JAIST Repository
https://dspace.jaist.ac.jp/

Title	Type-directed Compilation of ML Supporting Interoperable Memory Management System
Author(s)	Huu-Duc, Nguyen
Citation	
Issue Date	2006-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/987
Rights	
Description	Supervisor:Atsushi Ohori，情報科学研究科，博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Type-directed Compilation of ML Supporting Interoperable Memory Management System

by

Nguyen Huu Duc

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Atsushi Ohori

School of Information Science
Japan Advanced Institute of Science and Technology

February 15, 2006

Abstract

The major weakness due to that functional languages such as ML become less popular than other imperative languages such as C is the inefficiency in senses of performance and memory usage. Developing a function language that can seamlessly interoperate with C (and other imperative languages) would help programmers to take advantage of both programming styles.

In this thesis[1], I present a type-directed compilation method of ML for achieving a high level of interoperability. This compilation method supports a memory management system, where integers, floating point values and other atomic data have the same natural representations as in other language implementations. This allows ML and other languages sharing the same heap space without any additional performance cost. Another advantage of this memory management model is that run-time system can have a better performance by eliminating all "boxing" and "tagging" operations that appear in most of conventional implementations.

In order to achieve this, I first consider an "unboxed", "non-tag" data representation model in which

- integers, floating point values and other atomic values are naturally represented;
- each a heap block or run-time environment (stack frame) has a "bitmap" that describes the pointer positions in the block;

then develop a compilation algorithm to support this model.

Since a polymorphic function may produce runtime objects of different types with different sizes, the compiler should be able to generate a function's code so that it has the same behavior for all instance types and it can compute a correct bitmap for each memory block. This would require us to insert extra lambda abstractions and applications to pass the bit tags required in bitmap computation and the sizes required in manipulation on unboxed values.

This compilation process should be done for both stack frames and heap-allocated objects including functions' closures and their environment records. I solve the problem of mutual dependency between this compilation method and closure conversion by developing a combined algorithm that plays both of the roles. The resulting compilation process is shown to be sound with respect to an untyped operational semantics with bitmap-inspecting garbage collection.

I also consider several optimization techniques for reducing run-time overhead arising from bitmap computation and unboxed manipulation. This compilation method, together with the proposed optimizations, has been implemented in our SML# compiler for the full set of Standard ML language, demonstrating its practical feasibility.

[1] This research is conducted as a program for the "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology.

Acknowledgments

I would like to extend my sincere gratitude and appreciation to many people who made this thesis possible. Special thanks are due to my principal advisor Professor Atsushi Ohori from Tohoku University whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis. Thanks also to Yamatodani Kyoshi, Liu Bochao and other members of the SML# project for interesting discussions and e-mail exchanges that directly or indirectly pertain to the present work.

I greatefully acknowledge the finalcial support of the Graduate Research Program. I am also grateful to the School of Information Science, Japan Advanced Institute of Science and Technology for providing me an excellent work environment during the past years.

I would like to thank the members of my PhD committee who monitored my work and took effort in reading and providing me with valuable comments on earlier versions of this thesis: Professor Takuya Katayama, Professor Kokichi Futatsugi, Professor Yasushi Hibino, Professor Mizuhito Ogawa, and Professor Yasuhiko Minamide. I thank you all.

I wish to express my warm and sincere thanks to Professor Nguyen Thanh Thuy, Head of the Department of Information Systems, Hanoi University of Technology, who gave me important guidance during my first steps into computer science studies.

Many thanks go to professor Ho Tu Bao and all vietnamese people at JAIST for giving me the feeling of being at home.

I feel a deep sense of gratitude for my father and mother who formed part of my vision and taught me the good things that really matter in life. I am grateful to my sisters Minh and Mai, for rendering me the sense and the value of blood relative. I am happy to be one of them.

Contents

Abstract i

Acknowledgments ii

1 Introduction 1
1.1 Method Outline 4
1.1.1 Bitmap-passing Compilation 5
1.1.2 Unboxed Compilation 6
1.1.3 Implementation Issues 7
1.2 Related Work 10
1.3 Organization of Thesis 12

2 The Source Calculus – λ^{ML} 13
2.1 Conventional ML Type System 13
2.1.1 ML type system at a glance 13
2.1.2 Problem with value polymorphism 14
2.1.3 Problem with type-directed compilation 15
2.2 Rank-1 Polymorphism 16
2.3 The Source Calculus – λ^{ML} 17
2.3.1 Types 17
2.3.2 Syntax 18
2.3.3 Typing Environment 19
2.3.4 Typing Rules 21

3 Bitmap-passing Compilation 22
3.1 The Target Calculus – Λ^{B} 22
3.1.1 Types 22
3.1.2 Syntax 23
3.1.3 Typing Environment 24
3.1.4 Typing Rules 25
3.1.5 Semantics 29
3.2 The Compilation Algorithm 35
3.2.1 The Explicitly Typed Bitmap-passing Calculus – λ^{B} 35
3.2.2 Compilation from λ^{ML} to λ^{B} 35
3.2.3 Transformation from λ^{B} terms to Λ^{B} terms 41

4 Unboxed Compilation **44**
4.1 The Target Calculus – Λ^U . . . 44
4.1.1 Types . . . 45
4.1.2 Syntax . . . 46
4.1.3 Typing Environment . . . 47
4.1.4 Typing Rules . . . 47
4.1.5 Semantics . . . 52
4.2 The Compilation Algorithm . . . 58
4.2.1 The Explicitly Typed Unboxed Calculus – λ^U . . . 58
4.2.2 Compilation from λ^{ML} to λ^U . . . 60
4.2.3 Transformation from λ^U terms to Λ^U terms . . . 64

5 The Combined Algorithm **66**
5.1 Combination with Closure Conversion . . . 66
5.1.1 Bitmap-passing unboxed closure calculus – Λ^{BUC} . . . 67
5.1.2 Explicitly Typed BUC calculus – λ^{BUC} . . . 72
5.1.3 λ^{ML} to λ^{BUC} transformation . . . 73
5.1.4 λ^{BUC} to Λ^{BUC} transformation . . . 77
5.2 Extension for Generating Stack Frame Layout . . . 79

6 Separate Compilation and Module Language **83**

7 Implementation and Optimizations **87**
7.1 Mutually Recursive Function Definition . . . 87
7.2 Double-Word Alignment . . . 90
7.3 Optimizations . . . 91
7.3.1 Uncurrying optimization . . . 91
7.3.2 Sharing bitmap/offset computation . . . 92
7.3.3 Arithmetic Optimization . . . 94

8 Conclusion and Future Directions **95**

Chapter 1

Introduction

In recent years, modern functional programming languages (FLs) such as ML, Haskell become more and more attractive to programmers with many powerful features which are not available in conventional imperative programming languages (ILs) such as C, Pascal:

- most of FLs are strongly typed, eliminating a huge class of easy-to-make errors at compile time;
- higher order parametric polymorphic functions and powerful module system of FLs allow programmers to “glue” their code together easily;
- symbolic data types provide a more abstract tool for programmers to describe their developing systems for both rapid prototyping and stand-alone applications.

Despite of such considerable advantages, FLs are still less popular than ILs due to several weaknesses they have. FLs run slower and consume much memory space. In developing a real-life application where user interests are put in the first place, programmers are often unwilling to trade the efficiency for the advantages that FLs bring to.

Clearly, we want to have the best of both programming models. Since there does not exist an all-in-one language, an alternative way for enjoying the benefits from ILs and FLs is to make them *interoperable*, e.g. ML programmers can write code to call functions from libraries implemented in C.

This is, however, not quite an easy task. The major obstacle preventing us from achieving the desired interoperability is the mismatch between data representations in FLs and ILs. Let’s consider a typical example of the interoperability between ML and C. Taking advantage of a low-level, monomorphic language, compilers of C can produce code which can run with natural data representation and efficient calling conventions. ML compilers, in contrast, often sacrifice the natural representation of data for compiling polymorphism and for supporting tracing garbage collection. More specifically, ML often assumes “boxed” and “tagged” representations for run-time objects.

- **Boxed representation.** A boxed representation of a run-time object is a pointer to a memory block where the actual content of the object resides. The point of this representation is that objects of any types can have the same size, i.e. one word data. ML compilers, therefore, can compile a polymorphic function into a simple code which has the same behavior for for all possible instance types.

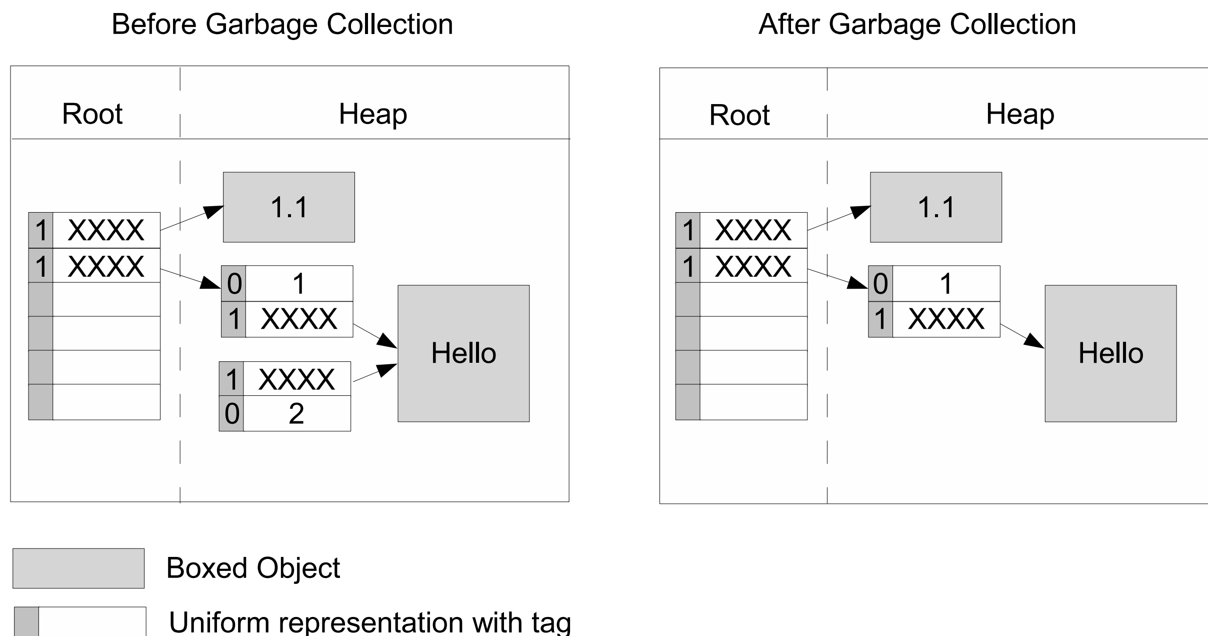

Figure 1.1: Example of boxed and tagged representation

- **Tagged representation.** A tagged representation is a word containing one bit tag for indicating whether it is a pointer or not. This yields particularly simple memory management: the compiler only needs to set one bit when emitting code, and the tracing garbage collector can locate all the pointers in a heap block by simply scanning the tag bit in each word in the block.

Figure 1.1 gives an example of a memory model with boxed and tagged representation. In this example, the floating point number `1.1` and the string `''Hello''` are boxed, i.e. they are allocated in a separated heap blocks and pointers to these block are used in place. Tags (`1` or `0`) are included in each word in an ordinary block. Pointers to heap blocks have tag `1`. Non-pointers have tag `0`. These bit tags are useful for tracing garbage collection which traverses over the memory space to find out and eliminate "garbage" blocks (in the example, there is a "garbage" block that contains a pointer to the string block `''Hello''` and an interger `2`).

In the presence of data representation mismatch, ML and C run-time systems do not accept direct function calls from one to other. In this case, to achieve the desired interoperability, SML/NJ (a famous dialect of ML) integrates the so-call *foreign function interface*. This allows ML users accessing to external libraries by performing conversion of data every time they are passed through the language boundary. An obvious drawback of this method is that the conversion may exhibit serious performance overhead when exchanging a large amount of data. Another drawback, which comes from the nature of boxed and tagged representations, is the inefficiency of runtime and memory usage in maintaining boxed and tagged objects. OCaml, another well-known dialect of ML, also allows to call C's functions from ML programs. In order to access to values passed from ML, C programmer must used a special accessing tools provided in the OCaml's compiler package. This would be a more efficient solution in sense of performance. However, the called C programs must be written with a specialized code for ML. This implies that we

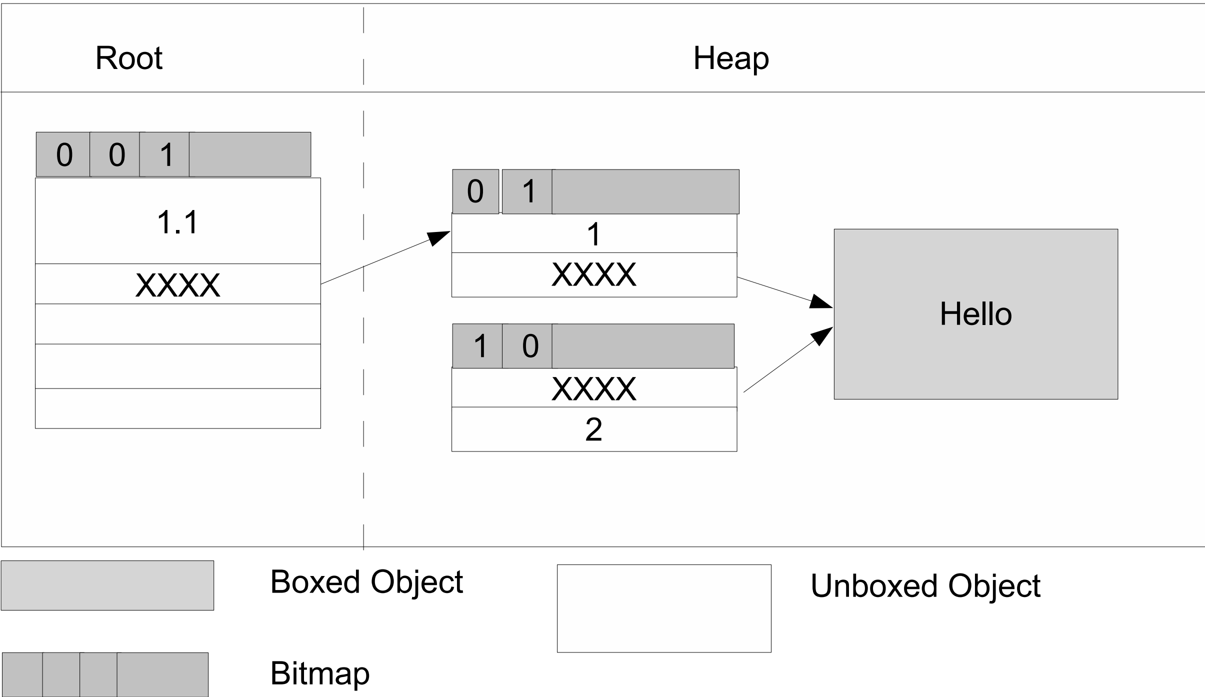

Figure 1.2: Example of a natuaral representation memory model

can not achieve a real interoperability where ML programs can call any C function.

The goal of my thesis is to develop a compilation method of ML so that we can achieve a higher level of interoperability and more efficient run-time system. First of all, I assume an *interoperable memory management* for ML run-time system in which a common heap space can be shared by ML and other languages including C. The following two features are considered to be prerequisites in this memory management system:

- integers, floating point values, and other atomic data have the same natural representations as those in C;
- each memory block includes its layout information for garbage collection.

One natural strategy that satisfies these criteria is to include in each heap-allocated block a *bitmap* that describes the pointer positions in the block. Figure 1.2 depicts a memory model that satisfies the described criteria. In this example, atomic objects such as intergers (`1, 2`), floating point numbers (`1.1`) are naturally represented. Strings and other structured objects such as records are boxed. A bitmap are generated for each record, indicating pointer positions in the record. In the example, bitmap of the records $(1,''Hello'')$ and $(``Hello'',2)$ are $[0,1]$ and $[1,0]$, respectively. Runtime environment (root) also requires bitmap. For the sake of simplicity, in the above example, I consider the bitmap of runtime environment as the same as the bitmap of the record $(1.1,(1,''Hello''))$ whose value is $[0,0,1]$. Since the floating point value 1.1 is inlined and occupies two words, we need to reserve two zero bits in the bitmap for this value. In a practical development, bitmaps for runtime environment are different. I shall described this later in Chapter 5.

Since this representation only includes the necessary information for garbage collection and does not place any constraint on scanning strategy of garbage collection, it should be

efficient and also compatible with heap management of most of typed languages whose object creation is monomorphic such as Java.

The major task of the thesis is to develop a compilation method that support the proposed memory management system. For monomorphic language where all type information can statically determined, allocating heap block with unboxed representation and correct layout bitmap is easy. This, however, is challenging for a polymorphic language such as ML. Two major problems we have to face when compiling polymorphic functions are: generating code for computing a correct layout bitmap for each memory block, generating code for manipulating unboxed values. To see the problems, let us consider the following program in Standard ML syntax,

```
let fun f x y = (x,y)
in (f 1 1 ; f 1.1 1; f "a" 1) end
```

where (e_1,e_2) is a record to be allocated in a heap, and (e_1;e_2) is sequential evaluation. Although there is only one heap allocation code, namely `(x,y)`, it will allocate 3 different records `([0,0]; 1, 1)`, `([0,0,0]; 1.1, 1)`, and `([1,0]; p, 1)` (p is the pointer to the string block "a").

The first problem is to set a correct bitmap for each instance of the function. In order to do this, it is necessary to evaluate the instance types of `x` and `y` to get the corresponding portions of the entire bitmap, compose them, and pass them to the code of `(x,y)`.

The second problem is to generate code for manipulating unboxed values. In the above example, allocation code for the record `(x,y)` consists of the following steps.

1. allocating a fresh memory block whose size is the total size of `x` and `y`, plus size of the bitmap.

2. generating bitmap (as shown above) and copy the resulting value to the first word of the block

3. copy the value of `x` from runtime environment to its location in the block

4. copy the value of `y` from runtime environment to its location in the block

In the above steps, size information of `x` and `y` are required. Therefore it is also necessary to evaluate the instance types of `x` and `y` to get their sizes and propagate these sizes into appropriate operations.

My strategy to deal with these two problems is to develop a type-directed compilation algorithm that statically transforms types into terms that can evaluate at runtime to compute the necessary information.

The remainder of this chapter serves as an overview of this strategy. In Section 1.1, I briefly introduces the method applied in this research. In Section 1.2, I review several related works, compare them with the proposed method, and claim the contributions of the thesis. Section 1.3 gives a comprehensive overview for the rest of the thesis.

1.1 Method Outline

The general idea behind this research is to follow type-passing approach presented in several existing works including intentional type analysis [HM95], compilation of type

classes [HHJW96, PJ93], polymorphic record compilation [Oho95], and mixed representation optimization [Ler92]. In order to pass types to run-time, I develop a type-directed compilation scheme in which type information are encoded as terms in the target language, and the compilation algorithm takes responsibility for this transformation.

As mentioned above, type information are useful for computing layout bitmaps and manipulating of unboxed values. Unlike *dynamic type dispatch* approach in [HM95], we do not need full type information for compiling polymorphism; size and tag information of types are sufficient for generating bitmaps and locations. The compiler, therefore, only encodes types as size and tag terms in the target language. For the types whose data layouts are statically determined, the compiler can statically generate tag and size values. For abstract types such as type variables, we need to introduce tag/size variables. In this case, the compiler generates tag/size abstractions at type abstractions and tag/size applications at type instantiations.

This compilation scheme is seemingly complex for generating code for both of computing bitmaps and manipulating unboxed values. For the sake of readability, in the thesis, I separately present the compilation method for generating bitmaps (namely *bitmap-passing compilation*) and the compilation method for manipulating unboxed value (*unboxed compilation*). After giving a formal representation of these methods, I shall show how to combine these methods in a practical implementation of ML. The rest of this section gives an overview for each parts of the presentation.

1.1.1 Bitmap-passing Compilation

The goal of the bitmap-passing compilation is to generate code for computing necessary bitmap for each memory block. Since bitmaps are required for all memory blocks including heap-allocated blocks and run-time environments (stack frames), we need to compute bitmaps for all of them. As a formal representation, in the bitmap-passing compilation, I only consider the generation of bitmaps for heap-allocated blocks. Generation of bitmaps for run-time environments shall be discussed later in the implementation issues.

Let us consider the following explicitly typed polymorphic expression in ML-style.

```
let f:∀t.t → t × t = Λt.λx:t.(x,x)
in (f int 1; f string "1") end
```

Since the type of the record expression `(x,x)` is $t \times t$, we know that this expression requires a bitmap corresponding to $t \times t$. This bitmap can be composed from bit tag information of objects of types t. We extend the function `f` by introducing new parameters `a` and type this new variable with a special *bit tag type* of the forms $\langle t \rangle$. In general, $\langle \sigma \rangle$ denotes (the singleton set of) 0 if σ is a type of *unboxed* objects (i.e. non-pointer atoms), and 1 if σ is a type of *boxed* objects (i.e. pointers). In ML, if σ is not a type variable then $\langle \sigma \rangle$ is determined by the outermost type constructor of σ.

Using bit tag type, we can obtain the necessary bit tag for each usage of `f` by inspecting the instance of the bit tag types.

For example, if t is instantiated to *int* then the corresponding bit tag type is $\langle t \rangle[int/t] = \langle int \rangle$ which denotes the (singleton) value 0. The value of bitmap `(x,x)` is therefore $[0, 0]$. Based on this idea, the *bitmap-passing compiler* with a typing derivation algorithm will transform the above program to the following explicitly typed term.

```
let f:∀t.⟨t⟩ → t → t × t = Λt.λa:⟨t⟩.λx:t.([a,a]; x,x)
in (f int 0 1; f string 1 "1") end
```

where the syntax $[e_1, \ldots, e_n]$ is the term constructor for bitmaps and $(e_{bm}; e_1, \ldots, e_n)$ is the term constructor for a record where the first component e_{bm} is the bitmap term of the record.

In the above example, new bit tag parameter a is introduced. Then, bitmap of the record (x, x) is generated as $[a, a]$. At the type instantiation, this bit tag parameter will be given actual value, i.e 0 for the argument's instance type *int* and 1 for the argument's instance type *string*.

1.1.2 Unboxed Compilation

Unboxed compilation takes responsibility for generating code for manipulating unboxed values. In order to support recursively user-defined data, we put a restriction on unboxing rules: only multi-word atomic constants such as floating point values are unboxed, structured data such as records, strings remain boxed. We also assume that polymorphic operators never inspect the content of unboxed values. Common copying and comparing operators satisfy this assumptions. Other polymorphic operators such as "+" can be implemented by overload operators. Under these restrictions, the only information which are really needed for manipulating an unboxed value in the presence of polymorphism are location and size of the value.

In ML, a run-time value is allocated either in a heap block or in a run-time environment (stack frame). The location of the value (or the offset of the value related to the memory block pointer) is computed as the total sizes of its predecessors in the block. For example, let us consider a block of three unboxed value `(1.1,2,3)`. Offset of the third element related to the block pointer is `3` since the floating point value `1.1` requires two words. In the presence of polymorphism, types of the value and its predecessors may not be statically determined. Their sizes, therefore, are needed to be evaluated from their instance types. We can apply the same strategy as bitmap-passing compilation for computing offsets and sizes: statically compute offset and size as far as possible, introduce extra size parameter for each type variable, and pass the neccessary sizes to offset computation.

Here, we meet a problem of performance inefficiency. Since offset computations are required for all values resided in memory, these computation would be heavy in the presence of polymorphism. In order to reduce the run-time overhead arising by offset computation of stack frame values, we consider a simple model of stack frame where all variables are allocated in static locations (therefore their offset can be statically computed). We can do this by reserving maximum space for each polymorphic variables, i.e. two words in our compiler implementation. Since all polymorphic variables have fixed sizes, we can easily computed their offset without knowing their actual types. This strategy would require extra memory space for runtime environment. However, I believe that the redundancy problem arising from variable allocations is not serious with a variable liveness analysis optimization. Adopting this model of runtime environment, the only kind of offset computation we have to deal with are those of unboxed values resided in heap-allocated blocks.

