On the Runtime Complexity of Type-Directed Unboxing

Yasuhiko Minamide
nan@kurims.kyoto-u.ac.jp

Jacques Garrigue
garrigue@kurims.kyoto-u.ac.jp

Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606-8502, Japan

Abstract

Avoiding boxing when representing native objects is essential for the efficient compilation of any programming language. For polymorphic languages this task is difficult, but several schemes have been proposed that remove boxing on the basis of type information. Leroy's type-directed unboxing transformation is one of them. One of its nicest properties is that it relies only on visible types, which makes it compatible with separate compilation. However it has been noticed that it is not safe both in terms of time and space complexity —*i.e.* transforming a program may raise its complexity. We propose a refinement of this transformation, still relying only on visible types, and prove that it satisfies the safety condition for time complexity. The proof is an extension of the usual logical relation method, in which correctness and safety are proved simultaneously.

1 Introduction

Compared to explicitly typed first order traditional languages, polymorphically typed functional programming languages have to face a number of challenges for their efficient compilation. One of them is choosing the right representation for data whereas, due to polymorphism, its actual type cannot be completely known at compile time.

In dynamically typed languages like Lisp, the basic approach to data polymorphism has been to make the representation homogeneous, that is to cover differences between various data types by making them all fit into a uniform representation. Then polymorphism is not a problem since one knows about the representation of objects even without knowing their actual types. This approach is simple, but is also utterly inefficient since it means that any data which cannot fit directly into this representation, being too large for instance, has to be coerced, generally by allocating the real data somewhere (e.g. in the heap), and by using a handle (e.g. a pointer) in place of it. Any access to the real data will then require a level of indirection, particularly costly when compared with directly passing the data via a register.

The above method, *i.e.* allocating data structures in the heap and passing a pointer instead, is called boxing, and is even used in statically typed polymorphic languages. Despite their static typing, the presence of polymorphic functions makes difficult the use of heterogeneous data representation. In the context of polymorphic languages, choosing more efficient heterogeneous representations is called unboxing.

There are many approaches to perform unboxing, but we will here concern ourselves with only one of them, type-directed unboxing transformation. The idea, formalized by Leroy [8], is to actually make data representation fit its static type, exactly like it would be done in a monomorphically typed language. However, since polymorphism means that the same value may have different static types according to its context, coercions between different representations are needed. This optimization's main advantage is that it is both cheap and effective. Since it relies only on types, no special analysis is needed, and separate compilation is still possible. Simultaneously, types capture the whole control flow of a program, making it possible to apply the transformation almost everywhere.

While this transformation has been proved correct in terms of denotational semantics, it is known that it is incorrect in terms of complexity: optimizing may result in raising the complexity order of a program. Counter examples were found after a similar optimization was introduced in a publicly released compiler [20].

We think that this fact reveals the need for a formal treatment of the complexity induced by program transformation based optimization techniques. In general such techniques are only proved correct in terms of semantics, but not in terms of complexity. Their runtime behavior is only demonstrated by benchmarks, which tell about their normal case behavior, but nothing about worst case.

As an example of such a treatment, we present here a refinement of Leroy's type-directed unboxing transformation, but this time we also give a proof that this new transformation preserves the time complexity of programs. Notice that the transformation we propose here is not guaranteed to improve the performance of programs (actually its performance is worse than Leroy's in most cases), but only not to raise their complexity order. The results given here are the-

To appear in the proceedings of the 1998 International Conference on Functional Programming (ICFP).

oretical, but we believe that our approach and proof method can be extended to more efficient optimizations, making it possible to have both performance and theoretical results.

This paper is organized as follows. We start with an informal discussion describing type-directed unboxing and its pitfall, then introducing our approach. In Section 3 we define the language we will use in the rest of the paper. We recall Leroy's approach in Section 4. Our new transformation and its proof of correctness are given in Section 5. Finally we review related works, and give directions for future work. Detailed proofs are included in Appendix.

2 Informal Discussion

2.1 Type-Directed Unboxing Based on Coercions

Compilation is a translation from a source language to a lower-level target language. It is however important to consider that this target language has in fact more structure than a simple assembly language. That is, for a compiled program to run correctly, all compiled code and data must not only be syntactically correct, but must also share the same conventions.

Let us consider a function $f_{\mathsf{real}} : \mathit{real} \to \mathit{real}$. In monomorphic languages, this function will be sensibly implemented as a procedure taking its argument and returning its result in a floating point register. Let us suppose we do the same thing in a polymorphic language. Notice first that in polymorphic languages, a polymorphic function $f_{\mathsf{poly}} : \forall\alpha.\alpha \to \alpha$ must use a more generic calling convention than f_{real} since nothing is known about the runtime type for α. Using boxing, f_{poly} will take boxed values which can be represented in one word, and are in the best case passed through a general purpose register. However, it may happen that at runtime f_{poly} is used with type $\mathit{real} \to \mathit{real}$ by instantiation.

We have now two possible calling conventions for the type $\mathit{real} \to \mathit{real}$: unboxed real in a floating point register for f_{real}, or boxed real through a general purpose register for f_{poly}. Raw type information is not enough to choose the right calling convention. For this very reason, many implementations of polymorphic languages have just chosen a single calling convention, that of f_{poly}, which is universal but requires all data types that do not fit in one word to be always boxed.

In order to solve this problem, Leroy [8] has shown that by introducing coercions when a polymorphic function is specialized, one can have a function always match the calling convention of its type. His transformation acts on the program annotated with polymorphic types, and inserts boxing and unboxing so that monomorphic parts of a program can use unboxed representations of values.

The key point here is to distinguish between boxed and unboxed versions of types: we have both unboxed *real* type and boxed *boxreal* type for floating point numbers; and consider coercions between them: `wrapreal` coerces *real* into *boxreal* and `unwrapreal` coerces *boxreal* into *real*. Now a function `f` of type $\forall\alpha.\alpha \to \alpha$ can be safely instantiated into $\mathit{boxreal} \to \mathit{boxreal}$, since *boxreal* uses the same calling convention as type variables. However if we want to give it the type $\mathit{real} \to \mathit{real}$, we have to introduce coercions where representations change:

```
wrap_{real→real}(f) : real -> real ≡
        fn x =>  unwrapreal(f(wrapreal x))
```

Thanks to the coercions, the calling convention of this new function matches its type.

Boxing and unboxing are necessary not only for specialization of polymorphic functions, but also for constructing and accessing some data types. It is not desirable, not even possible, to apply coercions to all data types: coercing a whole list is too expensive, and references cannot be coerced since one would have to make copies of them, changing their semantics. For such data constructors, values have to be coerced to their fully boxed representation before putting them in the constructor. Shao called these data types *incoercible* [19] and we follow his terminology. We give here the example of a function `g` of type $\mathit{real} \to \mathit{real}$ being coerced to boxed representation, before assigning it to a reference for instance.

```
wrap_{real→real}(g) : boxreal -> boxreal ≡
    fn x =>  wrapreal(g(unwrapreal x))
```

This idea of introducing coercions according to types can be extended to full ML and is implemented in [8] and [20]. Benchmarks show that it works very well in general. However, recently several researchers have noticed undesirable behavior of some programs when compiled by this method.

2.2 Problem

Let us put it short: the type-directed unboxing transformation changes the complexity of some programs with respect to time and space. The problem stems from the behavior of boxing and unboxing on functional values. When acting on non-functional values, boxing and unboxing behave as inverses, so that we have the following property: `unwrap(wrap(v))` and `wrap(unwrap(v))` are both evaluated to `v`. However, this does not hold for function types. For example, by applying boxing and unboxing to `f` of type $\mathit{real} \to \mathit{real}$ we obtain the following function.

```
unwrap_{real→real}(wrap_{real→real}(f)) ≡
 fn x => unwrapreal((fn y =>
                     wrapreal(f(unwrapreal y)))
                     (wrapreal x))
```

This expression is not evaluated to the value of `f`, but to a value wrapped with two lambda abstractions. From a denotational semantics point of view, they have the same behavior. However, this changes the behavior of programs with respect to time and space. This situation occurs in several ways in programs obtained by Leroy's transformation.

Let us consider the specialization of the polymorphic identity function to some monomorphic type. If we coerce the polymorphic identity function to the type $(\mathit{real} \to \mathit{real}) \to (\mathit{real} \to \mathit{real})$, we obtain the following function

```
id_{real→real} ≡
  fn x => unwrap_{real→real} (id (wrap_{real→real}(x)))
```

Applying this identity function to a function `f` : $\mathit{real} \to \mathit{real}$ results in applying wrapping and unwrapping to `f`; we obtain `unwrap`$_{\mathit{real}\to\mathit{real}}$`(wrap`$_{\mathit{real}\to\mathit{real}}$`(f))`. Once we have noticed this effect of boxing and unboxing, we can easily construct a program where the type-directed unboxing transformation changes the complexity. Here is such a program.

```
let val id = fn x => x
    fun iter f 0 = 0.0
      | iter f n = f (iter (id f) (n - 1))
in
    iter (fn x => x + 1.0) n
end
```

By applying type-directed unboxing `id` is replaced by $\texttt{id}_{real \to real}$. Thus each recursive call wraps the function `f` with coercions and closures and increases its size. All these wrappings make the complexity of `f` change from constant to linear in n, so that program will run in order n^2 after transformation, while its naive execution would have resulted in linear time. The same problem also occurs with respect to space. The direct execution of the program requires only a constant amount of space independent of n, though it requires space proportional to n if we use Leroy's representation. This violates the rule of safety for space complexity [1]. In this specific example, coercions can be eliminated by partially evaluating $\texttt{(id (wrap}_{real \to real}\ \texttt{f))}$, but in general this is not possible: the code of `id` might be unknown —defined in another compilation unit (this is an advantage of type-directed unboxing), or abstracted as parameter to a module—, or the reduction might increase the size of a program in an unacceptable way (e.g. if `id` contains side-effects).

