Record Unboxing

Huu-Duc Nguyen
Research Institute of Electrical Communication
Tohoku University
ducnh@riec.tohoku.ac.jp

Atsushi Ohori
Research Institute of Electrical Communication
Tohoku University
ohori@riec.tohoku.ac.jp

Abstract

This paper proposes a type-based optimization method that "unboxes" records whenever possible by flattening nested records and changing top-level tuples to multiple value passing. We first develop a type-based algorithm that infers whether each record expression is "rigid" or not, i.e. whether it requires heap allocation or not. This is based on the observation that the changes the representations of those record types that do not interact with external functions or polymorphic functions do not affect the behavior of the program. We then develop a record unboxing algorithm that translates a functional language with records to a multiple value calculus in which a function can take/return multiple values. During this translation, the algorithm transforms all non rigid records into multiple values. The combination of the two yields a simple and yet effective optimization method. We have implemented the proposed method in the SML# (an extension of the Standard ML) compiler, and have evaluated it with typical benchmark programs. The results show significant speed ups for non trivial benchmarks, including 21% speed up of mlyacc and 27% speed up of lexgen.

1. Introduction

In polymorphic functional languages, records (tuples) are implemented as pointers to heap-allocated memory blocks. This "boxed" representation is necessary for passing a record to a higher-order polymorphic function, whose operational semantics is to take and return one object in uniform representation, typically a one-word value. However, naively adopting this strategy introduces unnecessary runtime overhead in creating and accessing heap allocated memory blocks. The motivation of this work is to develop a systematic method that suppresses unnecessary heap allocation of records whenever possible.

Records are used to bundle multiple values as parameters to a function, return values from a function, or multiple attributes of an object. For each of these purposes, heap allocation is often unnecessary. Let us first consider the case of function parameter using the following simple example (written in Standard ML syntax).

```
fun fib n =
  let fun fibPair (n, x, y) =
          if n = 0 then x
          else fibPair(n - 1, y, x + y)
  in  fibPair(n, 0, 1)
  end
```


Since the tuple argument of `fibPair` is only used to pass three values, repeated heap block creation is totally unnecessary and yields unacceptable runtime overhead. For this simple case, most of optimizing compilers would eliminate heap allocation by converting `fibPair` to the following three argument function:

```
fun fib n =
  let fun fibPair {n, x, y} =
          if n = 0 then x
          else fibPair {n - 1, y, x + y}
  in fibPair {n, 0, 1}
  end
```

where the notation `fibPair {n,x,y}` represents application of `fibPair` to three arguments `n, x, y`. This optimization is done by analyzing all possible caller sites of each function. If all actual arguments in these call sites have a common syntactic pattern that matches with the formal argument, then function's argument can be split according to the pattern. This ad hoc approach works well for first order known functions. In the presence of higher-order and unknown functions, analyzing all call sites of a function is not trivial. Hannan and Hicks [7] proposed a type-based method to solve this problem. It first infers a kind information, which is either compile time `c` (flattenable) or runtime `r` (non-flattenable), for each subexpression type, and then systematically transforms all formal arguments of tuple type marked with kind `c` into sequences of formal arguments, and all actual tuple arguments marked with kind `c` into sequences of arguments.

This is a satisfactory solution to unboxing records used as function parameters. However, unnecessary heap allocation of records is not limited to function parameters. To see this, consider the following program.

```
let
  fun iterate isFinished next x =
      if isFinished x
      then x
      else iterate isFinished next (next x)
  fun isFinished_R (a,(b,c)) = a + b + c = 0
  fun next_R (a,(b,c)) = (a + 1, (b - 1, c - 1))
  val r = iterate isFinished_R next_R (1,(2,3))
in #1 r end
```

In this example, all three functions manipulate objects of type $int \times (int \times int)$ but none of these objects is returned as the result of the program. In principle, it is therefore possible to unbox and flatten all objects as follows.

```
let
  fun iterate isFinished next {a,b,c} =
      if isFinished {a,b,c}
```

```
      then {a,b,c}
      else iterate isFinished next (next {a,b,c})
  fun isFinished_R {a,b,c} = a + b + c = 0
  fun next_R {a,b,c} = {a + 1, b - 1, c - 1}
  val {r1,r2,r3} =
      iterate isFinished_R next_R {1,2,3}
in r1 end
```

Note that this process requires us to extend functions to have multiple return values and multiple value bindings. One would speculate that the work of [7] and others on arity raising could be extended to deal with this simple monomorphic case. It is however not at all clear whether existing approaches can be extended to unbox and flatten records that are passed or returned by foreign polymorphic functions. For example, consider the case where `iterate` is a polymorphic function defined in another compilation unit, for which only the following signature is available:

`iterate` : $('a \to bool) \to ('a \to' a) \to' a \to' a$

In this case, the arguments of `isFinished_R` and `next_R` need to be boxed, but their second components can be unboxed since they are only manipulated locally inside the program. So the ideal optimizer should produce the following code.

```
iterate : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
let
  fun isFinished_R (a,b,c) = a + b + c = 0
  fun next_R (a,b,c) = (a + 1, b - 1, c - 1)
  val r = iterate isFinished_R next_R (1,2,3)
in #1 r end
```

As far as we can see, no existing method can produce such optimized code.

An alternative approach to unboxing object is to decompose a function into a "worker" that takes unboxed objects and a "wrapper" that unboxes an argument and passes it to the worker. This approach has been investigated using type information [14, 10]. This would work fine for atomic objects such as floating point numbers, but it is impractical for nested records. With the existence of polymorphic functions, this approach sometimes produces unexpected runtime overhead [12].

We note that runtime representation of an object can be freely changed as long as this change is consistent and the object does not interact with environment (only be manipulated locally). If a tuple satisfies this criteria then the compiler can safely unbox it irrespective of whether it is used as an argument to a function or not. The goal of this paper is to develop an optimization method that uniformly unbox all tuples that satisfy this general criteria. Furthermore, we require that the optimization method should scale up to a full scale ML style language without introducing any extra runtime overhead. In this paper, we develop such an optimization method based on the following observations.

1. We call a record *rigid* if it requires boxed representation, otherwise we call it *flexible*. By extending the target intermediate language to a multiple value calculus in which a function can take/return multiple values, all flexible records can be compiled to efficient multiple value bindings.
2. A record is rigid if it appears in one of the following contexts.
   - in an argument or in the return value of an external (or separately compiled) function,
   - in a final result value of the program, or
   - as the out-most constructor of an argument to a polymorphic function.
3. It is possible to statically approximate the set of rigid record expressions by refining a type system to attach a "rigidity attribute" to each record type.
4. All the non rigid record expressions can be "unboxed" by performing one of the following transformation depending on the contexts they appear.
   - Change record-parameter functions to multiple parameter functions, and application to a tuple parameter to multiple parameter application.
   - Change record return expression to multiple value return expression, and the corresponding value binding to multiple variable bindings.

These simple and natural observations indeed yield an effective optimization method. We have worked out the details and have developed a record unboxing optimization algorithm. The required type-based analysis is rather simple, and yet it scales up to the full Standard ML language and supports incremental or separate compilation. It is also easily incorporated in a compiler as one transformation phase. We have implemented the algorithm in a full scale compiler of Standard ML, and have conducted a typical set of benchmarks for Standard ML. The result demonstrates that this optimization indeed significantly improves the speed of the compiled code. We therefore claim that the optimization method proposed in this paper is a simple but effective one that can be readily incorporated in an optimizing compiler for an ML style language.

The rest of this paper is organized as follows. Section 2 presents a type-based static analysis to approximate the set of rigid record expressions. Section 3 presents our record unboxing transformation algorithm. Section 4 describes the necessary extension to deal with polymorphism. Section 5 describes our implementation and reports benchmark results. Section 6 compares this work with related works. Section 7 concludes the paper.

2. Representation Analysis

The first step toward the development of unboxing record optimization is to determine the set of record expressions whose representations can be changed without affecting the (observable) meaning of the program.

For an entire program, the observable meaning is simply the (possible) value computed by the program. Let us consider the following simple program.

```
let
   val r1 = (1,2)
   val r2 = (3,4)
in (r1, #1 r2) end
```

The final result computed by this program contains value of the record `r1` and value of the first component of the record `r2`, but not the value of `r2` itself. The observable meaning of the program therefore depends on the representation of the record `r1` but does not depend on the representation of `r2`. In this case we say `r1` is *rigid* and `r2` is *flexible*.

It is intuitively obvious that some record expression r is *rigid* if the type of the program contains a record type "originated" from r. The above program has type `(int * int) * int`, and the first type component `(int * int)` is "originated" from the record `r1`. From this simple analysis, we obtain the following strategy for statically estimating the set of rigid record expressions in a given entire program.

1. We define a type system where each record type is annotated with a rigidity attribute indicating whether this is rigid or flexible.

2. We set the rigid annotation to all record types appearing in the program type, and then apply a constraint-based annotation inference algorithm to infer annotation types of all the program's sub-expressions. The rigidity constrained by the program type will be propagated to all rigid record sub-expressions during the inference process.
3. A record sub-expression of the program is rigid if its inferred annotated type is marked as rigid. Otherwise it is flexible.

This strategy is easily generalized to a compilation unit that contain external functions (free variables). The only necessary extension is to distinguish external names from internal names, and to require annotations in types of arguments or return values of an external function to be rigid. Based on this strategy, in this section, we develop a type system that performs rigidity annotation inference. To simplify the presentation, we first develop a type system for a simply typed calculus. Extension to an ML style polymorphic calculus shall be given in section 4.