Let us consider the following explicitly typed polymorphic expression in ML-style.

`let` $f : \forall t.t \times int \times t \to t = \Lambda t.\lambda x : t \times int \times t.\ \#3\ \ x$

```
in (f int × int × int (1,2,3); f real × int × real (1.1,2,3.3)) end
```

Function f performs the projection for the third element of a given record x of type $t \times int \times t$. Offset of the third element depends on size of values of type t, i.e. $sizeOf(t) + 1$. Following the strategy of bitmap-passing compilation, we extend the function f by introducing new parameters a and type this new variable with a special *size type* of the forms $|t|$. In general, $|\sigma|$ denotes (the singleton set of) 1 if objects of type σ are *single* (i.e. have one word size such as integers, pointers), and 2 if objects of type σ are *double* (i.e. have two word size such as floating points). In ML, if σ is not a type variable then $|\sigma|$ is also determined by the outermost type constructor of σ.

Using size type, we can obtain the necessary size for each usage of f by inspecting the instance of the size types.

For example, if t is instantiated to $real$ then the corresponding size type is $|t|[real/t] = |real|$ which denotes the (singleton) value 2. The offset of the third element of x is therefore $2 + 1 = 3$. Based on this idea, the compiler can transform the above program into the following explicitly typed term.

$$\texttt{let}\ f : \forall t.|t| \rightarrow t \times int \times t \rightarrow t = \Lambda t.\lambda a : |t|.\lambda x : t \times int \times t.\ \pi^{a}_{a+1}(x)$$
$$\texttt{in}\ (f\ \ int \times int \times int\ \ 1\ \ (1^1, 2^1, 3^1);\ \ f\ \ real \times int \times real\ \ 2\ \ (1.1^2, 2^1, 3.3^2))\ \texttt{end}$$

In the above code, the term $\pi^{a}_{a+1}(x)$ is compiled from the projection #3 x. The superscript a represents actual size of the selected field, the subscript $a+1$ represents the offset term of the selected field. This offset term is generated from type information of predecessor of the field (t and int). In record allocations $(1^1, 2^1, 3^1)$ and $(1.1^2, 2^1, 3.3^2)$, the superscripts represent size of each element required in the allocation of the record at runtime.

At type instantiation, actual sizes are given to the size parameter though funtion application. In the first instantiation, actual size is given to 1 corresponding to the actual type of t is int. In the second instantiation, actual size is given to 2 corresponding to the actual type of t is $real$.

1.1.3 Implementation Issues

We have successfully implemented a compiler of ML, namely SML#. Beside several powerful features such as rank-1 polymorphism [OY99], polymorphic record compilation [Oho95], the compiler also archives a high-degree of interoperability based on the proposed compilation methods. During the development of this compiler, we have worked out several implementation issues related to bitmap-passing compilation and unboxed compilation.

The first one is the mutual dependency problem among bitmap-passing compilation, unboxed compilation and closure conversion.

- **Dependency between bitmap-passing compilation and unboxed compilation.** Bitmap-passing compilation generates bitmaps for heap-allocated blocks. The composition of these bitmaps also require size information generated from unboxed compilation. For example a bitmap of a record of type $t \times t$ may have three different values depending on the instance type of t: $[0, 0]$ for single, unboxed type t (e.g. int), $[1, 1]$ for single, boxed type t (e.g. $string$), and $[0, 0, 0, 0]$ for double, unboxed type t (e.g. $real$). In the first two cases, the generated bitmap consists of two bits. In the last cases, the generated bitmap consists of four bits. In general, generating

a correct bitmap of an unboxed memory block in bitmap-passing compilation, we also need size information of each element of the block. These size information may need to be generalized by unboxed compilation.

- **Dependency between bitmap-passing compilation and closure conversion.** A practical compiler of functional languages often implements closure conversion as a crucial immediate compilation step which transforms functions into data structures (closures) that encapsulates the function's code and its environments (often implemented as records). Bitmap-passing compilation introduces bit tag abstractions which are treated as ordinary lambda abstractions. These abstractions need to be closure converted. In the other hand, a closure and its environment record are also heap-allocated objects. Bitmap-passing compilation is required for generating bitmaps for them.

- **Dependency between unboxed compilation and closure conversion.** Similar to bit tag abstractions introduced in bitmap-passing compilation, size abstractions introduced in unboxed compilation also need to be closure converted. A closure and its environment records are heap-allocated blocks. Normally, a closure is just a pair of the pointer to code and the pointer to its environment record. It does not need any special treatment in unboxed manipulation. However, an environment record may consist of polymorphic unboxed values. In this case, allocation of an environment record and accessing to its components require to be done in unboxed compilation.

Due to the mutual dependency among these compilation processes, in SML# compiler, we combine them into a single compilation step. The resulting algorithm is complex but it does not add much complexity, and it does not violate the correctness of the type-directed compilation.

The second implementation issue is the problem of how to generate bitmap information for stack frames. We consider a stack-based implementation, where a stack frame is allocated for each function, and all the temporary variables used by the function are allocated in the stack frame. Suppose that the function code is implemented by a sequence of instructions in three address format $x_1 = op(x_2, x_3)$ where x_1, x_2, x_3 are locations in a stack frame relative to the frame pointer. Compiling a lambda term to three address code can be done by translating it to A-normal form [FSDF93] and minimizing the number of stack slots. Our strategy for setting up a bitmap for such a stack frame is first to type each local variable to one of the following types: $\{boxed, unboxed, t\}$ where t stands for type variables. Let $\{t_1, \ldots, t_n\}$ be the set of type variable appearing in a stack frame. We represent the layout information of a stack frame by the following data:

1. the number of slots of type *unboxed*,
2. the number of slots of type *boxed*,
3. a set of bit tags types $\{\langle t_1 \rangle, \ldots, \langle t_n \rangle\}$, and
4. the number of slots of each t_i.

Secondly, we refine the type-directed compilation algorithm to compute the necessary layout information by the following steps.

1. In the type-directed compilation, we identify the set of type variables $\{t_1, \ldots, t_n\}$ that will appear as types of local variables and arguments, and record it in a function definition. This is easily done by analyzing the types of the set of sub-terms in the function body,

2. The type-directed compilation algorithm is refined so that it regards the function body to create an additional record of type $t_1 \times \cdots \times t_n$.

3. After the type-directed compilation algorithm, we perform type-preserving A-normalization [Oho99] and code generation to obtain three address code.

4. The type of each address of instruction is evaluated to one of *boxed*, *unboxed*, and t. We then perform variable liveness analysis and compute the maximal number of simultaneously live variables for each type, and assign variables to a frame index. This problem can be regarded as register allocation with unbounded number of different kinds of registers (for each type), and can be solved by the technique register-allocation by proof-transformation [Oho04].

5. We produce the stack frame layout by counting the number of slots for each type.

The third implementation issue relates to separate compilation and module language. The standard algorithm of bitmap-passing compilation and unboxed compilation can be extended to work well with the full set of ML Core Language where top-level objects are closed in sense of type. However, in separate compilation and the presence of functor, top-level objects in one module may involve abstract types which are specialized in another modules. Tags and sizes of these types, therefore, are undetermined, and the compilation algorithm will be unsound. I solve this by introducing top-level tag and size variables (corresponding to each abstract user-defined type). Actual values of these variables will be determined at the time we link modules together (or at the time of functor application).

The forth issue is the treatment of mutually recursive function definitions. In a polymorphic mutually recursive function definition, all functions share the same type abstraction. Bitmap-passing compilation and unboxed compilation generate extra lambda abstractions (bit tag abstractions and size abstractions) at type abstractions. This implies that all functions must share the same tag and size abstractions generated from the common type abstractions. Passing tags and sizes among these functions may introduce significant run-time overhead. We solve this by wrapping each function in the definition by a polymorphic function. The recursive definition becomes monomorphic inside each wrapper, therefore no more tag and size passing are required.

The fifth issue relates to the implementation of interoperability feature in some specific run-time architecture. Some architectures (e.g. SPARC) assume that floating point values are double-word aligned. An interoperable compiler should, therefore, generate data that compromise with this assumption. We solve this by refining bitmap-compilation and unboxed compilation for inserting a dummy word before each possition which is need to be double-word aligned.

Last but not least, we considered several optimizations for reducing run-time overhead arising by bitmap-passing compilation and unboxed compilation. Both of these methods introduce extra abstractions, extra applications and extra computation of bitmaps and

offsets. Run-time overhead arising by these computation might be significant in comparison to the main task. Reducing this overhead, therefore, is an important issue for a practical compiler. The optimizations we considered are:

- **Uncurrying.** Traditional uncurrying optimization is often performed before closure conversion. This transforms a sequence of ordinary lambda abstractions (curry function) into a single, multiple argument function (uncurry function). In the development of SML#, extra bit tag/size abstractions are generated at type abstraction. In addition, rank-1 polymorphic type reconstruction often introduces type abstractions at ordinary abstractions. We can, therefore, perform the optimization again for uncurrying the extra bit tag/size parameters with the ordinary lambda parameter.

- **Sharing bitmap/offset computation.** Bitmap and offset computation are heavy in presence of polymorphism. Reducing the number of identical bitmap/offset computation would help us to generate a better compilation results. This optimization have the same spirit with the well-known common expression elimination technique. We specialize this for bitmaps and offsets by introducing efficient comparison rules on bitmap/offset based on type information. In addition, compiler can chose the best place to put common bitmaps/offsets without adding much complexity.

- **Arithmetic optimization.** Bitmap and offset computation can be implemented by simpler arithmetic operations such as addition, logical bitwise operations. This optimization first break down a set of bitmap/offset computation into a set of simpler operations, then remove unnecessary duplicate operations. This process is directed by types of bitmaps/offsets.

1.2 Related Work

The problem of boxed and tagged data representation has attracted attention of several researchers, and several methods on *unboxed representation* and *tag-free garbage collection* have been proposed for run-time efficiency of polymorphic languages. Let us summarize these existing methods and compare them to the proposed compilation method.

- **Tag-free garbage collection.** Most relevant to the bitmap-passing compilation is the work on tag-free garbage collection, where the garbage collector traces data using type information generated by the compiler. Goldberg [Gol91] has proposed a tracing method by inspecting the call frame of the current function to determine the type of each runtime object. If the current function is a polymorphic function, then the type information is computed from the types of its arguments. Tolmach [Tol94] has proposed a refined approach to pass types of arguments to each polymorphic function. This approach requires runtime construction and inspection of type information, which may be large and inefficient. In [MMH96, SS98], methods for optimizing type-passing compilation have been proposed.

  One drawback to this approach is the runtime overhead arising from manipulating type information, which may sometimes become very large compared to allocated data. Another drawback, which we regard as a serious obstacle in achieving high-degree of interoperability, is the close dependency between the garbage collector and the compiler. The garbage collector needs to know the semantics of types and

their precise runtime representations. This also implies that the traversal strategy is constrained by type structures. For example, Tolmach's algorithm relies on depth-first traversal and does not necessarily compatible with commonly used efficient strategies, including the most notable one of Cheney's collector [Che70].

Compared with these approaches, the proposed method can be regarded as an optimized version of a type-passing approach by statically computing most the layout information at compile time. This reduces the runtime time and space overhead of type-passing. Another important feature is that our method produces self-contained layout information for each runtime object so that the garbage collector may be implemented independently of the language type system.

- **Unboxed Representation.** To solve the problem of boxed data representation, Morrison et. al. [MDCB91] have described several optimization techniques for runtime specialization of functions using type information at run-time. Peyton Jones and Launchbury [JL91] have proposed a calculus where boxed and unboxed objects are explicitly manipulated and polymorphism is restricted to boxed objects. They have also proposed several optimization strategies by source-to-source program translation. Leroy [Ler92] proposed a systematic method to transform ML into a calculus similar to that of Peyton Jones and Launchbury. His strategy is to keep objects unboxed as long as its type is statically determined. To combine this strategy with polymorphic functions, his algorithm inserts representation conversion functions before and after each invocation of a polymorphic function. Leroy's method is further refined in [HJ94, Thi95].

  These methods of "mixed representation" try to minimize boxed representation overhead by localizing them to polymorphic functions. This approach should be effective for those application whose cost is largely determined by monomorphic manipulation of multi-word data such as arithmetic computation loops. One apparent limitation of this approach is that it does not eliminate the mismatch between the boxed representation required by polymorphic functions and various unboxed data; polymorphic functions still require their arguments to be boxed. Another weakness of this approach, as Leroy pointed out, is that it does not work well for recursively defined data types such as lists or trees, since these data structures require boxing/unboxing coercions to be lifted to the structures by recursively applying the necessary coercions.

  Ohori and Katamizawa [OT97] proposed an operational semantics for full unboxed representation. This serves as an abstract machine where unboxed values are manipulated through their actual positions in the memory block. Size information of types are encoded and passed to run-time to compute correct positions of unboxed values. This method, however, exposes several limitations. One of them is the poor support for different kinds of memory blocks. Only pairs are accepted, other kinds such as multi-field records would require must more works on the formalism of the operational semantics. Another serious drawback is that the implicit rules of location computation given by the unboxed operational semantics prevent us from developing optimization in source level.

  Our compilation method, in contrast, supports full unboxed representation for multi-word atomic constants. This should be efficient for the case of combining unboxed

representation with recursively defined data. More important, this fully supports natural data representation, and therefore encourage a high-degree of interoperability.

Our method yields an accurate collector based on reachability. There have also been several attempts [Fra94, MFH95, IK00] to develop more powerful GC exploiting type informations at runtime. It is an interesting open issue whether we can combine these approach and our type-based compilation approach.

Our method uses type information at runtime. This technique is related to previous work on type-directed compilation, including intentional type analysis [HM95], compilation of type classes [HHJW96, PJ93], and mixed representation optimization [Ler92]. In comparison with these approaches, one type theoretical feature our method relies on is the introduction of a family of bit tag/size types, each of which denotes a singleton set of bit tags/sizes. This idea is first presented in [Oho95]. Crary, Weirich, and Morriset [CWM98] have proposed a similar mechanism.

We have implemented the mechanism presented here in our Standard ML compiler. The compiler is composed of a series of type-directed type-preserving transformation steps. This approach has the same spirit of the TIL compiler [TMC+96] and the compilation of System F to a typed assembly language [MWCG99].

1.3 Organization of Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents an ML-style polymorphic calculus which serves as the source calculus in the proposed type-directed compilation. Unlike in most conventional ML compiler, this calculus is equipped with rank-1 polymorphism to suppress unnecessary type abstraction and type application. I formalize type system of the source calculus and show several property of this type system.

Chapter 3 and Chapter 4 independently show the theoretical aspect of the bitmap-passing compilation and unboxed compilation. In each chapter, I present a target calculus as an extension of the source calculus with new factors involved in the corresponding methods. I formalize semantics and typing rules of each target calculus, and show the usual soundness property with respect to the evaluation rules. Bitmap-passing compilation algorithm and unboxed compilation algorithm are given, and I show that both of them are type-preserved. As in usual type-directed compilation method, the soundness property of the target calculus and the type-preserved property of the transformation guarantee that the source calculus is sound with respect to a semantics realized by the transformation followed by the evaluation rules.

Chapter 5 presents a combination scheme of bitmap-passing compilation, unboxed compilation and closure conversion. In this chapter, I also present an extension of this combined algorithm for generating layout information of stack frame.

In Chapter 6, I show how to extend the compilation method for separate compilation and module language.

Chapter 7 presents several implementation issues including the treatment of mutually recursive function definition, double-word alignment, and optimizations.

Finally, Chapter 8 concludes the thesis and gives some future directions for the research.

Chapter 2

The Source Calculus – λ^{ML}

This chapter presents an explicitly typed ML-style polymorphic calculus, written λ^{ML}, which serves as the source calculus for the type-directed compilation presented in the next chapters. While most of ML dialects adopt conventional ML type system [DM82], I choose an extended version that supports full rank-1 polymorphism [OY99] for relaxing the "value-only" restriction of ML type system, and for suppressing unnecessary type abstractions/type applications introduced by constructing polymorphic terms from other polymorphic terms.

In the rest of this chapter, I briefly review the conventional ML type system, analise its problems related to value polymorphism and type passing semantics, and introduce the source calculus with rank-1 polymorphic type systems.

2.1 Conventional ML Type System

Damas-Milner type discipline [DM82] has been chosen as the basis for type system of ML (Standard ML, Ocaml) and several other functional languages (Miranda, Hope, Haskell). The widespread use of this type system is due to the benefits achieved from the flexibility of programming through ML's polymorphic `let` construct, and the efficiency of implementation of type inference through the restricted treatment of polymorphism. However, as pointed out by Ohori and Yoshida in [OY99], this type system exposes some weaknesses relating to value polymorphism and type-directed compilation.

The following three sub-sections briefly summarize the benefits and limitations of this type system.

2.1.1 ML type system at a glance

Assuming a set of base types o and type variables t, the syntax of ML *types* (ranged over by τ) and *type schemes* (ranged over by σ) is given by

$$\begin{array}{rcl} \tau & ::= & o \mid t \mid \tau \to \tau \\ \sigma & ::= & \tau \mid \forall t.\sigma \end{array}$$

where o represents a base type, and $\tau \to \tau$ stands for function types.

A type scheme $\forall t_1 \ldots \forall t_n.\tau$ (also written as $\forall t_1 \ldots t_n.\tau$) represents a set of types (instance types) achieved from τ by substituting each free occurrence of t_i in τ by a type τ_i.

Polymorphism can be achieved by giving a type scheme to the bound variable of an ML `let` construct. In the following context of the variable binding, this variable can be used polymorphically, i.e. it can be assumed to different instance types of the type scheme.

Let us consider the following example

```
let
 val f = fn x => x
in
 (f 1, f "a")
end
```

A type scheme $\forall t.t \rightarrow t$ is given to the variable `f`, and in the record `(f 1, f "a")` `f` is used with different types, i.e. $int \rightarrow int$ and $string \rightarrow string$.

Despite the existence of more powerful type systems, ML type system has been chosen as the basis for implementation of many functional languages including Standard ML of New Jersey, Ocaml, Haskell. This is due to the fact that ML types can be reconstructed efficiently by the unification-based type inference algorithm presented in [DM82].

2.1.2 Problem with value polymorphism

The interaction between polymorphism and imperative features of ML such as reference, communication and continuation has always been a troublesome problem. Let us consider the following example taken from Standard ML of New Jersey documentation.

```
val r = ref(fn x => x);
r := (fn x => x+1);
!r true;
```

The basic rules of ML type inference give the following polymorphic type to r:

$$r \ : \ \forall t.(t \rightarrow t) \ ref.$$

Given that r has this polymorphic type, the occurrence of r in the assignment expression would have the type $(int \rightarrow int) \ ref$, while in the third line r would have the type $(bool \rightarrow bool) \ ref$. This typing of the program is clearly unsound, since it allows a function on integers to be applied to a boolean value.

This problem happens because of the mutable value r is given a polymorphic type. This allows the mutable value's type to be instantiated differently at the points where the contents are updated and where they are fetched, with the consequence that the value stored and retrieved changes type in the process.

In order to avoid this problem, ML adopt the so-call "value polymorphism" scheme so that polymorphism is restricted to syntactic values (also known as non-expansive expressions). A non-expansive expression is identified if it has one of the following forms.

- constant
- variable
- function
- nullary constructor

```
let
   val x = fn x => x
   val x = (x,x)
   ...
   val x = (x,x)
in
   (#1 (...(#1 x)...))
end
```

The source term

```
let
   val x = Λt1.λx : t1.x
   val x = Λt1Λt2.(x t1, x t2)
   ...
   val x =
      Λt1 ... Λt2^n.(x t1 ··· t2^(n-1), x t2^(n-1)+1 ··· t2^n)
in
   (#1 (...(#1 (x int s1 ··· s2^n-1))...)) 1
end
```

The explicitly typed term

Figure 2.1: An example of ML polymorphism related to type passing implementation

For relaxing these restrictions, some languages find more rules to enlarge the class of non-expansive expressions. For example, Standard ML of New Jersey 97' adds the following two rules.

- record or tuple with non-expansive fields
- constructor (except ref) applied to non-expansive arguments

However, the combination of value polymorphism and ML type inference still excludes a number of safe programs. Let us consider the following example.

```
val r = ((fn x => x) 1, fn x => x)
```

In this example, the variable r is given a monomorphic type, i.e. $int \times (X \to X)$ (X is a dummy type variable), since the first component is an expansive expression. The second component is therefore given a monomorphic function type where the argument is instantiated to a dummy type X. This implies that any application of the second component should be rejected by the type checking system. This situation is unreasonable because the second component `fn x => x` can be safely given a polymorphic type $\forall t.t \to t$, and it could be used polymorphically in the following context.

For the same reason, ML type inference system may reject a large number of safe programs involving rich data structures which contain high-order objects. This is an unexpected behavior of such a modern language.

2.1.3 Problem with type-directed compilation

The second problem addresses to the inefficiency of the current ML type system in type-passing implementation of many type-directed compilation methods including the one I am going to develop in the following chapters.

To identify the problem, let us consider the example depicted in Figure 2.1 (the example is taken from [OY99]). As seen from the example, the explicitly typed term achieved from ML type inference system contains many unnecessary type abstractions and type applications. This is due to the ML restriction of polymorphism: type abstractions are only allowed at top-level. ML type inference should therefore perform both type application and type abstraction when constructing a polymorphic term from other polymorphic term, e.g. making polymorphic records in the given example.

In type-directed compilation approaches including intensional type analysis [HM95], polymorphic record compilation [Oho95], and the compilation method I am going to present, static type information are encoded to use at run-time for dealing with polymorphism. Type abstractions and type applications would therefore be compiled into lambda abstractions and lambda applications, respectively. Compilation and execution of the given example may suffer from exponential overhead problem arising from the explosion of type abstractions and type applications.

In real-life applications, this extreme case may not often occur. But at least, run-time overhead increases in proportion to the number of nested type abstractions and type applications. In a large and modular program, this may result in a significant reduction of the run-time efficiency.

2.2 Rank-1 Polymorphism

Both of the described drawbacks of conventional ML type system can be removed by the solution proposed in [OY99]. In this research, the authors first extend the set of ML types to rank-1 polymorphic types, and then propose an efficient unification-based type inference algorithm for implementing practical compilers.

Following [KT92], the set of rank-k polymorphic types is inductively defined as follows.

$R(0)$	$=$	τ	Monomorphic type
$R(k)$	$=$	$R(k-1)$	Rank-(k-1) polymorphic type
	$\mid$	$\forall t.R(k)$	Rank-k second-order type
	$\mid$	$R(k-1) \to R(k)$	Rank-k function type
	$\mid$	$R(k) \times \cdots \times R(k)$	Rank-k product type

Under the definition, ML monomorphic types can be regarded as rank-0 type ($R(0)$) and type quantifier ($\forall$) can appear at strictly positive positions (i.e. those that are not at the left of function arrow) of any type constructors.

Apparently, the set of rank-1 polymorphic types is a super set of the set of ML types. Supporting rank-1 polymorphic types, an ML compiler can accept a more larger class of programs, e.g. the ones containing high-order polymorphic objects inside data structures. Moreover, an ML type inference system equipped with rank-1 polymorphism can avoid a large number of unnecessary type abstractions and type applications. Considering again the example depicted in Figure 2.1, the following explicitly typed term can be achieved by such a type inference system.

```
let
   val x = Λt₁.λx : t₁.x
   val x = (x,x)
   ...
   val x = (x,x)
in
   (#1 (...(#1 x))...))  int 1
end
```

When being compiled by a type-directed compilation method, this result would give a much more efficient code than the one resulted by conventional ML type inference system.

There is one question arising to be answered for implementing a practical compiler: "how can such a type inference system be implemented efficiently?". Note that there exists more powerful type systems than that of rank-1 polymorphism, but designing efficient type inference algorithm for them was always problematic. The solution proposed in [OY99] is a reasonable answer for the question. The general idea behind this solution is to introduce type abstractions at the time of lambda abstraction instead of at the variable binding of `let` terms as conventional ML type inferencer does. Type applications are delayed until they are really needed, e.g. at function application. In the above example, the proposed type inference algorithm allows polymorphic records to be constructed without performing extra type abstractions and type applications.

I shall not go further in details of this solution since this is out of the thesis concerns and this may distract readers from the subject matter of the thesis.