The same problem occurs when we use incoercible data types such as references and lists. Let us consider the program: `!(ref f)`. Before creating the reference cell `f` must be coerced into fully boxed representation and after dereference it must be coerced back into unboxed representation. Thus after the execution of this program we obtain not `f` but the following value.

$\texttt{unwrap}_{real \to real}\texttt{(wrap}_{real \to real}\texttt{(f))} \equiv$
```
fn x =>
  unwrapreal((fn y => wrapreal(f(unwrapreal y)))
             (wrapreal x))
```

The same problem occurs for a pair of dereference and assignment `r := !r`. In practice this situation occurs quite frequently and may cause some programs to abort unexpectedly running out of memory.

2.3 Our Approach

If we trace back what is happening in the above examples, we see that we have an unlimited number of coercions applied successively to the original function. The natural answer is to cut such infinite chains.

The way we do cut these infinite chains is by always keeping an untouched reference version of the original function, together with the specialized version. From this reference version, we can obtain any degree of specialization of the function by applying a single coercion. That is, we can obtain any form of our function by applying at most two coercions to the original code: one to build the reference version from the original function, and one to build the specialized function from the reference version.

There is still a question: what should be this reference version? The simplest answer is also the only possible one: the fully boxed form of the function. Indeed if we do not want to use any runtime information, there is no other reasonable way to define a reference version, exactly for the same reasons some compilers choose to use only the fully boxed form.

In our scheme, there are two different representations for a function: the *generic* representation is just its fully boxed version, while *specialized* representations are pairs of a specialized version and the fully boxed version. Going back to our running example, the specialized representation of `f` is a pair of functions $\texttt{(f}_1\texttt{,f}_2\texttt{)}$ where $\texttt{f}_1$ and $\texttt{f}_2$ have type $real \to real$ and $boxreal \to boxreal$ respectively. Application is performed by extracting the first component of the pair before applying it to an argument `v`: $(\pi_1(\texttt{f}_1\texttt{,f}_2))\texttt{v}$. Using this specialized representation, the generic representation of a function can be obtained by just extracting the second component of the pair: $\pi_2(\texttt{f}_1\texttt{,f}_2)$. This makes wrapping a function particularly simple.

Conversely, from the generic representation, one can obtain a specialized version by applying coercions as before. Let $\texttt{f}_0$ be a function of type $boxreal \to boxreal$. The specialized representation is obtained by the following expression.

$\texttt{(fn x => unwrapreal(f}_0\texttt{(wrapreal x)), f}_0\texttt{)}$

Now the first component is wrapped by coercion and lambda abstraction. Our idea is summarized in Figure 1.

In this approach boxing and unboxing will not cause a function to be wrapped by coercion repeatedly, growing forever: every time we build a new specialized version, we start from the same untouched generic representation. As a result, our type-directed transformation preserves the complexity of programs.

The translation of our previous example using this representation of functions is shown in Figure 2. Details of the translation process will be explained in the formal development. Recursive calls of `iter` from its body of `iter` do not change the second component of `f` and pass the following value for `f`.

$\texttt{(unwrap}_{real \to real}\texttt{(wrap}_{real \to real}\texttt{(fn x => x + 1.0)),}$
$\texttt{wrap}_{real \to real}\texttt{(fn x => x + 1.0))}$

Although the first component of this value is wrapped in $\texttt{unwrap}_{real \to real}$ and $\texttt{wrap}_{real \to real}$, it remains the same for each recursive call. Hence the complexity of the program is still order n with respect to evaluation steps and it runs in a constant amount of memory.

This approach does not solve another problem of coercion based unboxing. This problem is related with space-complexity in an execution model using tail-call optimization. Let us see the following program.

```
fun apply (f, x) = f x
fun f x =
    if x < 0.0 then x else apply (f, x - 1.0)
```

Since `apply (f, x - 1.0)` is a tail call in `f` and `f x` a tail call in `apply`, it shall be possible to execute `f` in constant space. The type of `apply` is $(\alpha \to \beta) \times \alpha \to \beta$, but it is used at type $(real \to real) \times real \to real$. As a result, our transformation (as Leroy's) will insert coercions around `apply`'s call in `f`, making it a non-tail call. The transformed program will need linear space.

Still, this problem shall not be overstated. It may appear with some special tail calls, like the one above, but it does not appear with tail recursion, at least in a framework without polymorphic recursion. That is, in usual ML recursion, all recursively defined functions are monomorphic, and no coercion needs to be inserted in recursive calls. Recursive tail calls are correctly transformed into tail calls. For this reason we believe that our transformation is space-safe, even in a model containing tail-recursion optimization.

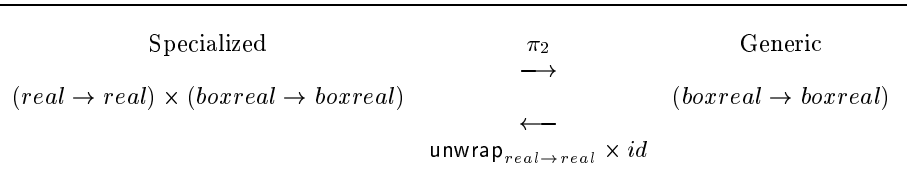

Figure 1: Represention of Functions

```
let val id = (fn x => x, fn x => x)
    fun iter f 0 = 0.0
      | iter f n =  π1(f) (iter (let val f' = π2(id) (π2(f))
                                 in
                                     (unwrap_{real→real}(f'), f')
                                 end)
                                (n - 1))
in
    iter (fn x => x + 1.0, wrap_{real→real}(fn x => x + 1.0)) n
end
```

Figure 2: Example of Translation

3 Source and Target Languages

In this section we define a language to formalize our type-directed unboxing transformation and discuss the relation between the complexity of source programs and their translations. In order to make possible discussing the complexity of programs, we adopt an operational semantics which counts evaluation steps.

The source language is basically the core language of ML. Our integers require boxing. Effects of incoercible data types such as reference or lists are simulated by a single incoercible type τ *pack* with constructor pack and destructor unpack [19].

$$\begin{array}{rcl} \tau & ::= & \alpha \mid int \mid \tau \to \tau \mid \tau \times \tau \mid \tau\ pack \\ \sigma & ::= & \forall \alpha_1 \ldots \alpha_n.\tau \\ i & ::= & 0 \mid 1 \mid 2 \mid \ldots \\ e & ::= & i \mid x \mid ee \mid \lambda x.e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid (e,e) \mid \\ & & \pi_1(e) \mid \pi_2(e) \mid \mathsf{pack}(e) \mid \mathsf{unpack}(e) \mid \\ & & \mathtt{iter}(e,e,e) \end{array}$$

Instead of recursive definitions, which would make complexity semantics rather involved, we include $\mathtt{iter}(n, v, f)$ which applies the function f to v repeatedly n times. This is enough to write critical examples and simplifies our presentation. For instance, translation using Leroy's method would change the complexity for both of the following programs (they correspond to the two examples presented above).

$$\mathtt{let}\ id = \lambda x.x\ \mathtt{in}$$
$$\mathtt{iter}(n, (0, \lambda x.x), \lambda y.(\pi_2(y)\pi_1(y), id(\pi_2(y))))$$
$$\mathtt{iter}(n, (0, \lambda x.x), \lambda y.(\pi_2(y)\pi_1(y), \mathsf{unpack}(\mathsf{pack}(\pi_2(y)))))$$

The target language is extended with the type *boxint* for boxed integers and the coercions $\mathsf{wrapint}(e)$ and $\mathsf{unwrapint}(e)$ between *int* and *boxint*.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid int \mid \tau \to \tau \mid \tau \times \tau \mid \tau\ pack \mid boxint \\ e & ::= & i \mid x \mid ee \mid \lambda x.e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \\ & & (e,e) \mid \pi_1(e) \mid \pi_2(e) \mid \mathsf{pack}(e) \mid \mathsf{unpack}(e) \mid \\ & & \mathsf{wrapint}(e) \mid \mathsf{unwrapint}(e) \mid \mathtt{iter}(e,e,e) \end{array}$$

The source language is considered as a subset of the target language. Thus we will present the type system and operational semantics only for the target language.

To this language we apply Milner's type discipline [10] and use the syntax-directed typing rules presented in [3]. Typing judgments have the following form:

$$\Gamma \vdash e : \tau$$

where Γ is a finite mapping from variables to polytypes. Important rules of the type system are shown in Figure 3. Other rules are standard, and we omit them to concentrate on essential parts.