2.1 The Source Calculus – λ^{ML}

We define the source calculus, λ^{ML}, as a simple explicitly typed ML-style lambda calculus. Syntax of programs (ranged over by P), expressions (ranged over by e), types (ranged over by τ) of λ^{ML} are given by the following grammar.

$$\begin{array}{rcl} e & ::= & c^o \mid x \mid X \mid \lambda x:\tau.e \mid (e\ e) \mid (e,\ldots,e) \mid \pi_i(e) \\ & & \mid \texttt{let}\ x:\tau = e\ \texttt{in}\ e\ \texttt{end} \\ P & ::= & e \\ \tau & ::= & o \mid \tau \to \tau \mid \tau \times \cdots \times \tau \end{array}$$

Most of the term constructions of λ^{ML} are standard. c^o stands for a constant of the base type o. x and X stand for an internal variable (local or argument) and an external identifier, respectively. This distinction is necessary for investigating the rigidity of records since only the external identifiers place rigidity constraint.

Term constructions $\lambda x : \tau.e$ and $(e_1\ e_2)$ define a lambda abstraction and a lambda application. Term constructions $(e_1, \ldots, e_n)$ and $\pi_i(e)$ define a record and a projection. The ML-let construction $\texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2\ \texttt{end}$ is also given in the syntax of the calculus for demonstrating multiple value local binding in the target calculus.

Type context is defined as a pair (Δ, Γ) where Δ is a sequence of type assumptions for external identifiers, and Γ is a sequence of type assumption for internal variables.

Figure 1 gives a set of typing rules for terms to derive typing judgments of the form $(\Delta, \Gamma) \vdash e : \tau$.

A program $P = e$ is typed by the judgment $\Delta \vdash P : \tau$ which is derived from $(\Delta, \emptyset) \vdash e : \tau$.

$$(\Delta,\Gamma) \vdash c^o : o$$

$$(\Delta,\Gamma) \vdash x : \tau \quad \text{if } (x:\tau) \in \Gamma$$

$$(\Delta,\Gamma) \vdash X : \tau \quad \text{if } (X:\tau) \in \Delta$$

$$\frac{(\Delta,\Gamma[x:\tau_1]) \vdash e : \tau_2}{(\Delta,\Gamma) \vdash \lambda x:\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{(\Delta,\Gamma) \vdash e_1 : \tau_1 \to \tau_2 \quad (\Delta,\Gamma) \vdash e_2 : \tau_1}{(\Delta,\Gamma) \vdash (e_1\ e_2) : \tau_2}$$

$$\frac{(\Delta,\Gamma) \vdash e_i : \tau_i \quad \text{for all } 1 \le i \le n}{(\Delta,\Gamma) \vdash (e_1,\ldots,e_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{(\Delta,\Gamma) \vdash e : \tau_1 \times \cdots \times \tau_n}{(\Delta,\Gamma) \vdash \pi_i(e) : \tau_i}$$

$$\frac{(\Delta,\Gamma) \vdash e_1 : \tau \quad (\Delta,\Gamma[x:\tau]) \vdash e_2 : \tau_2}{(\Delta,\Gamma) \vdash \texttt{let}\ x:\tau = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2}$$

Figure 1. Typing Rules for Terms in λ^{ML}

$$(\Delta,\Gamma) \vdash_A c^o : o$$

$$(\Delta,\Gamma) \vdash_A x : \tau \quad \text{if } (x:\tau) \in \Gamma$$

$$(\Delta,\Gamma) \vdash_A X : \tau \quad \text{if } (X:\tau) \in \Delta$$

$$\frac{(\Delta,\Gamma[x:\tau_1]) \vdash_A e : \tau_2}{(\Delta,\Gamma) \vdash_A \lambda x:\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{(\Delta,\Gamma) \vdash_A e_1 : \tau_1 \to \tau_2 \quad (\Delta,\Gamma) \vdash_A e_2 : \tau_1}{(\Delta,\Gamma) \vdash_A (e_1\ e_2) : \tau_2}$$

$$\frac{(\Delta,\Gamma) \vdash_A e_i : \tau_i \quad \text{for all } 1 \le i \le n}{(\Delta,\Gamma) \vdash_A (e_1:\tau_1,\ldots,e_n:\tau_n)^\varphi : (\tau_1 \times \cdots \times \tau_n)^\varphi}$$

$$\frac{(\Delta,\Gamma) \vdash_A e : (\tau_1 \times \cdots \times \tau_n)^\varphi}{(\Delta,\Gamma) \vdash_A \pi_i(e : (\tau_1 \times \cdots \times \tau_n)^\varphi) : \tau_i}$$

$$\frac{(\Delta,\Gamma) \vdash_A e_1 : \tau \quad (\Delta,\Gamma[x:\tau]) \vdash_A e_2 : \tau_2}{(\Delta,\Gamma) \vdash_A \texttt{let}\ x:\tau = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2}$$

Figure 2. Typing Rules for Terms in λ^A

2.2 The Annotated Calculus – λ^A

The representation analysis does not change any program construction. It only infers rigidity information and records them in types of target terms. The following grammar gives the syntax of terms, types, and programs of the target calculus.

$$\begin{array}{rcl} e & ::= & c^o \mid x \mid X \mid \lambda x:\tau.e \mid (e\ e) \mid (e:\tau,\ldots,e:\tau)^\varphi \\ & & \mid \pi_i(e:\tau) \mid \texttt{let}\ x:\tau = e\ \texttt{in}\ e\ \texttt{end} \\ P & ::= & e \\ \tau & ::= & o \mid \tau \to \tau \mid (\tau \times \cdots \times \tau)^\varphi \\ \varphi & ::= & R \mid F \end{array}$$

Most of term constructions in λ^A are similar to those in λ^{ML}, except for records and projections. To facilitate the development of the record unboxing algorithm presented in the next section, full type information is given in the constructions of records and projections.

A record expression has form $(e_1 : \tau_1, \ldots, e_n : \tau_n)^\varphi$ where τ_i is the type of element e_i. $(\tau_1 \times \cdots \times \tau_n)^\varphi$ is a type of records having rigidity φ. The annotation φ is either R (stands for rigid record types) or F (stands for flexible record types).

A type context is also defined by a pair (Δ, Γ) for type assumptions of external identifiers and bound variables. Since the consistency between the implementation of an external object and its usages inside a program is guaranteed by an interface (signature or type), we uniformly assume a rigid type for any record type appearing in an external identifier type. Thus all annotation φ appearing in Δ should be R.

Figure 2 gives the set of typing rules for terms in λ^A. The annotations on record types and terms add additional constraint about rigidity to the type consistency. For example, a function that

$$(G,E) \vdash_A c^o \Downarrow c^o$$

$$(G,E) \vdash_A X \Downarrow G(x)$$

$$(G,E) \vdash_A x \Downarrow E(x)$$

$$(G,E) \vdash_A \lambda x.e \Downarrow \langle\!\langle (G,E); \lambda x.e \rangle\!\rangle$$

$$\frac{(G,E) \vdash_A e_1 \Downarrow \langle\!\langle (G_0,E_0); \lambda x.e_0 \rangle\!\rangle \quad (G,E) \vdash_A e_2 \Downarrow v_2 \quad (G_0, E_0[x \mapsto v_2]) \vdash_A e_0 \Downarrow v_0}{(G,E) \vdash_A (e_1\ e_2) \Downarrow v_0}$$

$$\frac{(G,E) \vdash_A e_i \Downarrow v_i \quad \text{for each } 1 \leq i \leq n}{(G,E) \vdash_A (e_1 : \tau_1, \ldots, e_n : \tau_n)^{\varphi} \Downarrow (v_1, \ldots, v_n)^{\varphi}}$$

$$\frac{(G,E) \vdash_A e \Downarrow (v_1, \ldots, v_n)^{\varphi}}{(G,E) \vdash_A \pi_i(e : \tau) \Downarrow v_i}$$

$$\frac{(G,E) \vdash_A e_1 \Downarrow v_1 \quad (G, E[x \mapsto v_1]) \vdash_A e_2 \Downarrow v_2}{(G,E) \vdash_A \texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow v_2}$$

Figure 3. Operational Semantics of λ^A

$$\models_A c^o : o$$

$$\frac{\text{There exists } (\Delta, \Gamma) \text{ so that } \models_A G : \Delta \text{ and } \models_A E : \Gamma \quad (\Delta, \Gamma) \vdash_A \lambda x : \tau_1.e : \tau_1 \to \tau_2}{\models_A \langle\!\langle (G,E); \lambda x : \tau_1.e \rangle\!\rangle : \tau_1 \to \tau_2}$$

$$\frac{\models_A v_i : \tau_i \ \ (\text{for all } 1 \leq i \leq n)}{\models_A (v_1, \ldots, v_n)^{\varphi} : (\tau_1 \times \cdots \times \tau_n)^{\varphi}}$$

$$\models_A \emptyset : \emptyset$$

$$\frac{\models_A G : \Delta \quad \models_A v : \tau}{\models_A G[X \mapsto v] : \Delta[X : \tau]}$$

$$\frac{\models_A E : \Gamma \quad \models_A v : \tau}{\models_A E[x \mapsto v] : \Gamma[x : \tau]}$$

Figure 4. Typing Rules for Values of λ^A

has rigid record argument type should never take a flexible record argument, since the record argument may be unboxed which causes a disagreement between the function definition and its usages.

The typing rules for programs is given below

$$\frac{(\Delta, \emptyset) \vdash_A e : \tau \quad \text{all } \varphi \text{ in } \tau \text{ is } R}{\Delta \vdash e : \tau}$$

where we require that in addition to Δ the final type only contain R.

2.3 Operational Semantics and Type Soundness

We define a set of values for the annotated calculus by the following syntax.

$$\begin{array}{rcl} v & ::= & c^o \mid \langle\!\langle (G, E; \lambda x : \tau.e \rangle\!\rangle \mid (v, \ldots, v)^{\varphi} \\ G & ::= & \emptyset \mid G[X \mapsto v] \\ E & ::= & \emptyset \mid E[x \mapsto v] \end{array}$$

$\langle\!\langle (G,E); \lambda x : \tau.e \rangle\!\rangle$ is a function closure that encloses a runtime environment (G,E) and a function code $\lambda x : \tau.e$. G is the external runtime environment that maps external identifier to values. E is an internal runtime environment that maps internal variables to values.