2.3 The Source Calculus – λ^{ML}

Now I am going to define an explicitly typed polymorphic calculus equipped with rank-1 polymorphism. This calculus serves as the source calculus for both type-directed compilation methods (bitmap-passing compilation and unboxed compilation) presented in next two chapters. Without considering the semantic correctness of the type-directed compilations, in this section, I only present the syntactic behaviors of the source calculus.

2.3.1 Types

Let t range over an infinite set of type variables and o range over a set of base type, the sets monomorphic types (ranged over by τ) and polymorphic types (ranged over by σ) are given by the following syntax.

$$
\begin{array}{rcll}
\tau & ::= & o & \text{base type} \\
 & | & t & \text{type variable} \\
 & | & \tau \to \tau & \text{monomorphic function type} \\
 & | & \tau \times \cdots \times \tau & \text{monomorphic product type} \\
\sigma & ::= & \tau & \\
 & | & \forall t.\sigma & \text{second-order type} \\
 & | & \tau \to \sigma & \text{polymorphic function type} \\
 & | & \sigma \times \cdots \times \sigma & \text{polymorphic product type}
\end{array}
$$

Let $\overline{a}$ be a sequence of objects denoted by the meta-variable a. We write $\overline{\tau} \to \sigma$ and $\forall\overline{t}.\sigma$ as shorthands for $\tau_1 \to \cdots \to \tau_n \to \sigma$ and $\forall t_1 \cdots \forall t_m.\sigma$ where $\overline{\tau} = \{\tau_1, \ldots, \tau_n\}$ and $\overline{t} = \{t_1, \ldots, t_m\}$.

Let σ be a type, the set of type variables that are free in σ (written $FTV(\sigma)$) is defined as follows.

$$
\begin{array}{rcl}
FTV(o) & = & \emptyset \\
FTV(t) & = & \{t\} \\
FTV(\tau \to \sigma) & = & FTV(\tau) \cup FTV(\sigma) \\
FTV(\sigma_1 \times \cdots \times \sigma_n) & = & \bigcup_{i=1}^{n} FTV(\sigma_i) \\
FTV(\forall t.\sigma) & = & FTV(\sigma) - \{t\}
\end{array}
$$

We write $FTV(\overline{\sigma})$ for the union set of all $FTV(\sigma_i)$ where $\sigma_i \in \overline{\sigma}$. A *type substitution*, or simply *substitution*, is a function that maps from a finite set of type variables to monotypes. We write $S = [\tau_1/t_1, \ldots, \tau_n/t_n]$ for the substitution that maps t_i to τ_i, and define $dom(S)$ for the set $\{t_1, \ldots t_n\}$. A substitution S is extended to set of all type variables by letting $S(t) = t$ for all $t \notin dom(S)$. By this extension, the application of a substitution S to a type σ, written $S(\sigma)$, is defined as follows.

$$\begin{array}{rcl} S(o) & = & o \\ S(t) & = & S(t) \\ S(\tau \rightarrow \sigma) & = & S(\tau) \rightarrow S(\sigma) \\ S(\sigma_1 \times \cdots \times \sigma_n) & = & S(\sigma_1) \times \cdots \times S(\sigma_n) \\ S(\forall t.\sigma) & = & \forall t.S(\sigma) \text{ (for } t \notin dom(S)) \end{array}$$

Under the bound type variable convention, we can assume that the condition in the last rule always satisfies.

2.3.2 Syntax

The set of terms of λ^{ML} is given by the following grammar.

$$\begin{array}{rcll} e & ::= & c^o & \text{constant} \\ & | & x & \text{variable} \\ & | & \lambda\overline{x:\tau}.e & \text{function} \\ & | & (e\ \overline{e}) & \text{application} \\ & | & \Lambda\overline{t}.e & \text{type abstraction} \\ & | & (e\ \overline{\tau}) & \text{type application} \\ & | & (e, \ldots, e) & \text{record} \\ & | & \pi_i(e) & \text{projection} \\ & | & \texttt{let}\ x : \sigma = e\ \texttt{in}\ e\ \texttt{end} & \text{let binding} \end{array}$$

c^o is a constant of the base type o. x ranges over an infinite set of identifiers. Lambda abstractions ($\lambda\overline{x:\tau}.e$) and lambda applications ($(e_1\ \overline{e_2})$) are generalized with multiple arguments. Since bit tag/size abstractions and bit tag/size applications usual occur together with ordinary abstractions and applications, this generalization is useful for generating optimized code by uncurrying extra arguments to the ordinary lambda argument. I shall comment on this issue in Section 7.3. Type abstractions $\Lambda\overline{t}.e$ and type applications $(e\ \overline{\tau})$ are explicitly defined in the syntax of terms. $(e_1, \ldots, e_n)$ and $\pi_i(e)$ are term constructors for records and projections, respectively. ML `let` expressions are represented by `let` $x : \sigma = e$ `in` e `end`. Parentheses may be used for eliminating syntactic ambiguity.

Example. Here are two examples of syntactically correct expressions.

`let` $f : \forall t_1.t_1 \rightarrow \forall t_2.t_2 \rightarrow t_1 \times t_2$ `=` $\Lambda t_1.\lambda x : t_1.\Lambda t_2.\lambda y : t_2.(x, y)$
`in f` *int* `1` *string* `"a" end;`

`let` $r : (\forall t_1.t_1 \rightarrow t_1) \times (\forall t_2.t_2 \rightarrow t_2)$ `=` $(\Lambda t_1.\lambda x : t_1.x, \Lambda t_2.\lambda y : t_2.y)$
`in` $\pi_1(r)$ *int* `1 end;`

In the first example, f is given a nested polymorphic type. In the second one, r is a record consisting of two polymorphic fields. Comparing with the results achieved from

$$\vdash_{ML} \emptyset$$

$$\frac{\vdash_{ML} \Gamma \quad FTV(\overline{\tau}) \subseteq TV(\Gamma) \quad \overline{x} \cap dom(\Gamma) = \emptyset}{\vdash_{ML} \Gamma, arg(\overline{x : \tau})}$$

$$\frac{\vdash_{ML} \Gamma \quad TV(\Gamma) \cap \overline{t} = \emptyset}{\vdash_{ML} \Gamma, tvar(\overline{t})}$$

$$\frac{\vdash_{ML} \Gamma \quad FTV(\sigma) \subseteq TV(\Gamma) \quad x \notin dom(\Gamma)}{\vdash_{ML} \Gamma, local(x : \sigma)}$$

Figure 2.2: Context Formation Rules

conventional ML type system, f and r can be partially instantiated without introducing any extra type abstraction or type application.

For the sake of readability, recursions are omitted from the syntax of the calculus. In Chapter 7, I shall give a presentation of how to incorporate mutually recursive function definitions into the language.

2.3.3 Typing Environment

In order to type source expressions, let us define typing environments (or contexts) of λ^{ML} as follows.

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, arg(\overline{x : \tau}) \mid \Gamma, local(x : \sigma) \mid \Gamma, tvar(\overline{t})$$

Intuitively, a context Γ is a sequence of assumptions of the forms $arg(\overline{x : \tau})$, $local(x : \sigma)$, and $tvar(\overline{t})$ for lambda variables, `let` variables, and type variables. To introduce bit tag/size abstractions, it is necessary to record type variables explicitly in the context.

Let $TV(\Gamma)$ be the set of type variables declared in Γ.

$$\begin{array}{rcl} TV(\emptyset) & = & \emptyset \\ TV(\Gamma, arg(\overline{x : \tau})) & = & TV(\Gamma) \\ TV(\Gamma, local(x : \sigma)) & = & TV(\Gamma) \\ TV(\Gamma, tvar(\overline{t})) & = & TV(\Gamma) \cup \overline{t} \end{array}$$

A type σ is *well-formed* under a well-formed context Γ, written $\Gamma \vdash_{ML} \sigma$, if $FTV(\sigma) \subseteq TV(\Gamma)$. We write $\vdash_{ML} \Gamma$ if a context Γ is well-formed. Variables (as well as type variables) recorded in a well-formed context are distinct. We write $x : \sigma \in \Gamma$ for checking the appearance of $x : \sigma$ in assumptions $arg()$ and $local()$ of Γ. We also write $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$, and define $dom(\Gamma)$ as the set of all variables recorded in Γ. The well-formedness of a context is derived by the set of rules depicted in Figure 2.2.

Given a type substitution S and a context Γ, we define the type substitution of the context, i.e. $S(\Gamma)$, by the following rules.

$$\begin{array}{rcl} S(\emptyset) & = & \emptyset \\ S(\Gamma, arg(\overline{x : \tau})) & = & S(\Gamma), arg(\overline{x : S(\tau)}) \\ S(\Gamma, local(x : \sigma)) & = & S(\Gamma), local(x : S(\sigma)) \\ S(\Gamma, tvar(\overline{t})) & = & S(\Gamma), tvar(\overline{t}) \end{array}$$

Since type variables are recorded explicitly in contexts, the well-formedness of context may not be preserved under type substitution. In order to keep this property, we put a particular restriction on type substitutions.

Firstly, we define orders on type variables declared in a context Γ as:

$$t_1 \prec_\Gamma t_2 \text{ if } t_1 \text{ is recorded before } t_2 \text{ in } \Gamma$$

Note that $t_1 \prec_\Gamma t_2$ implies that both t_1 and t_2 are recorded in the context Γ.

Secondly, we define that a substitution S *respects* a context Γ if for all $t_1 \in TV(\Gamma) \cap dom(S)$, and for all $t_2 \in FTV(S(t_1))$, we have $t_2 \prec_\Gamma t_1$. Intuitively, if S respects Γ then for any type variable t in the domain of S and being recorded in Γ, $S(t)$ should be well-formed under Γ and does not contain any free type variable declared after t. The following properties are easy to check.

Lemma 2.1 *Let* Γ *be a well-formed context,* S *be a type substitution that respects* Γ*, and* σ *be a well-formed type under* Γ*. Then* $\Gamma \vdash_{ML} S(\sigma)$*.*

PROOF. Since σ is well-formed under Γ, then $FTV(\sigma) \subseteq TV(\Gamma)$. The substitution S to σ only replaces free type variables, e.g. t, in $FTV(\sigma)$ by the corresponding type obtained from S, e.g. $S(t)$. Since S respects Γ, then we have $FTV(S(t)) \subseteq TV(\Gamma)$. This implies that $FTV(S(\Gamma)) \subseteq TV(\Gamma)$. Then we have $S(\Gamma)$ is well-formed under Γ as desired. $\square$

Lemma 2.2 *Let* Γ *be a well-formed context,* S *be a type substitution that respects* Γ*. Then* $S(\Gamma)$ *is also a well-formed context.*

PROOF. This is proved by induction on the structure of Γ.

Case $\Gamma = \emptyset$**.** Straightforward

Case $\Gamma = \Gamma', arg(\overline{x : \tau})$**.** Because S respects Γ then S must also respect Γ'. Applying induction hypothesis for Γ', we have $S(\Gamma')$ is well-formed. Since Γ is well-formed, we have $\Gamma' \vdash_{ML} \tau$. Because S respects Γ', by the Lemma 2.1 we have $\Gamma' \vdash_{ML} \overline{S(\tau)}$. Furthermore, Γ' and $S(\Gamma')$ have the same set of type variables, we can easily derive $S(\Gamma') \vdash_{ML} \overline{S(\tau)}$. By the context formation rules, we have $\vdash_{ML} S(\Gamma'), arg(\overline{x : S(\tau)})$. By substitution rules for contexts, we obtain $\vdash_{ML} S(\Gamma', arg(\overline{x : \tau}))$ as desired.

Case $\Gamma = \Gamma', local(x : \sigma)$**.** Similar to the previous case.

Case $\Gamma = \Gamma', tvar(\overline{t})$**.** Since S respects Γ, we have that S also respects Γ'. Applying induction hypothesis for Γ', we obtain that $S(\Gamma')$ is well-formed. Since $TV(\Gamma') = TV(S(\Gamma'))$, then $\overline{t} \cap TV(S(\Gamma')) = \emptyset$. Then we have $\vdash_{ML} S(\Gamma'), tvar(\overline{t})$. By type substitution rules for context, we obtain $\vdash_{ML} S(\Gamma', tvar(\overline{t}))$ as desired $\square$

Lemma 2.3 *Let* Γ *be a well-formed context,* S *be a type substitution that respects* Γ*, and* σ *be a well-formed type under* Γ*. Then* $S(\Gamma) \vdash_{ML} S(\sigma)$*.*

PROOF. Firstly, by applying Lemma 2.1, we have $\Gamma \vdash_{ML} S(\sigma)$. By Lemma 2.2, we have $S(\Gamma)$ is well-formed. Since $TV(\Gamma) = TV(S(\Gamma))$, then $FTV(\sigma) \subseteq TV(S(\Gamma))$. We obtain $S(\Gamma) \vdash_{ML} S(\sigma)$ as desired. $\square$

Let's define *ground substitution* as a type substitution that assign each type variable t to a *closed type* σ ($FTV(\sigma) = \emptyset$). Apparently, a ground substitution respects any context.

$$\Gamma \vdash_{ML} c^o : o$$

$$\Gamma \vdash_{ML} x : \sigma \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{ML} e : \sigma}{\Gamma \vdash_{ML} \lambda \overline{x : \tau}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\overline{t}) \vdash_{ML} e : \sigma}{\Gamma \vdash_{ML} \Lambda \overline{t}.e : \forall \overline{t}.\sigma}$$

$$\frac{\Gamma \vdash_{ML} e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_{ML} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{\Gamma \vdash_{ML} e : \forall \overline{t}.\sigma \quad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{\tau}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_{ML} e_i : \sigma_i \quad \text{for all } 1 \le i \le n}{\Gamma \vdash_{ML} (e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_{ML} e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_{ML} \pi_i(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_{ML} e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_{ML} e_2 : \sigma_2}{\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Figure 2.3: The Type System of The Source Calculus

Lemma 2.4 *For a ground substitution S and a context Γ, if $TV(\Gamma) \subseteq dom(S)$ then*

1. *for all σ so that $\Gamma \vdash_{ML} \sigma$, then $S(\sigma)$ is a closed type*
2. $S(\Gamma)$ *is a* ground context *which assigns a closed type to each variable*

2.3.4 Typing Rules

We write $\Gamma \vdash_{ML} e : \sigma$ if e has type σ under a well-formed context Γ. The set of rules to derive this judgment is given in Figures 2.3. In the typing rule for variables, we check $x : \sigma \in \Gamma$ by looking up the appearance of $x : \sigma$ in both $arg()$ and $local()$ assumptions of Γ. If x is defined in a $local()$ assumption, x can be given a polymorphic type. In the rule for type instantiation terms, the instance type $\sigma[\overline{\tau}/\overline{t}]$ should be well-formed under the context Γ by the given condition $\Gamma \vdash_{ML} \overline{\tau}$ (we write this for the logical conjunction of all conditions $\Gamma \vdash_{ML} \tau_i$, $\tau_i \in \overline{\tau}$).

As seen from the typing rules, polymorphic type can be given to any term (except constants and arguments). This would be more flexible than conventional ML type system where polymorphic types are only given to local variables.

Chapter 3

Bitmap-passing Compilation

In the present chapter I introduce the bitmap-passing compilation method. Firstly, I define a target calculus where bitmap information of heap-allocated objects are formulated. Then I establish operational semantics to give the meaning of terms in the calculus, and show that the typing system of the calculus is sound with respect to the given operational calculus. Finally, I develop a type-preserved compilation algorithm which translates terms in the source calculus (presented in previous chapter) into terms in the target calculus. The combination of these development steps guarantees that the type system of the source calculus is sound with respect to the operational semantics realized by bitmap-passing compilation followed by evaluation of the compiled term.

3.1 The Target Calculus – Λ^B

I am going to define the target calculus, namely *bitmap-passing calculus* or Λ^B, for the bitmap-passing compilation method. Firstly, I define the syntax and the type system of the calculus involving bitmaps and bit tags. Then, I define operational semantics that faithfully models the evaluation of expressions under bitmap-inspecting garbage collection. The type system of the calculus is shown to be sound with respect to the operational semantics to ensure that the evaluation of a typable expression in Λ^B never fails.

Different from λ^{ML}, Λ^B is formulated as an implicitly typed calculus in ML style for the convenience of establishing the soundness property. In a practical implementation, several type-based optimizations may require type information from target terms. Later in the presentation of bitmap-passing compilation algorithm, I will introduce an explicitly typed calculus, written λ^B, as an immediate language for the compilation method. Terms in this calculus contain enough type information for optimizations, and we can prove that if we "erase" all type annotations in an well-typed λ^B term, we can achieve an Λ^B term of the same type.

3.1.1 Types

The set of types of terms in Λ^B is extended for bitmaps and bit tags as follows.

$$
\begin{array}{rcll}
\tau & ::= & o & \text{base type} \\
 & | & t & \text{type variable} \\
 & | & \langle\sigma\rangle & \text{bit tag type} \\
 & | & \langle\!\langle \sigma, \ldots, \sigma \rangle\!\rangle & \text{bitmap type} \\
 & | & \overline{\tau} \rightarrow \tau & \text{monomorphic function type} \\
 & | & \tau \times \cdots \times \tau & \text{monomorphic product type} \\
\sigma & ::= & \tau & \\
 & | & \forall \overline{t}.\sigma & \text{second-order type} \\
 & | & \overline{\tau} \rightarrow \sigma & \text{polymorphic function type} \\
 & | & \sigma \times \cdots \times \sigma & \text{polymorphic product type}
\end{array}
$$

$\langle\sigma\rangle$ is a bit tag type which denotes a singleton set of bit tag ($\{0\}$ or $\{1\}$) for objects of type σ. In ML Core language, for any type σ other than type variable, whether σ is a boxed type or an unboxed type is determined by its outermost type constructor. I define the function $\texttt{tagOf}(\sigma)$ which determines the tag value of objects of type σ as follows.

$$
\begin{array}{rcl}
\texttt{tagOf}(o) & = & 0 \text{ if } o \text{ is an unboxed base type} \\
\texttt{tagOf}(o) & = & 1 \text{ if } o \text{ is an boxed base type} \\
\texttt{tagOf}(\overline{\tau} \rightarrow \sigma) & = & 1 \\
\texttt{tagOf}(\forall \overline{t}.\sigma) & = & 1 \\
\texttt{tagOf}(\sigma_1 \times \cdots \times \sigma_n) & = & 1 \\
\texttt{tagOf}(\langle\sigma\rangle) & = & 0 \\
\texttt{tagOf}(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) & = & 0 \\
\texttt{tagOf}(t) & = & \text{undetermined}
\end{array}
$$

The case for $\forall \overline{t}.\sigma$ requires some explanation: this type refers to a polymorphic function type since bit tag abstractions are always inserted at type abstractions. Then we can safely assume that $\forall \overline{t}.\sigma$ is a boxed type.

A bitmap type $\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle$ denotes a singleton set of the bitmap of records of type $\sigma_1 \times \cdots \times \sigma_n$. Value of the bitmap can be composed from tag values of objects of types $\sigma_1, \ldots, \sigma_n$.

The function FTV is extended for bit tag and bitmap types as:

$$
\begin{array}{rcl}
FTV(\langle\sigma\rangle) & = & FTV(\sigma) \\
FTV(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) & = & \bigcup FTV(\sigma_i)
\end{array}
$$

The set of substitution rules for types is also extended for bitmap and bit tag types as follows.

$$
\begin{array}{rcl}
S(\langle\sigma\rangle) & = & \langle S(\sigma)\rangle \\
S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) & = & \langle\!\langle S(\sigma_1), \ldots, S(\sigma_n) \rangle\!\rangle
\end{array}
$$

3.1.2 Syntax

The set of terms of Λ^B is given by the following syntax:

$$
\begin{array}{rclll}
e & ::= & c^o & & \text{constant} \\
 & | & x & & \text{variable} \\
 & | & \lambda\overline{x}.e & & \text{function} \\
 & | & (e\ \overline{e}) & & \text{application} \\
 & | & (e; e, \ldots, e) & & \text{record} \\
 & | & \pi_i(e) & & \text{projection} \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e\ \texttt{end} & & \text{let binding} \\
 & | & [e, \ldots, e] & & \text{bitmap} \\
 & | & \langle B \rangle & & \text{constant bit tag } (B \in \{0, 1\})
\end{array}
$$

In this implicitly typed calculus, type information are hidden from terms. The lambda abstraction $\lambda\overline{x}.e$ plays dual roles: this term can represent the resulting term of either an ordinary lambda abstraction or a type abstraction in λ^{ML}. More clearly, since bit tag abstractions are inserted at type abstractions and they are encoded as the same as ordinary lambda abstractions, a type abstraction $\Lambda\overline{t}.e$ in λ^{ML} will be compiled into the term $\lambda\overline{b}.e'$ in Λ^B where $\overline{b}$ represent the formal bit tag parameters corresponding to $\overline{t}$.

Similarly, the lambda application $(e_1\ \overline{e_2})$ also plays dual roles: this can represent the resulting term of either an ordinary lambda application or a type instantiation in λ^{ML}. A type instantiation $(e\ \overline{\tau})$ in λ^{ML} will be compiled into $(e'\ \overline{e_b})$ where $\overline{e_b}$ are the actual bit tag parameters generated from $\overline{\tau}$.

Bit tag constants ($\langle B \rangle$) and bitmaps ($[e_1, \ldots, e_n]$) are formulated as first class objects in Λ^B. We distinguish between bit tag constants and integer numbers. $\langle 0 \rangle$ and $\langle 1 \rangle$ are syntactic bit tag values for *unboxed* objects (non-pointers) and *boxed* objects, respectively. $[e_1, \ldots, e_n]$ is the term constructor for a bitmap composed from n bit tags $\{e_1, \ldots, e_n\}$. The first component e_0 in a record $(e_0; e_1, \ldots, e_n)$ is a bitmap term representing the bitmap information of this record.

3.1.3 Typing Environment

Typing environment (context) of Λ^B is defined as the same as in λ^{ML}. Let us distinguish the well-formednesses of types and contexts of Λ^B to those of λ^{ML} by judgments of forms $\Gamma \vdash_B \sigma$, $\vdash_B \Gamma$. The former is for well-formedness of types, the latter is for well-formedness of contexts.

The rules to derive these judgments are defined as the same as in λ^{ML}. The following properties still hold.

Lemma 3.1 *Let Γ be a well-formed context, σ be a type, and S be a substitution that respects Γ. If $\Gamma \vdash_B \sigma$ then $S(\Gamma) \vdash_B S(\sigma)$.*

Lemma 3.2 *Let Γ be a well-formed context and S be a substitution that respects Γ. If $\vdash_B \Gamma$ then $\vdash_B S(\Gamma)$.*

Lemma 3.3 *For a ground substitution S and a context Γ, if $TV(\Gamma) \subseteq dom(S)$ then*

1. *for all σ so that $\Gamma \vdash_B \sigma$, then $S(\sigma)$ is a closed type*
2. *$S(\Gamma)$ is a* ground context

$$\Gamma \vdash_B c^o : o$$

$$\Gamma \vdash_B x : \sigma \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda \overline{x}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda \overline{b}.e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma}$$

$$\frac{\Gamma \vdash_B e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_B \overline{e_2} : \overline{\tau}}{\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{\Gamma \vdash_B e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma \quad \Gamma \vdash_B \overline{e_b} : \overline{\langle \tau \rangle} \quad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_B (e\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_B e_i : \sigma_i\ (1 \le i \le n) \quad \Gamma \vdash_B e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_B e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_B \pi_i(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_B e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_B e_2 : \sigma_2}{\Gamma \vdash_B \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} : \sigma_2}$$

$$\frac{\texttt{tagOf}(\sigma) = B}{\Gamma \vdash_B \langle B \rangle : \langle \sigma \rangle}$$

$$\frac{\Gamma \vdash_B e_i : \langle \sigma_i \rangle\ (1 \le i \le n)}{\Gamma \vdash_B [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

Figure 3.1: Typing Rules of The Target Calculus

3.1.4 Typing Rules

Typing rules for the target calculus are formulated as judgments of the form $\Gamma \vdash_B e : \sigma$, which read e has type σ under Γ. Figure 3.1 gives the set of typing rules of the target calculus. Since $\lambda\overline{x}.e$ plays dual roles, we may have two different typing derivations for this term. One is to give a function type to the term, and other is to give a second order type to the term (we call these two rules are typing rule for lambda abstraction and typing rule for type abstraction). Similarly, $(e_1\ \overline{e_2})$ may also have two typing derivations: one is a typing derivation for ordinary lambda application and other for type instantiation with bit tag application (we call these two rules are typing rule for lambda application and typing rule for type instantiation). In the rule for typing records, we check the consistency of bitmap type and record's field types by adding $\Gamma \vdash_B e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle$ in the premises. This bitmap type check is proceeded by checking type of each bit tag in the bitmap. Careful readers may notice that a bit tag is either a variable or a constant. Therefore the bit tag type is either recorded in the context or statically given by the function `tagOf()`.