We define the operational semantics of the source and target languages by natural semantics. For integers we consider unboxed integers $\underline{i}$ and boxed integers $\overline{i}$. The values are defined as follows:

$$v ::= \underline{i} \mid \overline{i} \mid \langle\!\langle \gamma, x, e \rangle\!\rangle \mid (v, v) \mid \langle\!\langle v \rangle\!\rangle$$

where γ is an *environment*, *i.e.* a finite map from variables to values, $\langle\!\langle \gamma, x, e \rangle\!\rangle$ is a closure with an environment γ, and $\langle\!\langle v \rangle\!\rangle$ is a value of type τ *pack*. Then the operational semantics is defined as the following relation:

$$\gamma \vdash e \downarrow_n v$$

The rules are given in Figure 4. Their meaning is standard except for the subscript n. This subscript indicates the number of evaluation steps needed to obtain the value v. The definition of evaluation of $\mathtt{iter}(e_1, e_2, e_3)$ requires two auxiliary rules with judgments of the form $\vdash_i v_1, v_2 \downarrow_n v$. This

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{pack}(e) : \tau\ pack} \qquad \frac{\Gamma \vdash e : \tau\ pack}{\Gamma \vdash \mathsf{unpack}(e) : \tau} \qquad \frac{\Gamma \vdash e : boxint}{\Gamma \vdash \mathsf{unwrapint}(e) : int} \qquad \frac{\Gamma \vdash e : int}{\Gamma \vdash \mathsf{wrapint}(e) : boxint}$$

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \quad Dom(\rho) = \{\alpha_1, \ldots, \alpha_n\}}{\Gamma \vdash x : \rho(\tau)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : Clos(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \to \tau}{\Gamma \vdash \mathtt{iter}(e_1, e_2, e_3) : \tau}$$

Figure 3: Typing Rules

$$\gamma \vdash x \downarrow_1 \gamma(x) \qquad \gamma \vdash i \downarrow_1 \underline{i} \qquad \frac{\gamma \vdash e_1 \downarrow_n v_1 \quad \gamma \vdash e_2 \downarrow_m v_2}{\gamma \vdash (e_1, e_2) \downarrow_{n+m+1} (v_1, v_2)} \qquad \frac{\gamma \vdash e \downarrow_n (v_1, v_2)}{\gamma \vdash \pi_i(e) \downarrow_{n+1} v_i}$$

$$\gamma \vdash \lambda x.e \downarrow_1 \langle\!\langle \gamma, x, e \rangle\!\rangle \qquad \frac{\gamma \vdash e_1 \downarrow_l \langle\!\langle \gamma', x, e \rangle\!\rangle \quad \gamma \vdash e_2 \downarrow_m v_2 \quad \gamma'[v_2/x] \vdash e \downarrow_n v}{\gamma \vdash e_1 e_2 \downarrow_{l+n+m+1} v}$$

$$\frac{\gamma \vdash e \downarrow_n \underline{i}}{\gamma \vdash \mathsf{wrapint}(e) \downarrow_{n+1} \overline{i}} \qquad \frac{\gamma \vdash e \downarrow_n \overline{i}}{\gamma \vdash \mathsf{unwrapint}(e) \downarrow_{n+1} \underline{i}} \qquad \frac{\gamma \vdash e \downarrow_n v}{\gamma \vdash \mathsf{pack}(e) \downarrow_{n+1} \langle\!\langle v \rangle\!\rangle} \qquad \frac{\gamma \vdash e \downarrow_n \langle\!\langle v \rangle\!\rangle}{\gamma \vdash \mathsf{unpack}(e) \downarrow_{n+1} v}$$

$$\frac{\gamma \vdash e_1 \downarrow_l v' \quad \gamma[v'/x] \vdash e_2 \downarrow_m v}{\gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \downarrow_{l+m+1} v} \qquad \frac{\gamma \vdash e_1 \downarrow_l \underline{i} \quad \gamma \vdash e_2 \downarrow_m v_2 \quad \gamma \vdash e_3 \downarrow_n v_3 \quad \vdash_i v_2, v_3 \downarrow_p v}{\gamma \vdash \mathtt{iter}(e_1, e_2, e_3) \downarrow_{l+m+n+p+1} v}$$

$$\vdash_0 v, v' \downarrow_1 v \qquad \frac{i \geq 1 \quad \vdash_{i-1} v_2, \langle\!\langle \gamma, x, e \rangle\!\rangle \downarrow_m v' \quad \gamma[v'/x] \vdash e \downarrow_n v}{\vdash_i v_2, \langle\!\langle \gamma, x, e \rangle\!\rangle \downarrow_{m+n+1} v}$$

Figure 4: Operational Semantics

relation means that v is obtained by applying v_1 to the closure v_2 repeatedly i times. We write $\gamma \vdash e \Downarrow_n v$ if $\gamma \vdash e \downarrow_k v$ for some $k \leq n$.

Semantic typing of values is defined as follows:

$\models \underline{i} : int$	
$\models \overline{i} : boxint$	
$\models (v_1, v_2) : \tau_1 \times \tau_2$	if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$.
$\models \langle\!\langle \gamma, x, e \rangle\!\rangle : \tau_1 \to \tau_2$	if there exists Γ such that $\models \gamma : \Gamma$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.
$\models \langle\!\langle v \rangle\!\rangle : \tau\ pack$	if $\models v : \tau$.

Then the standard type soundness holds for both source and target languages: if an expression has type τ and it is evaluated to v, then v has type τ.

4 Leroy's Type-Directed Unboxing

Before formalizing our approach we briefly review Leroy's type-directed unboxing transformation. For details see Leroy's paper [8].

First, we define two translations of types $|\cdot|$ and $[\cdot]$, where $|\tau|$ is the type of the specialized representation of τ and $[\tau]$ is the type of the generic representation of τ. A term of type τ in the source language is translated into a term of type $|\tau|$ in the target language.

$$\begin{array}{rcl} |\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\ |\tau_1 \times \tau_2| & = & |\tau_1| \times |\tau_2| \\ |\alpha| & = & \alpha \\ |\tau\ pack| & = & [\tau]\ pack \\ |int| & = & int \end{array}$$

$$\begin{array}{rcl} [\tau_1 \to \tau_2] & = & [\tau_1] \to [\tau_2] \\ [\tau_1 \times \tau_2] & = & [\tau_1] \times [\tau_2] \\ [\alpha] & = & \alpha \\ [\tau\ pack] & = & [\tau]\ pack \\ [int] & = & boxint \end{array}$$

The specialized representation of τ in $\tau\ pack$ is fully boxed since $\tau\ pack$ is an incoercible type.

We then define coercions between specialized and generic types: wrap_τ coerces a value of $|\tau|$ into $[\tau]$ and unwrap_τ coerces a value of $[\tau]$ into $|\tau|$.

$$\begin{array}{rcl} \mathsf{wrap}_\alpha(e) & = & e \\ \mathsf{wrap}_{int}(e) & = & \mathsf{wrapint}(e) \\ \mathsf{wrap}_{\tau_1 \to \tau_2}(e) & = & \lambda y.\mathsf{wrap}_{\tau_2}(e(\mathsf{unwrap}_{\tau_1}(y))) \\ \mathsf{wrap}_{\tau\ pack}(e) & = & e \\ \mathsf{wrap}_{\tau_1 \times \tau_2}(e) & = & \mathtt{let}\ x = e\ \mathtt{in} \\ & & (\mathsf{wrap}_{\tau_1}(\pi_1(x)), \mathsf{wrap}_{\tau_2}(\pi_2(x))) \end{array}$$

$$\begin{array}{rcl} \mathsf{unwrap}_\alpha(e) & = & e \\ \mathsf{unwrap}_{int}(e) & = & \mathsf{unwrapint}(e) \\ \mathsf{unwrap}_{\tau_1 \to \tau_2}(e) & = & \lambda y.\mathsf{unwrap}_{\tau_2}(e(\mathsf{wrap}_{\tau_1}(y))) \\ \mathsf{unwrap}_{\tau\ pack}(e) & = & e \\ \mathsf{unwrap}_{\tau_1 \times \tau_2}(e) & = & \mathtt{let}\ x = e\ \mathtt{in} \\ & & (\mathsf{unwrap}_{\tau_1}(\pi_1(x)), \mathsf{unwrap}_{\tau_2}(\pi_2(x))) \end{array}$$

Programs are translated by rules of the form $\Gamma \vdash e : \tau \rightsquigarrow e'$. The only case where something special happens is when a variable gets specialized:

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \quad Dom(\rho) = \{\alpha_1, \ldots, \alpha_n\}}{\Gamma \vdash x : \rho(\tau) \rightsquigarrow S_\rho(x : \tau)}$$

where specialization S_ρ is defined by mutual recursion with generalization G_ρ as follows:

$$\begin{array}{rcl}
S_\rho(e:\alpha) & = & \mathsf{unwrap}_{\rho(\alpha)}(e) \\
S_\rho(e:int) & = & e \\
S_\rho(e:\tau\ pack) & = & e \\
S_\rho(e:\tau_1 \to \tau_2) & = & \lambda x.S_\rho(e(G_\rho(x:\tau_1)):\tau_2) \\
S_\rho(e:\tau_1 \times \tau_2) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (S_\rho(\pi_1(x):\tau_1), S_\rho(\pi_2(x):\tau_2))
\end{array}$$

$$\begin{array}{rcl}
G_\rho(e:\alpha) & = & \mathsf{wrap}_{\rho(\alpha)}(e) \\
G_\rho(e:int) & = & e \\
G_\rho(e:\tau\ pack) & = & e \\
G_\rho(e:\tau_1 \to \tau_2) & = & \lambda x.G_\rho(e(S_\rho(x:\tau_1)):\tau_2) \\
G_\rho(e:\tau_1 \times \tau_2) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (G_\rho(\pi_1(x):\tau_1), G_\rho(\pi_2(x):\tau_2))
\end{array}$$

The other rules of the translation are straightforward.