Each record value of the form $(v_1, \ldots, v_n)^{\varphi}$ has an annotation φ indicating whether this is a rigid or flexible record.

The operational semantics is defined in the style of [8] by giving a set of rules to derive a evaluation relation of the form $(G,E) \vdash_A e \Downarrow v$, which reads: "$e$ evaluates to v under (G,E)". Figure 3 gives the set of evaluation rules.

The evaluation rule for records indicates that a rigid record expression will be evaluated to a rigid record value, and a flexible record expression will be evaluated to a flexible record value.

Traditionally, we have to check that the type system of λ^A is sound. For this purpose, we define the set of typing rules for values in figure 4.

The type system of the calculus is sound as shown in the following soundness theorem.

THEOREM 2.1. *Suppose* $(\Delta, \Gamma) \vdash_A e : \tau$*, For any* (G,E) *so that* $\models_A G : \Delta$ *and* $\models_A E : \Gamma$ *and* $(G,E) \vdash_A e \Downarrow v$ *succeeds, we also have* $\models_A v : \tau$.

PROOF. The proof is by induction on the derivation of the evaluation. We do case analysis on e. We only shows the cases of records and projections where annotation type appeared. Other cases can simply hold as shown similarly in conventional proof of the type soundness.

Case $e = (e_1 : \tau_1, \ldots, e_n : \tau_n)^{\varphi}$. Suppose $(G,E) \vdash_A (e_1 : \tau_1, \ldots, e_n : \tau_n)^{\varphi} \Downarrow (v_1, \ldots, v_n)^{\varphi}$ which is derived from $(G,E) \vdash_A e_i \Downarrow v_i$. Also suppose $(\Delta, \Gamma) \vdash_A (e_1 : \tau_1, \ldots, e_n : \tau_n)^{\varphi} : (\tau_1 \times \cdots \times \tau_n)^{\varphi}$ where $(\Delta, \Gamma) \vdash_A e_i : \tau_i$. Applying induction hypothesis for each evaluation of e_i, we have $\models_A v_i : \tau_i$. Then applying value typing rule for record, we obtain $\models_A (v_1, \ldots, v_n)^{\varphi} : (\tau_1 \times \cdots \times \tau_n)^{\varphi}$ as desired.

Case $e = \pi_i(e_r : \tau_r)$. By the typing rule for projection, e is typable only if τ_r has form $(\tau_1 \times \cdots \times \tau_n)^{\varphi}$. Suppose $(G,E) \vdash_A \pi_i(e_0) \Downarrow v_i$ is derived from $(G,E) \vdash_A e_r \Downarrow (v_1, \ldots, v_{n'})^{\varphi'}$. Also suppose $(\Delta, \Gamma) \vdash_A \pi_i(e_r : \tau_r) : \tau_i$ which is derived from $(\Delta, \Gamma) \vdash_A e_r : (\tau_1 \times \cdots \times \tau_n)^{\varphi}$. Applying induction hypothesis on the evaluation of e_r, we obtain $\models_A (v_1, \ldots, v_{n'})^{\varphi'} : (\tau_1 \times \cdots \times \tau_n)^{\varphi}$. By record value typing rule, we should have $n = n'$, $\varphi = \varphi'$, and $\models_A v_i : \tau_i$ as desired. □

2.4 Annotation Inference

This subsection presents an annotation inference algorithm which aims to infer all annotation for a given source program.

Obviously, if we set the annotation in each record type and subexpression of the program to R (rigid), the type checking for the resulting program should be correct. This strategy, however, does not give any useful information for the record unboxing algorithm presented in the next section since all record expressions are considered as rigid.

The main goal of this inference algorithm is to find as many flexible records as possible in the given program. For this goal, we develop an annotation inference algorithm in the style of type-based control flow analysis [13].

First, we extend the notion of annotation so that a record type or term can take an *annotation variable* as its rigidity information. The annotation inference algorithm starts with a simple annotated program (a program obtained from the source code by assigning fresh annotation variable to each record type and term) and an external annotated type context where all record types are rigid (annotated with R). Then the algorithm infers a set of *rigidity constraints* on these annotation variables. Finally we solve the set of constraints to find the best solution for each annotation variable.

In [13] the annotation inference algorithm has been done together with ordinary type inference algorithm. By this way, annotations and the standard types (*underlying types*) are inferred simultaneously. In this paper, we consider another approach: separating the annotation inference from the conventional type inference. More specifically, our annotation inference algorithm assumes underlying explicitly typed expressions as inputs and produces annotated well-typed expressions as outputs.

The reason behind this decision is that most of modern compilers have a powerful and complicated type inference algorithm for inferring underlying type. Separating annotation inference from type inference would simplify and modularize the development of the compiler.

For example, in our compiler, the type inference is performed in a very early stage of the compilation process. We integrate the annotation inference after several intermediate compilation steps, such as pattern matching, module flattening, which normalize the source code to a simple typed intermediate code. Performing annotation inference in this place should therefore be much easier than in an early stage where records may appear in some derived forms.

The rest of this subsection presents two phases of the annotation inference algorithm: finding the constraint set and solving these constraints.

2.4.1 Simple Annotation Inference

We define a simple annotated calculus which serves as an intermediate calculus for the annotation inference algorithm by extending the notion of annotation as described above. The syntax of annotation is now defined as

$$\varphi \quad ::= \quad R \mid F \mid \alpha$$

where α stands for an annotation variable.

Given a well-typed source program P under an external type context Δ, i.e. $\Delta \vdash P : \tau$, the goal of the simple annotation inference is to generate an intermediate code P' of P, and a set of constraints on annotation variables appearing in P'.

Each annotation constraint appears in the form $\varphi_1 = \varphi_2$ which indicates that all records annotated by φ_1 and φ_2 should have the same rigidity attribute (of course, we should not accept a constraint like $R = F$ or $F = R$).

Existence of annotation variables is the only difference between this simple annotation calculus and the annotated calculus λ^A presented in the previous sub-section. If we substitute all the annotation variables in a term in the simple annotation calculus with concrete annotations (i.e., R or F), we obtain a term in the annotated calculus. We define an *annotation substitution* θ as a map from the set of annotation variables to the set of rigidity $\{R, F\}$. θ extends to the set of rigidities by setting $\theta(R) = R$ and $\theta(F) = F$. We say that θ satisfies a constraint $\varphi_1 = \varphi_2$, written $\theta \models \varphi_1 = \varphi_2$ if $\theta(\varphi_1) = \theta(\varphi_2)$. We also extend this predicate for the set of constraints C. A annotation substitution θ satisfies a set of constraints C, written $\theta \models C$ if θ satisfies all constraints in C.

Let θ be an annotation substitution, we say θ *covers* a simple annotated context Γ, a simple annotated expression e, or a simple annotated type τ, if $dom(\theta)$ includes all annotation variables appearing in Γ, e, or τ, respectively.

Similar to a unification-based inferencer, we define a unification algorithm for two simple annotated types τ_1 and τ_2, which have the same underlying type, as follows.

$$\begin{array}{rcl}
\mathcal{U}(o,o) & = & \emptyset \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & \mathcal{U}(\tau_1,\tau_1') \cup \mathcal{U}(\tau_2,\tau_2') \\
\mathcal{U}((\tau_1 \times \cdots \times \tau_n)^{\alpha}, (\tau_1' \times \cdots \times \tau_n')^{\alpha'}) & = & \{\alpha = \alpha'\} \cup \bigcup \mathcal{U}(\tau_i,\tau_i')
\end{array}$$

$\mathcal{W}_e(\Delta, \Gamma, c^o) = (c^o, o, \emptyset)$

$\mathcal{W}_e(\Delta, \Gamma, X) = (X, rigid(\Delta(x)), \emptyset)$

$\mathcal{W}_e(\Delta, \Gamma, x) = (x, \Gamma(x), \emptyset)$

$\mathcal{W}_e(\Delta, \Gamma, \lambda x : \tau.e_0) =$
  $\texttt{let}\ \tau' = fresh(\tau)$
    $(e_0', \tau_0, C) = \mathcal{W}_e(\Delta, \Gamma[x : \tau'], e_0)$
  $\texttt{in}\ (\lambda x : \tau'.e_0', \tau' \to \tau_0, C)$
  $\texttt{end}$

$\mathcal{W}_e(\Delta, \Gamma, (e_1\ e_2)) =$
  $\texttt{let}\ (e_1', \tau_1 \to \tau_2, C_1) = \mathcal{W}_e(\Delta, \Gamma, e_1)$
    $(e_2', \tau_1', C_2) = \mathcal{W}_e(\Delta, \Gamma, e_2)$
    $C_3 = \mathcal{U}(\tau_1, \tau_1')$
  $\texttt{in}\ ((e_1'\ e_2'), \tau_2, C_1 \cup C_2 \cup C_3)$
  $\texttt{end}$

$\mathcal{W}_e(\Delta, \Gamma, (e_1, \ldots, e_n)) =$
  $\texttt{let}\ (e_i', \tau_i, C_i) = \mathcal{W}_e(\Delta, \Gamma, e_i)$ for all $1 \leq i \leq n$
    α is fresh
  $\texttt{in}\ ((e_1' : \tau_1, \ldots, e_n' : \tau_n)^{\alpha}, (\tau_1 \times \cdots \times \tau_n)^{\alpha},$
    $C_1 \cup \cdots \cup C_n)$
  $\texttt{end}$

$\mathcal{W}_e(\Delta, \Gamma, \pi_i(e)) =$
  $\texttt{let}\ (e', (\tau_1 \times \cdots \times \tau_n)^{\alpha}, C) = \mathcal{W}_e(\Delta, \Gamma, e)$
  $\texttt{in}\ (\pi_i(e'), \tau_i, C)$
  $\texttt{end}$

$\mathcal{W}_e(\Delta, \Gamma, \texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2\ \texttt{end}) =$
  $\texttt{let}\ (e_1', \tau', C_1) = \mathcal{W}_e(\Delta, \Gamma, e_1)$
    $(e_2', \tau_2', C_2) = \mathcal{W}_e(\Delta, \Gamma[x \mapsto \tau'], e_2)$
  $\texttt{in}\ (\texttt{let}\ x : \tau' = e_1'\ \texttt{in}\ e_2'\ \texttt{end}, \tau_2', C_1 \cup C_2)$
  $\texttt{end}$

Figure 5. Annotation Inference Algorithm

The unification algorithm produces a set of annotation constraints from two given simple annotated types τ_1 and τ_2. Since we restrict that τ_1 and τ_2 have the same underlying type, the unification algorithm always succeeds and have the following property.