Now we show that the type system of Λ^B is stable under type substitution in the following lemma.

Lemma 3.4 *Let e be an expression, σ be a type, Γ be a well-formed context and S be a substitution that respects Γ. If $\Gamma \vdash_B e : \sigma$ then $S(\Gamma) \vdash_B e : S(\sigma)$.*

PROOF. This is proved by induction on the derivation of the typing rule $\Gamma \vdash_B e : \sigma$.

Case $\Gamma \vdash_B c^o : o$**.** Straightforward.

Case $\Gamma \vdash_B x : \sigma$**.** By typing rule for variable, $\Gamma(x) = \sigma$. By Lemma 3.2, we have $S(\Gamma)$ is a well-formed context. By the typing rule for variable, $S(\Gamma) \vdash_B x : S(\Gamma)(x)$. By the definition of type substitution for context, $S(\Gamma)(x) = S(\Gamma(x)) = S(\sigma)$ as desired.

Case $\Gamma \vdash_B \lambda\overline{x}.e : \overline{\tau} \to \sigma$**.** Suppose this is derived from

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda\overline{x}.e : \overline{\tau} \to \sigma}$$

Since S respects Γ, we have that S also respects $\Gamma, arg(\overline{x : \tau})$. Applying induction hypothesis for $\Gamma, arg(\overline{x : \tau}) \vdash_B e : \sigma$, we obtain

$$S(\Gamma, arg(\overline{x : \tau})) \vdash_B e : S(\sigma)$$

By definition of type substitution for context, we have

$$S(\Gamma, arg(\overline{x : \tau})) = S(\Gamma), arg(\overline{x : S(\tau)})$$

Thus $S(\Gamma), arg(\overline{x : S(\tau)}) \vdash_B e : S(\sigma)$. Applying typing rule for lambda abstraction, we obtain

$$S(\Gamma) \vdash_B \lambda\overline{x}.e : \overline{S(\tau)} \to S(\sigma)$$

By type substitution for type, we have $\overline{S(\tau)} \to S(\sigma) = S(\overline{\tau} \to \sigma)$ as desired.

Case $\Gamma \vdash_B \lambda\overline{b}.e : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma$**.** By bound type variable convention, we assume that $\overline{t} \cap dom(\Gamma) = \emptyset$ and $\overline{t} \cap dom(S) = \emptyset$. Suppose this typing is derived from

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda\overline{b}.e : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma}$$

Since $\overline{t} \cap dom(S) = \emptyset$ and S respects Γ, we also have that S respects $\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle})$. Applying induction hypothesis for $\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle}) \vdash_B e : \sigma$, we obtain

$$S(\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle})) \vdash_B e : S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then

$$S(\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle})) = S(\Gamma), tvar(\overline{t}), arg(\overline{b : \langle t\rangle})$$

By typing rule for type abstraction, we have

$$S(\Gamma) \vdash_B \lambda\overline{b}.e : \forall\overline{t}.\overline{\langle t\rangle} \to S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, we have $\forall\overline{t}.\overline{\langle t\rangle} \to S(\sigma) = S(\forall\overline{t}.\overline{\langle t\rangle} \to \sigma)$ as desired.

Case $\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_B \overline{e_2} : \overline{\tau}}{\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma}$$

Applying induction hypothesis for $\Gamma \vdash_B e_1 : \overline{\tau} \to \sigma$ and $\Gamma \vdash_B \overline{e_2} : \overline{\tau}$, we obtain

$$S(\Gamma) \vdash_B e_1 : S(\overline{\tau} \to \sigma)$$
$$S(\Gamma) \vdash_B \overline{e_2} : \overline{S(\tau)}$$

Since $S(\overline{\tau} \to \sigma) = \overline{S(\tau)} \to S(\sigma)$, applying typing rule for lambda application, we obtain

$$S(\Gamma) \vdash_B (e_1\ \overline{e_2}) : S(\sigma)$$

as desired.

Case $\Gamma \vdash_B (e\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma \quad \Gamma \vdash_B \overline{e_b} : \overline{\langle \tau \rangle} \quad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_B (e\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for $\Gamma \vdash_B e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma$ and $\Gamma \vdash_B \overline{e_b} : \overline{\langle \tau \rangle}$, we obtain

$$S(\Gamma) \vdash_B e : S(\forall \overline{t}.\overline{\langle t \rangle} \to \sigma)$$
$$S(\Gamma) \vdash_B \overline{e_b} : \overline{S(\langle \tau \rangle)}$$

By bound type variable convention, assume that $\overline{t} \cap dom(S) = \emptyset$. Thus $S(\forall \overline{t}.\overline{\langle t \rangle} \to \sigma) = \forall \overline{t}.\overline{\langle t \rangle} \to S(\sigma)$. By type substitution for type, $\overline{S(\langle \tau \rangle)} = \overline{\langle S(\tau) \rangle}$. Applying Lemma 3.1, we also have $S(\Gamma) \vdash_B \overline{\tau}$. By typing rule for type instantiation, we obtain

$$S(\Gamma) \vdash_B (e\ \overline{e_b}) : S(\sigma)[\overline{S(\tau)}/\overline{t}].$$

Since $\overline{t} \cap dom(S) = \emptyset$, we have $S(\sigma)[\overline{S(\tau)}/\overline{t}] = S(\sigma[\overline{\tau}/\overline{t}])$ as desired

Case $\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e_i : \sigma_i, \text{ for all } 1 \leq i \leq n \quad \Gamma \vdash_B e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

Applying induction hypothesis for each e_i, we have

$$S(\Gamma) \vdash_B e_i : S(\sigma_i)$$
$$S(\Gamma) \vdash_B e_0 : S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle)$$

Since $S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) = \langle\!\langle S(\sigma_1), \ldots, S(\sigma_n) \rangle\!\rangle$, we have

$$S(\Gamma) \vdash_B e_0 : \langle\!\langle S(\sigma_1), \ldots, S(\sigma_n) \rangle\!\rangle$$

Applying the typing rule for records, we have

$$S(\Gamma) \vdash_B (e_0; e_1, \ldots, e_n) : S(\sigma_1) \times \cdots \times S(\sigma_1)$$

Since $S(\sigma_1) \times \cdots \times S(\sigma_1) = S(\sigma_1 \times \cdots \times \sigma_n)$, we obtain

$$S(\Gamma) \vdash_B (e_0; e_1, \ldots, e_n) : S(\sigma_1 \times \cdots \times \sigma_1)$$

as desired.

Case $\Gamma \vdash_B \pi_i(e) : \sigma_i$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_B \pi_i(e) : \sigma_i}$$

Applying induction hypothesis for $\Gamma \vdash_B e : \sigma_1 \times \cdots \times \sigma_n$, we have

$$S(\Gamma) \vdash_B e : S(\sigma_1 \times \cdots \times \sigma_n), \text{ we have}$$

Since $S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$, applying typing rule for projection, we obtain $S(\Gamma) \vdash_B \pi_i(e) : S(\sigma_i)$ as desired.

Case $\Gamma \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e_1 : \sigma_1 \quad \Gamma, \mathit{local}(x : \sigma_1) \vdash_B e_2 : \sigma_2}{\Gamma \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for $\Gamma \vdash_B e_1 : \sigma_1$, we obtain

$$S(\Gamma) \vdash_B e_1 : S(\sigma_1)$$

Since S respects Γ, then S also respects $\Gamma, \mathit{local}(x : \sigma)$. Applying induction hypothesis for $\Gamma, \mathit{local}(x : \sigma_1) \vdash_B e_2 : \sigma_2$, we obtain

$$S(\Gamma, \mathit{local}(x : \sigma_1)) \vdash_B e_2 : S(\sigma_2)$$

Since $S(\Gamma, \mathit{local}(x : \sigma_1)) = S(\Gamma), \mathit{local}(x : S(\sigma_1))$, by applying typing rule for `let` term, we obtain

$$S(\Gamma) \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : S(\sigma_2)$$

as desired.

Case $\Gamma \vdash_B [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_B e_i : \langle \sigma_i \rangle \ (1 \le i \le n)}{\Gamma \vdash_B [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

Applying induction hypothesis for each e_i, we have $S(\Gamma) \vdash_B e_i : S(\langle \sigma_i \rangle)$. Since $S(\langle \sigma_i \rangle) = \langle S(\sigma_i) \rangle$, applying the typing rule for bitmap, we have

$$S(\Gamma) \vdash_B [e_i, \ldots, e_n] : \langle\!\langle S(\sigma_1), \ldots, S(\sigma_n) \rangle\!\rangle$$

Applying substitution rule for bitmap type , we can conclude

$$S(\Gamma) \vdash_B [e_i, \ldots, e_n] : S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle)$$

Case $e = \langle B \rangle$**.** Suppose $\Gamma \vdash_B \langle B \rangle : \langle \sigma \rangle$. Since $\langle B \rangle$ is a constant bit tag, type σ must have an appropriate outermost type constructor. Type substitution does not modify the outermost type constructor, then the substituted type $S(\sigma)$ must have the same outermost constructor as σ, therefore $\texttt{tagOf}(\sigma) = \texttt{tagOf}(S(\sigma))$. This implies $S(\Gamma) \vdash_B \langle B \rangle : S(\langle \sigma \rangle)$ as desired. □

3.1.5 Semantics

I define semantics of Λ^B, in style of natural semantic [Kah87], that faithfully models evaluation of expressions under bitmap-inspecting garbage collection. The sets of runtime values (ranged over by v) and runtime environments (ranged over by E) are given by the following grammar.

$$
\begin{array}{lcll}
v & ::= & c^o & \text{constant value} \\
 & | & i & \text{unsigned integer} \\
 & | & cls(E, \lambda\overline{x}.e) & \text{closure} \\
 & | & (v; v, \ldots, v) & \text{record value} \\
 & | & wrong & \text{runtime error} \\
E & ::= & \emptyset \ | \ E, x : v & \text{run-time environment}
\end{array}
$$

Bit tags and bitmaps are evaluated into unsigned integers (denoted by i). A bit tag value is either 0 or 1. A bitmap value represents a sequence of bit tags. We use the notation $[B_1 \circ \cdots \circ B_n]$ for the unsigned integer obtained by setting the least i^{th} bit to the bit tag value B_i.

A bitmap value may exceed one word representation if the number of bit tag components is greater than 32. In order to make sure that bitmaps always fit into one word data, I assume that the maximum number of fields in a record is always smaller than 32 or equal 32. Long record can be converted into a nested record by applying a source-to-source transformation before passed to the compilation algorithm. For example, the record $(e_1, \ldots, e_{40})$ in the source language can be transformed into $(e_1, \ldots, e_{31}, (e_{32}, \ldots, e_{40}))$. Nested representation of records may introduce extra performance cost. But I believe that this extreme case does not often occur in a real-life application.

$cls(E, \lambda\overline{x}.e)$ is a function closure, where E is an environment assigning values to variables. $(v_0; v_1, \ldots, v_n)$ represents a record value whose first component v_0 is the bitmap value of the record block. *wrong* represents a runtime type error.

Instead of explicitly modeling bitmap-inspecting garbage collection, I define the operational semantics in such a way that it checks the correctness of a bitmap every time a record is created or used. If this check fails then the evaluation halts with *wrong*. For performing this check, I define the trace bit $\mathtt{tagOf}(v)$ corresponding to v as follows.

$$
\begin{array}{rcl}
\mathtt{tagOf}(c^o) & = & 0 \text{ if } c^o \text{ is an unboxed constant} \\
\mathtt{tagOf}(c^o) & = & 1 \text{ if } c^o \text{ is a boxed constant} \\
\mathtt{tagOf}(i) & = & 0 \\
\mathtt{tagOf}(cls(E, \lambda\overline{x : \tau}.e)) & = & 1 \\
\mathtt{tagOf}(tcls(E, \Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.e)) & = & 1 \\
\mathtt{tagOf}((v_0; v_1, \ldots, v_n)) & = & 1
\end{array}
$$

The operational semantics are defined in the style of [Kah87] by giving a set of rules to derive a evaluation relation of the form $E \vdash_B e \Downarrow v$, which reads: "$e$ evaluates to v under E". Figure 3.2 gives the set of evaluation rules. In evaluating a record, runtime system first evaluates the record's bitmap into an integer. After evaluating the record's fields, runtime system checks the bit tag consistency by comparing the bit tags of the actual field's values ($\mathtt{tagOf}(v_i)$) with the corresponding bit tags obtained from the bitmap's value (B_i). If this check fails, then the evaluation process will go wrong. Later in the

$$E \vdash_B c^o \Downarrow c^o$$

$$E \vdash_B x \Downarrow E(x)$$

$$E \vdash_B \lambda\overline{x}.e \Downarrow cls(E, \lambda\overline{x}.e)$$

$$\frac{E \vdash_B e_1 \Downarrow cls(E_0, \lambda\overline{x}.e_0) \quad E \vdash_B \overline{e_2} \Downarrow \overline{v_2} \quad E_0, \overline{x : v_2} \vdash_B e_0 \Downarrow v_0}{E \vdash_B (e_1\ \overline{e_2}) \Downarrow v_0}$$

$$\frac{\begin{array}{c} E \vdash_B e_0 \Downarrow i \quad \text{where } i = [B_1 \circ \cdots \circ B_n] \\ E \vdash_B e_j \Downarrow v_j \quad \texttt{tagOf}(v_j) = B_j \text{ for all } 1 \leq j \leq n \end{array}}{E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow (i; v_1, \ldots, v_n)}$$

$$\frac{E \vdash_B e \Downarrow (v_0; v_1, \ldots, v_n)}{E \vdash_B \pi_i(e) \Downarrow v_i}$$

$$\frac{E \vdash_B e_1 \Downarrow v_1 \quad E, x : v_1 \vdash_B e_2 \Downarrow v_2}{E \vdash_B \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v_2}$$

$$E \vdash_B \langle B \rangle \Downarrow B$$

$$\frac{E \vdash_B e_i \Downarrow B_i \quad (1 \leq i \leq n)}{E \vdash_B [e_1, \ldots, e_n] \Downarrow [B_1 \circ \cdots \circ B_n]}$$

Figure 3.2: Operational Semantics of The Target Calculus

soundness theorem I will show that, for a well typed term, this check never fails. This implies that the evaluation of well typed program never go wrong. Hence, in a practical runtime system, we do not need to implement this check.

In the rule for evaluating bitmap, the value of each bit tag component, i.e. B_i, is evaluated. The bitmap's value is obtained by setting the least i^{th} bit of an zero word to B_i. This would be done by a sequence of logical bitwise operations. This is the only kind of computation left for the bitmap-passing compilation method. In Chapter 7, I shall present several optimizations for reducing runtime overhead arising by this kind of computation.

This set of rules should be taken with the following implicit rules yielding *wrong*: if evaluation of any component yields *wrong* or undefined or does not satisfy the specified condition then the entire term will yield *wrong*. For example, the rules for $(e_0; e_1, \ldots, e_n)$ include, among others, the following one:

$$\frac{E \vdash_B e_0 \Downarrow [B_1 \circ \cdots \circ B_n] \quad E \vdash_B e_1 \Downarrow v_1 \quad \texttt{tagOf}(v_1) \neq B_1}{E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow wrong}$$

To show that the type system is sound with respect to this operational semantics, I define typing judgments on runtime values and environments of the forms $\models_B v : \sigma$ and $\models_B E : \Gamma$ where σ is a closed type and Γ is a ground context. The set of rules to derive these judgments is given in Figure 3.4 and Figure 3.3.

The following properties holds.

Lemma 3.5 *If* $\models_B v : \sigma$ *then* $\texttt{tagOf}(v) = \texttt{tagOf}(\sigma)$.

$$\models_B \emptyset : \emptyset$$

$$\frac{\models_B E : \Gamma \quad \models_B \overline{v} : \overline{\tau}}{\models_B (E, \overline{x : v}) : (\Gamma, arg(\overline{x : \tau}))}$$

$$\frac{\models_B E : \Gamma \quad \models_B v : \sigma}{\models_B (E, x : v) : (\Gamma, local(x : \sigma))}$$

$$\frac{\models_B E : \Gamma}{\models_B E : (\Gamma, tvar(\overline{t}))}$$

Figure 3.3: Typing rules on environments

$$\models_B c^o : o$$

$$\models_B i : \langle\sigma\rangle \text{ if } i = \texttt{tagOf}(\sigma)$$

$$\frac{i = [B_1 \circ \cdots \circ B_n] \quad \models_B B_j : \langle\sigma_j\rangle \quad \text{for all } 1 \leq j \leq n}{\models_B i : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ and closed types } \overline{\tau} \text{ so that} \\ \models_B E : \Gamma \quad \Gamma, arg(\overline{x : \tau}) \vdash_B e : \sigma\end{array}}{\models_B cls(E, \lambda\overline{x}.e) : \overline{\tau} \to \sigma}$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ so that} \\ \models_B E : \Gamma \quad \Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_B e : \sigma\end{array}}{\models_B cls(E, \lambda\overline{b}.e) : \forall\overline{t}.\overline{\langle t \rangle} \to \sigma}$$

$$\frac{\models_B v_i : \sigma_i \text{ for all } 1 \leq i \leq n \quad \models_B v_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\models_B (v_0; v_1, \ldots, v_n) : \sigma_0 \times \cdots \times \sigma_n}$$

Figure 3.4: Typing rules on values

PROOF. From the value typing rules, σ must have a proper outermost type constructor. By the definition of bit tag value of runtime values and types, we can easily derive the expected result. □

As in a conventional type system, the type system of the target calculus is sound, i.e. it guarantees type-error free evaluation of any type correct expression. We show this in the following theorem.

Theorem 3.1 *Let Γ be a context, e be an expression, σ be a type so that $\Gamma \vdash_B e : \sigma$. Let S be a ground substitution so that $TV(\Gamma) \subseteq dom(S)$. Let E be a run-time environment so that $\models_B E : S(\Gamma)$. If $E \vdash_B e \Downarrow v$ then $\models_B v : S(\sigma)$.*

PROOF. We prove this theorem by induction on the derivation of the evaluation rule.

Case $e \vdash_B c^0 \Downarrow c^o$**.** Straightforward.

Case $E \vdash_B x \Downarrow E(x)$**.** Suppose $\Gamma \vdash_B x : \sigma$ is derived from $\Gamma(x) = \sigma$. Since $\models_B E : S(\Gamma)$, we have $\models_B E(x) : S(\Gamma)(x)$. Since $S(\Gamma)(x) = S(\Gamma(x))$, we obtain $\models_B E(x) : S(\sigma)$ as

desired.

Case $E \vdash_B \lambda\overline{x}.e \Downarrow cls(E, \lambda\overline{x}.e)$. We have two possible typing derivation of $\lambda\overline{x}.e$
<u>*Sub-case 1*</u> $\Gamma \vdash_B \lambda\overline{x}.e : \overline{\tau} \to \sigma$. Suppose this typing is derived from

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda\overline{x}.e : \overline{\tau} \to \sigma}$$

S is a ground substitution, then S should respect $\Gamma, arg(\overline{x : \tau})$. Applying substitution lemma 3.4, we obtain

$$S(\Gamma, arg(\overline{x : \tau})) \vdash_B e : S(\sigma)$$

Since $S(\Gamma, arg(\overline{x : \tau})) = S(\Gamma), arg(\overline{x : S(\tau)})$, then

$$S(\Gamma), arg(\overline{x : S(\tau)}) \vdash_B e : S(\sigma)$$

By context formation rule, we have $\Gamma \vdash_B \overline{\tau}$. By lemma 3.3 we have $S(\tau)$ are closed types and $S(\Gamma)$ is a ground context. Then by applying value typing rule for closure, we obtain

$$\models_B cls(E, \lambda\overline{x}.e) : \overline{S(\tau)} \to S(\sigma)$$

Since $\overline{S(\tau)} \to S(\sigma) = S(\overline{\tau} \to \sigma)$, then we have

$$\models_B cls(E, \lambda\overline{x}.e) : S(\overline{\tau} \to \sigma)$$

as desired.
<u>*Sub-case 2*</u> $\Gamma \vdash_B \lambda\overline{x}.e : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma$ Suppose this typing is derived from

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{x : \langle t\rangle}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda\overline{x}.e : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma}$$

By bound type variable convention, suppose that $\overline{t} \cap dom(S) = \emptyset$. Since S is a ground substitution then S respects $\Gamma, tvar(\overline{t}), arg(\overline{x : \langle t\rangle})$. Applying substitution lemma 3.4, we obtain

$$S(\Gamma, tvar(\overline{t}), arg(\overline{x : \langle t\rangle})) \vdash_B e : S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then

$$S(\Gamma), tvar(\overline{t}), arg(\overline{x : \langle t\rangle}) \vdash_B e : S(\sigma)$$

By lemma 3.3 we have that $S(\Gamma)$ is a ground context. By value typing for closure, we obtain

$$\models_B cls(E, \lambda\overline{x}.e) : \forall\overline{t}.\overline{\langle t\rangle} \to S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then we have $\forall\overline{t}.\overline{\langle t\rangle} \to S(\sigma) = S(\forall\overline{t}.\overline{\langle t\rangle} \to \sigma)$ as desired.

Case $E \vdash_B (e_1\ \overline{e_2}) \Downarrow v_0$. Suppose that this is derived from

$$\frac{E \vdash_B e_1 \Downarrow cls(E_0, \lambda\overline{x}.e_0) \quad E \vdash_B \overline{e_2} \Downarrow \overline{v_2} \quad E_0, \overline{x : v_2} \vdash_B e_0 \Downarrow v_0}{E \vdash_B (e_1\ \overline{e_2}) \Downarrow v_0}$$

There are two possible typing derivations for $(e_1\ \overline{e_2})$
<u>*Sub-case 1*</u>.

$$\frac{\Gamma \vdash_B e_1 : \overline{\tau} \rightarrow \sigma \quad \Gamma \vdash_B \overline{e_2} : \overline{\tau}}{\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma}$$

Applying induction hypothesis for e_1 and $\overline{e_2}$, we obtain

$$\models_B cls(E_0, \lambda\overline{x}.e_0) : S(\overline{\tau} \rightarrow \sigma)$$
$$\models_B \overline{v_2} : \overline{S(\tau)}$$

Since $S(\overline{\tau} \rightarrow \sigma) = \overline{S(\tau)} \rightarrow S(\sigma)$, by value typing rule for closure, there must exist a ground context Γ_0 so that $\models_B E_0 : \Gamma_0$ and $\Gamma_0, arg(\overline{x : S(\tau)}) \vdash_B e_0 : S(\sigma)$. This is easy to choose Γ_0 so that it does not contain any $tvar()$ assumption. By typing rule for runtime environment, we have $\models_B (E_0, \overline{x : v_2}) : \Gamma_0, arg(\overline{x : S(\tau)})$ We choose an identical substitution S_0 ($dom(S_0) = \emptyset$), Since Γ_0 does not contain any $tvar()$ assumption, then we can apply induction hypothesis for e_0 to obtain $\models_B v_0 : S(\sigma)$ as desired.
<u>*Sub-case 2*</u>.

$$\frac{\Gamma \vdash_B e_1 : \forall\overline{t}.\overline{\langle t\rangle} \rightarrow \sigma \quad \Gamma \vdash_B \overline{e_2} : \overline{\langle\tau\rangle} \quad \Gamma \vdash_B \overline{\tau}}{\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for e_1 and $\overline{e_2}$, we obtain

$$\models_B cls(E_0, \lambda\overline{x}.e_0) : S(\forall\overline{t}.\overline{\langle t\rangle} \rightarrow \sigma)$$
$$\models_B \overline{v_2} : \overline{S(\langle\tau\rangle)}$$

By the bound type variable convention, assume that $\overline{t} \cap dom(S) = \emptyset$. Therefore $S(\forall\overline{t}.\overline{\langle t\rangle} \rightarrow \sigma) = \forall\overline{t}.\overline{\langle t\rangle} \rightarrow S(\sigma)$. By value typing rule for closure, there must exist a ground context Γ_0 so that $\models_B E_0 : \Gamma_0$ and $\Gamma_0, tvar(\overline{t}), arg(\overline{x : \langle t\rangle}) \vdash_B e_0 : S(\sigma)$. This is easy to choose Γ_0 so that it does not contain any $tvar()$ assumption. Since $\overline{S(\langle\tau\rangle)} = \overline{\langle S(\tau)\rangle}$, then we have

$$\models_B \overline{v_2} : \overline{\langle S(\tau)\rangle}$$

Since $\Gamma \vdash_B \overline{\tau}$, then by lemma 3.3, we have $\overline{S(\tau)}$ are closed types. Choose $S_0 = [\overline{S(\tau)}/\overline{t}]$, S_0 is a ground substitution and $TV(\Gamma_0, tvar(\overline{t}), arg(\overline{x : \langle t\rangle})) \subseteq dom(S_0)$. Since Γ_0 is a ground context and does not contain any $tvar()$ assumption, we have

$$S_0(\Gamma_0, tvar(\overline{t}), arg(\overline{b : \langle t\rangle})) = \Gamma_0, tvar(\overline{t}), arg(\overline{x : \langle S(\tau)\rangle})$$

Since $\models_B E_0 : \Gamma_0$ then

$$\models_B (E_0, \overline{x : v_2}) : (\Gamma_0, tvar(\overline{t}), arg(\overline{x : \langle S(\tau)\rangle}))$$

Now, we apply induction hypothesis for e_0 to obtain $\models_B v_0 : S_0(S(\sigma))$. Since $S_0 = [\overline{S(\tau)}/\overline{t}]$ and $\overline{t} \cap dom(S) = \emptyset$, we can conclude that $\models_B v_0 : S(\sigma[\overline{\tau}/\overline{t}])$ as desired.