To illustrate this transformation, the two programs in Section 3 are translated as follows:

$$\begin{array}{l}
\texttt{let } id = \lambda x.x \texttt{ in} \\
\quad \texttt{iter}(n, (0, \lambda x.x), \\
\qquad \lambda y.(\pi_2(y)\pi_1(y), \\
\qquad\quad (\lambda x.\mathsf{unwrap}_{int\to int}(id(\mathsf{wrap}_{int\to int}(x))))\pi_2(y))
\end{array}$$

$$\begin{array}{l}
\texttt{iter}(n, (0, \lambda x.x), \\
\quad \lambda y.(\pi_2(y)\pi_1(y), \\
\qquad \mathsf{unwrap}_{int\to int}(\mathsf{unpack}(\mathsf{pack}(\mathsf{wrap}_{int\to int}(\pi_2(y))))))
\end{array}$$

As we explained earlier, $\mathsf{wrap}_{int\to int}$ and $\mathsf{unwrap}_{int\to int}$ introduced by the translation make the complexity of both programs change from linear to n^2.

5 Formalization of Our Approach

Along the lines of the formalization of Leroy's type-directed unboxing transformation in the previous section, we formalize our approach and prove that it preserves the complexity of a program with respect to evaluation steps.

5.1 Translation and Type Correctness

In this section we formalize the unboxing transformation based on our representation of functions and prove its type correctness. First, as for Leroy's translation we define two translations of types: $|\tau|$ is the type of the specialized representation of τ and $[\tau]$ is the type of the generic representation of τ.

$$\begin{array}{rcl}
|\tau_1 \to \tau_2| & = & (|\tau_1| \to |\tau_2|) \times [\tau_1 \to \tau_2] \\
|\tau_1 \times \tau_2| & = & |\tau_1| \times |\tau_2| \\
|\alpha| & = & \alpha \\
|\tau\ pack| & = & [\tau]\ pack \\
|int| & = & int
\end{array}$$

$$\begin{array}{rcl}
[\tau_1 \to \tau_2] & = & [\tau_1] \to [\tau_2] \\
[\tau_1 \times \tau_2] & = & [\tau_1] \times [\tau_2] \\
[\alpha] & = & \alpha \\
[\tau\ pack] & = & [\tau]\ pack \\
[int] & = & boxint
\end{array}$$

As you can see, these translations only differ from Leroy's translations by the specialized representation of functions. As we explained informally, this representation is a pair of a specialized function of type $|\tau_1| \to |\tau_2|$ and the fully boxed function of type $[\tau_1 \to \tau_2]$. You may notice also that the generic translation is closed under substitution:

$$[\rho]([\tau]) = [\rho(\tau)]$$

where $[\rho]$ is a type substitution such that $[\rho](\alpha) = [\rho(\alpha)]$.

We then define the operations $\mathsf{wrap}_\tau(e)$ and $\mathsf{unwrap}_\tau(e)$ where τ is a type in the source language and e is an expression of the target language: wrap_τ coerces a value into fully boxed form and unwrap_τ coerces a value into the unboxed form for τ, starting from fully boxed form.

$$\begin{array}{rcl}
\mathsf{wrap}_\alpha(e) & = & e \\
\mathsf{wrap}_{int}(e) & = & \mathsf{wrapint}(e) \\
\mathsf{wrap}_{\tau_1\to\tau_2}(e) & = & \pi_2(e) \\
\mathsf{wrap}_{\tau\ pack}(e) & = & e \\
\mathsf{wrap}_{\tau_1\times\tau_2}(e) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (\mathsf{wrap}_{\tau_1}(\pi_1(x)), \mathsf{wrap}_{\tau_2}(\pi_2(x)))
\end{array}$$

$$\begin{array}{rcl}
\mathsf{unwrap}_\alpha(e) & = & e \\
\mathsf{unwrap}_{int}(e) & = & \mathsf{unwrapint}(e) \\
\mathsf{unwrap}_{\tau_1\to\tau_2}(e) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (\lambda y.\mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y))), x) \\
\mathsf{unwrap}_{\tau\ pack}(e) & = & e \\
\mathsf{unwrap}_{\tau_1\times\tau_2}(e) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (\mathsf{unwrap}_{\tau_1}(\pi_1(x)), \mathsf{unwrap}_{\tau_2}(\pi_2(x)))
\end{array}$$

In this definition $\mathsf{unwrap}_{\tau_1\to\tau_2}(e)$ creates a specialized version of the function as in Leroy's method, but retains the fully boxed version in the second component of the pair. The coercion $\mathsf{wrap}_{\tau_1\to\tau_2}(e)$ can be performed by just extracting the second component of the pair[1].

These operations satisfy the following lemma which states that they coerce values into their expected type.

Lemma 1 *1. If $\Gamma \vdash e : [\rho]|\tau|$, then $\Gamma \vdash \mathsf{wrap}_\tau(e) : [\rho(\tau)]$.*

2. If $\Gamma \vdash e : [\rho(\tau)]$, then $\Gamma \vdash \mathsf{unwrap}_\tau(e) : [\rho]|\tau|$.

The next step is to define the operation $S_\rho(e:\tau)$ which is used to coerce values of polymorphic type τ into a more specialized type $\rho(\tau)$. With our representation it can be easily defined by using unwrap_τ without introducing the generalization $G_\rho(e:\tau)$. This comes from the fact we always build new versions of a function by specializing its fully boxed version, rather than generalizing another specialized version.

$$\begin{array}{rcl}
\text{if } \rho(\tau) \equiv \tau & & \\
\qquad S_\rho(e:\tau) & = & e \\
\text{otherwise} & & \\
\qquad S_\rho(e:\alpha) & = & \mathsf{unwrap}_{\rho(\alpha)}(e) \\
\qquad S_\rho(e:int) & = & e \\
\qquad S_\rho(e:\tau_1 \to \tau_2) & = & \mathsf{unwrap}_{\rho(\tau_1\to\tau_2)}(\pi_2(e)) \\
\qquad S_\rho(e:\tau_1 \times \tau_2) & = & \texttt{let } x = e \texttt{ in} \\
 & & \quad (S_\rho(\pi_1(x):\tau_1), S_\rho(\pi_2(x):\tau_2)) \\
\qquad S_\rho(e:\tau\ pack) & = & e
\end{array}$$

This operation coerces a value into the proper type as stated below.

Lemma 2 *If $\Gamma \vdash e : [\rho](|\tau|)$, then $\Gamma \vdash S_\rho(e:\tau) : |\rho(\tau)|$.*

[1] These new definitions of wrap and unwrap are not compatible with those used in the informal presentation: there we use Leroy's definitions.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2'}{\Gamma \vdash e_1 e_2 : \tau_2 \rightsquigarrow (\pi_1 e_1') e_2'} \qquad \frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \quad Dom(\rho) = \{\alpha_1, \ldots, \alpha_n\}}{\Gamma \vdash x : \rho(\tau) \rightsquigarrow S_\rho(x : \tau)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \rightsquigarrow \texttt{let } y = \lambda x.e' \texttt{ in } (y, \lambda z.\mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))))}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow e_1' \quad \Gamma, x : Clos(\tau_1, \Gamma) \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2 \rightsquigarrow \texttt{let } x = e_1' \texttt{ in } e_2'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \rightsquigarrow (e_1', e_2')} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow e'}{\Gamma \vdash \pi_i(e) : \tau_i \rightsquigarrow \pi_i(e')}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e'}{\Gamma \vdash \mathsf{pack}(e) : \tau\ pack \rightsquigarrow \mathsf{pack}(\mathsf{wrap}_\tau(e'))} \qquad \frac{\Gamma \vdash e : \tau\ pack \rightsquigarrow e'}{\Gamma \vdash \mathsf{unpack}(e) : \tau \rightsquigarrow \mathsf{unwrap}_\tau(\mathsf{unpack}(e'))}$$

$$\frac{\Gamma \vdash e_1 : int \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \tau \rightsquigarrow e_2' \quad \Gamma \vdash e_3 : \tau \to \tau \rightsquigarrow e_3'}{\Gamma \vdash \texttt{iter}(e_1, e_2, e_3) : \tau \rightsquigarrow \texttt{iter}(e_1', e_2', \pi_1(e_3'))}$$

Figure 5: Translation of Expressions

Now we present the translation of expressions as a deductive system with judgments of the form

$$\Gamma \vdash e : \tau \rightsquigarrow e'.$$

The rules of the translation are shown in Figure 5. The translation is uniquely determined by the typing derivation of e. Let us explain a few rules:

Variable A variable x is translated to $S_\rho(x)$ which specializes the representation of x.

Application After translation the specialized version of a function is obtained by extracting the first component of its representation.

Abstraction A lambda abstraction is translated to a pair of lambda abstractions. The fully boxed version is obtained by coercing the specialized version by unwrap_{τ_1} and wrap_{τ_2}.

We extend the translation to polytypes and type environments as $|\forall\overline{\alpha}.\tau| = \forall\overline{\alpha}.|\tau|$ and $|\Gamma|(x) = |\Gamma(x)|$. Then the type correctness of this translation is formulated as the following lemma and proved by induction on the derivation of $\Gamma \vdash e : \tau \rightsquigarrow e'$.