LEMMA 2.2. *Given τ_1 and τ_2 be two simple annotated types which have the same underlying type. The following properties hold.*

- *$\mathcal{U}(\tau_1, \tau_2)$ succeeds, and*
- *for any θ such that $\theta \models \mathcal{U}(\tau_1, \tau_2)$, we have $\theta(\tau_1) = \theta(\tau_2)$.*

Based on this unification algorithm, we design an annotation inference algorithm for expressions in the form $(e', \tau', C) = \mathcal{W}_e(\Delta, \Gamma, e)$, where e is a source expression in λ^{ML}, Δ is a source external type context, Γ is a simple annotated type context for internal variables, e', τ' are the target term and type of e in simple annotation calculus, and C is the derived set of constraints.

The algorithm is given in figure 5. In the case of external identifiers, the algorithm infers type of X as $rigid(\Delta(X))$. The function $rigid(\tau)$ takes an underlying type τ and produces a simple annotated type τ' by annotating all record types appearing in τ with R. This function is defined as follows.

$$\begin{array}{rcl}
rigid(o) & = & o \\
rigid(\tau_1 \to \tau_2) & = & rigid(\tau_1) \to rigid(\tau_2) \\
rigid(\tau_1 \times \cdots \times \tau_n) & = & (rigid(\tau_1) \times \cdots \times rigid(\tau_n))^R
\end{array}$$

Similarly, in the case for functions, a fresh simple annotated type is created from the underlying type τ by the function

$fresh(\tau)$ which is defined as

$$\begin{array}{rcl} fresh(o) & = & o \\ fresh(\tau_1 \to \tau_2) & = & fresh(\tau_1) \to fresh(\tau_2) \\ fresh(\tau_1 \times \cdots \times \tau_n) & = & (fresh(\tau_1) \times \cdots \times fresh(\tau_n))^\alpha \\ & & \text{where } \alpha \text{ is a fresh annotation variable} \end{array}$$

The annotation inference algorithm generates the target expression, its type, and a set of constraints, which is accumulated during unification.

We extend the function $rigid$ for external type context so that it transforms an external type context Δ in λ^{ML} to an external type context Δ' in λ^A by setting all annotations to R.

We define $\lfloor\tau\rfloor$ as the underlying type of τ, and define $\lfloor\Gamma\rfloor$ as the underlying context of Γ.

The annotation inference algorithm for expressions satisfies the following property.

LEMMA 2.3. *Given a source expression e, an external type context Δ in λ^{ML}, and a simple annotated type context Γ such that $(\Delta, \lfloor\Gamma\rfloor) \vdash e : \tau$.*

- *the annotation inference algorithm succeeds as $(e', \tau', C) = \mathcal{W}_e(\Delta, \Gamma, e)$ and $\tau = \lfloor\tau'\rfloor$*
- *for any annotation substitution θ that covers Γ ,e', τ', and $\theta \models C$, we have $(rigid(\Delta), \theta(\Gamma)) \vdash_A \theta(e') : \theta(\tau')$.*

PROOF. Due to the space limit, we only sketch out the proof.

Firstly, the inference algorithm for expressions always succeeds. Although there are some forms of pattern matching on types in the inference rules (the rule for applications and the rule for projections), these cases should succeed since we assume that source expressions are well typed.

The algorithm only infers annotations, then the underlying type of the target term should be the same as type of the source term, i.e. $\tau = \lfloor\tau'\rfloor$.

The second conclusion can be proved by induction on the structure of e. Let us show one typical case of application $e = (e_1\ e_2)$.

Suppose $(\Delta, \lfloor\Gamma\rfloor) \vdash (e_1\ e_2) : \tau$ is derived from $(\Delta, \lfloor\Gamma\rfloor) \vdash e_1 : \tau_1 \to \tau$ and $(\Delta, \lfloor\Gamma\rfloor) \vdash e_2 : \tau_1$. Since the algorithm succeeds, we have

- $(e_1', \tau_1' \to \tau', C_1) = \mathcal{W}_e(\Delta, \Gamma, e_1)$
- $(e_2', \tau_2', C_2) = \mathcal{W}_e(\Delta, \Gamma, e_2)$
- $C_3 = \mathcal{U}(\tau_1', \tau_2')$
- $e' = (e_1'\ e_2')$, $C = C_1 \cup C_2 \cup C_3$.

θ covers e so it also covers e_1 and e_2. Then by induction hypothesis for e_1 and e_2, we have

- $(rigid(\Delta), \theta(\Gamma)) \vdash_A \theta(e_1') : \theta(\tau_1') \to \theta(\tau')$
- $(rigid(\Delta), \theta(\Gamma)) \vdash_A \theta(e_2') : \theta(\tau_2')$

Since $\theta \models C$, we also have $\theta \models C_3$. From the first conclusion of the property, we can derive that τ_1' and τ_2' has the same underlying type, i.e. τ_1. Then by lemma 2.2, we achieve that $\theta(\tau_1') = \theta(\tau_2')$. Applying the typing rule for application, we obtain $(rigid(\Delta), \theta(\Gamma)) \vdash_A \theta(e') : \theta(\tau')$ as desired. □

Based on the inference algorithm for expression, we design the inference algorithm for programs as $(P', \tau', C) = \mathcal{W}_P(\Delta, P)$. Since the result of a program can be used by another compilation unit, we need to put the constraint that the type of the program only contain rigid record types.

We define a function $rigidCon(\tau)$ for producing such constraints as follows.

$$\begin{array}{rcl} rigidCon(o) & = & \emptyset \\ rigidCon(\tau_1 \to \tau_2) & = & rigidCon(\tau_1) \cup rigidCon(\tau_2) \\ rigidCon((\tau_1 \times \cdots \times \tau_n)^\varphi) & = & \{\varphi = R\} \cup \bigcup rigitCon(\tau_i) \end{array}$$

The inference algorithm for a program is therefore defined as

$$\begin{array}{l} \mathcal{W}_P(\Delta, P) = \\ \quad \texttt{let} \\ \qquad (P', \tau', C_1) = \mathcal{W}_e(\Delta, \emptyset, P) \\ \qquad C_2 = rigidCon(\tau') \\ \quad \texttt{in } (P', \tau', C_1 \cup C_2) \texttt{ end} \end{array}$$

The following theorem holds.

THEOREM 2.4. *Given a source program P and an external type context Δ such that $\Delta \vdash P : \tau$. We have*

- *the annotation inference algorithm for programs succeeds as $(P', \tau', C) = \mathcal{W}_P(\Delta, P)$, and $rigid(\tau) = \tau'$*
- *for any annotation substitution θ that covers P', and $\theta \models C$, we have $rigid(\Delta) \vdash_A \theta(P') : \theta(\tau')$.*

PROOF. This theorem can be easily proved by applying the lemma 2.3. The only extra thing we have to show here is to prove that the target type τ' of the program P only contains rigid annotation. This is also trivial since the set of inferred constraints C includes $rigidCon(\tau')$ by the definition of the annotation inference algorithm for programs. □

2.4.2 Constraint Resolution

Given a program P, application of the annotation inference algorithm $\mathcal{W}_P(\Delta, P)$ produces (P', τ', C). Let V be the set of annotation variables in C. To obtain an annotated program, we need to construct the largest substitution θ on V that satisfy C point wise ordered as $R \sqsubseteq F$.

Since Δ only contain rigidity R, constraint in C is either of one of the following forms: $R = R$, $\alpha = R$, $R = \alpha$, or $\alpha_1 = \alpha_2$, So the best substitution θ is obtained by first unifying all C and then setting all the remaining annotation variables to F.

3. Record Unboxing

In previous section, we developed a representation analysis that classifies a record appearing in a given source program into either a rigid record or a flexible one.

Based on this static information, we develop an optimization method, namely record unboxing, which changes representations of all flexible records by using the following transformation strategy.

- A flexible record is transformed into multiple values whose values are either allocated on stack or flattened on the heap block containing the record.
- A function that takes a flexible record argument is transformed into a multiple argument function.
- A function that returns a flexible record is transformed into a multiple return value function.

In the rest of this section, we introduce a multiple value calculus λ^M as the target calculus of the transformation. We then develop the transformation algorithm, and show that it preserves typing. Due to the limitation of space, the semantic correctness is not included in this paper. We would like to present this in an extend version of the paper.

3.1 Multiple Value Calculus – λ^M

We define a calculus that includes multiple argument/multiple return value functions and simultaneous multiple value `let` bindings. Since some constructs are restricted to single values, the language consists of two classes of terms: single value terms (ranged over by e^1) and multiple value terms (ranged over by e^n). Top level terms (ranged over by P) must be a single value term. They are defined by the following syntax.

$$\begin{array}{rcl} e^1 & ::= & c^o \mid x \mid X \mid \lambda\{x,\ldots,x\}.e^n \mid (e^n,\ldots,e^n) \mid \pi_i(e^1) \\ e^n & ::= & e^1 \mid \{e^n,\ldots,e^n\} \mid (e^1\; e^n) \\ & & \mid \texttt{let}\,\{x,\ldots,x\} = e^n \;\texttt{in}\; e^n \;\texttt{end} \\ P & ::= & e^1 \end{array}$$

Some explanations of the language are in order.