Case $E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow (i; v_1, \ldots, v_n)$**.** Suppose this is derived from

$$\frac{\begin{array}{c} E \vdash_B e_0 \Downarrow i \quad \text{where } i = [B_1 \circ \cdots \circ B_n] \\ E \vdash_B e_j \Downarrow v_j \quad \texttt{tagOf}(v_j) = B_j \text{ for all } 1 \leq j \leq n \end{array}}{E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow (i; v_1, \ldots, v_n)}$$

Also suppose $\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n$. This is derived from

$$\frac{\Gamma \vdash_B e_j : \sigma_j \text{ for all } 1 \leq j \leq n \quad \Gamma \vdash_B e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n.}$$

We have to prove: $\texttt{tagOf}(v_j) = B_j$ for each j, and $\models_B (i; v_1, \ldots, v_n) : S(\sigma_1 \times \cdots \times \sigma_n)$. Applying induction hypothesis for e_0 and each e_j, we have

$$\models_B v_j : S(\sigma_j)$$
$$\models_B i : S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle)$$

By lemma 3.5, we have $\texttt{tagOf}(v_i) = \texttt{tagOf}(S(\sigma_i))$. Since

$$S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle = \langle\!\langle S(\sigma_1), \ldots, S(\sigma_n) \rangle\!\rangle$$
$$i = [B_1 \circ \cdots \circ B_n]$$

by value typing rule for bitmap and bit tag, we have $B_j = \texttt{tagOf}(S(\sigma_j))$. Therefore $\texttt{tagOf}(v_j) = B_j$ for each j. Besides, applying value typing rule for records, we obtain,

$$\models_B (i; v_1, \ldots, v_n) : S(\sigma_1) \times \cdots \times S(\sigma_n)$$

Since $S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$, the proof for this case is done.

Case $E \vdash_B \pi_i(e') \Downarrow v_i$**.** This should be derived from

$$E \vdash_B e' \Downarrow (v_0; v_1, \ldots, v_n).$$

Also suppose $\Gamma \vdash_B \pi_i(e') : \sigma_i$. By the typing rule for projection, $\Gamma \vdash_B e' : \sigma_1 \times \cdots \times \sigma_n$. Applying the induction hypothesis for e', $\models_{ML} (v_0; v_1, \ldots, v_n) : S(\sigma_1 \times \cdots \times \sigma_n)$. Since $S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$, by the value typing rule for record, we have $\models_{ML} v_i : S(\sigma_i)$ as desired.

Case $E \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow v_2$**.** Suppose that this is derived from

$$\frac{E \vdash_B e_1 \Downarrow v_1 \quad E, x : v_1 \vdash_B e_2 \Downarrow v_2}{E \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow v_2}$$

Also suppose $\Gamma \vdash_B \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$. By the typing rule for `let` term, we have

$$\Gamma \vdash_B e_1 : \sigma_1$$
$$(\Gamma, \mathit{local}(x : \sigma_1)) \vdash_B e_2 : \sigma_2.$$

Applying induction hypothesis for e_1, we obtain $\models_{ML} v_1 : S(\sigma_1)$. Let's define $\Gamma_1 = \Gamma, \mathit{local}(x : \sigma_1)$, We have $TV(\Gamma_1) = TV(\Gamma) \subseteq \mathit{dom}(S)$. We also have $S(\Gamma_1) = S(\Gamma), \mathit{local}(x : S(\sigma_1))$. Since $\models_B E : S(\Gamma)$, we obtain $\models_B (E, x : v_1) : S(\Gamma), \mathit{local}(x : S(\sigma_1))$. Applying induction hypothesis for e_2 , we obtain $\models_B v_2 : S(\sigma_2)$ as desired.

Case $E \vdash_B \langle B \rangle \Downarrow B$**.** immediate by definition.

Case $E \vdash_B [e_1, \ldots, e_n] \Downarrow [B_1 \circ \cdots \circ B_n]$**.** This is derived from $E \vdash_B e_i \Downarrow B_i$ for each i. Also suppose $\Gamma \vdash_B [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle$. By the bitmap typing rule, $\Gamma \vdash_B e_i : \langle \sigma_i \rangle$, for each $1 \leq i \leq n$. Applying induction hypothesis for each e_i, we obtain $\models_B B_i : \langle S(\sigma_i) \rangle$. By value typing for bitmap and type substitution rule, we have $\models_{ML} [B_1 \circ \cdots \circ B_n] : S(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle)$ as desired. □

3.2 The Compilation Algorithm

I'm going to define a type-directed compilation algorithm that transforms terms in the source calculus (λ^{ML}) into term in the target calculus (Λ^B). This would be done in a single phase. However, as I have briefly described, some optimization technique may need type information for generating a more efficient code. Thus I designed the compilation algorithm in two phases: compilation of the source expressions into an immediate explicitly typed language, written λ^B, and the transformation from λ^B terms into Λ^B terms. The later phase is easy done by eliminating all type annotations in the given λ^B term.

The syntactic correctness for the whole compilation process can be proved by a combination of syntactic correctnesses of each phase.

3.2.1 The Explicitly Typed Bitmap-passing Calculus – λ^B

The set of terms of λ^B is given by the following syntax:

e	$::=$	c^o	constant
	$\mid$	x	variable
	$\mid$	$\lambda\overline{x:\tau}.e$	function
	$\mid$	$(e\ \overline{e})$	application
	$\mid$	$\Lambda\overline{t}.\lambda\overline{x:\langle t\rangle}.e$	type abstraction and bit tag abstraction
	$\mid$	$(e\ \overline{\tau}\ \overline{e})$	type instantiation and bit tag application
	$\mid$	$(e; e, \ldots, e)$	record
	$\mid$	$\pi_i(e)$	projection
	$\mid$	$\texttt{let}\ x:\sigma = e\ \texttt{in}\ e\ \texttt{end}$	let binding
	$\mid$	$[e, \ldots, e]$	bitmap
	$\mid$	$\langle B\rangle$	bit tag constant ($B \in \{0,1\}$)

In this calculus, we distinguish between two kinds of function: ordinary function (represented by $\lambda\overline{x:\tau}.e$) and bit tag function (represented by $\Lambda\overline{t}.\lambda\overline{x:\langle t\rangle}.e$). A bit tag function $\Lambda\overline{t}.\lambda\overline{x:\langle t\rangle}.e'$ can be obtained from the compilation result of a type abstraction $\Lambda\overline{t}.e$ in λ^{ML} where $\overline{x}$ are the inserted bit tag parameters corresponding to $\overline{t}$.

We also distinguish two kinds of application: ordinary lambda application ($(e_1\ \overline{e_2})$) and bit tag application ($e\ \overline{\tau}\ \overline{e_B}$). A bit tag application can be generated from a type instantiation ($e\ \overline{\tau}$) in λ^{ML} where $\overline{e_B}$ are actual bit tag parameters generated from $\overline{\tau}$.

In order to etablish the type system for λ^B, We define the typing context, type well-formedness and context well-formedness as the same as in Λ^B. We define the typing judgment of the form $\Gamma \vdash_{TB} e : \sigma$ for λ^B. The set of rules to derive this judgment is given in Figure 3.5.

The typing rule for bit tag function is just a combination of typing rules for type abstraction and lambda abstraction in λ^{ML}. The typing rule for bit tag application is just a combination of typing rules for type instantiation and lambda application in λ^{ML}. Other cases are easily understandable.

3.2.2 Compilation from λ^{ML} to λ^B

Now, I show the compilation algorithm from λ^{ML} terms into λ^B terms. The algorithm is formulated by judgment of the forms $\Gamma \vdash_{TB} e \leadsto e'$ where Γ is a context of the target

$$\Gamma \vdash_{TB} c^o : o$$

$$\Gamma \vdash_{TB} x : \sigma \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TB} e : \sigma}{\Gamma \vdash_{TB} \lambda \overline{x : \tau}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_{TB} e : \sigma}{\Gamma \vdash_{TB} \Lambda \overline{t}.\lambda \overline{b : \langle t \rangle}.e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma}$$

$$\frac{\Gamma \vdash_{TB} e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_{TB} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{TB} (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{\Gamma \vdash_{TB} e : \forall \overline{t}.\overline{\langle t \rangle} \to \sigma \quad \Gamma \vdash_{TB} \overline{e_b} : \overline{\langle \tau \rangle} \quad \Gamma \vdash_{B} \overline{\tau}}{\Gamma \vdash_{TB} (e\ \overline{\tau}\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_{TB} e_i : \sigma_i\ (1 \leq i \leq n) \quad \Gamma \vdash_{TB} e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_{TB} (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_{TB} e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_{TB} \pi_i(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_{TB} e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_{TB} e_2 : \sigma_2}{\Gamma \vdash_{TB} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

$$\frac{\texttt{tagOf}(\sigma) = B}{\Gamma \vdash_{TB} \langle B \rangle : \langle \sigma \rangle}$$

$$\frac{\Gamma \vdash_{TB} e_i : \langle \sigma_i \rangle\ (1 \leq i \leq n)}{\Gamma \vdash_{TB} [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

Figure 3.5: Typing Rules of λ^B

calculus, e is a source expression and e' is a target expression. The set of compilation rules is given in Figure 3.6.

In the compilation rule for type abstraction $\Lambda\overline{t}.e$, the algorithm introduces new bit tag parameters $\overline{b : \langle t \rangle}$ corresponding to type variables $\overline{t}$. In the rule for type instantiation $(e\ \overline{\tau})$, actual bit tag parameters $\overline{e_b}$ are generated and inserted after type instantations. Each of $\overline{e_b}$ is generated by inspecting the corresponding instance type in $\overline{\tau}$. In the compilation rule for records, the bitmap $[e^b_1, \ldots, e^b_n]$ is generated by composing bit tags $e^b_1, \ldots, e^b_n$. Each e^b_i is also generated by inspecting the corresponding field type σ_i.

We formalize the creation of $\overline{e_b}$ in type instantiation compilation rule and e^b_i in record compilation rule by introducing *bit tag creation algorithm* of the form $\Gamma \vdash_{TB} \sigma \leadsto e$ as follows.

$\Gamma \vdash_{TB} \sigma \leadsto \langle \texttt{tagOf}(\sigma) \rangle$	if σ has a proper outermost type constructor
$\Gamma \vdash_{TB} t \leadsto b$	where $b : \langle t \rangle \in \Gamma$

This algorithm generate a bit tag term corresponding to a given type σ. In the first rule, σ has a proper outermost type constructor. Therefore bit tag of objects of this type is statically determined by $\langle \texttt{tagOf}(\sigma) \rangle$. If σ does not have proper outermost type

$$\Gamma \vdash_{TB} c^o \rightsquigarrow c^o$$

$$\Gamma \vdash_{TB} x \rightsquigarrow x \quad \text{If } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TB} e \rightsquigarrow e'}{\Gamma \vdash_{TB} \lambda\overline{x : \tau}.e \rightsquigarrow \lambda\overline{x : \tau}.e'}$$

$$\frac{\Gamma \vdash_{TB} e_1 \rightsquigarrow e'_1 \quad \Gamma \vdash_{TB} \overline{e_2} \rightsquigarrow \overline{e'_2}}{\Gamma \vdash_{TB} (e_1\ \overline{e_2}) \rightsquigarrow (e'_1\ \overline{e'_2})}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_{TB} e \rightsquigarrow e' \quad \overline{b} \text{ are fresh variables}}{\Gamma \vdash_{TB} \Lambda\overline{t}.e \rightsquigarrow \Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.e'}$$

$$\frac{\Gamma \vdash_{TB} e \rightsquigarrow e' \quad \Gamma \vdash_{TB} \overline{\tau} \rightsquigarrow \overline{e_b}}{\Gamma \vdash_{TB} (e\ \overline{\tau}) \rightsquigarrow (e'\ \overline{\tau}\ \overline{e_b})}$$

$$\frac{\Gamma \vdash_{TB} e_i \rightsquigarrow e'_i \quad \Gamma \vdash_{TB} e'_i : \sigma_i \quad \Gamma \vdash_{TB} \sigma_i \rightsquigarrow e^b_i \quad (1 \leq i \leq n)}{\Gamma \vdash_{TB} (e_1, \ldots, e_n) \rightsquigarrow ([e^b_1, \ldots, e^b_n]; e'_1, \ldots, e'_n)}$$

$$\frac{\Gamma \vdash_{TB} e \rightsquigarrow e'}{\Gamma \vdash_{TB} \pi_i(e) \rightsquigarrow \pi_i(e')}$$

$$\frac{\Gamma \vdash_{TB} e_1 \rightsquigarrow e'_1 \quad \Gamma \vdash_{TB} e_1 : \sigma' \quad \Gamma, local(x : \sigma') \vdash_{TB} e_2 \rightsquigarrow e'_2}{\Gamma \vdash_{TB} \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end} \rightsquigarrow \texttt{let}\ x : \sigma' = e'_1\ \texttt{in}\ e'_2\ \texttt{end}}$$

Figure 3.6: Bitmap-passing Compilation

constructor, i.e. σ is a type variable t, the algorithm will looking for a bit tag variable of type $\langle t \rangle$ recorded in the context. If this search returns no result, the bit tag creation algorithm will fail. To ensure that the bit tag creation algorithm never fails, we assume that the compile context Γ is constructed in such a way where any $tvar(\overline{t})$ assumption is followed by a $arg(\overline{b : \langle t \rangle})$ assumption (this is guaranteed by the simulation relations defined later). The following proposition demonstrates this feature.

Lemma 3.6 *Let Γ be an well-formed context and each $tvar(\overline{t})$ assumption of Γ is followed by a $arg(\overline{b : \langle t \rangle})$ assumption, σ be a type. If $\Gamma \vdash_B \sigma$ then $\Gamma \vdash_{TB} \sigma \rightsquigarrow b$ always succeeds and $\Gamma \vdash_{TB} b : \langle \sigma \rangle$.*

PROOF. There are two cases.

Case 1. σ has a proper outermost type constructor, i.e $\sigma \neq t$. In this case the bit tag creation succeeds with a constant bit tag $B = \texttt{tagOf}(\sigma)$. Obviously, $\Gamma \vdash_{TB} \langle B \rangle : \langle \sigma \rangle$.

Case 2. σ is a type variable, i.e. $\sigma = t_i$. By the assumption, $\Gamma \vdash_B t_i$, Γ must contain a $tvar(\overline{t})$ assumption where $t_i \in \overline{t}$. Also by the assumption, there must be a $arg(\overline{b : \langle t \rangle})$ followed $tvar(\overline{t})$. Then the algorithm succeeds with the variable b_i corresponding to t_i, and we have $\Gamma \vdash_{TB} b_i : \langle t_i \rangle$ as desired. □

The bitmap compilation algorithm only inserts bitmap abstractions and bitmap applications in target types and terms. Erasing all bitmap abstractions from the type of a target expression should therefore recover the original type of the source expression. To

$$\tau \sim_B \tau$$

$$\frac{\sigma \sim_B \sigma'}{\overline{\tau} \to \sigma \sim_B \overline{\tau} \to \sigma'}$$

$$\frac{\sigma \sim_B \sigma'}{\forall \overline{t}.\sigma \sim_B \forall \overline{t}.\overline{\langle t \rangle} \to \sigma'}$$

$$\frac{\sigma_i \sim_B \sigma'_i \text{ for each } 1 \leq i \leq n}{\sigma_1 \times \cdots \times \sigma_n \sim_B \sigma'_1 \times \cdots \times \sigma'_n}$$

$$\emptyset \sim_B \emptyset$$

$$\frac{\Gamma \sim_B \Gamma' \quad x \notin dom(\Gamma') \quad \sigma \sim_B \sigma'}{(\Gamma, local(x : \sigma)) \sim_B (\Gamma', local(x : \sigma'))}$$

$$\frac{\Gamma \sim_B \Gamma' \quad \overline{x} \cap dom(\Gamma') = \emptyset}{(\Gamma, arg(\overline{x : \tau})) \sim_B (\Gamma', arg(\overline{x : \tau}))}$$

$$\frac{\Gamma \sim_B \Gamma' \quad \overline{b} \cap dom(\Gamma') = \emptyset}{(\Gamma, tvar(\overline{t})) \sim_B (\Gamma', tvar(\overline{t}), arg(\overline{b : \langle t \rangle}))}$$

Figure 3.7: Simulation Relations on Types and Contexts

formalize this property, we define simulations relations $\sigma \sim_B \sigma'$ between source and target types, and $\Gamma \sim_B \Gamma'$ between source and target contexts in Figure 3.7. The simulation relations satisfy following properties.

Lemma 3.7 *Let Γ be a well-formed context in λ^{ML} and Γ' be a context in λ^B. If $\Gamma \sim_B \Gamma'$, then Γ' is well-formed and any $tvar(\overline{t})$ assumption in Γ' must have an $arg(\overline{b : \langle t \rangle})$ assumption follows.*

PROOF. Straightforward by structural induction on Γ. □

Lemma 3.8 *Let Γ be a well-formed context in λ^{ML}, Γ' be a context in λ^B, σ be a type in λ^{ML}, and σ' be a type in λ^B. If $\Gamma \sim_B \Gamma'$, $\sigma \sim_B \sigma'$ and $\Gamma \vdash_{ML} \sigma$ then $\Gamma' \vdash_B \sigma'$.*

PROOF. Straightforward. □

Lemma 3.9 *If $\Gamma \sim_B \Gamma'$ then $\Gamma(x) \sim_B \Gamma'(x)$ for all $x \in dom(\Gamma)$*

PROOF. Straightforward by structural induction on Γ.

□

Lemma 3.10 *The simulation relations on types are stable under monomorphic substitution. If $\sigma \sim_B \sigma'$ then for any monomophic substitution $S = [\overline{\tau}/\overline{t}]$, $S(\sigma) \sim_B S(\sigma')$.*

PROOF. This is proved by induction on the derivation of $\sigma \sim_B \sigma'$. We do case analysis on σ.

Case $\sigma = o$**.** Straightforward.

Case $\sigma = t$**.** From the simulation relation rule, $\sigma' = \sigma = t$. Therefore $S(\sigma) = S(\sigma')$ is a monomorphic type. Then we have $S(\sigma) \sim_B S(\sigma')$ as desired.

Case $\sigma = \tau \to \sigma_1$**.** Suppose $\tau \to \sigma_1 \sim_B \tau \to \sigma_1'$. This is derived from $\sigma_1 \sim_B \sigma_1'$. Applying induction hypothesis for $\sigma_1 \sim_B \sigma_1'$, $S(\sigma_1) \sim S(\sigma_1')$. In addition, $S(\tau) \sim_B S(\tau)$, therefore $S(\tau) \to S(\sigma_1) \sim_B S(\tau) \to S(\sigma_1')$. This implies $S(\tau \to \sigma_1) \sim_B S(\tau \to \sigma_1')$.

Case $\sigma = \sigma_1 \times \cdots \times \sigma_n$**.** Suppose $\sigma_1 \times \cdots \times \sigma_n \sim_B \sigma_1' \times \cdots \times \sigma_n'$. This is derived from $\sigma_i \sim_B \sigma_i'$, for all $1 \leq i \leq n$. Applying induction hypothesis for each σ_i, we have $S(\sigma_i) \sim_B S(\sigma_i')$. Therefore, $S(\sigma_1) \times \cdots \times S(\sigma_n) \sim_B S(\sigma_1') \times \cdots \times S(\sigma_n')$. By the definition of type substitution, we have $S(\sigma_1 \times \cdots \times \sigma_n) \sim_B S(\sigma_1' \times \cdots \times \sigma_n')$ as desired.

Case $\sigma = \forall \overline{t}.\sigma_1$**.** By bound type variable convention, suppose $\overline{t} \cap dom(S) = \emptyset$. Also suppose $\forall \overline{t}.\sigma_1 \sim_B \forall \overline{t}.\overline{\langle t \rangle} \to \sigma_1'$. This is derived from $\sigma_1 \sim_B \sigma_1'$. Applying induction hypothesis for σ_1, we obtain $S(\sigma_1) \sim_B S(\sigma_1')$. Therefore, $\forall \overline{t}.S(\sigma_1) \sim_B \forall \overline{t}.\overline{\langle t \rangle} \to S(\sigma_1')$. Since $\overline{t} \cap dom(S) = \emptyset$, by the definition of type substitution, we have $S(\forall \overline{t}.\sigma_1) \sim_B S(\forall \overline{t}.\overline{\langle t \rangle} \to \sigma_1')$ as desired. □

Now we show the type preservation property of the compilation algorithm by the following theorem:

Theorem 3.2 *Suppose* $\Gamma \vdash_{ML} e : \sigma$. *For any* Γ' *so that* $\Gamma \sim_B \Gamma'$ *the compilation algorithm succeeds as* $\Gamma' \vdash_{TB} e \rightsquigarrow e'$ *with* $\Gamma' \vdash_{TB} e' : \sigma'$ *where* $\sigma \sim_B \sigma'$

PROOF. This is proved by induction on derivation of the typing rule. We proceed by do case analysis on e.
Case $e = c^o$**.** Straighforward.

Case $e = x$**.** Suppose $\Gamma \vdash_{ML} x : \sigma$, then by the λ^{ML} typing rule for variable, we have $\Gamma(x) = \sigma$. Since $\Gamma \sim_B \Gamma'$, by Lemma 3.9, $\Gamma(x) \sim_B \Gamma'(x)$. Then the compilation succeeds as $\Gamma' \vdash_{TB} x \rightsquigarrow x$. By the λ^B typing rule for variable, we have $\Gamma' \vdash_{TB} x : \Gamma'(x)$. Since $\Gamma(x) \sim_B \Gamma'(x)$, we get the desired result.