Lemma 3 (Type Correctness) *If $\Gamma \vdash e : \tau \rightsquigarrow e'$, then $|\Gamma| \vdash e' : |\tau|$.*

As an example, let us consider the specialization of a variable z of polymorphic type $\forall\alpha.\alpha \to \alpha$ to $(int \to int) \to (int \to int)$. We do not expand $\mathsf{wrap}_{int\to int}$ and $\mathsf{unwrap}_{int\to int}$ for the sake of readability.

$$\begin{aligned} e_0 &\equiv S_{[int\to int/\alpha]}(z : \alpha \to \alpha) \\ &\equiv \mathsf{unwrap}_{(int\to int)\to(int\to int)}(\pi_2(z)) \\ &\equiv \texttt{let } x = \pi_2(z) \texttt{ in} \\ &\qquad (\lambda y.\mathsf{unwrap}_{int\to int}(x(\mathsf{wrap}_{int\to int}(y))), x) \end{aligned}$$

In our representation, application of e_0 to an expression e_1 will be written $\pi_1(e_0)e_1$. After a few administrative reductions, let us see what it looks like.

$$\begin{aligned} \pi_1(e_0)e_1 &\equiv \pi_1(\texttt{let } x = \pi_2(z) \texttt{ in} \\ &\qquad (\lambda y.\mathsf{unwrap}_{int\to int}(x(\mathsf{wrap}_{int\to int}(y))), x))e_1 \\ &= (\lambda y.\mathsf{unwrap}_{int\to int}(\pi_2(z)(\mathsf{wrap}_{int\to int}(y))))e_1 \\ &= \mathsf{unwrap}_{int\to int}(\pi_2(z)(\mathsf{wrap}_{int\to int}(e_1))) \end{aligned}$$

Finally, the two programs in Section 3 are translated as follows:

$$\begin{aligned} &\texttt{let } id = \mathsf{wrap'}_{\alpha\to\alpha}(\lambda x.x) \texttt{ in} \\ &\quad \texttt{iter}(n, (0, \mathsf{wrap'}_{int\to int}(\lambda x.x)), \\ &\qquad \lambda(y_1, y_2).(\pi_1(y_2)y_1, \\ &\qquad\quad \mathsf{unwrap}_{int\to int}(\pi_2(id)(\mathsf{wrap}_{int\to int}(y_2))))) \end{aligned}$$

$$\begin{aligned} &\texttt{iter}(n, (0, \mathsf{wrap'}_{int\to int}(\lambda x.x)), \\ &\quad \lambda(y_1, y_2).(\pi_1(y_2)y_1, \\ &\qquad \mathsf{unwrap}_{int\to int}(\mathsf{unpack}(\mathsf{pack}(\mathsf{wrap}_{int\to int}(y_2)))))) \end{aligned}$$

where $\mathsf{wrap'}_{\tau_1\to\tau_2}(e)$ is a shorthand for $\texttt{let } y = e \texttt{ in } (y, \lambda z.\mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))))$ and $\lambda(y_1, y_2).e$ for $\lambda y.e[\pi_1(y)/y_1, \pi_2(y)/y_2]$. We also simplified $\pi_1(\mathsf{wrap'}_{\tau_1\to\tau_2}(e))$ into e. As for Leroy's transformation, at runtime these programs successively apply $\mathsf{unwrap}_{int\to int}$ and $\mathsf{wrap}_{int\to int}$. However, this does not raise the original complexity anymore, as we explained earlier.

5.2 Correctness and Complexity

In this section we prove the main theorem of this paper: our version of the type-directed unboxing transformation is correct and does not raise the complexity of programs with respect to evaluation steps.

Formalizing the fact our translation preserves the complexity of programs with respect to evaluation steps is not that easy. For instance, there exists no constant C such that for any closed program e the following holds.

Property 4 *If $\emptyset \vdash e : int \rightsquigarrow e'$ and $\emptyset \vdash e \downarrow_n \underline{i}$, then $\emptyset \vdash e' \Downarrow_{Cn} \underline{i}$.*

This impossibility is related to the fact $\mathsf{wrap}_\tau(e)$ and $\mathsf{unwrap}_\tau(e)$ may require a number of evaluation steps proportional to the size of τ. This becomes clear if you consider the evaluation of $\mathsf{wrap}_{\tau_n}(e)$ where $\tau_0 = int$ and $\tau_{n+1} = \tau_n \times \tau_n$.

Notice also that, if we let the ratio C depend on the closed program to be transformed, the above property is trivial: just choose C such that Cn is larger than the number of steps needed to evaluate e'!

Thus, while the ratio C needs to be program dependent to avoid the above impossibility, we will consider a larger

class of programs, *i.e.* programs with a single free variable of type *int*, rather than only closed ones. Then we can formalize our main theorem.

Theorem 5 *Let e be a program such that $x : int \vdash e : int \rightsquigarrow e'$. Then there exists a constant C such that for any integer i, if $[\underline{i}/x] \vdash e \downarrow_n \underline{i'}$ then $[\underline{i}/x] \vdash e' \Downarrow_{Cn} \underline{i'}$.*

This means that the time complexity of e is preserved up to C, independently of the value of x.

Several studies related to program transformation have used logical relations to build correctness proofs [14, 8, 6, 11]. For our main theorem we extend the standard logical relation framework so that the relation between evaluation steps is taken into account. Our relations depend on a constant C which indicates the maximum ratio between the number of evaluation steps in the source program and in the translated program. We define relations between values $v : \tau \approx_C v : \tau'$ indexed by closed source and target types τ and τ'. The definition of the relations $v : \tau \approx_C v : \tau'$ is shown in Figure 6 where we write $\vdash vv' \downarrow_n v''$ if v is a closure $\langle\!\langle \gamma, x, e \rangle\!\rangle$ and $\gamma[v'/x] \vdash e \downarrow_{n-1} v''$.

The key point in this definition is that in $(*)$ the evaluation of the application v'_1 to v' is required to terminate not in less than $2Cn - 1$ steps, but in less than $2Cn - C$ steps. Thanks to this extra $C-1$, one can obtain an unboxed version of v' without violating the condition on evaluation steps. The above relations are well-defined by induction on the structure of indexing types.

Our strategy to prove the main theorem is to find for each program a constant C such that the program and its translation satisfy the relation $\approx_C$. We determine C by inspecting the maximal size of the types appearing in the typing derivation of the program. The size of types is defined as follows:

$$\begin{aligned} size(\alpha) &= 1 \\ size(int) &= 1 \\ size(\tau\ pack) &= size(\tau) + 1 \\ size(\tau_1 \times \tau_2) &= size(\tau_1) + size(\tau_2) + 1 \\ size(\tau_1 \to \tau_2) &= size(\tau_1) + size(\tau_2) + 1 \end{aligned}$$

The following lemmas give an upper bound for the cost of three basic operations: boxing, unboxing and specialization. We use a constant natural number $R = 6$, determined by inspecting the evaluation of $\mathsf{wrap}_\tau(e)$, $\mathsf{unwrap}_\tau(e)$, and $S_\rho(e : \tau)$.

Lemma 6 *Let M be a constant natural number and C be a natural number such that $C \geq RM$. Then the following hold for any τ such that $size(\tau) \leq M$.*

1. *If $\gamma' \vdash e' \downarrow_m v'$ and $v : \rho(\tau) \approx_C v' : [\rho]|\tau|$, then $\gamma' \vdash \mathsf{wrap}_\tau(e') \Downarrow_{m+R(size(\tau))} v''$ and $v : \rho(\tau) \approx_C v'' : [\rho(\tau)]$.*
2. *If $\gamma' \vdash e' \downarrow_m v'$ and $v : \rho(\tau) \approx_C v' : [\rho(\tau)]$, then $\gamma' \vdash \mathsf{unwrap}_\tau(e') \Downarrow_{m+R(size(\tau))} v''$ and $v : \rho(\tau) \approx_C v'' : [\rho]|\tau|$.*

Lemma 7 *Let M be a constant natural number and C be a natural number such that $C \geq RM$. For any τ and ρ such that $size(\rho(\tau)) \leq M$, if $\gamma' \vdash e' \downarrow_m v'$ and $v : \delta(\rho(\tau)) \approx_C v' : [\delta]([\rho](|\tau|))$, then $\gamma' \vdash S_\rho(e' : \tau) \Downarrow_{m+R(size(\rho(\tau)))} v''$ and $v : \delta(\rho(\tau)) \approx_C v'' : [\delta](|\rho(\tau)|)$.*

The proof of these lemmas appears in Appendix.

The next lemma tells that we can choose a constant C such that the evaluation of a source program and its translation are related by C. For γ and γ' two environments with same domain, $\gamma : \Gamma \approx_C \gamma' : \Gamma'$ means that they are pointwise related at types corresponding to the signatures Γ and Γ'.

Lemma 8 *Let $\Gamma \vdash e : \tau \rightsquigarrow e'$. Then there exists a constant C such that if $\gamma : \delta(\Gamma) \approx_C \gamma' : [\delta](|\Gamma|)$ and $\gamma \vdash e \downarrow_n v$, then $\gamma' \vdash e' \Downarrow_{2Cn} v'$ and $v : \delta(\tau) \approx_C v' : [\delta](|\tau|)$.*

This lemma is proved by induction on the derivation of $\Gamma \vdash e : \tau \rightsquigarrow e'$ by using Lemma 6 and 7. An outline of the proof appears in Appendix. The main theorem is obtained by restricting this lemma to $\Gamma = x : int$ and $\tau = int$.

6 Related Work

The formal study of unboxing for functional programming languages started at the beginning of the 1990's. Peyton Jones and Launchbury [16] extended a non-strict functional language and its type system to handle unboxed values. By making all boxing and unboxing explicit in an intermediate language, they are able to express optimizations in terms of program transformation. However their transformations are not type-directed, and their unboxing is only local: thanks to a worker-wrapper model of functions they are able to handle unboxing in up to recursive function calls —the worker calls itself recursively with unboxed values while the wrapper does the necessary unboxing and boxing—, but they cannot handle cross-module optimizations for instance.