- Sometimes we write e instead of e^1 or e^n in a context that clearly distinguishes between these two cases.
- This syntax restricts multiple values to be non first class objects. Note also that nested multiple value expressions are allowed. A component e_i of a multiple value expression $\{e_1,\ldots,e_n\}$ may also be another multiple value term. In such case $\{e_1,\ldots,e_n\}$ is evaluated to a value sequence which is the concatenation of the results of its components.
- Variables (external identifiers and internal variables) are restricted to single value terms, i.e. they can only be bound to single values.
- Functions and records are single value terms since their runtime representations are just pointers (i.e. pointers to closure blocks or record blocks). A function $\lambda\{x_1,\ldots,x_m\}.e^n$ takes a sequence of m values as arguments to produce another sequence of values resulted from the evaluation of e^n.
- A record can also be composed from multiple value expressions. Value of a record $(e_1,\ldots,e_n)$ is a heap block whose content is the concatenation of the value sequences obtained from evaluating $e_1,\ldots,e_n$.
- The language only includes single value projection. $\pi_i(e)$ extracts the i^{th} (single value) element from the record value obtained from evaluating e. As a consequence, the result of $\pi_i((e_1,\ldots,e_n))$ is the ith value in the resulting single value sequence and is not necessarily the value denoted by e_i.
- A `let` expression $\texttt{let}\;\{x_1,\ldots,x_m\} = e_1\;\texttt{in}\;e_2\;\texttt{end}$ represents a multiple value binding which simultaneously binds m values obtained from evaluating e_1 to m local variables $x_1,\ldots,x_m$ for the computation of e_2.

A single value term denotes a single value (ranged over by v^1) and a multiple value term e^n denotes a sequence of (single) values (ranged over by $\{v^1,\ldots,v^1\}$). They are defined by the following syntax.

$$\begin{array}{rcl} v^1 & ::= & c^o \mid \langle\!\langle (G,E);\lambda\{x,\ldots,x\}.e^n\rangle\!\rangle \mid (v^1,\ldots,v^1) \\ v^n & ::= & v^1 \mid \{v^1,\ldots,v^1\} \\ G & ::= & \emptyset \mid G[X \mapsto v^1] \\ E & ::= & \emptyset \mid E[x \mapsto v^1] \end{array}$$

Figure 6 defines a set of evaluation rules for terms in λ^M. For the consistency, we assume that the following syntaxes are equivalent: e and $\{e\}$, v and $\{v\}$ for single value v and single value expression e.

$$G,E \vdash_M c^o \Downarrow c^o$$

$$G,E \vdash_M x \Downarrow E(x)$$

$$G,E \vdash_M X \Downarrow G(X)$$

$$G,E \vdash_M \lambda\{x_1,\ldots,x_m\}.e \Downarrow \langle\!\langle G,E;\lambda\{x_1,\ldots,x_m\}.e\rangle\!\rangle$$

$$\frac{G,E \vdash_M e_i \Downarrow \{v_{1,i},\ldots,v_{k_i,i}\} \quad \text{for each } 1 \le i \le m}{\begin{array}{l} G,E \vdash_M \{e_1,\ldots,e_m\} \Downarrow \\ \quad \{v_{1,1},\ldots,v_{k_1,1},\ldots,v_{1,m},\ldots,v_{k_m,m}\} \end{array}}$$

$$\frac{\begin{array}{c} G,E \vdash_M e_1 \Downarrow \langle\!\langle G_0,E_0;\lambda\{x_1,\ldots,x_m\}.e^n\rangle\!\rangle \\ G,E \vdash_M e_2 \Downarrow \{v_1,\ldots,v_m\} \\ G_0,E_0[x_1 \mapsto v_1]\cdots[x_m \mapsto v_m] \vdash_M e_0 \Downarrow v^n \end{array}}{G,E \vdash_M (e_1\; e_2) \Downarrow v^n}$$

$$\frac{G,E \vdash_M e_i \Downarrow \{v_{1,i},\ldots,v_{k_i,i}\} \quad \text{for each } 1 \le i \le m}{\begin{array}{l} G,E \vdash_M (e_1,\ldots,e_m) \Downarrow \\ \quad (v_{1,1},\ldots,v_{k_1,1},\ldots,v_{1,m},\ldots,v_{k_m,m}) \end{array}}$$

$$\frac{G,E \vdash_M e \Downarrow (v_1,\ldots,v_n)}{G,E \vdash_M \pi_i(e) \Downarrow v_i}$$

$$\frac{\begin{array}{c} G,E \vdash_M e_1 \Downarrow \{v_1,\ldots,v_m\} \\ G,E[x_1 \mapsto v_1]\cdots[x_m \mapsto v_m] \vdash_M e_2 \Downarrow v \end{array}}{G,E \vdash_M \texttt{let}\;\{x_1,\ldots,x_m\} = e_1\;\texttt{in}\;e_2\;\texttt{end} \Downarrow v}$$

Figure 6. Operational semantics

3.2 Type System

The sets of single value types (ranged over by τ^1) and multiple value types (ranged over by τ^n) are given by the following grammar.

$$\begin{array}{rcl} \tau^1 & ::= & o \mid \tau^n \to \tau^n \mid (\tau^1,\ldots,\tau^1) \\ \tau^n & ::= & \tau^1 \mid \{\tau^1,\ldots,\tau^1\} \end{array}$$

A multiple value type $\{\tau_1,\ldots,\tau_n\}$ is just a sequence of single value types $\tau_1,\ldots,\tau_n$ denoting sequences of single values. $\tau_1 \to \tau_2$ is a multiple argument/multiple return value function type which denotes functions that can take m arguments of the multiple value type τ_1, and return n values of the multiple value type τ_2. Record type $\tau_r = (\tau_1,\ldots,\tau_n)$ denotes a set of record values whose i^{th} element has the single value type τ_i.

We define type context as a pair (Δ,Γ), where Δ and Γ are the external and internal type contexts that map external identifiers and internal variables to single values.

Figure 7 gives a set of typing rules to derive typing judgments for terms of the form $(\Delta,\Gamma) \vdash_M e : \tau$.

We also define a set of typing rules for values which is depicted in figure 8.

The type system of λ^M is sound with respect to the operational semantics as shown by the following theorem.

THEOREM 3.1. *Suppose* $(\Delta,\Gamma) \vdash_M e : \tau$. *For any* (G,E) *such that* $\models_M G : \Delta$ *and* $\models_M E : \Gamma$, *if* $(G,E) \vdash_M e \Downarrow v$ *succeeds then we have* $\models_M v : \tau$.

PROOF. We prove this theorem by induction on the length of the evaluation. We only show the cases of multiple value expressions and records. Other cases are almost standard.

Case $e = \{e_1,\ldots,e_m\}$.

Suppose $(G,E) \vdash_M \{e_1,\ldots,e_m\} \Downarrow v$. This must be derived from $(G,E) \vdash_M e_i \Downarrow \{v_{1,i},\ldots,v_{k_i,i}\}$ for each i such that

$$(\Delta, \Gamma) \vdash_M c^o : o$$

$$(\Delta, \Gamma) \vdash_M x : \Gamma(x)$$

$$(\Delta, \Gamma) \vdash_M X : \Delta(X)$$

$$\frac{(\Delta, \Gamma) \vdash_M e_i : \{\tau_{1,i}, \ldots, \tau_{k_i,i}\} \quad \text{for each } 1 \leq i \leq m}{(\Delta, \Gamma) \vdash_M \{e_1, \ldots, e_m\} : \{\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,m}, \ldots, \tau_{k_m,m}\}}$$

$$\frac{(\Delta, \Gamma[x_1 : \tau_1] \cdots [x_m : \tau_m]) \vdash_M e^n : \tau^n}{(\Delta, \Gamma) \vdash_M \lambda\{x_1, \ldots, x_m\}.e^n : \{\tau_1, \ldots, \tau_m\} \to \tau^n}$$

$$\frac{(\Delta, \Gamma) \vdash_M e_1 : \tau_1 \to \tau_2 \quad (\Delta, \Gamma) \vdash_M e_2 : \tau_1}{(\Delta, \Gamma) \vdash_M (e_1\ e_2) : \tau_2}$$

$$\frac{(\Delta, \Gamma) \vdash_M e_i : \{\tau_{1,i}, \ldots, \tau_{k_i,i}\} \quad \text{for each } 1 \leq i \leq m}{(\Delta, \Gamma) \vdash_M (e_1, \ldots, e_m) : (\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,m}, \ldots, \tau_{k_m,m})}$$

$$\frac{(\Delta, \Gamma) \vdash_M e : (\tau_1, \ldots, \tau_n)}{(\Delta, \Gamma) \vdash_M \pi_i(e) : \tau_i}$$

$$\frac{(\Delta, \Gamma) \vdash_M e_1 : \{\tau_1, \ldots, \tau_m\} \quad (\Delta, \Gamma[x_1 : \tau_1] \cdots [x_m : \tau_m]) \vdash_M e_2 : \tau^n}{(\Delta, \Gamma) \vdash_M \texttt{let}\ \{x_1, \ldots, x_m\} = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau^n}$$