Case $e = \lambda \overline{x : \tau}.e_1$**.** Suppose $\Gamma \vdash_{ML} \lambda \overline{x : \tau}.e_1 : \overline{\tau} \to \sigma_1$. This is derived from $\Gamma, arg(\overline{x : \tau}) \vdash_{ML} e_1 : \sigma_1$. By the context simulation relation, we have $(\Gamma, arg(\overline{x : \tau})) \sim_B (\Gamma', arg(\overline{x : \tau}))$. Applying induction hypothesis for e_1, we have $\Gamma', arg(\overline{x : \tau}) \vdash_{TB} e_1 \rightsquigarrow e_1'$ succeeds with $\Gamma', arg(\overline{x : \tau}) \vdash_{TB} e_1' : \sigma_1'$ where $\sigma_1 \sim_B \sigma_1'$. Applying the compilation algorithm for function, we have $\Gamma' \vdash_{TB} \lambda \overline{x : \tau}.e_1 \rightsquigarrow \lambda \overline{x : \tau}.e_1'$ succeeds. By the λ^B typing rule for function, we have $\Gamma' \vdash_{TB} \lambda \overline{x : \tau}..e_1' : \overline{\tau} \to \sigma'$. Since $\sigma_1 \sim_B \sigma_1'$, by the simulation relation for type, we have $(\overline{\tau} \to \sigma_1) \sim_B (\overline{\tau} \to \sigma_1')$ as desired.

Case $e = (e_1\ \overline{e_2})$**.** Suppose $\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \overline{\tau} \to \sigma \qquad \Gamma \vdash_{ML} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma}$$

Applying induction hypothesis for e_1 and $\overline{e_2}$, we have

- $\Gamma' \vdash_{TB} e_1 \rightsquigarrow e_1'$ succeeds with $\Gamma' \vdash_{TB} e_1' : \sigma_1$ where $(\overline{\tau} \to \sigma) \sim_B \sigma_1$
- $\Gamma' \vdash_{TB} \overline{e_2} \rightsquigarrow \overline{e_2'}$ succeeds with $\Gamma' \vdash_{TB} \overline{e_2'} : \overline{\sigma_2}$ where $\overline{\tau} \sim_B \overline{\sigma_2}$.

By simulation relations on types, we have $\sigma_1 \equiv \overline{\tau} \to \sigma'$ where $\sigma \sim_B \sigma'$, and $\overline{\sigma_2} \equiv \overline{\tau}$. Applying the compilation rule for lambda application, we have $\Gamma' \vdash_{TB} (e_1\ \overline{e_2}) \rightsquigarrow (e_1'\ \overline{e_2'})$ succeeded. Applying λ^B typing rule for application, we obtain $\Gamma' \vdash_{TB} (e_1'\ \overline{e_2'}) : \sigma'$ where $\sigma \sim_B \sigma'$ as desired.

Case $e = \Lambda\overline{t}.e_1$**.** Suppose $\Gamma \vdash_{ML} \Lambda\overline{t}.e_1 : \forall\overline{t}.\sigma_1$ This is derived from $\Gamma, tvar(\overline{t}) \vdash_{ML} e_1 : \sigma_1$. Let $\overline{b}$ be a sequence of fresh variables, $\overline{b} \cap dom(\Gamma) = \emptyset$. By the context simulation relation, we have $(\Gamma, tvar(\overline{t})) \sim_B (\Gamma', tvar(\overline{t}), arg(\overline{b : \langle t\rangle}))$. Applying induction hypothesis for e_1, we have $\Gamma', tvar(\overline{t}), arg(\overline{b : \langle t\rangle}) \vdash_{TB} e_1 \rightsquigarrow e_1'$ succeeds with $\Gamma', tvar(\overline{t}), arg(\overline{b : \langle t\rangle}) \vdash_{TB} e_1 : \sigma_1'$ where $\sigma_1 \sim_B \sigma_1'$. Applying the compilation algorithm, we have

$$\Gamma' \vdash_{TB} \Lambda\overline{t}.e_1 \rightsquigarrow \Lambda\overline{t}.\lambda\overline{b : \langle t\rangle}.e_1'.$$

By the target typing rule for type abstraction, we have

$$\Gamma' \vdash_{TB} \Lambda\overline{t}.be_1' : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma_1'$$

Since $\sigma_1 \sim_B \sigma_1'$, by the simulation relation on type, we have $\forall\overline{t}.\sigma_1 \sim_B \forall\overline{t}.\overline{\langle t\rangle} \to \sigma_1'$ as desired.

Case $e = \Lambda\overline{e_1}.\tau$**.** Suppose $\Gamma \vdash_{ML} \Lambda\overline{e_1}.\tau : \sigma_1[\overline{\tau}/\overline{t}]$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \forall\overline{t}.\sigma_1 \qquad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_{ML} \Lambda\overline{e_1}.\tau : \sigma_1[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for e_1, we have $\Gamma' \vdash_{TB} e_1 \rightsquigarrow e_1'$ with $\Gamma' \vdash_{TB} e_1' : \sigma_2$ where $\forall\overline{t}.\sigma_1 \sim_B \sigma_2$. By the simulation relation for types, σ_2 must have form $\forall\overline{t}.\overline{\langle t\rangle} \to \sigma_1'$ where $\sigma_1 \sim_B \sigma_1'$. Since $\Gamma \vdash_{ML} \overline{\tau}$, by Lemma 3.8, we have $\Gamma' \vdash_B \overline{\tau}$. By Lemma 3.7 and Lemma 3.6, $\Gamma' \vdash_{TB} \overline{\tau} \rightsquigarrow \overline{e_b}$ must succeeds and $\Gamma' \vdash_{TB} \overline{e_b} : \overline{\langle\tau\rangle}$. Applying the compilation rule for type instantiation, we have $\Gamma' \vdash_{TB} \Lambda\overline{e_1}.\tau \rightsquigarrow \Lambda\overline{e_1'}.\lambda\overline{\tau : \langle e_1'\rangle}.e_b$. Applying the λ^B typing rule for type instantiation, we have $\Gamma' \vdash_{TB} \Lambda\overline{e_1'}.\lambda\overline{\tau : \langle e_1'\rangle}.e_b : \sigma_1'[\overline{\tau}/\overline{t}]$. By Lemma 3.10, $\sigma_1[\overline{\tau}/\overline{t}] \sim_B \sigma_1'[\overline{\tau}/\overline{t}]$ as desired.

Case $e = (e_1, \ldots, e_n)$**.** Suppose $\Gamma \vdash_{ML} (e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n$. This is derived from $\Gamma \vdash_{ML} e_i : \sigma_i$, for all $1 \le i \le n$. Applying induction hypothesis for each e_i, we have $\Gamma \vdash_{TB} e_i \rightsquigarrow e_i'$ succeeds with $\Gamma' \vdash_{TB} e_i' : \sigma_i'$ where $\sigma_i \sim_B \sigma_i'$. Since σ_i is well-formed under Γ, by Lemma 3.8, we have $\Gamma' \vdash_B \sigma_i'$. By Lemma 3.7 and Lemma 3.6, $\Gamma' \vdash_{TB} \sigma_i' \rightsquigarrow e_i^b$ must succeeds and $\Gamma' \vdash_{TB} e_i^b : \langle\sigma_i'\rangle$. Applying the compilation rule for record, we obtain $\Gamma' \vdash_{TB} (e_1, \ldots, e_n) \rightsquigarrow ([e_1^b, \ldots, e_n^b]; e_1', \ldots, e_n')$. Applying the λ^B typing rule for bitmap, we have $\Gamma' \vdash_{TB} [e_1^b, \ldots, e_n^b] : \langle\langle\sigma_1', \ldots, \sigma_n'\rangle\rangle$. Applying the λ^B typing rule for record, we have $\Gamma' \vdash_{TB} ([e_1^b, \ldots, e_n^b]; e_1', \ldots, e_n') : \sigma_1' \times \cdots \times \sigma_n'$. By the simulation relation on types, we have $(\sigma_1 \times \cdots \times \sigma_n) \sim_B (\sigma_1' \times \cdots \times \sigma_n')$ as desired.

Case $e = \pi_i(e_1)$**.** Suppose $\Gamma \vdash_{ML} \pi_i(e_1) : \sigma_i$. This is derived from $\Gamma \vdash_{ML} e_1 : \sigma_1 \times \cdots \times \sigma_n$. Applying the induction hypothesis for e_1, we have $\Gamma' \vdash_{TB} e_1 \rightsquigarrow e_1'$ with $\Gamma' \vdash_{TB} e_1' : \sigma'$ where $(\sigma_1 \times \cdots \times \sigma_n) \sim_B \sigma'$. Then σ' must have the form $\sigma_1' \times \cdots \times \sigma_n'$ where $\sigma_i \sim_B \sigma_i'$ for each i. Applying the compilation rule for projection, we have $\Gamma' \vdash_{TB} \pi_i(e_1) \rightsquigarrow \pi_i(e_1')$. Applying the λ^B typing rule for projection, we obtain $\Gamma' \vdash_{TB} \pi_i(e_i') : \sigma_i'$ where $\sigma_i \sim_B \sigma_i'$ as desired.

Case $e = \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end}$**.** Suppose $\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \sigma_1 \;\; \Gamma, local(x : \sigma_1) \vdash_{ML} e_2 : \sigma_2}{\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for e_1, we have $\Gamma' \vdash_{TB} e_1 \leadsto e_1'$ succeeds with $\Gamma' \vdash_{TB} e_1' : \sigma_1'$ where $\sigma_1 \sim_B \sigma_1'$ By the simulation relation for context, we have

$$(\Gamma, local(x : \sigma_1)) \sim_B (\Gamma', local(x : \sigma_1')).$$

Applying induction hypothesis for e_1, we obtain $\Gamma', local(x : \sigma_1') \vdash_{TB} e_2 \leadsto e_2'$ succeeds with $\Gamma', local(x : \sigma_1') \vdash_{TB} e_2' : \sigma_2'$ where $\sigma_2 \sim_B \sigma_2'$. By the compilation algorithm, we have

$$\Gamma' \vdash_{TB} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} \leadsto \texttt{let}\ x : \sigma_1' = e_1'\ \texttt{in}\ e_2'\ \texttt{end}.$$

Finally, by the λ^B typing rule for `let` expression, we have $\Gamma' \vdash_{TB} \texttt{let}\ x : \sigma_1' = e_1'\ \texttt{in}\ e_2'\ \texttt{end} : \sigma_2'$ where $\sigma_2 \sim_B \sigma_2'$ as desired. □

3.2.3 Transformation from λ^B terms to Λ^B terms

After obtaining the explicitly typed term in λ^B by the above compilation algorithm, we can easily get the target term in Λ^B by using the $erase_B(e)$ function which erases all type annotation in the given term e in λ^B. The $erase_B$ function can be inductively defined as follows

$$\begin{array}{rcl}
erase_B(c^0) & = & c^o \\
erase_B(x) & = & x \\
erase_B(\lambda \overline{x : \tau}.e) & = & \lambda \overline{x}.erase_B(e) \\
erase_B((e_1\ \overline{e_2})) & = & (erase_B(e_1)\ \overline{erase_B(e_2)}) \\
erase_B(\Lambda \overline{t}.\lambda \overline{x : \langle t \rangle}.e) & = & \lambda \overline{x}.erase_B(e) \\
erase_B((e\ \overline{\tau}\ \overline{e_b})) & = & (erase_B(e)\ \overline{erase_B(e_b)}) \\
erase_B((e_{bm}; e_1, \ldots, e_n)) & = & (erase_B(e_{bm}); erase_B(e_1), \ldots, erase_B(e_n)) \\
erase_B(\pi_i(e)) & = & \pi_i(erase_B(e)) \\
erase_B(\texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let}\ x = erase_B(e_1)\ \texttt{in}\ erase_B(e_2)\ \texttt{end} \\
erase_B([e_1, \ldots, e_n]) & = & [erase_B(e_1), \ldots, erase_B(e_n)] \\
erase_B(\langle B \rangle) & = & \langle B \rangle
\end{array}$$

The function $erase_B$ just simply eliminates type annotation in the given term. Typing derivation should be the same in the given term and the result. This property is shown by the following theorem

Theorem 3.3 *Suppose Γ, e, σ is a well-formed context, a term and a well-formed type in λ^B so that $\Gamma \vdash_{TB} e : \sigma$. Then we also have $\Gamma \vdash_B erase_B(e) : \sigma$.*

PROOF. This is proved by induction on derivation of the typing rule $\Gamma \vdash_{TB} e : \sigma$. We proceed by case analysis on e.

Case $e = c^o$**,** $e = x$**,** $e = \langle B \rangle$**.** Straightforward.

Case $e = \lambda \overline{x : \tau}.e'$**.** Suppose that we have the following type derivation

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TB} e' : \sigma}{\Gamma \vdash_{TB} \lambda\overline{x : \tau}.e' : \overline{\tau} \to \sigma}$$

Applying induction hypothesis for e', we obtain $\Gamma, arg(\overline{x : \tau}) \vdash_B erase_B(e') : \sigma$. Applying the first Λ^B typing rule for function, we have $\Gamma \vdash_B \lambda\overline{x}.erase_B(e') : \overline{\tau} \to \sigma$. Since $erase_B(\lambda\overline{x : \tau}.e') = \lambda\overline{x}.erase_B(e')$, we have $\Gamma \vdash_B erase_B(\lambda\overline{x : \tau}.e') : \overline{\tau} \to \sigma$ as desired.

Case $e = (e_1\ \overline{e_2})$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e_1 : \overline{\tau} \to \sigma \qquad \Gamma \vdash_{TB} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{TB} (e_1\ \overline{e_2}) : \sigma}$$

Applying induction hypothesis for e_1 and $\overline{e_2}$, we have

$$\Gamma \vdash_B erase_B(e_1) : \overline{\tau} \to \sigma$$
$$\Gamma \vdash_B \overline{erase_B(e_2)} : \overline{\tau}$$

By the first typing rule for application in Λ^B, we obtain $\Gamma \vdash_B (erase_B(e_1)\ \overline{erase_B(e_2)}) : \sigma$. Since $erase_B((e_1\ \overline{e_2})) = (erase_B(e_1)\ \overline{erase_B(e_2)})$, then we have $\Gamma \vdash_B erase_B((e_1\ \overline{e_2})) : \sigma$ as desired.

Case $e = \Lambda\overline{t}.\lambda\overline{x : \langle t\rangle}.e'$**.** Suppose

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{x : \langle t\rangle}) \vdash_{TB} e' : \sigma}{\Gamma \vdash_{TB} \Lambda\overline{t}.\lambda\overline{x : \langle t\rangle}.e' : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma}$$

Applying induction hypothesis for e' we have $\Gamma, tvar(\overline{t}), arg(\overline{x : \langle t\rangle}) \vdash_B erase_B(e') : \sigma$. By the second typing rule for function in Λ^B, we have $\Gamma \vdash_B \lambda\overline{x}.erase_B(e') : \sigma$. Since $erase_B(\Lambda\overline{t}.\lambda\overline{x : \langle t\rangle}.e') = \lambda\overline{x}.erase_B(e')$, then we have $\Gamma \vdash_B erase_B(\Lambda\overline{t}.\lambda\overline{x : \langle t\rangle}.e') : \sigma$ as desired.

Case $e = (e'\ \overline{\tau}\ \overline{e_b})$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e' : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma \qquad \Gamma \vdash_{TB} \overline{e_b} : \overline{\langle \tau\rangle} \qquad \Gamma \vdash_B \overline{\tau}}{\Gamma \vdash_{TB} (e'\ \overline{\tau}\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for e' and $\overline{e_b}$ we obtain

$$\Gamma \vdash_B erase_B(e') : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma$$
$$\Gamma \vdash_B \overline{erase_B(e_b)} : \overline{\langle \tau\rangle}$$

By the second typing rule for application in Λ^B, we have $\Gamma \vdash_B (erase_B(e')\ \overline{erase_B(e_b)}) : \sigma[\overline{\tau}/\overline{t}]$. Since $erase_B((e'\ \overline{\tau}\ \overline{e_b})) = (erase_B(e')\ \overline{erase_B(e_b)})$, the we have $\Gamma \vdash_B erase_B((e'\ \overline{\tau}\ \overline{e_b})) : \sigma[\overline{\tau}/\overline{t}]$ as desired.

Case $e = (e_0; e_1, \ldots, e_n)$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e_i : \sigma_i\ (1 \leq i \leq n) \qquad \Gamma \vdash_{TB} e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_{TB} (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

Applying induction hypothesis for each e_0, we obtain

$$\Gamma \vdash_B erase_B(e_i) : \sigma_i\ (1 \leq i \leq n)$$
$$\Gamma \vdash_B erase_B(e_0) : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle$$

By Λ^B typing rule for record, we have

$$\Gamma \vdash_B (erase_B(e_0); erase_B(e_1), \ldots, erase_B(e_n)) : \sigma_1 \times \cdots \times \sigma_n$$

Since $erase_B((e_0; e_1, \ldots, e_n)) = (erase_B(e_0); erase_B(e_1), \ldots, erase_B(e_n))$, the we have

$$\Gamma \vdash_B erase_B((e_0; e_1, \ldots, e_n)) : \sigma_1 \times \cdots \times \sigma_n$$

as desired.

Case $e = \pi_i(e')$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e' : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_{TB} \pi_i(e') : \sigma_i}$$

Applying induction hypothesis for e', we have $\Gamma \vdash_B erase_B(e') : \sigma_1 \times \cdots \times \sigma_n$. By Λ^B typing rule for projection, we have $\Gamma \vdash_{TB} \pi_i(erase_B(e')) : \sigma_i$. Since $erase_B(\pi_i(e')) = \pi_i(erase_B(e'))$, then we obtain $\Gamma \vdash_{TB} erase_B(\pi_i(e')) : \sigma_i$ as desired.

Case $e = \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end}$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_{TB} e_2 : \sigma_2}{\Gamma \vdash_{TB} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for e_1 and e_2, we obtain

$$\Gamma \vdash_B erase_B(e_1) : \sigma_1$$
$$\Gamma, local(x : \sigma_1) \vdash_B erase_B(e_2) : \sigma_2$$

By the Λ^B typing rule for `let` term, we have

$$\Gamma \vdash_B \texttt{let}\ x = erase_B(e_1)\ \texttt{in}\ erase_B(e_2)\ \texttt{end} : \sigma_2$$

Since $erase_B(\texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end}) = \texttt{let}\ x = erase_B(e_1)\ \texttt{in}\ erase_B(e_2)\ \texttt{end}$, Then we have

$$\Gamma \vdash_B erase_B(\texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$$

as desired.

Case $e = [e_1, \ldots, e_n]$**.** Suppose

$$\frac{\Gamma \vdash_{TB} e_i : \langle\sigma_i\rangle\ (1 \leq i \leq n)}{\Gamma \vdash_{TB} [e_1, \ldots, e_n] : \langle\!\langle\sigma_1, \ldots, \sigma_n\rangle\!\rangle}$$

Applying induction hypothesis for each e_i, we obtain $\Gamma \vdash_B erase_B(e_i) : \langle\sigma_i\rangle$. By Λ^B typing rule for bitmap, we have

$$\Gamma \vdash_B [erase_B(e_1), \ldots, erase_B(e_n)] : \langle\!\langle\sigma_1, \ldots, \sigma_n\rangle\!\rangle.$$

Since $erase_B([e_1, \ldots, e_n]) = [erase_B(e_1), \ldots, erase_B(e_n)]$, then we have

$$\Gamma \vdash_B erase_B([e_1, \ldots, e_n]) : \langle\!\langle\sigma_1, \ldots, \sigma_n\rangle\!\rangle.$$

as desired. □

This result together with the type soundness theorem of the target calculus (Theorem 3.1) and the type preservation theorem (Theorem 3.2) establishes that the type system of the source calculus is sound with respect to the operational semantics realized by λ^{ML} to λ^B to Λ^B compilation followed by evaluation of the compiled term.

Chapter 4

Unboxed Compilation

In previous chapter, we have seen how to generate bitmap information for heap-allocated objects – the first step in my development process for achieving a high-degree of interoperability. In this chapter, I present the second step: generating information for manipulating unboxed values. Presentation strategy in this chapter is as the same as previous one. First, I define a target calculus where the necessary information are formalized. Then, I establish an operational semantics to give the meaning of terms in the calculus, and show that typing system of the calculus is sound with respect to the given operational calculus. Finally, I develop a type-preserved compilation algorithm which translates terms in the source calculus (presented in previous chapter) into terms in the target calculus.

4.1 The Target Calculus – Λ^U

I'm going to define the target calculus, namely *unboxed calculus* or Λ^U, for the unboxed compilation method. In order to identify the necessary information to be formalized in the calculus, let me first describe several preliminaries related to unboxed manipulation.

This compilation method aims to support fully unboxed representation for multi-word atomic values such as floating point numbers. These values may reside in either a heap block or in a run-time environment (e.g. stack frame). In order to manipulate these values, runtime system must know their sizes and exact locations of these values. This task would not be difficult for a monomorphic language where all size information are statically determined. For a polymorphic language, computing sizes and locations is challenge due to the abstraction of types (and therefore the abstraction of sizes). This problem can be solved by following the strategy of bitmap-passing compilation presented in previous chapter: statically computing sizes for layout-fixed objects; introducing a size parameter for each type variable (by inserting a size abstraction/size application at each type abstraction/type instantiation), and passing size information to the necessary computation (re-formulating language constructions with size information).

To determine which language constructions need size information, and how to reformulate them is not so simple. In fact, almost every run-time operations that directly manipulate unboxed values require sizes and locations of the values. In the presence of polymorphism where sizes and locations may not be statically determined, the computation of size and locations is heavy. For the sake of efficiency, as mentioned in the Chapter 1, we assume a simple model of run-time environment: polymorphic variables are allocated with maximum size, i.e. two words in our implementation. This assumption allows the

compiler to generate locations of variables in run-time environment statically. I exclude this computation from the formalism. Under this assumption, only size information are required for manipulating immediate variables (arguments and local variables). For heap objects, both size and location are needed. An alternative representation of location for an unboxed value which resides in a heap block is the offset of the value related to block pointer. From now on, we use the term “offset” instead of “location”.

In the rest of this section, I define the syntax and type system of the target calculus involving size and offset. Then, I define operational semantics that faithfully models the evaluation of expressions under unboxed manipulation. The type system of the calculus is shown to be sound with respect to the operational semantics to ensure that the evaluation of a typable expression in the target calculus never fails.

4.1.1 Types

The sets of monomorphic types (ranged over by τ) and polymorphic types (ranged over by σ) of Λ^U is defined as follows.

$$
\begin{array}{rcll}
\tau & ::= & o & \text{base type} \\
 & \mid & t & \text{type variable} \\
 & \mid & |\sigma| & \text{size type} \\
 & \mid & \|\,\sigma, \ldots, \sigma\,\| & \text{offset type} \\
 & \mid & \overline{\tau} \to \tau & \text{monomorphic function type} \\
 & \mid & \tau \times \cdots \times \tau & \text{monomorphic product type} \\
\sigma & ::= & \tau & \\
 & \mid & \forall \overline{t}.\sigma & \text{second-order type} \\
 & \mid & \overline{\tau} \to \sigma & \text{polymorphic function type} \\
 & \mid & \sigma \times \cdots \times \sigma & \text{polymorphic product type}
\end{array}
$$

$|\sigma|$ denotes a singleton set of size for objects of type σ. For example, if σ is a *single type* (e.g. $int, word, boxed$), $|\sigma|$ denotes $\{1\}$. if σ is a *double type* (e.g. $real$), $|\sigma|$ denotes $\{2\}$. In ML Core language, for any σ type other than type variable, whether σ is a single type or a double type is determined by its outermost type constructor. We define the size value $\texttt{sizeOf}(\sigma)$ of a type σ as follows.

$$
\begin{array}{rcl}
\texttt{sizeOf}(o) & = & 1 \text{ if } o \text{ is a single base type} \\
\texttt{sizeOf}(o) & = & 2 \text{ if } o \text{ is an double base type} \\
\texttt{sizeOf}(\tau \to \sigma) & = & 1 \\
\texttt{sizeOf}(\sigma_1 \times \cdots \times \sigma_n) & = & 1 \\
\texttt{sizeOf}(|\sigma|) & = & 1 \\
\texttt{sizeOf}(t) & = & \text{undetermined} \\
\texttt{sizeOf}(\|\,\sigma_1, \ldots, \sigma_n\,\|) & = & 1 \\
\texttt{sizeOf}(\forall t.\sigma) & = & 1
\end{array}
$$

In the cases for $\texttt{sizeOf}(\overline{\tau} \to \sigma)$ and $\texttt{sizeOf}(\sigma_1 \times \cdots \times \sigma_n)$, the size value is 1 because all these types denote sets of pointer values (to a function’s closure and to a heap-allocated block). In the case for size and offset types, the size is 1 because the actual runtime representation of a size or an offset is an integer number. The case for $\forall \overline{t}.\sigma$ needs more explanation: this type refers to a polymorphic function type since size abstractions are

always inserted at type abstractions. Then we can safely assume $\forall \overline{t}.\sigma$ to be a single type (pointer).