By using a type-directed transformation, Leroy avoids such limitations [8]. We presented his transformation thoroughly in the body of this paper.

From then this area has been stimulating several attempts to effectively use unboxed representations in the implementation of polymorphic languages. Basically two approaches have been proposed so far: the coercion based approach following Leroy, and another approach based on runtime type passing.

Thiemann showed that by making some "mild" assumption on the calling convention of the underlying language, and using a yet more refined type system, not only monomorphic functions but also some polymorphic functions can get rid of boxing [22]. Henglein and Jørgensen formalized optimality of boxing and unboxing [7]. However, their optimality criterion is based on a rewriting of programs which eliminates coercions and does not capture the runtime behavior of boxing and unboxing. Shao proposed to mix with the second approach: unboxing based on both coercion and runtime type passing [19]. His method can use partially unboxed representation even for incoercible data types such as reference and lists.

The main topic of these studies has been to reduce boxing and unboxing operations and to extend the use of unboxed representations. However, none of them was aware that unboxing transformation may raise a program's complexity.

The coercion based approach is still limited by the barrier of polymorphism: once we pass a function to a higher-order polymorphic functional, there is no way we can use its unboxed version. Runtime type passing can handle this case. Ohori and Takamizawa [15] showed that it is possible to pass unboxed values to polymorphic functions by parameterizing them on the size of their arguments. Harper and Morrisett

$$
\begin{array}{cl}
\underline{i} : int \approx_C \underline{i} : int & \\
\underline{i} : int \approx_C \overline{i} : boxint & \\
(v_1, v_2) : \tau_1 \times \tau_2 \approx_C (v_1', v_2') : \tau_1' \times \tau_2' & \text{if } v_1 : \tau_1 \approx_C v_1' : \tau_1' \text{ and } v_2 : \tau_2 \approx_C v_2' : \tau_2' \\
\langle\langle v \rangle\rangle : \tau\ pack \approx_C \langle\langle v' \rangle\rangle : \tau'\ pack & \text{if } v : \tau \approx_C v' : \tau' \\
v : \tau_1 \to \tau_2 \approx_C v' : \tau_1' \to \tau_2' & \begin{cases} \text{for all } v_1 : \tau_1 \approx_C v_1' : \tau_1', \text{ if } \vdash v v_1 \downarrow_n v_2 \\ \quad \text{then } \vdash v' v_1' \Downarrow_{2Cn-C} v_2' \text{ and } v_2 : \tau_2 \approx_C v_2' : \tau_2' \end{cases} \quad (*) \\
v : \tau_1 \to \tau_2 \approx_C (v', v'') : \tau_1' \to \tau_2' \times \tau_1'' \to \tau_2'' & \begin{cases} \text{for all } v_1 : \tau_1 \approx_C v_1' : \tau_1', \text{ if } \vdash v v_1 \downarrow_n v_2 \\ \quad \text{then } \vdash v' v_1' \Downarrow_{2Cn-1} v_2' \text{ and } v_2 : \tau_2 \approx_C v_2' : \tau_2' \\ v : \tau_1 \to \tau_2 \approx_C v'' : \tau_1'' \to \tau_2'' \end{cases} \\
v : \forall \alpha_i . \tau \approx_C v' : \forall \alpha_i . \tau' & \text{for all } \tau_i,\ v : [\tau_i/\alpha_i]\tau \approx_C v' : [[\tau_i]/\alpha_i]\tau'
\end{array}
$$

Figure 6: Logical Relations

presented a general framework to utilize runtime type information by dynamically passing it [6]. The TIL ML compiler [21, 12] is developed based on this framework and uses unboxed representations. Runtime type passing does not seem to change the complexity of programs. However, there are difficult implementation issues in this approach [12].

Recently, the complexity problem we pointed here, together with the difficulties inherent to other methods, have stimulated stronger interest for untyped flow analysis based optimizations. In his ML compiler, Leroy abandons type-directed unboxing for a set of local untyped optimizations combined with a simple flow analysis, and reports encouraging results [9]. Goubault [4] goes further by refining the worker-wrapper model, and suggesting to inline the wrapper, cut into pre-processing and post-processing parts. He can then eliminate most of the boxing on the basis of a control-flow analysis.

While there was no concern about complexity in the unboxing optimization area, there have been several studies on how to formalize complexity in functional programming languages. Santos [18] enriches natural semantics with a notion of cost in a way very similar to ours, and studies time-complexity properties of several transformations in a lazy functional programming language. However, the transformations he studied are local and change only the constant amount of costs. Thus he did not have to consider asymptotic complexity of programs as we do. Roe [17], and Greiner and Blelloch [5, 2] defined profiling operational semantics for parallel functional languages. The later have used it to prove that the translation of the parallel speculative λ-calculus and NESL into abstract machines preserve asymptotic complexity of programs.

7 Future Work

This work is still ongoing, and there are problems left to solve, both theoretical and practical.

We have proved that our unboxing transformation preserves the complexity of programs with respect to evaluation steps, that is safety for time. The next natural step is to prove the same safety property for space. We believe our approach also preserves the space complexity of programs, in the sense of amount of live heap needed. However formalizing that property would need a more refined operational semantics such as proposed in [13], and the proof ought to be more involved.

Our unboxing transformation still shares another undesirable property with other type-directed unboxing transformations: it may convert some tail calls into non-tail calls, which can also change the space complexity of programs (in a model including tail-call optimization). We are working at solving this problem.

Even if we can solve all these questions, our transformation is useless if it does not indeed increase the performance of programs. As we stated in the introduction, on purely monomorphic programs (*i.e.* without specialization at all) it is by definition less efficient than Leroy's, since we are adding projections at every application step. Even for polymorphic ones, our coercions are generally more complex.

A first remark is that it is not really as bad as it seems. For instance, our specialized representation for function types is a pair. Since it is independent from the generic representation, nothing opposes unboxing this pair, and getting rid of the extra indirection needed for boxed pairs. It means that we loose almost nothing on monomorphic programs, and gain when the fully boxed version is needed, since we already have it for free. To get into more details we need to actually implement this transformation.

Another remark is that we shall be able to extend this proof to further refinements of the unboxing transformation, this time oriented towards efficiency. One of the problems we have is that we specialize a function always from its fully boxed form. However, this can be modified without violating safety so that a function is specialized from its current specialized form. To achieve this scheme the only thing to modify is the definition of specialization.

$$
\begin{array}{rcl}
S_\rho(e : \tau_1 \to \tau_2) & = & (\lambda x . S_\rho(\pi_1(e)(G_\rho(x : \tau_1)) : \tau_2), \pi_2(e)) \\
G_\rho(e : \tau_1 \to \tau_2) & = & (\lambda x . \mathsf{unwrap}_{\tau_2}(\pi_2(e)(\mathsf{wrap}_{\tau_1}(x))), \pi_2(e))
\end{array}
$$

The other cases are taken from Leroy's definitions. In this scheme, specialization increases the size of functions. However, this expansion does not go on forever, since generalization restarts from the generic representation. For this reason we believe this method still preserves the complexity of programs. This is yet to be proved.

Acknowledgements

This work is partially supported by Grant-in-Aid for Encouragement of Young Scientists of Japan No. 09780271. We would like to thank Masahito Hasegawa, Susumu Nishimura, Atsushi Ohori, and the anonymous reviewers for their many helpful comments and suggestions.

References

[1] A. W. Appel. *Compiling with Continuation*. Cambridge University Press, 1992.

[2] G. E. Blelloch and J. Greiner. A provably time and space efficient implementation of NESL. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 13 – 27, 1986.

[4] J. Goubault. Generalized unboxing, congruences and partial inlining. In *Proc. Static Analysis Symposium*, pages 147–161, 1994.

[5] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 309 – 321, 1996.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 130–141, 1995.

[7] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 213 – 226, 1994.

[8] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 177 – 188, 1992.

[9] X. Leroy. The effectiveness of type-based unboxing. In *Proc. International Workshop on Types in Compilation*, pages 1–8, 1997.

[10] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 271 – 283, 1996.

[12] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science Carnegie Mellon University, 1995.

[13] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. ACM Symposium on Functional Programming Languages and Computer Architecture*, pages 66–77, 1995.

[14] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems*, 17(6):844–895, 1995.

[15] A. Ohori and T. Takamizawa. An unboxed operational semantics for ML polymorphism. *Journal of Lisp and Symbolic Computation*, 10(1):61 – 91, 1997.

[16] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc. ACM Symposium on Functional Programming Languages and Computer Architecture*, pages 636 – 666, 1991.

[17] P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1991.

[18] A. L. Santos. *Compilation by Transformation in Non-strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1995.

[19] Z. Shao. Flexible representation analysis. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 85 – 98, 1997.

[20] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116 – 129, 1995.

[21] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[22] P. Thiemann. Polymorphic typing and unboxed values revisited. In *Proc. ACM Symposium on Functional Programming Languages and Computer Architecture*, pages 24 – 35, 1995.

A Proofs

A.1 Proof of Lemma 6

Proof. By induction on the structure of types. We will show some important cases. In this proof we write $\approx$ for $\approx_C$.

Case: τ is $\tau_1 \to \tau_2$.