Figure 7. Typing Rules for Terms of λ^M

$$\models_M c^o : o$$

$$\frac{\models_M v_i : \tau_i \quad 1 \leq i \leq n}{\models_M \{v_1, \ldots, v_n\} : \{\tau_1, \ldots, \tau_n\}}$$

$$\frac{\text{There exists } (\Delta, \Gamma) \text{ so that } \models_M G : \Delta \text{ and } \models_M E : \Gamma \quad (\Delta, \Gamma) \vdash_M \lambda\{x_1, \ldots, x_m\}.e^n : \tau^m \to \tau^n}{\models_M \langle\!\langle (G, E); \lambda\{x_1, \ldots, x_m\}.e^n \rangle\!\rangle : \tau^m \to \tau^n}$$

$$\frac{\models_M v_i : \tau_i \quad 1 \leq i \leq n}{\models_M (v_1, \ldots, v_n) : (\tau_1, \ldots, \tau_n)}$$

$$\models_M \emptyset : \emptyset$$

$$\frac{\models_M G : \Delta \quad \models_M v : \tau}{\models_M G[X \mapsto v] : \Delta[X : \tau]}$$

$$\frac{\models_M E : \Gamma \quad \models_M v : \tau}{\models_M E[x \mapsto v] : \Gamma[x : \tau]}$$

Figure 8. Value typing

$$o \sim o$$

$$\frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

$$\frac{\tau_i \sim \{\tau_{1,i}, \ldots, \tau_{k_i,i}\} \quad \text{for all } 1 \leq i \leq n}{(\tau_1 \times \cdots \times \tau_n)^F \sim \{\tau_{(1,1)}, \ldots, \tau_{(k_1,1)}, \ldots, \tau_{(1,n)}, \ldots, \tau_{(k_n,n)}\}}$$

$$\frac{\tau_i \sim \{\tau_{1,i}, \ldots, \tau_{k_i,i}\} \quad \text{for all } 1 \leq i \leq n}{(\tau_1 \times \cdots \times \tau_n)^R \sim (\tau_{(1,1)}, \ldots, \tau_{(k_1,1)}, \ldots, \tau_{(1,n)}, \ldots, \tau_{(k_n,n)})}$$

Figure 9. Simulation Relation on Types

$v = \{v_{1,1}, \ldots, v_{k_1,1}, \ldots, v_{1,m}, \ldots, v_{k_m,m}\}$.
Also suppose $(\Delta, \Gamma) \vdash_M \{e_1, \ldots, e_m\} : \tau$ which must be derived from $(\Delta, \Gamma) \vdash_M e_i : \{\tau_{1,i}, \ldots, \tau_{k_i',i}\}$ for each i such that
$\tau = \{\tau_{1,1}, \ldots, \tau_{k_1',1}, \ldots, \tau_{1,m}, \ldots, \tau_{k_m',m}\}$.
Applying induction hypothesis for each evaluation of e_i, we have $\models_M \{v_{1,i}, \ldots, v_{k_i,i}\} : \{\tau_{1,i}, \ldots, \tau_{k_i',i}\}$. By the value typing rule for multiple value, we have $k_i = k_i'$ and $\models_M v_{j,i} : \tau_{j,i}$ for all $1 \leq i \leq m$ and $1 \leq j \leq k_i$. Applying the multiple value typing rule, we can conclude $\models_M v : \tau$ as desired.

Case $e = (e_1, \ldots, e_m)$.
Suppose $(G, E) \vdash_M (e_1, \ldots, e_m) \Downarrow v$. This must be derived from $(G, E) \vdash_M e_i \Downarrow \{v_{1,i}, \ldots, v_{k_i,i}\}$ for each i such that
$v = (v_{1,1}, \ldots, v_{k_1,1}, \ldots, v_{1,m}, \ldots, v_{k_m,m})$.
Also suppose $(\Delta, \Gamma) \vdash_M (e_1, \ldots, e_m) : \tau$ which must be derived from $(\Delta, \Gamma) \vdash_M e_i : \{\tau_{1,i}, \ldots, \tau_{k_i',i}\}$ for each i such that
$\tau = (\tau_{1,1}, \ldots, \tau_{k_1',1}, \ldots, \tau_{1,m}, \ldots, \tau_{k_m',m})$.
Applying induction hypothesis for each evaluation of e_i, we have $\models_M \{v_{1,i}, \ldots, v_{k_i,i}\} : \{\tau_{1,i}, \ldots, \tau_{k_i',i}\}$. By the value typing rule for multiple value, we have $k_i = k_i'$ and $\models_M v_{j,i} : \tau_{j,i}$ for all $1 \leq i \leq m$ and $1 \leq j \leq k_i$. Applying the value typing rule for records, we can conclude $\models_M v : \tau$ as desired. □

3.3 Record Unboxing Algorithm

The record unboxing algorithm is developed based on the annotations on types inferred from the representation analysis presented in the previous section.

Given a well-typed source expression e under a type context (Δ, Γ) in λ^A, the algorithm is formulated as $\Sigma \vdash e \rightsquigarrow e'$ where e' is the target expression in λ^M, and Σ is the compile context that maps each free variable of e to a sequence of distinct variables in the target calculus, i.e. $[x \mapsto \{x_1, \ldots, x_n\}]$.

In the source program, a flexible record may be bound to a variable x. This record is unboxed by the algorithm, i.e. transformed into a multiple value term. Values evaluated from the record should also be bound to variables in the target code. The variable mapping $[x \mapsto \{x_1, \ldots, x_n\}]$ records the target variables $x_1, \ldots, x_n$ for such a source variable x.

Types of the target expressions simulate types of the source expressions as shown by simulation relations defined in figure 9

The compile context Σ should respect the internal typing context Γ of the source expression. For example, if a variable x has a flexible record type τ_r and $\{\tau_1, \ldots, \tau_n\}$ is the target type of x, i.e. $\tau_r \sim \{\tau_1, \ldots, \tau_n\}$, then in Σ, x should map to exactly n variables, i.e. $[x \mapsto \{x_1, \ldots, x_n\}]$, and each x_i should have type τ_i. The context simulation relation $\Gamma \sim_\Sigma \Gamma'$ defined below indicates this constraints.

$$\Sigma \vdash c^o \leadsto c^o$$

$$\Sigma \vdash X \leadsto X$$

$$\Sigma \vdash x \leadsto \{x_1, \ldots, x_n\} \quad \text{For } [x \mapsto \{x_1, \ldots, x_n\}] \in \Sigma$$

$$\frac{\tau \sim \tau' \quad m = \mathcal{K}(\tau') \quad \{x_1, \ldots, x_m\} \text{ are fresh} \quad \Sigma[x \mapsto \{x_1, \ldots, x_m\}] \vdash e \leadsto e'}{\Sigma \vdash \lambda x : \tau.e \leadsto \lambda\{x_1, \ldots, x_m\}.e'}$$

$$\frac{\Sigma \vdash e_1 \leadsto e_1' \quad \Sigma \vdash e_2 \leadsto e_2'}{\Sigma \vdash (e_1\ e_2) \leadsto (e_1'\ e_2')}$$

$$\frac{\Sigma \vdash e_i \leadsto e_i' \quad \text{for each } 1 \leq i \leq m}{\Sigma \vdash (e_1 : \tau_1, \ldots, e_m : \tau_m)^R \leadsto (e_1', \ldots, e_m')}$$

$$\frac{\Sigma \vdash e_i \leadsto e_i' \quad \text{for each } 1 \leq i \leq m}{\Sigma \vdash (e_1 : \tau_1, \ldots, e_m : \tau_m)^F \leadsto \{e_1', \ldots, e_m'\}}$$

$$\frac{\Sigma \vdash e \leadsto e' \quad \tau_j \sim \{\tau_{1,j}, \ldots, \tau_{k_j,j}\} \quad s_0 = 0 \quad s_j = s_{j-1} + k_j \quad x \text{ is fresh}}{\Sigma \vdash \pi_i(e : (\tau_1 \times \cdots \times \tau_n)^R) \leadsto \texttt{let } [x] = e' \texttt{ in } \{\pi_{s_{i-1}+1}(x), \ldots, \pi_{s_i}(x)\} \texttt{ end}}$$

$$\frac{\Sigma \vdash e \leadsto e' \quad \tau_j \sim \{\tau_{1,j}, \ldots, \tau_{k_j,j}\} \quad x_{i,j} \text{ fresh}}{\Sigma \vdash \pi_i(e : (\tau_1 \times \cdots \times \tau_n)^F) \leadsto \texttt{let } [x_{1,1}, \ldots, x_{k_1,1}, \ldots, x_{1,n}, \ldots, x_{k_n,n}] = e' \texttt{ in } \{x_{1,i}, \ldots, x_{k_i,i}\} \texttt{ end}}$$

$$\frac{\Sigma \vdash e_1 \leadsto e_1' \quad \tau \sim \tau' \quad k = \mathcal{K}(\tau') \quad x_i \text{ fresh} \quad \Sigma[x \mapsto \{x_1, \ldots, x_k\}] \vdash e_2 \leadsto e_2'}{\Sigma \vdash \texttt{let } x : \tau = e_1 \texttt{ in } e_2 \texttt{ end} \leadsto \texttt{let } \{x_1, \ldots, x_k\} = e_1' \texttt{ in } e_2' \texttt{ end}}$$

Figure 10. Record Unboxing Algorithm

$$\emptyset \sim_\Sigma \emptyset$$

$$\frac{\Gamma \sim_\Sigma \Gamma' \quad \tau \sim \{\tau_1, \ldots, \tau_n\} \quad [x \mapsto \{x_1, \ldots, x_n\}] \in \Sigma}{\Gamma[x : \tau] \sim_\Sigma \Gamma'[x_1 : \tau_1] \cdots [x_n : \tau_n]}$$

(for consistency, we also consider that $\{\tau\}$ is equivalent to τ for any single type τ).

Let us define cardinality of a multiple value type τ, written $\mathcal{K}(\tau)$, as follows

$$\begin{array}{rcl} \mathcal{K}(o) & = & 1 \\ \mathcal{K}(\tau_1 \to \tau_2) & = & 1 \\ \mathcal{K}((\tau_1, \ldots, \tau_n)) & = & 1 \\ \mathcal{K}(\{\tau_1, \ldots, \tau_n\}) & = & n \end{array}$$

$\mathcal{K}(\tau)$ computes the number of single values contained in a sequence of values denoted by τ.