A offset type $\| \sigma_1, \ldots, \sigma_n \|$ denotes a singleton set of offset of fields whose predecessors (with respect to the heap block) have types $\{\sigma_1, \ldots, \sigma_n\}$. Value of an offset can be computed by a summation of all sizes of values of these types.

The well-formedness of types is similarly defined as one in the source language with an extension of the function FTV for size and offset types:

$$
\begin{array}{rcl}
FTV(|\sigma|) & = & FTV(\sigma) \\
FTV(\| \sigma_1, \ldots, \sigma_n \|) & = & \bigcup FTV(\sigma_i)
\end{array}
$$

We also extend the set of substitution rules for bitmap and bit tag types as follows.

$$
\begin{array}{rcl}
S(|\sigma|) & = & |S(\sigma)| \\
S(\| \sigma_1, \ldots, \sigma_n \|) & = & \| S(\sigma_1), \ldots, S(\sigma_n) \|
\end{array}
$$

4.1.2 Syntax

The set of terms of Λ^U is given by the following syntax:

$$
\begin{array}{rcll}
e & ::= & c^o & \text{constant} \\
& | & x^e & \text{variable} \\
& | & \lambda \overline{x}.e & \text{function} \\
& | & (e\ \overline{e^e}) & \text{application} \\
& | & (e^e, \ldots, e^e) & \text{record} \\
& | & \pi^e_e(e) & \text{projection} \\
& | & \texttt{let}\ x^e = e\ \texttt{in}\ e\ \texttt{end} & \text{let binding} \\
& | & [e + \cdots + e] & \text{offset} \\
& | & |B| & \text{constant size } (B \in \{1, 2\})
\end{array}
$$

Λ^U serves as a low-level calculus closed to an abstract machine of ML. Each term constructor in the syntax represents a (or a set of) instructions in the machine. I will explain the meaning of each syntax of terms in an intuitive way.

Constant c^o corresponds to a constant loading instruction, this requires size information of the constant which can statically determined from the base type o.

A variable x^s represents an access instruction to a value in run-time environment corresponding to the variable x. As we mentioned above, we omit the location information of the variable which can be statically computed by the compiler. Only size information is needed. The subscript expression s in the variable term x^s represents size information of this variable.

As the same as in Λ^B, a function construction $\lambda \overline{x}.e$ has dual roles: this represents a resulting term of either a lambda abstraction or a type abstraction in the source calculus (λ^{ML}). The type abstraction $\Lambda \overline{t}.e$ in λ^{ML} will be compiled into $\lambda \overline{s}.e$ in Λ^U where $\overline{s}$ is the size parameters corresponding to $\overline{t}$.

$(e_1\ \overline{e_2{}^{e_s}})$ represents a resulting term of either a lambda application or a type instantiation in the source calculus where e_1 is a function, $\overline{e_2}$ are actual parameters (in the case of type instantiation, they are actual size parameters generated from the instance types), $\overline{e_s}$ are size information of actual parameters. These information are needed for copying the values of actual parameters to the function's stack frame.

A record expression $(e_1^{s_1}, \ldots, e_n^{s_n})$ corresponds to an allocation of heap block with the field values taken from the evaluation of $e_1, \ldots, e_n$. This requires size information $s_1, \ldots, s_n$ of each $e_1, \ldots, e_n$ for computing the total size of the block, and for copying the values of $e_1, \ldots, e_n$ into correct locations in the block.

The most interesting case is the projection of the form $\pi_o^s(e)$. Given a record e, size s and offset o, this term will be transformed into an instruction that extracts an unboxed value of size s resided at the offset o of the block e. o is an offset term of form $[s_1 + \cdots + s_n]$ where $\{s_1, \ldots, s_n\}$ are size expressions. o represents the summation of values of $\{s_1, \ldots, s_n\}$. In a projection term $\pi_o^s(e)$, o represents the total sizes of the predecessors of the selected field.

A `let` expression of the form `let` $x^s = e_1$ `in` e_2 `end` can be evaluated by a sequence of operations in the target abstract machine. Firstly, the value v_1 of e_1 is computed. Secondly, the machine binds this value to the variable x by copying it to the corresponding location of x in the runtime environment. Finally, e_2 is evaluated under the updated runtime environment. Since we assume that the location of x is fixed, the machine only need to know size of x (or size of v_1). We provide this information by introducing s in the syntax of the `let` term.

4.1.3 Typing Environment

Typing environment of Λ^U is defined as the same as in λ^{ML}. We distinguish the well-formednesses of type and context of Λ^U to those of λ^{ML} (and also those of the Λ^B) by judgments of forms $\Gamma \vdash_U \sigma$, $\vdash_U \Gamma$. The former is for well-formedness of types, the latter is for well-formedness of context. The rules to derived these judgments are similarly defined as those in the source calculus.

The following properties still hold in the target calculus.

Lemma 4.1 *Let Γ be a well-formed context, σ be a type, and S be a substitution that respects Γ. If $\Gamma \vdash_U \sigma$ then $S(\Gamma) \vdash_U S(\sigma)$.*

Lemma 4.2 *Let Γ be a well-formed context and S be a substitution that respects Γ. If $\vdash_U \Gamma$ then $\vdash_U S(\Gamma)$.*

Lemma 4.3 *For a ground substitution S and a context Γ, if $TV(\Gamma) \subseteq dom(S)$ then*

1. *for all σ so that $\Gamma \vdash_U \sigma$, then $S(\sigma)$ is a closed type*
2. *$S(\Gamma)$ is a* ground context

4.1.4 Typing Rules

I formalize typing rules for Λ^U as judgments of the form $\Gamma \vdash_U e : \sigma$, read e has type σ under Γ. Figure 4.1 gives the set of rules to derive these judgment.

In the typing rule for variable, we check the consistency between type of the variable and type of the variable size by $\Gamma \vdash_U s : |\sigma|$.

Similar to Λ^B type system, a function term $\lambda\overline{x}.e$ may have two different typing derivations: one for lambda abstraction and other for type abstraction (together with size abstraction).

$$\Gamma \vdash_U c^o : o$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_U s : |\sigma|}{\Gamma \vdash_U x^s : \sigma}$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{x}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{x : |t|}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{x}.e : \forall\overline{t}.\overline{|t|} \to \sigma}$$

$$\frac{\Gamma \vdash_U e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_U \overline{e_2} : \overline{\tau} \quad \Gamma \vdash_U \overline{e_s} : \overline{|\tau|}}{\Gamma \vdash_U (e_1\ \overline{{e_2}^{e_s}}) : \sigma}$$

$$\frac{\Gamma \vdash_U e : \forall\overline{t}.\overline{|t|} \to \sigma \quad \Gamma \vdash_U \overline{e_s} : \overline{|\tau|} \quad \Gamma \vdash_U \overline{\tau}}{\Gamma \vdash_U (e\ \overline{{e_s}^{|1|}}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_U e_i : \sigma_i \quad \Gamma \vdash_U s_i : |\sigma_i| \text{ for } (1 \leq i \leq n)}{\Gamma \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_U e : \sigma_1 \times \cdots \times \sigma_n \quad \Gamma \vdash_U s : |\sigma_i| \quad \Gamma \vdash_U o : \| \sigma_1, \ldots, \sigma_{i-1} \|}{\Gamma \vdash_U \pi_o^s(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_U e_1 : \sigma_1 \quad \Gamma \vdash_U s : |\sigma_1| \quad \Gamma, local(x : \sigma_1) \vdash_U e_2 : \sigma_2}{\Gamma \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

$$\frac{\texttt{sizeOf}(\sigma) = S}{\Gamma \vdash_U |S| : |\sigma|}$$

$$\frac{\Gamma \vdash_U e_i : |\sigma_i|\ (1 \leq i \leq n)}{\Gamma \vdash_U [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|}$$

Figure 4.1: Typing Rules of The Target Calculus

$(e_1\ \overline{{e_2}^{e_s}})$ may also have two different typing derivations: one for lambda application and other for type instantiation. In the case of lambda application, we check the consistency between the type of actual arguments and the type of their sizes by $\Gamma \vdash_U \overline{e_s} : \overline{|\tau|}$. In the case of type instantiation, since actual arguments are always size values whose sizes are always one word, we do not need to perform these checks.

In the case of record, we check the consistency between type of each element and type of its size by $\Gamma \vdash_U s_i : |\sigma_i|$. In the case of projection, the consistency checks are performed for both size and offset of the selected fields ($\Gamma \vdash_U s : |\sigma_i|$ and $\Gamma \vdash_U o : \| \sigma_1, \ldots, \sigma_{i-1} \|$). For `let` expression, the consistency of type of the bound variable's size is performed ($\Gamma \vdash_U s : |\sigma_1|$).

Now I show that the type system of Λ^B is stable under type substitution by the following lemma.

Lemma 4.4 *Let e be an expression, σ be a type, Γ be a well-formed context and S be a substitution that respects Γ. If $\Gamma \vdash_U e : \sigma$ then $S(\Gamma) \vdash_U e : S(\sigma)$.*

PROOF. This is proved by induction on the derivation of the typing rule $\Gamma \vdash_U e : \sigma$.

Case $\Gamma \vdash_U c^o : o$**.** Straightforward.

Case $\Gamma \vdash_U x^s : \sigma$**.** This is derived from

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_U s : |\sigma|}{\Gamma \vdash_U x^s : \sigma}$$

By induction hypothesis for $\Gamma \vdash_U s : |\sigma|$ and the definition of substitution for types, we have $S(\Gamma) \vdash_U s : |S(\sigma)|$. By the definition of type substitution for context, $S(\Gamma)(x) = S(\Gamma(x)) = S(\sigma)$. By typing rule for variable, we obtain $S(\Gamma) \vdash_U x^s : S(\sigma)$ as desired.

Case $\Gamma \vdash_U \lambda\overline{x}.e : \overline{\tau} \to \sigma$**.** Suppose this is derived from

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{x}.e : \overline{\tau} \to \sigma}$$

Since S respects Γ, we have that S also respects $\Gamma, arg(\overline{x : \tau})$. Applying induction hypothesis for $\Gamma, arg(\overline{x : \tau}) \vdash_U e : \sigma$, we obtain

$$S(\Gamma, arg(\overline{x : \tau})) \vdash_U e : S(\sigma)$$

By definition of type substitution for context, we have

$$S(\Gamma, arg(\overline{x : \tau})) = S(\Gamma), arg(\overline{x : S(\tau)})$$

Thus $S(\Gamma), arg(\overline{x : S(\tau)}) \vdash_U e : S(\sigma)$. Applying typing rule for lambda abstraction, we obtain

$$S(\Gamma) \vdash_U \lambda\overline{x}.e : \overline{S(\tau)} \to S(\sigma)$$

By type substitution for type, we have $\overline{S(\tau)} \to S(\sigma) = S(\overline{\tau} \to \sigma)$ as desired.

Case $\Gamma \vdash_U \lambda\overline{s}.e : \forall\overline{t}.\overline{|t|} \to \sigma$**.** By bound type variable convention, we assume that $\overline{t} \cap dom(\Gamma) = \emptyset$ and $\overline{t} \cap dom(S) = \emptyset$. Suppose this typing is derived from

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{s}.e : \forall\overline{t}.\overline{|t|} \to \sigma}$$

Since $\overline{t} \cap dom(S) = \emptyset$ and S respects Γ, we also have that S respects $\Gamma, tvar(\overline{t}), arg(\overline{s : |t|})$. Applying induction hypothesis for $\Gamma, tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_U e : \sigma$, we obtain

$$S(\Gamma, tvar(\overline{t}), arg(\overline{s : |t|})) \vdash_U e : S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then

$$S(\Gamma, tvar(\overline{t}), arg(\overline{s : |t|})) = S(\Gamma), tvar(\overline{t}), arg(\overline{s : |t|})$$

By typing rule for type abstraction, we have

$$S(\Gamma) \vdash_U \lambda\overline{s}.e : \forall\overline{t}.\overline{|t|} \to S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, we have $\forall\overline{t}.\overline{|t|} \to S(\sigma) = S(\forall\overline{t}.\overline{|t|} \to \sigma)$ as desired.

Case $\Gamma \vdash_U (e_1\ \overline{e_2{}^{e_s}}) : \sigma$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e_1 : \overline{\tau} \rightarrow \sigma \quad \Gamma \vdash_U \overline{e_2} : \overline{\tau} \quad \Gamma \vdash_U \overline{e_s} : \overline{|\tau|}}{\Gamma \vdash_U (e_1\ \overline{e_2{}^{e_s}}) : \sigma}$$

Applying induction hypothesis for $\Gamma \vdash_B e_1 : \overline{\tau} \rightarrow \sigma$, $\Gamma \vdash_B \overline{e_2} : \overline{\tau}$, and $\Gamma \vdash_U \overline{e_s} : \overline{|\tau|}$, we obtain

$$\begin{array}{c} S(\Gamma) \vdash_U e_1 : S(\overline{\tau} \rightarrow \sigma) \\ S(\Gamma) \vdash_U \overline{e_2} : \overline{S(\tau)} \\ S(\Gamma) \vdash_U \overline{e_s} : \overline{S(|\tau|)} \end{array}$$

Since $S(\overline{\tau} \rightarrow \sigma) = \overline{S(\tau)} \rightarrow S(\sigma)$ and $\overline{S(|\tau|)} = \overline{|S(\tau)|}$, applying typing rule for lambda application, we obtain

$$S(\Gamma) \vdash_U (e_1\ \overline{e_2{}^{e_s}}) : S(\sigma)$$

as desired.

Case $\Gamma \vdash_U (e\ \overline{e_s{}^{|1|}}) : \sigma[\overline{\tau}/\overline{t}]$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e : \forall \overline{t}.\overline{|t|} \rightarrow \sigma \quad \Gamma \vdash_U \overline{e_s} : \overline{|\tau|} \quad \Gamma \vdash_U \overline{\tau}}{\Gamma \vdash_U (e\ \overline{e_s{}^{|1|}}) : \sigma[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for $\Gamma \vdash_U e : \forall \overline{t}.\overline{|t|} \rightarrow \sigma$ and $\Gamma \vdash_U \overline{e_s} : \overline{|\tau|}$, we obtain

$$\begin{array}{c} S(\Gamma) \vdash_U e : S(\forall \overline{t}.\overline{|t|} \rightarrow \sigma) \\ S(\Gamma) \vdash_U \overline{e_s} : \overline{S(|\tau|)} \end{array}$$

By bound type variable convention, assume that $\overline{t} \cap dom(S) = \emptyset$. Thus $S(\forall \overline{t}.\overline{|t|} \rightarrow \sigma) = \forall \overline{t}.\overline{|t|} \rightarrow S(\sigma)$. By type substitution for type, $\overline{S(|\tau|)} = \overline{|S(\tau)|}$. Applying Lemma 4.1, we also have $S(\Gamma) \vdash_U \overline{\tau}$. By typing rule for type instantiation, we obtain

$$S(\Gamma) \vdash_U (e\ \overline{e_b{}^{|1|}}) : S(\sigma)[\overline{S(\tau)}/\overline{t}].$$

Since $\overline{t} \cap dom(S) = \emptyset$, we have $S(\sigma)[\overline{S(\tau)}/\overline{t}] = S(\sigma[\overline{\tau}/\overline{t}])$ as desired

Case $\Gamma \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e_i : \sigma_i \quad \Gamma \vdash_U s_i : |\sigma_i| \text{ for } (1 \leq i \leq n)}{\Gamma \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

Applying induction hypothesis for each e_i and s_i, we have

$$\begin{array}{c} S(\Gamma) \vdash_U e_i : S(\sigma_i) \\ S(\Gamma) \vdash_U s_i : S(|\sigma_i|) \end{array}$$

Since $S(|\sigma_i|) = |S(\sigma_i)|$, we have

$$S(\Gamma) \vdash_U s_i : |S(\sigma_i)|$$

Applying the typing rule for records, we have

$$S(\Gamma) \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : S(\sigma_1) \times \cdots \times S(\sigma_1)$$

Since $S(\sigma_1) \times \cdots \times S(\sigma_1) = S(\sigma_1 \times \cdots \times \sigma_n)$, we obtain

$$S(\Gamma) \vdash_U (e_1^{s_1}, \ldots, e_n^{e_n}) : S(\sigma_1 \times \cdots \times \sigma_1)$$

as desired.

Case $\Gamma \vdash_U \pi_o^s(e) : \sigma_i$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e : \sigma_1 \times \cdots \times \sigma_n \quad \Gamma \vdash_U s : |\sigma_i| \quad \Gamma \vdash_U o : \| \sigma_1, \ldots, \sigma_{i-1} \|}{\Gamma \vdash_U \pi_o^s(e) : \sigma_i}$$

Applying induction hypothesis for $\Gamma \vdash_U e : \sigma_1 \times \cdots \times \sigma_n$, $\Gamma \vdash_U s : |\sigma_i|$, and $\Gamma \vdash_U o : \| \sigma_1, \ldots, \sigma_{i-1} \|$, we have

$S(\Gamma) \vdash_U e : S(\sigma_1 \times \cdots \times \sigma_n)$, $S(\Gamma) \vdash_U s : S(|\sigma_i|)$, and $S(\Gamma) \vdash_U o : S(\| \sigma_1, \ldots, \sigma_{i-1} \|)$, we have

By definition of substitution for types, we have

$$S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$$
$$S(|\sigma_i|) = |S(\sigma_i)|$$
$$S(\| \sigma_1, \ldots, \sigma_{i-1} \|) = \| S(\sigma_1), \ldots, S(\sigma_{i-1}) \|$$

applying typing rule for projection, we obtain $S(\Gamma) \vdash_U \pi_o^s(e) : S(\sigma_i)$ as desired.

Case $\Gamma \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e_1 : \sigma_1 \quad \Gamma \vdash_U s : |\sigma_1| \quad \Gamma, local(x : \sigma_1) \vdash_U e_2 : \sigma_2}{\Gamma \vdash_U \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for $\Gamma \vdash_U e_1 : \sigma_1$ and $\Gamma \vdash_U s : |\sigma_1|$, we obtain

$$S(\Gamma) \vdash_U e_1 : S(\sigma_1)$$
$$S(\Gamma) \vdash_U s : S(|\sigma_1|)$$

Since S respects Γ, then S also respects $\Gamma, local(x : \sigma)$. Applying induction hypothesis for $\Gamma, local(x : \sigma_1) \vdash_U e_2 : \sigma_2$, we obtain

$$S(\Gamma, local(x : \sigma_1)) \vdash_U e_2 : S(\sigma_2)$$

Since $S(\Gamma, local(x : \sigma_1)) = S(\Gamma), local(x : S(\sigma_1))$, by applying typing rule for `let` term, we obtain

$$S(\Gamma) \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} : S(\sigma_2)$$

as desired.

Case $\Gamma \vdash_U [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|$**.** Suppose this typing is derived from

$$\frac{\Gamma \vdash_U e_i : |\sigma_i| \ (1 \leq i \leq n)}{\Gamma \vdash_U [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|}$$

Applying induction hypothesis for each e_i, we have $S(\Gamma) \vdash_U e_i : S(|\sigma_i|)$. Since $S(|\sigma_i|) = |S(\sigma_i)|$, applying the typing rule for offset, we have

$$S(\Gamma) \vdash_U [e_i + \cdots + e_n] : \| S(\sigma_1), \ldots, S(\sigma_n) \|$$

Applying substitution rule for bitmap type , we can conclude

$$S(\Gamma) \vdash_U [e_i + \cdots + e_n] : S(\|\sigma_1, \ldots, \sigma_n\|)$$

Case $e = |B|$**.** Suppose $\Gamma \vdash_U |B| : |\sigma|$. Since $|B|$ is a constant size, type σ must have an appropriate outermost type constructor. Type substitution does not modify the outermost type constructor, then the substituted type $S(\sigma)$ must have the same outermost constructor as σ, therefore $\texttt{sizeOf}(\sigma) = \texttt{sizeOf}(S(\sigma))$. This implies $S(\Gamma) \vdash_U |B| : S(|\sigma|)$ as desired. □

4.1.5 Semantics

We define semantics of Λ^U, in style of natural semantic [Kah87], that faithfully models evaluation of expressions involving unboxed manipulation. The sets of runtime values (ranged over by v) and runtime environments (ranged over by E) are given by the following grammar.

v	$::=$	c^o	constant value
	$\mid$	i	integer
	$\mid$	$cls(E, \lambda\overline{x}.e)$	function closure
	$\mid$	$(v, \ldots, v)$	record value
	$\mid$	$wrong$	runtime error
E	$::=$	$\emptyset \mid E, x : v$	run-time environment

Size terms or offset terms in the calculus are evaluated to integers (denoted by i).

$cls(E, \lambda\overline{x}.e)$ is a function closure, where E is an environment assigning values to variables.

$(v_1, \ldots, v_n)$ is a record (heap block) consisting of n unboxed values $\{v_1, \ldots, v_n\}$.

wrong represents a runtime type error.

We define the operational semantics, which serves as an abstract machine (VM), in such a way that it checks the consistency of unboxed values and their sizes every time they are manipulated. If this check fails then the evaluation halts with *wrong*. In order to formalize this check, we define the size value $\texttt{sizeOf}(v)$ corresponding to v as follows.

$$\begin{aligned}
\texttt{sizeOf}(c^o) &= 1 \text{ if } c^o \text{ is a single constant} \\
\texttt{sizeOf}(c^o) &= 2 \text{ if } c^o \text{ is a double constant} \\
\texttt{sizeOf}(i) &= 1 \\
\texttt{sizeOf}(cls(E, \lambda\overline{x}.e)) &= 1 \\
\texttt{sizeOf}((v_1, \ldots, v_n)) &= 1
\end{aligned}$$

Closure and record values have one word size since we do not unbox structured data. In our development, only floating point numbers have two word size.

The operational semantics is defined in the style of [Kah87] by giving a set of rules to derive a evaluation relation of the form $E \vdash_U e \Downarrow v$, which reads: "$e$ evaluates to v under E". Figure 4.2 gives the set of evaluation rules. This set of rules should be taken with the following implicit rules yielding *wrong*: if evaluation of any component yields *wrong* or undefined or does not satisfy the specified condition then the entire term will yield *wrong*.

$$E \vdash_U c^o \Downarrow c^o$$

$$\frac{E \vdash_U s \Downarrow v \quad v = \texttt{sizeOf}(E(x))}{E \vdash_U x^s \Downarrow E(x)}$$

$$\frac{E \vdash_U \overline{s} \Downarrow \overline{i}}{E \vdash_U \lambda \overline{x}.e \Downarrow cls(E, \lambda \overline{x}.e)}$$

$$\frac{\begin{array}{c} E \vdash_U e_1 \Downarrow cls(E_0, \lambda \overline{x}.e_0) \quad E \vdash_U \overline{e_2} \Downarrow \overline{v_2} \\ E_0, \overline{x : v_2} \vdash_U e_0 \Downarrow v_0 \quad E_0 \vdash_U \overline{e_s} \Downarrow \overline{v_s} \quad \overline{v_s = \texttt{sizeOf}(v_2)} \end{array}}{E \vdash_U (e_1 \ \overline{e_2{}^{e_s}}) \Downarrow v_0}$$

$$\frac{E \vdash_U s_i \Downarrow v_i^s \quad E \vdash_U e_i \Downarrow v_i \quad \texttt{sizeOf}(v_i) = v_i^s \text{ for all } 1 \leq i \leq n}{E \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) \Downarrow (v_1, \ldots, v_n)}$$

$$\frac{\begin{array}{c} E \vdash_U e \Downarrow (v_1, \ldots, v_n) \quad E \vdash_U s \Downarrow v_s \quad E \vdash_U o \Downarrow v_o \\ \texttt{sizeOf}(v_i) = v_s \quad \texttt{sizeOf}(v_1) + \cdots + \texttt{sizeOf}(v_{i-1}) = v_o \end{array}}{E \vdash_U \pi_o^s(e) \Downarrow v_i}$$

$$\frac{E \vdash_U e_1 \Downarrow v_1 \quad E \vdash_U s \Downarrow v_s \quad \texttt{sizeOf}(v_1) = v_s \quad E, x : v_1 \vdash_U e_2 \Downarrow v_2}{E \vdash_U \texttt{let } x^s = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v_2}$$

$$E \vdash_U |i| \Downarrow i$$

$$\frac{E \vdash_U e_i \Downarrow B_i \quad (1 \leq i \leq n)}{E \vdash_U [e_1 + \cdots + e_n] \Downarrow B_1 + \cdots + B_n}$$

Figure 4.2: Operational Semantics of The Unboxed Calculus

In the rule for variable x^s, VM checks the consistency between the value $E(x)$ and its size by first evaluating the size expression s, then comparing the result with the actual size of $E(x)$.