Subcase: $\mathsf{wrap}_{\tau_1\to\tau_2}(e')$ is $\pi_2(e')$. By definition of $\approx$, v' can be written as (v'_1, v'_2) and $v : \rho(\tau_1 \to \tau_2) \approx v'_2 : [\rho(\tau_1 \to \tau_2)]$. This case is proved since $\gamma' \vdash \pi_2(e') \Downarrow_{m+1} v'_2$.

Subcase: $\mathsf{unwrap}_{\tau_1\to\tau_2}(e')$ is $\mathtt{let}\ x = e'\ \mathtt{in}\ (\lambda y.\mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y))), x)$.
By definition of the operational semantics,

$$\gamma'[v'/x] \vdash (\lambda y.\mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y))), x) \downarrow_3 (\langle\!\langle \gamma'[v'/x], y, \mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y)))\rangle\!\rangle, v')$$

Let v'' be $\langle\!\langle \gamma'[v'/x], y, \mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y)))\rangle\!\rangle$. Then

$$\gamma' \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ (\lambda y.\mathsf{unwrap}_{\tau_2}(x(\mathsf{wrap}_{\tau_1}(y))), x) \downarrow_{m+3+1} (v'', v')$$

where $3 + 1 \le R(size(\tau_1 \to \tau_2))$.

Now we have to prove that $v : \rho(\tau_1 \to \tau_2) \approx (v'', v') : [\rho]|\tau_1 \to \tau_2|$. By the hypothesis of this lemma, $v : \rho(\tau_1 \to \tau_2) \approx v' : [\rho][\tau_1 \to \tau_2]$. Thus the only thing we have to prove is that for all $v_1 : \rho(\tau_1) \approx v'_1 : [\rho]|\tau_1|$, if $\vdash v v_1 \downarrow_n v_2$ then $\vdash v'' v'_1 \Downarrow_{2Cn-1} v'_2$ and $v_2 : \rho(\tau_2) \approx v'_2 : [\rho]|\tau_2|$. Now we will prove this claim.

By induction hypothesis,

$$[v'_1/z] \vdash \mathsf{wrap}_{\tau_1}(z) \Downarrow_{1+R(size(\tau_1))} v''_1$$

and $v_1 : \rho(\tau_1) \approx v''_1 : [\rho(\tau_1)]$.

By $v : \rho(\tau_1 \to \tau_2) \approx v' : [\rho(\tau_1 \to \tau_2)]$ and $v_1 : \rho(\tau_1) \approx v_1'' : [\rho(\tau_1)]$ and $\vdash v v_1 \downarrow_n v_2$, we obtain $\vdash v' v_1'' \Downarrow_{2Cn-C} v_2'$ and $v_2 : \rho(\tau_2) \approx v_2' : [\rho(\tau_2)]$. Then

$$\gamma'[v'/y][v_1'/z] \vdash y(\mathsf{wrap}_{\tau_1}(z)) \Downarrow_{1+(1+R(size(\tau_1)))+2Cn-C} v_2'$$

Then by induction hypothesis,

$$\gamma'[v'/y][v_1'/z] \vdash \mathsf{unwrap}_{\tau_2}(y(\mathsf{wrap}_{\tau_1}(z))) \Downarrow_{(2Cn-C+R(size(\tau_1))+2)+R(size(\tau_2))} v_2''$$

and

$$v_2 : \rho(\tau_2) \approx v_2'' : [\rho]|\tau_2| \tag{1}$$

By the operational semantics,

$$\vdash \langle\!\langle \gamma'[v'/y'], z, \mathsf{unwrap}_{\tau_2}(y(\mathsf{wrap}_{\tau_1}(z))) \rangle\!\rangle v_1' \Downarrow_{2Cn-C+R(size(\tau_1)+size(\tau_2))+3} v_2''$$

Knowing that $R(size(\tau_1) + size(\tau_2)) + 3 - C \leq RM - R + 3 - C \leq -1$,

$$\vdash \langle\!\langle \gamma'[v'/y'], z, \mathsf{unwrap}_{\tau_2}(y(\mathsf{wrap}_{\tau_1}(z))) \rangle\!\rangle v_1' \Downarrow_{2Cn-1} v_2'' \tag{2}$$

(1) and (2) complete the proof of this case.

Case: τ is $\tau_1 \times \tau_2$.

Subcase: $\mathsf{wrap}_{\tau_1 \times \tau_2}(e')$ is $\mathtt{let}\ x = e'\ \mathtt{in}\ (\mathsf{wrap}_{\tau_1}(\pi_1(x)), \mathsf{wrap}_{\tau_2}(\pi_2(x)))$.

By definition of $\approx$, v and v' can be written as (v_1, v_2) and (v_1', v_2') such that $v_1 : \rho(\tau_1) \approx v_1' : [\rho]|\tau_1|$ and $v_2 : \rho(\tau_2) \approx v_2' : [\rho]|\tau_2|$. Then $\gamma'[(v_1', v_2')/x] \vdash \pi_1(x) \Downarrow_2 v_1'$. By induction hypothesis, $\gamma'[(v_1', v_2')/x] \vdash \mathsf{wrap}_{\tau_1}(\pi_1(x)) \Downarrow_{2+R(size(\tau_1))} v_1''$ and $v_1 : \rho(\tau_1) \approx v_1'' : [\rho(\tau_1)]$. In the same way, $\gamma'[(v_1', v_2')/x] \vdash \mathsf{wrap}_{\tau_2}(\pi_2(x)) \Downarrow_{2+R(size(\tau_2))} v_2''$ and $v_2 : \rho(\tau_2) \approx v_2'' : [\rho(\tau_2)]$. Then

$$\gamma'[(v_1', v_2')/x] \vdash (\mathsf{wrap}_{\tau_1}(\pi_1(x)), \mathsf{wrap}_{\tau_2}(\pi_2(x))) \Downarrow_{5+R(size(\tau_1)+size(\tau_2))} (v_1'', v_2'')$$

and $(v_1, v_2) : \rho(\tau_1 \times \tau_2) \approx_C (v_1'', v_2'') : [\rho(\tau_1 \times \tau_2)]$. Finally,

$$\gamma' \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ (\mathsf{wrap}_{\tau_1}(\pi_1(x)), \mathsf{wrap}_{\tau_2}(\pi_2(x))) \Downarrow_{m+5+R(size(\tau_1)+size(\tau_2))+1} (v_1'', v_2'')$$

where $m+5+R(size(\tau_1)+size(\tau_2))+1 \leq m+R(size(\tau_1)+size(\tau_2)+1) = m+R(size(\tau_1 \times \tau_2))$. Here we used the fact $R = 6$.

The same proof works for $\mathsf{unwrap}_{\tau_1 \times \tau_2}(e')$.

□

A.2 Proof of Lemma 7

Proof. By induction on the structure of τ. We will show some important cases. In this prove we write $\approx$ for $\approx_C$.

Case: $\rho(\tau) \equiv \tau$. This case is clear since $S_\rho(e' : \tau)$ is e'.

Case: τ is α and $S_\rho(e' : \alpha)$ is $\mathsf{unwrap}_{\rho(\alpha)}(e')$.

By $[\rho]|\alpha| \equiv [\rho(\alpha)]$ and Lemma 6,

$$\gamma' \vdash \mathsf{unwrap}_{\rho(\alpha)}(e') \Downarrow_{m+R(size(\rho(\alpha)))} v''$$

and $v : \delta(\rho(\alpha)) \approx v'' : [\delta](|\rho(\alpha)|)$.

Case: τ is $\tau_1 \to \tau_2$ and $S_\rho(e' : \tau_1 \to \tau_2)$ is $\mathsf{unwrap}_{\rho(\tau_1 \to \tau_2)}(\pi_2(e'))$.

By definition of $\approx$, $\gamma' \vdash \pi_2(e') \Downarrow_{(m+1)} v''$ and $v : \delta(\rho(\tau_1 \to \tau_2)) \approx v'' : [\delta][\rho][\tau_1 \to \tau_2]$

By $[\delta][\rho][\tau_1 \to \tau_2] \equiv [\delta][\rho(\tau_1 \to \tau_2)]$ and Lemma 6, we obtain

$$v : \delta(\rho(\tau_1 \to \tau_2)) \approx v''' : [\delta](|\rho(\tau_1 \to \tau_2)|) \tag{3}$$

Moreover, $\mathsf{unwrap}_{\rho(\tau_1 \to \tau_2)}(\pi_2(e'))$ is actually the following expression.

$$e'' \equiv \mathtt{let}\ x = \pi_2(e')\ \mathtt{in}\ (\lambda y.\mathsf{unwrap}_{\rho(\tau_2)}(x(\mathsf{wrap}_{\rho(\tau_1)}(y))), x)$$

Then it is clear that $\gamma' \vdash e'' \Downarrow_{(m+1)+3+1} v'''$ where $(m+1)+3+1 \leq m+R(size(\rho(\tau_1 \to \tau_2)))$. This statement and (3) complete the proof of this case.

□

A.3 Proof of Lemma 8

Proof. There is a natural number M such that the size of any type appearing the derivation $\Gamma \vdash e : \tau \rightsquigarrow e'$ is less than M. Let C be a natural number such that $C \geq RM$.

Now C is fixed, and we will prove the property by induction for every judgment in the derivation of $\Gamma \vdash e : \tau \rightsquigarrow e'$. We write $\approx$ for $\approx_C$.