Figure 10 depicts the set of rules to derive the compilation judgment $\Sigma \vdash e \leadsto e'$.

Since we assume that an external identifier always has rigid type, so we never "unbox" it, i.e. transform it into a sequence of variables.

A variable x may have flexible record type. This should be "unboxed", i.e. transformed into a sequence of variables as shown in the transformation rule.

A function $\lambda x : \tau.e$ is transformed into a multiple argument function $\lambda\{x_1, \ldots, x_n\}.e'$ if τ is a flexible record type. In this case n is the cardinality of the target type τ' of τ. If e' has a multiple value type, then the function will return multiple values to its caller.

Similar to the transformation of functions, we may also have to transform the bound variable of a `let` expression to a sequence of bound variables in the target code, if the bound expression has a multiple value type.

More complicated is the transformation of records. Supposed we have a source record $r = (e_1 : \tau_1, \ldots, e_n : \tau_n)^\varphi$, The annotation φ provides information for the decision of unboxing this record. If φ is R, i.e. the record is rigid, we should keep this record boxed. Otherwise, we unbox the record by transforming it into a multiple value term.

Transforming a projection term also needs an explanation. If we want to extract a component from an unboxed record , the system just simply returns the corresponding values taken from the value sequence of the unboxed record. Otherwise, if the record expression is marked with R, we need to extract these values by performing projections with properly adjusted indexes. Note that the selected component may be another unboxed record. In this case we have to extract all values corresponding to this component.

We achieve the following property of the transformation algorithm for expression.

LEMMA 3.2. *Given a typing $(\Delta, \Gamma) \vdash_A e : \tau$. Suppose we have a type context (Δ', Γ') in λ^M and a compile context Σ, such that $\Delta \sim_\emptyset \Delta'$ and $\Gamma \sim_\Sigma \Gamma'$. Then the following properties hold.*

1. *$\Sigma \vdash e \leadsto e'$ succeeds, and*
2. *$(\Delta', \Sigma') \vdash_M e' : \tau'$ where $\tau \sim \tau'$.*

PROOF. We prove this lemma by induction on the derivation of the annotated typing rule. Let us show one typical case for a record $e = (e_1, \ldots, e_n)^\varphi$. We have two sub-cases

Subcase 1: $e = (e_1 : \tau_1, \ldots, e_n : \tau_n)^R$. Suppose that $(\Delta, \Sigma) \vdash_A e : \tau$. By the rigid record typing rule on annotated terms, we can infer $(\Delta, \Gamma) \vdash_A e_i : \tau_i$ and $\tau = (\tau_1 \times \cdots \times \tau_n)^R$. Applying induction hypothesis for each compilation of e_i, we obtain $\Sigma \vdash e_i \leadsto e_i'$ succeeds, and $(\Delta', \Gamma') \vdash_M e_i' : \tau_i'$, and $\tau_i \sim \tau_i'$ for each i. Therefore, applying the transformation rule for rigid records, we obtain $\Sigma \vdash (e_1 : \tau_1, \ldots, e_n : \tau_n)^R \leadsto (e_1', \ldots, e_n')$. Suppose that each τ_i' has form $\{\tau_{1,i}, \ldots, \tau_{k_i,i}\}$ where each $\tau_{i,j}$ is a single value type. Then, by the record typing rule for multiple value term, we have $(\Delta', \Gamma') \vdash_M (e_1', \ldots, e_n') : (\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,n}, \ldots, \tau_{k_n,n})$. By the simulation on types, we can also derive
$(\tau_1 \times \cdots \times \tau_n)^R \sim (\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,n}, \ldots, \tau_{k_n,n})$ as desired.

Subcase 2: $e = (e_1 : \tau_1, \ldots, e_n : \tau_n)^F$. Suppose that $(\Delta, \Sigma) \vdash_A e : \tau$. By the rigid record typing rule on annotated terms, we can infer $(\Delta, \Gamma) \vdash_A e_i : \tau_i$ and $\tau = (\tau_1 \times \cdots \times \tau_n)^F$. Applying induction hypothesis for each compilation of e_i, we obtain $\Sigma \vdash e_i \leadsto e_i'$ succeeds, and $(\Delta', \Gamma') \vdash_M e_i' : \tau_i'$, and $\tau_i \sim \tau_i'$ for each i. Therefore, applying the transformation rule for flexible records, we obtain $\Sigma \vdash (e_1 : \tau_1, \ldots, e_n : \tau_n)^F \leadsto \{e_1', \ldots, e_n'\}$. Suppose that each τ_i' has form $\{\tau_{1,i}, \ldots, \tau_{k_i,i}\}$ where each $\tau_{i,j}$ is a single value type. Then, by the typing rule for multiple value terms, we have $(\Delta', \Gamma') \vdash_M \{e_1', \ldots, e_n'\} : \{\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,n}, \ldots, \tau_{k_n,n}\}$. By the simulation on types, we can also derive
$(\tau_1 \times \cdots \times \tau_n)^F \sim \{\tau_{1,1}, \ldots, \tau_{k_1,1}, \ldots, \tau_{1,n}, \ldots, \tau_{k_n,n}\}$ as desired. □

Based on the above transformation algorithm for expression, we develop a transformation algorithm for programs in the form $\vdash P \leadsto P'$ as follows

$$\frac{\emptyset \vdash P \leadsto P'}{\vdash P \leadsto P'}$$

In order to prove the type preservation of the program transformation, we define an equivalent relation on types of source programs and target programs as follows.

$$o \equiv o$$

$$\frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

$$\frac{\tau_i \equiv \tau_i'}{(\tau_1 \times \cdots \times \tau_n)^{\varphi} \equiv (\tau_1', \ldots, \tau_n')}$$

The type preservation of the program transformation is shown by the following theorem

THEOREM 3.3. *Given a well-typed program P under an external type context Δ, and Δ' is the external type context in target calculus such that $\Delta \sim_{\emptyset} \Delta'$. The following properties hold.*

1. *$\vdash P \rightsquigarrow P'$ succeeds, and*
2. *$\Delta' \vdash_M P' : \tau'$ where $\tau \equiv \tau'$.*

PROOF. Applying lemma 3.2, we can achieve that $\tau \sim \tau'$. As the result of the representation analysis, the annotated type of a program never contain flexible record types. Then $\tau \equiv \tau'$ should hold since there's no flexible record type involved in τ implying no unboxing rule appears on the derivation of $\tau \sim \tau'$. □

4. Extension to Polymorphism

The method so far developed extends smoothly to a language with a polymorphic type system by imposing another form of rigidity constraints.

To analyze the problem, let us consider a polymorphic function f of type of the form $\forall t.t \to \tau$. Unless we perform whole program analysis, its implementation is inherently limited to a single parameter function. This implies that if f is applied to a record, then the record must have a boxed representation, i.e. it must be rigid. However, different from external function, f only imposes that the out-most record constructor to be rigid. Also note that if a function g has type of the form $\forall(t_1, t_2).(t_1 \times t_2) \to \tau$, then it does not impose rigidity on the top-level record but it imposes rigidity on the two component records. From these analysis, we can conclude that a record type is rigid if it appears as a type argument to a polymorphic function in the following inference step.

$$\frac{(\Delta, \Gamma) \vdash_A e : \forall t.\sigma \quad \varphi = R}{(\Delta, \Gamma) \vdash_A e\ (\tau_1 \times \cdots \times \tau_n)^{\varphi} : \sigma[(\tau_1 \times \cdots \times \tau_n)^{\varphi}/t]}$$

The validity of this can be seem by observing the parametricity result [15, 23] of the polymorphic lambda calculus.

The necessary changes to polymorphic typing is then simply the following.

1. Extend the annotated calculus to the second order calculus.
2. For polymorphic type application rule, impose the constraint that if the instance type is a record type then the annotated outer-most record type constructor should be rigid.

According to this refined typing, the annotation inference algorithm for type instantiations is defined as follows

$\mathcal{W}_e(\Delta, \Gamma, e\ \tau) =$
    `let`
      $(e', \forall t.\sigma, C_1) = \mathcal{W}_e(\Delta, \Gamma, e)$
      $\tau' = fresh(\tau)$
      $C_2 = topRigid(\tau')$
    `in`
      $(e'\ \tau', \sigma[\tau'/t], C_1 \cup C_2)$
    `end`

where $topRigid$ is defined as follow

$$\begin{array}{rcl} topRigid((\sigma_1 \times \cdots \times \sigma_2)^{\alpha}) & = & \{\alpha = R\} \\ topRigid(\sigma) & = & \emptyset \end{array}$$

Other necessary changes are simple and we omit them from the paper due to the limitation of space.

5. Implementation and Performance Evaluation

The representation analysis and the record unboxing algorithm described in this paper has been implemented in our SML# compiler [19]. This section briefly discusses about several implementation issues and optimizations for supporting the full Standard ML and for obtaining the full benefit of record unboxing. The benchmark results presented in the end of this section shows the effectiveness of our record unboxing algorithm.

5.1 Implementation Issues and Optimizations

Supporting multiple argument/multiple return value functions The target calculus of our record unboxing algorithm is the multiple value calculus λ^M where a function in general takes multiple arguments and return multiple values. The compiler back-end needs to be extended so that it supports multiple-parameter multiple-valued functions. Since our current implementation is by a virtual machine, this is done by generalizing virtual machine instructions for call and return. Our virtual machine instructions are in three address code format operating on a call stack frame. We generalize those instructions to be ones whose operands and the targets are multiple variables. In the future, when we develop a native code generator, those instructions can be implemented by using either stack or registers for passing arguments/returning values between callers and callees.

Useless and duplicate code elimination The record unboxing algorithm transforms a flexible record into a sequence of values. In the latter phases of the compiler, when the front-end code is linearized into sequences of instructions, these values will be bound to local variables. This may introduce many dead code and duplicated code in the target code of the compilation.

