x := \text{box } e_4] \vdash v'$ and $v'' \vdash_{b,ox} v'$.

Thus, $v'' \vdash_{b,ox} v$ and

$$
\frac{e_1^\Box \vdash \text{box } e_4 \quad e_2[x := \text{box } e_4]^\Box \vdash v''}{\text{let box } x = e_1 \text{ in } e_2^\Box \vdash v''}
$$