Case: $\Gamma \vdash x : \rho(\tau) \rightsquigarrow S_\rho(x)$. We assume that $Dom(\delta) \cap \{\overline{\alpha}\} = \emptyset$ by variable convention and $Dom(\rho) = \{\overline{\alpha}\}$. By the hypothesis on Γ, $v : \forall\overline{\alpha}.\delta(\tau) \approx v' : \forall\overline{\alpha}.[\delta](|\tau|)$. Let ρ' be $\delta \circ \rho$. Then by the definition, $v : \rho'(\delta(\tau)) \approx v' : [\rho'][\delta](|\tau|)$. That is $v : \delta(\rho(\tau)) \approx v' : [\delta][\rho](|\tau|)$. On the other hand $\gamma \vdash x \downarrow_1 v$ and $\gamma' \vdash x \downarrow_1 v'$. Then by Lemma 7, $\gamma' \vdash S_\rho(x : \tau) \Downarrow_{1+R(size(\rho(\tau)))} v''$ such that $v : \delta(\rho(\tau)) \approx v'' : [\delta](|\rho(\tau)|)$ where $1 + R(size(\rho(\tau))) \leq 1 + RM \leq 2C$.

Case: $\Gamma \vdash e_1 e_2 : \tau_2 \rightsquigarrow \pi_1(e_1')e_2'$ is obtained from $\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1'$ and $\Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2'$. Let $\gamma \vdash e_1 \downarrow_l v_1$ and $\gamma \vdash e_2 \downarrow_m v_2$. Then by induction hypothesis for e_1, $\gamma' \vdash e_1' \Downarrow_{2Cl} (v_1', v_1'')$ and $v_1 : \delta(\tau_1 \to \tau_2) \approx (v_1', v_1'') : [\delta]((|\tau_1| \to |\tau_2|) \times [\tau_1 \to \tau_2])$. Then $\gamma' \vdash \pi_1(e_1') \Downarrow_{2Cl+1} v_1'$.

By induction hypothesis for e_2, $\gamma' \vdash e_2' \Downarrow_{2Cm} v_2'$ and $v_2 : \delta(\tau_1) \approx v_2' : [\delta](|\tau_1|)$.

Let $\vdash v_1 v_2 \downarrow_n v$. Then by the definition of $\approx$, $\vdash v_1' v_2' \Downarrow_{2Cn-1} v'$ and $v : \delta(\tau_2) \approx v' : [\delta](|\tau_2|)$.

This means that $\gamma \vdash e_1 e_2 \downarrow_{l+m+n} v$ and $\gamma' \vdash \pi_1(e_1')e_2' \Downarrow_{2Cl+1+2Cm+2Cn-1} v'$ where $2Cl + 1 + 2Cm + 2Cn - 1 = 2C(l + m + n)$.

Case: $\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \rightsquigarrow \mathtt{let}\ y = \lambda x.e'\ \mathtt{in}\ (y, \lambda z.\mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))))$ is obtained from $\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow e'$. Let e'' be $\mathtt{let}\ y = \lambda x.e'\ \mathtt{in}\ (y, \lambda z.\mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))))$.

$\gamma \vdash \lambda x.e \downarrow_1 \langle\!\langle \gamma, x, e \rangle\!\rangle$ and $\gamma' \vdash e'' \downarrow_5 (\langle\!\langle \gamma', x, e' \rangle\!\rangle, \langle\!\langle \gamma'[\langle\!\langle \gamma', x, e' \rangle\!\rangle / y], z, \mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))) \rangle\!\rangle)$ where $5 \leq 2C$.

Take $v : \delta(\tau_1) \approx v' : [\delta](|\tau_1|)$. Let $\vdash \langle\!\langle \gamma, x, e \rangle\!\rangle v \downarrow_n v_2$. That means $\gamma[v/x] \vdash e \downarrow_{n-1} v_2$. By induction hypothesis, $\gamma'[v'/x] \vdash e' \Downarrow_{2C(n-1)} v_2'$ and $v_2 : \delta(\tau_2) \approx v_2' : [\delta]|\tau_2|$. Then $\vdash \langle\!\langle \gamma', x, e' \rangle\!\rangle v' \Downarrow_{2C(n-1)+1} v_2'$ where $2C(n-1)+1 \leq 2Cn-1$.

Take $v : \delta(\tau_1) \approx v' : [\delta(\tau_1)]$ and let $\gamma[v/x] \vdash \langle\!\langle \gamma, v, e \rangle\!\rangle v \downarrow_n v_3$. That means

$$\gamma[v/x] \vdash e \downarrow_{n-1} v_3 \tag{4}$$

Now we have to consider the evaluation of $\langle\!\langle \gamma'[\langle\!\langle \gamma', x, e' \rangle\!\rangle / y], z, \mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))) \rangle\!\rangle v'$. Let γ'' be $\gamma'[\langle\!\langle \gamma', x, e' \rangle\!\rangle / y][v'/z]$. Then we will consider the evaluation of $\mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z)))$ under the environment γ''.

By Lemma 6, $\gamma'' \vdash \mathsf{unwrap}_{\tau_1}(z) \Downarrow_{1+R(size(\tau_1))} v''$ and $v : \delta(\tau_1) \approx v'' : [\delta](|\tau_1|)$.

By induction hypothesis for $\gamma[v/x] : \delta(\Gamma, x : \tau_1) \approx \gamma'[v''/x] : [\delta](|\Gamma|, x : |\tau_1|)$ and (4),

$$\gamma'[v''/x] \vdash e' \Downarrow_{2C(n-1)} v_3'.$$

Then $\gamma'' \vdash y(\mathsf{unwrap}_{\tau_1}(z)) \Downarrow_{1+(1+R(size(\tau_1)))+2C(n-1)+1} v_3'$ and $v_3 : \delta(\tau_2) \approx v_3' : [\delta](|\tau_2|)$.

By Lemma 6,

$$\gamma'' \vdash \mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))) \Downarrow_{(R(size(\tau_1))+2C(n-1)+3)+R(size(\tau_2))} v_3''$$

and $v_3 : \delta(\tau_2) \approx v_3'' : [\delta(\tau_2)]$. This means

$$\vdash \langle\!\langle \gamma'[\langle\!\langle x, \gamma', e' \rangle\!\rangle], z, \mathsf{wrap}_{\tau_2}(y(\mathsf{unwrap}_{\tau_1}(z))) \rangle\!\rangle v' \Downarrow_{(R(size(\tau_1))+2C(n-1)+3)+R(size(\tau_2))+1} v_3''$$

where $(R(size(\tau_1)) + 2C(n-1) + 3) + R(size(\tau_2)) + 1 = R(size(\tau_1 \to \tau_2)) + 2C(n-1) + 4 - R \leq RM + 2C(n-1) \leq 2Cn - C$

Case: $\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau \rightsquigarrow \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2'$ is derived from $\Gamma \vdash e_1 : \tau_1 \rightsquigarrow e_1'$ and $\Gamma, x : Clos(\tau_1, \Gamma) \vdash e_2 : \tau \rightsquigarrow e_2'$.

Let $\{\alpha_i\}$ be $FTV(\tau_1) \setminus FTV(\Gamma)$ and δ' be a ground substitution on α_i.

By the operational semantics $\gamma \vdash e_1 \downarrow_l v_1$. Then by induction hypothesis $\gamma' \vdash e_1' \Downarrow_{2Cl} v_1'$ and $v_1 : \delta'(\delta(\tau_1)) \approx v_1' : [\delta']([\delta](|\tau_1|))$. By definition $v_1 : \forall\alpha_i.\delta(\tau_1) \approx v_1' : \forall\alpha_i.[\delta](|\tau_1|)$. By the operational semantics $\gamma[v_1/x] \vdash e_2 \downarrow_m v$. Then by induction hypothesis, for $\gamma[v_1/x] : \rho(\Gamma, x : \forall\alpha_i.\tau_1) \approx \gamma'[v_1'/x] : [\rho](|\Gamma|, x : |\forall\alpha_i.\tau_1|)$ and $\gamma \vdash e_2 \downarrow_m v$, we obtain $\gamma'[v_1/x] \vdash e_2' \Downarrow_{2Cm} v'$ and $v : \delta(\tau_2) \approx v' : [\delta](|\tau_2|)$. This means that $\gamma \vdash e \downarrow_{l+m+1} v$ and $\gamma' \vdash e' \downarrow_{2Cl+2Cm+1} v$ and $v : \delta(\tau_2) \approx v' : [\delta](|\tau_2|)$. This case is proved since $2Cl + 2Cm + 1 \leq 2C(l + m + 1)$.

Case: $\Gamma \vdash \mathsf{pack}(e) : \tau\ pack \rightsquigarrow \mathsf{pack}(\mathsf{wrap}_\tau(e'))$ is derived from $\Gamma \vdash e : \tau \rightsquigarrow e'$. Let $\gamma \vdash e \downarrow_n v$. Then $\gamma \vdash \mathsf{pack}(e) \downarrow_{n+1} \langle\!\langle v \rangle\!\rangle$. By induction hypothesis, $\gamma' \vdash e' \downarrow_{2Cn} v'$ such that $v : \delta(\tau) \approx v' : [\delta]|\tau|$.

By Lemma 6, $\gamma' \vdash \mathsf{wrap}_\tau(e') \downarrow_{2Cn+R(size(\tau))} v''$ such that $v : \delta(\tau) \approx v' : [\delta(\tau)]$. Then $\gamma' \vdash \mathsf{pack}(\mathsf{wrap}_\tau(e')) \downarrow_{2Cn+R(size(\tau))+1} \langle\!\langle v'' \rangle\!\rangle$ where $2Cn + R(size(\tau)) + 1 \leq 2Cn + RM + 1 \leq 2Cn + C + 1 \leq 2C(n+1)$.

□