In our current compiler, the first optimization, duplicated code elimination, has been implemented. We plan to implement the second optimization, useless code elimination, soon based on one of known techniques [18, 24, 9, 4].

5.2 Experimental Results

The record unboxing algorithm has been successfully tested for full Standard ML. In order to evaluate the effectiveness of this optimization technique, sixteen benchmark programs have been given to produce the experimental results.

All the benchmarks are executed under Linux Fedora Core 1 installed under VMWare with 512 MByte memory RAM on an Intel Centrino 1.5Mhz. To avoid noises from computation environment, each program has been launched five times for each of two versions: with record unboxing and without record unboxing. The experimental results shown in figure 11 are average numbers of compilation times and execution times taken from the five launches.

In the table, the execution speedup is defined by $100 * T_N/T_U - 100$ where T_N and T_U are execution times of non-optimized version and optimized version respectively. The speedup figures show us that the record unboxing algorithm works effectively for most of the cases.

The variation of speedups reflects the number of records that have been unboxed. For example, `coresml` program is just an ordinary non-tail call fibonacci computation. There's nothing to deal with record unboxing in this program, thus the execution speedup figure shows no different between the two versions.

Benchmark	Compile		Execution		
	Without R.U (s)	With R.U (s)	Without R.U (s)	With R.U (s)	Speedup (%)
barnes_hut	1.82	1.83	16.14	13.5	19.58
boyer	1.2	1.27	1.08	0.91	18.48
coresml	0.01	0.01	53.59	53.6	-0.01
count_graph	0.3	0.31	162.91	141.37	15.24
fft	0.18	0.16	21.93	19.65	11.64
knuth_bendix	0.53	0.78	5.03	4.03	4.06
lexgen	1.32	1.37	17	13.32	27.56
life	0.23	0.24	1.75	1.71	2.7
logic	0.78	0.83	23.35	23.05	1.29
mandelbrot	0.04	0.03	8.76	8.06	8.66
mlyacc	13.78	14.71	2.17	1.78	21.89
nucleic	4.42	6.81	0.81	0.74	9.16
ray	0.53	0.55	11.87	9.85	20.55
simple	2.18	2.25	24.42	22.37	9.16
tsp	0.29	0.29	5.87	5.14	14.11
vliw	7.2	7.64	14.69	12.95	13.44

Figure 11. Experimental Results

In contrast, many flexible records are involved in `lexgen` and `mlyacc`. This brings us many chances for record unboxing, and as a consequence, the algorithm shows significant speedup in these cases (27% speed up of lexgen and 21% speed up of mlyacc).

In the final version of this paper, we plan to report more comprehensive evaluation including the number of heap allocations.

The experimental results are produced based on a byte code interpreter. But since our optimization is a source-to-source transformation, which does not assume any runtime architecture, similar results should be achieved for a native code compiler.

6. Related Works

As we mentioned in Introduction, one of the most relevant related works to our record unboxing may be Hannan's higher-order arity raising [7]. The kinded type system, which marks each type to either compile-time kind or run-time kind, in spirit, is similar to our annotated type system. In comparison to this work, our method can unbox more records, i.e. not only records appearing as arguments to functions but also records appearing in return values and in other data structures. Our record unboxing method can be naturally applied for separate compilation while this account has not been considered in [7].

The problem of arity raising has also been considered in the context of partial evaluation. Romanenko [16] developed an arity raiser which uses static data constructed from two global analysis of the program: one is to determine whether the argument splitting is feasible, other is to determine whether the argument splitting is useful. Thiemann [21] developed a partial evaluator for Scheme that can be able to split both argument of a higher-order function and its return value. It would be an interesting issue to combine our record unboxing method with these results. Since our record unboxing not only unboxes records appearing as arguments or return values but also unboxes other records in all other places, e.g. in other data structures, the combination would increase static computation in partial evaluation.

As we have mentioned in Introduction, the worker-wrapper approach such as those proposed in [14, 10] would not provide a practical solution to record unboxing. In [3], the authors considered the problem of applying this worker-wrapper approach to unboxing argument records and return records. This approach can only benefit if the subsequent inliner can successfully inline the wrappers introduced by the optimizer. However, in the context of separate compilation and in the presence of higher order functions where call-sites of a functional parameter may not easily be determined, developing an inliner that can inline all the wrappers introduced by the optimizer seems to be a difficult problem. In contrast, our technique works fine without requiring any other non-trivial optimization.

Object allocation optimization methods based on escape analysis [5, 6] share a similar motivation with ours of reducing heap allocation cost. Another related work in this direction is region based memory management [22]. Both of these approaches try to avoid heap allocation by estimating the lifetime of objects. Our system is based on the following different observation. Heap allocation of a record can be avoided by transforming it to multiple value bindings. This leads us to a particularly simple static analysis that only requires propagating one bit of information through unification. Yet, it successfully suppresses unnecessary heap allocation of records even if they are escaping from the creator functions.

In [17], a type-based method for unrolling lists is presented. In [11], a method to avoid intermediate closure creation is presented by allocating arguments of nested applications on a stack. In general perspective, both of them share the motivation similar to ours in trying to avoid memory allocation by unboxing a cons cell or by putting arguments in a stack. However, there does not seem to exist any successful attempt to transform records into multiple value sequences.

Our target calculus is a multiple value calculus where functions can take multiple parameters and can return multiple values. Multiple return values and multiple value bindings can be found in Scheme [1] and Common Lisp [20] and efficient implementation method for this feature has been studied [2]. As we have demonstrated in our development, this feature is useful in a compiler intermediate language for a typed functional language such as ML. However, there does not seem to exit much study on this feature in a typed setting. Our proposal would shed some light on the usefulness of this feature in type directed compilation.

7. Conclusion

In this paper, we have developed a type-based record unboxing optimization method that eliminates unnecessary heap allocation and associated memory access by transforming record-taking functions and record-returning functions into multiple-parameter functions and multiple-valued functions. This is done by the combination of type-based algorithm that statically estimates the set of rigid records that need to be boxed, and the type-directed algorithm that transforms all non-rigid record expressions and the corresponding functions to those multiple-parameter/multiple-valued functions. We have shown that these algorithms are sound. We have then implemented the algorithm in a compiler for SML#, an extension of the Definition of Standard ML, and have evaluated the algorithm through a typical benchmark suite. The results have shown the effectiveness of the algorithm.

There are several promising issues related to the proposed method that's worth considering:

Supporting flexible record types in signatures of external identifiers Our current development in this paper assumes that all external identifiers' signatures only contain rigid record types. This assumption places rigid constraints on every records that are passed to or returned from external functions. More heap allocations and heap accesses could be eliminated if the interface language to external functions can be redesigned to accept flexible record types in signatures of the external functions.

In this case, the current constraint resolution system may fail since a rigid constraint and a flexible constraint may be placed on the same annotation variable. One solution to this is that we introduce multiple argument/multiple return value functions and multiple value `let` bindings in the source language, and we change the type inference system for the source language so that only multiple values can be passed to/returned from external functions as flexible records. This solution also provides a better interoperability with foreign functions (e.g. C functions), which often take multiple arguments and returns multiple values.

Optimizing other data structures based on the representation analysis The record unboxing method is developed based on our observation: runtime representation of an object can be freely changed as long as this change is consistent and the object does not interact with the environment. Also based on this observation, we can develop further optimizations by changing the representations for other data structures. For example, a closed functions that is locally consumed inside a program can be efficiently represented by the function's code pointer instead of a closure. The representation analysis, in this case, can be adopted to detect all possible "flexible" closed functions for this purpose.

Acknowledgments

The authors thank John Hannan for his detailed comments on an earlier draft of this paper, which have been very helpful in improving the paper.

References

[1] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, E. Haynes, C. T.and Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised (5) report on the algorithmic language scheme. *ACM SIGPLAN Notices archive*, 33(9):26–76, 1998.

[2] J. M. Ashley and K. R. Dybvig. An efficient implementation of multiple return values in scheme. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 140–149, 1994.

[3] C Baker-Finch, K. Glynn, and S. Peyton Jones. Constructed product result analysis for haskell. *Journal of Functional Programming*, 14(2):211–245, 2004.

[4] Adam Fischbach and John Hannan. Type systems for useless-variable elimination. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 25–38, London, UK, 2001. Springer-Verlag.

[5] B. Goldberg and Y. G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of European Symposium on Programming*, pages 152–160, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[6] J. Hannan. A type-based escape analysis for functional languages. *J. Funct. Program.*, 8(3):239–273, 1998.

[7] J. Hannan and P. Hicks. Higher-order arity raising. In *Proceedings of International Conference on Functional Programming (ICFP)*, 1998.

[8] G. Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987.

[9] N. Kobayashi. Type-based useless-variable elimination. *Higher Order Symbol. Comput.*, 14(2-3):221–260, 2001.

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

[11] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1992.

[12] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *Proc. International Conference on Functional Programming (ICFP)*, pages 1–12, 1998.

[13] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[14] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer Lecture Notes in Computer Science, Vol 523, 1991.

[15] J.C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, 83:513–523, 1983.

[16] S. A. Romanenko. Arity raiser and use in program specialization. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 341–360, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[17] Z. Shao, J.H. Reppy, and A. W. Apple. Unrolling lists. In *Proc. ACM Conference on Lisp and Functional Programming*, 1994.

[18] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

[19] SML#. http://www.riec.tohoku.ac.jp/smlsharp/, 2006.

[20] G.L. Steele,Jr. *Common Lisp the Language, 2nd edition*. Thinking Machines, Inc., 1990.

[21] P. Thiemann. First-class polyvariant functions and co-arity raising. Unpublished Manuscript.

[22] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ calculus using a stack of region. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 150–161, 1994.

[23] P. Wadler. Theorems for free. In 347-359, editor, *Proceedings of International Conference on Functional Programming and Computer Architecture*, 1989.

[24] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 291–302, New York, NY, USA, 1999. ACM Press.