In the rule for application, VM checks the consistency between the sizes of actual arguments and the values of size terms $\overline{v_s}$.

For allocating a record, VM also has to check the consistency between layout of record given by sizes of its fields (s_i) and size the actual field values. This is done by first evaluating each s_i to v_i^s, then comparing v_i^s with the corresponding field size $\texttt{sizeOf}(v_i)$.

The evaluation rule for projection needs more explanations. VM obtains the selected value by first evaluate the record to get a block of memory, compute the value of offset and size, and extract the field by using the resulting offset value and size value. Suppose that the block consists of $(v_1, \ldots, v_n)$. For checking the consistency, VM compares the values of s and o (v_s, v_o) with the actual field size $\texttt{sizeOf}(v_i)$ and actual offset (computed from $\texttt{sizeOf}(v_1) + \cdots + \texttt{sizeOf}(v_{i-1})$)

In the evaluation rule for `let` expressions, VM first evaluates the bound expression e_1, binds its value to runtime environment slot x, then evaluate the main expression e_2 under the updated environment. To ensure that the value v_1 of e_1 has a correct size (with respect to size of e_1 given by s), VM checks $\texttt{sizeOf}(v_1) = v_s$.

To show that the type system is sound with respect to this operational semantics, we define typing judgments on runtime values and environments of the forms $\models_U v : \sigma$ and

$$\models_U \emptyset : \emptyset$$

$$\frac{\models_U E : \Gamma \quad \models_U \overline{v} : \overline{\tau}}{\models_U (E, \overline{x : v}) : (\Gamma, arg(\overline{x : \tau}))}$$

$$\frac{\models_U E : \Gamma \quad \models_U v : \sigma}{\models_U (E, x : v) : (\Gamma, local(x : \sigma))}$$

$$\frac{\models_U E : \Gamma}{\models_U E : (\Gamma, tvar(\overline{t}))}$$

Figure 4.3: Typing rules on environments

$$\models_U c^o : o$$

$$\models_U i : |\sigma| \quad \text{If } \mathtt{sizeOf}(\sigma) = i$$

$$\models_U i : \| \sigma_1, \ldots, \sigma_n \| \quad \text{If } \mathtt{sizeOf}(\sigma_1) + \cdots + \mathtt{sizeOf}(\sigma_n) = i$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ and closed types } \overline{\tau} \\ \models_U E : \Gamma \quad \Gamma, arg(\overline{x : \tau}) \vdash_U e : \sigma\end{array}}{\models_U cls(E, \lambda\overline{x}.e) : \overline{\tau} \rightarrow \sigma}$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ so that} \\ \models_U E : \Gamma \quad \Gamma, tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_U e : \sigma\end{array}}{\models_U cls(E, \lambda\overline{s}.e) : \forall\overline{t}.\overline{|t|} \rightarrow \sigma}$$

$$\frac{\models_U v_i : \sigma_i \text{ for all } 1 \leq i \leq n}{\models_U (v_1, \ldots, v_n) : \sigma_1 \times \cdots \times \sigma_n}$$

Figure 4.4: Typing rules on values

$\models_U E : \Gamma$ where σ represents a closed type and Γ represents a ground context. The set of rules to derive these judgments is given in Figure 4.4 and Figure 4.3.

The following property holds.

Lemma 4.5 *If* $\models_U v : \sigma$ *then* $\mathtt{sizeOf}(v) = \mathtt{sizeOf}(\sigma)$.

PROOF. Straightforward by definition. □

As in a conventional type system, the type system of Λ^U is sound, i.e. it guarantees type-error free evaluation of any type correct expression. We show this in the following theorem.

Theorem 4.1 *Let* Γ *be a context,* e *be an expression,* σ *be a type so that* $\Gamma \vdash_U e : \sigma$. *Let* S *be a ground substitution so that* $TV(\Gamma) \subseteq dom(S)$. *Let* E *be a run-time environment so that* $\models_U E : S(\Gamma)$. *If* $E \vdash_U e \Downarrow v$ *then* $\models_U v : S(\sigma)$.

PROOF. We prove this theorem by induction on the derivation of the evaluation rule.

Case $e \vdash_U c^0 \Downarrow c^o$**.** Straightforward.

Case $E \vdash_U x^s \Downarrow E(x)$**.** This is derived from

$$\frac{E \vdash_U s \Downarrow v \quad v = \texttt{sizeOf}(E(x))}{E \vdash_U x^s \Downarrow E(x)}$$

Suppose $\Gamma \vdash_U x^s : \sigma$ is derived from $\Gamma(x) = \sigma$ and $\Gamma \vdash_U s : |\sigma|$. By induction hypothesis for s, we obtain $\models_U v : S(|\sigma|)$. Since $S(|\sigma|) = |S(\sigma)|$, by value typing for size, we have $v = \texttt{sizeOf}(S(\sigma))$. We have to prove $v = \texttt{sizeOf}(E(x))$ and $\models_U E(x) : S(\sigma)$. Since $\models_U E : S(\Gamma)$, we have $\models_U E(x) : S(\Gamma)(x)$. Since $S(\Gamma)(x) = S(\Gamma(x))$, we obtain $\models_U E(x) : S(\sigma)$. By lemma 4.5, we also have $\texttt{sizeOf}(E(x)) = \texttt{sizeOf}(S(\sigma))$. Then we obtain $v = \texttt{sizeOf}(()E(x))$ as desired.

Case $E \vdash_U \lambda\overline{x}.e \Downarrow cls(E, \lambda\overline{x}.e)$**.** We have two possible typing derivation of $\lambda\overline{x}.e$
<u>*Sub-case 1*</u> $\Gamma \vdash_U \lambda\overline{x}.e : \overline{\tau} \to \sigma$. Suppose this typing is derived from

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{x}.e : \overline{\tau} \to \sigma}$$

S is a ground substitution, then S should respect $\Gamma, arg(\overline{x : \tau})$. Applying substitution lemma 3.4, we obtain

$$S(\Gamma, arg(\overline{x : \tau})) \vdash_U e : S(\sigma)$$

Since $S(\Gamma, arg(\overline{x : \tau})) = S(\Gamma), arg(\overline{x : S(\tau)})$, then

$$S(\Gamma), arg(\overline{x : S(\tau)}) \vdash_U e : S(\sigma)$$

By context formation rule, we have $\Gamma \vdash_U \overline{\tau}$. By lemma 4.3 we have $S(\tau)$ are closed types and $S(\Gamma)$ is a ground context. Then by applying value typing rule for closure, we obtain

$$\models_U cls(E, \lambda\overline{x}.e) : \overline{S(\tau)} \to S(\sigma)$$

Since $\overline{S(\tau)} \to S(\sigma) = S(\overline{\tau} \to \sigma)$, then we have

$$\models_U cls(E, \lambda\overline{x}.e) : S(\overline{\tau} \to \sigma)$$

as desired.
<u>*Sub-case 2*</u> $\Gamma \vdash_U \lambda\overline{x}.e : \forall\overline{t}.\overline{|t|} \to \sigma$ Suppose this typing is derived from

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{x : |t|}) \vdash_U e : \sigma}{\Gamma \vdash_U \lambda\overline{x}.e : \forall\overline{t}.\overline{|t|} \to \sigma}$$

By bound type variable convention, suppose that $\overline{t} \cap dom(S) = \emptyset$. Since S is a ground substitution then S respects $\Gamma, tvar(\overline{t}), arg(\overline{x : |t|})$. Applying substitution lemma 4.4, we obtain

$$S(\Gamma, tvar(\overline{t}), arg(\overline{x : |t|})) \vdash_U e : S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then

$$S(\Gamma), tvar(\overline{t}), arg(\overline{x : |t|}) \vdash_U e : S(\sigma)$$

By lemma 4.3 we have that $S(\Gamma)$ is a ground context. By value typing for closure, we obtain

$$\models_U cls(E, \lambda\overline{x}.e) : \forall\overline{t}.\overline{|t|} \to S(\sigma)$$

Since $\overline{t} \cap dom(S) = \emptyset$, then we have $\forall\overline{t}.\overline{|t|} \to S(\sigma) = S(\forall\overline{t}.\overline{|t|} \to \sigma)$ as desired.

Case $E \vdash_U (e_1\ \overline{e_2}^{e_s}) \Downarrow v_0$**.** Suppose that this is derived from

$$\frac{\begin{array}{c} E \vdash_U e_1 \Downarrow cls(E_0, \lambda\overline{x}.e_0) \quad E \vdash_U \overline{e_2} \Downarrow \overline{v_2} \\ E_0, \overline{x : v_2} \vdash_U e_0 \Downarrow v_0 \quad E_0 \vdash_U \overline{e_s} \Downarrow \overline{v_s} \quad \overline{v_s = \mathtt{sizeOf}(v_2)} \end{array}}{E \vdash_U (e_1\ \overline{e_2}^{e_s}) \Downarrow v_0}$$

There are two possible typing derivations for $(e_1\ \overline{e_2}^{e_s})$
<u>*Sub-case 1*</u>.

$$\frac{\Gamma \vdash_U e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_U \overline{e_2} : \overline{\tau} \quad \Gamma \vdash_U \overline{e_s} : \overline{|\tau|}}{\Gamma \vdash_U (e_1\ \overline{e_2}^{e_s}) : \sigma}$$

Applying induction hypothesis for e_1, $\overline{e_2}$ and $\overline{e_s}$, we obtain

$$\begin{array}{c} \models_U cls(E_0, \lambda\overline{x}.e_0) : S(\overline{\tau} \to \sigma) \\ \models_U \overline{v_2} : \overline{S(\tau)} \\ \models_U \overline{v_s} : \overline{S(|\tau|)} \end{array}$$

We have to prove that $\overline{v_s = \mathtt{sizeOf}(v_2)}$ and $\models_U v_0 : S(\sigma)$. By lemma 4.5, $\overline{v_2} = \overline{\mathtt{sizeOf}(S(\tau))}$. Since $\overline{S(|\tau|)} = \overline{|S(\tau)|}$, we also have $\overline{v_s} = \overline{\mathtt{sizeOf}(S(\tau))}$. Therefore $\overline{v_s = \mathtt{sizeOf}(v_2)}$. Since $S(\overline{\tau} \to \sigma) = \overline{S(\tau)} \to S(\sigma)$, by value typing rule for closure, there must exist a ground context Γ_0 so that $\models_B E_0 : \Gamma_0$ and $\Gamma_0, arg(\overline{x : S(\tau)}) \vdash_B e_0 : S(\sigma)$. This is also easy to choose that Γ_0 does not contain any $tvar()$ assumption. By typing rule for runtime environment, we have $\models_U (E_0, \overline{x : v_2}) : \Gamma_0, arg(\overline{x : S(\tau)})$ We choose an identical substitution S_0 ($dom(S_0) = \emptyset$), Since Γ_0 does not contain any $tvar()$ assumption, then we can apply induction hypothesis for e_0 to obtain $\models_U v_0 : S(\sigma)$ as desired.
<u>*Sub-case 2*</u>.

$$\frac{\Gamma \vdash_U e_1 : \forall\overline{t}.\overline{|t|} \to \sigma \quad \Gamma \vdash_U \overline{e_2} : \overline{|\tau|} \quad \Gamma \vdash_U \overline{\tau}}{\Gamma \vdash_U (e_1\ \overline{e_2}^{e_s}) : \sigma[\overline{\tau}/\overline{t}]}$$

where $\overline{e_s = |1|}$. Applying induction hypothesis for e_1 and $\overline{e_2}$, we obtain

$$\begin{array}{c} \models_U cls(E_0, \lambda\overline{x}.e_0) : S(\forall\overline{t}.\overline{|t|} \to \sigma) \\ \models_U \overline{v_2} : \overline{S(|\tau|)} \end{array}$$

By the bound type variable convention, assume that $\overline{t} \cap dom(S) = \emptyset$. Therefore $S(\forall\overline{t}.\overline{|t|} \to \sigma) = \forall\overline{t}.\overline{|t|} \to S(\sigma)$. By value typing rule for closure, there must exist a ground context Γ_0 so that $\models_U E_0 : \Gamma_0$ and $\Gamma_0, tvar(\overline{t}), arg(\overline{x : |t|}) \vdash_B e_0 : S(\sigma)$. This is easy to choose Γ_0 so that it does not contain any $tvar()$ assumption. Since $\overline{S(|\tau|)} = \overline{|S(\tau)|}$, then we have

$$\models_U \overline{v_2} : \overline{|S(\tau)|}$$

Since $\Gamma \vdash_U \overline{\tau}$, then by lemma 4.3, we have $\overline{S(\tau)}$ are closed types. Choose $S_0 = [\overline{S(\tau)}/\overline{t}]$, S_0 is a ground substitution and $TV(\Gamma_0, tvar(\overline{t}), arg(\overline{x : |t|})) \subseteq dom(S_0)$. Since Γ_0 is a ground context and does not contain any $tvar()$ assumption, we have

$$S_0(\Gamma_0, tvar(\overline{t}), arg(\overline{b : |t|})) = \Gamma_0, tvar(\overline{t}), arg(\overline{x : |S(\tau)|})$$

Since $\models_U E_0 : \Gamma_0$ then

$$\models_U (E_0, \overline{x : v_2}) : (\Gamma_0, tvar(\overline{t}), arg(\overline{x : |S(\tau)|}))$$

Now, we apply induction hypothesis for e_0 to obtain $\models_U v_0 : S_0(S(\sigma))$. Since $S_0 = [\overline{S(\tau)}/\overline{t}]$ and $\overline{t} \cap dom(S) = \emptyset$, we can conclude that $\models_U v_0 : S(\sigma[\overline{\tau}/\overline{t}])$ as desired.

Case $E \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) \Downarrow (v_1, \ldots, v_n)$**.** Suppose this is derived from

$$\frac{E \vdash_U s_i \Downarrow v_i^s \quad E \vdash_U e_i \Downarrow v_i \quad \texttt{sizeOf}(v_i) = v_i^s \text{ for all } 1 \leq i \leq n}{E \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) \Downarrow (v_1, \ldots, v_n)}$$

Also suppose $\Gamma \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n$. This is derived from

$$\frac{\Gamma \vdash_U e_i : \sigma_i \quad \Gamma \vdash_U s_i : |\sigma_i| \text{ for } (1 \leq i \leq n)}{\Gamma \vdash_U (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

We have to prove: $\texttt{tagOf}(v_i) = v_i^s$ for each i, and $\models_U (v_1, \ldots, v_n) : S(\sigma_1 \times \cdots \times \sigma_n)$. Applying induction hypothesis for each e_i and each e_j^s, we have

$$\models_U v_i : S(\sigma_i)$$
$$\models_U v_i^s : S(|\sigma_i|)$$

By lemma 4.5, we have $\texttt{tagOf}(v_i) = \texttt{tagOf}(S(\sigma_i))$. Since $S(|\sigma_i|) = |S(\sigma_i)|$ by value typing rule for size, we have $v_i^s = \texttt{sizeOf}(S(\sigma_i))$. Therefore $\texttt{tagOf}(v_i) = v_i^s$ for each i. Besides, applying value typing rule for records, we obtain,

$$\models_U (v_1, \ldots, v_n) : S(\sigma_1) \times \cdots \times S(\sigma_n)$$

Since $S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$, the proof for this case is done.

Case $E \vdash_U \pi_o^s(e) \Downarrow v_i$**.** This should be derived from

$$\frac{\begin{array}{c} E \vdash_U e \Downarrow (v_1, \ldots, v_n) \quad E \vdash_U s \Downarrow v_s \quad E \vdash_U o \Downarrow v_o \\ \texttt{sizeOf}(v_i) = v_s \quad \texttt{sizeOf}(v_1) + \cdots + \texttt{sizeOf}(v_{i-1}) = v_o \end{array}}{E \vdash_U \pi_o^s(e) \Downarrow v_i}$$

Also suppose $\Gamma \vdash_U \pi_o^s(e) : \sigma_i$. This is derived from

$$\frac{\Gamma \vdash_U e : \sigma_1 \times \cdots \times \sigma_n \quad \Gamma \vdash_U s : |\sigma_i| \quad \Gamma \vdash_U o : \| \sigma_1, \ldots, \sigma_{i-1} \|}{\Gamma \vdash_U \pi_o^s(e) : \sigma_i}$$

We have to prove $\texttt{sizeOf}(v_i) = v_s$, $\texttt{sizeOf}(v_1) + \cdots + \texttt{sizeOf}(v_{i-1}) = v_o$, and $\models_U v_i : S(\sigma_i)$ Applying the induction hypothesis for e, s, and o, we obtain

$$\models_U (v_1, \ldots, v_n) : S(\sigma_1 \times \cdots \times \sigma_n)$$
$$\models_U v_s : S(|\sigma_i|)$$
$$\models_U v_o : S(\| \sigma_1, \ldots, \sigma_{i-1} \|)$$

Since $S(\sigma_1 \times \cdots \times \sigma_n) = S(\sigma_1) \times \cdots \times S(\sigma_n)$, by the value typing rule for record, we have $\models_{ML} v_i : S(\sigma_i)$. Since $S(|\sigma_i|) = |S(\sigma_i)|$, then by value typing rule for size, we have $v_s = \texttt{sizeOf}(S(\sigma))$. Since $S(\| \sigma_1, \ldots, \sigma_{i-1} \|) = \| S(\sigma_1), \ldots, S(\sigma_{i-1}) \|$, by value typing rule for offset, we have $v_o = \texttt{sizeOf}(S(\sigma_1)) + \cdots + \texttt{sizeOf}(S(\sigma_{i-1}))$. By value typing rule for record , we have $\models_B v_j : S(\sigma_j)$ for each j. By lemma 4.5, we have $\texttt{sizeOf}(v_j) = \texttt{sizeOf}(S(\sigma_j))$ for each j. Therefore $v_s = \texttt{sizeOf}(v_i)$ and $v_0 = \texttt{sizeOf}(v_1) + \cdots + \texttt{sizeOf}(v_{i-1})$ as desired.

Case $E \vdash_U \texttt{let } x^s = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v_2$**.** Suppose that this is derived from

$$\frac{E \vdash_U e_1 \Downarrow v_1 \quad E \vdash_U s \Downarrow v_s \quad \texttt{sizeOf}(v_1) = v_s \quad E, x : v_1 \vdash_U e_2 \Downarrow v_2}{E \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow v_2}$$

Also suppose $\Gamma \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$. This is derived from

$$\frac{\Gamma \vdash_U e_1 : \sigma_1 \quad \Gamma \vdash_U s : |\sigma_1| \quad \Gamma, local(x : \sigma_1) \vdash_U e_2 : \sigma_2}{\Gamma \vdash_U \texttt{let}\ x^s = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for e_1 and s, we obtain $\models_{ML} v_1 : S(\sigma_1)$ and $\models_{ML} v_s : S(|\sigma_1|)$. Let's define $\Gamma_1 = \Gamma, local(x : \sigma_1)$, We have $TV(\Gamma_1) = TV(\Gamma) \subseteq dom(S)$. We also have $S(\Gamma_1) = S(\Gamma), local(x : S(\sigma_1))$. Since $\models_U E : S(\Gamma)$, we obtain $\models_U (E, x : v_1) : S(\Gamma), local(x : S(\sigma_1))$. Applying induction hypothesis for e_2 , we obtain $\models_U v_2 : S(\sigma_2)$. Since $\models_{ML} v_1 : S(\sigma_1)$, by lemma 4.5, we have $\texttt{sizeOf}(v_1) = \texttt{sizeOf}(S(\sigma_1))$. Since $\models_{ML} v_s : S(|\sigma_1|)$, by value typing rule for size, we obtain $v_s = \texttt{sizeOf}(()S(\sigma))$. Thus $v_s = \texttt{sizeOf}(v_1)$ as desired.

Case $E \vdash_U |B| \Downarrow B$**.** immediate by definition.

Case $E \vdash_U [e_1 + \cdots + e_n] \Downarrow B_1 + \cdots + B_n$**.** This is derived from $E \vdash_U e_i \Downarrow B_i$ for each i. Also suppose $\Gamma \vdash_U [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|$. By the offset typing rule, $\Gamma \vdash_U e_i : |\sigma_i|$, for each $1 \leq i \leq n$. Applying induction hypothesis for each e_i, we obtain $\models_U B_i : |S(\sigma_i)|$. By value typing for offset and type substitution rule, we have $\models_U B_1 + \cdots + B_n : S(\| \sigma_1, \ldots, \sigma_n \|)$ as desired. □

4.2 The Compilation Algorithm

Similar to the bitmap-passing compilation, I design the unboxed compilation in two phases: compilation from terms of the source calculus (λ^{ML}) into terms of an explicitly typed immediate calculus (λ^U), transformation from terms in λ^U to terms in Λ^U by erasing all type annotations.

The syntactic correctness for the whole compilation process can be proved by a combination of syntactic correctnesses of each phase.

4.2.1 The Explicitly Typed Unboxed Calculus – λ^U

The set of terms of λ^U is given by the following syntax:

e	$::=$	c^o	constant
	$\mid$	x^e	variable
	$\mid$	$\lambda \overline{x : \tau}.e$	function
	$\mid$	$(e\ \overline{e^e})$	application
	$\mid$	$\Lambda \overline{t}.\lambda \overline{x : \lvert t \rvert}.e$	type abstaction and size abstraction
	$\mid$	$(e\ \overline{\tau}\ \overline{e})$	type instantiation and size application
	$\mid$	$(e^e, \ldots, e^e)$	record
	$\mid$	$\pi^e_e(e)$	projection
	$\mid$	$\texttt{let}\ x^e : \sigma = e\ \texttt{in}\ e\ \texttt{end}$	let binding
	$\mid$	$[e + \cdots + e]$	offset
	$\mid$	$\lvert B \rvert$	constant size ($B \in \{1, 2\}$)

$$\Gamma \vdash_{TU} c^o : o$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_{TU} s : |\sigma|}{\Gamma \vdash_{TU} x^s : \sigma}$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TU} e : \sigma}{\Gamma \vdash_{TU} \lambda\overline{x : \tau}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_{TU} e : \sigma}{\Gamma \vdash_{TU} \Lambda\overline{t}.\lambda\overline{s : |t|}.e : \forall\overline{t}.\overline{|t|} \to \sigma}$$

$$\frac{\Gamma \vdash_{TU} e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_{TU} \overline{e_2} : \overline{\tau} \quad \Gamma \vdash_{TU} \overline{e_s} : \overline{|\tau|}}{\Gamma \vdash_{TU} (e_1\ \overline{e_2{}^{e_s}}) : \sigma}$$

$$\frac{\Gamma \vdash_{TU} e : \forall\overline{t}.\overline{|t|} \to \sigma \quad \Gamma \vdash_{TU} \overline{e_s} : \overline{|\tau|} \quad \Gamma \vdash_{U} \overline{\tau}}{\Gamma \vdash_{TU} (e\ \overline{\tau}\ \overline{e_s}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_{TU} e_i : \sigma_i \quad \Gamma \vdash_{TU} s_i : |\sigma_i| \text{ for } (1 \le i \le n)}{\Gamma \vdash_{TU} (e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_{TU} e : \sigma_1 \times \cdots \times \sigma_n \quad \Gamma \vdash_{TU} s : |\sigma_i| \quad \Gamma \vdash_{TU} o : \| \sigma_1, \ldots, \sigma_{i-1} \|}{\Gamma \vdash_{TU} \pi_o^s(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_{TU} e_1 : \sigma_1 \quad \Gamma \vdash_{TU} s : |\sigma_1| \quad \Gamma, local(x : \sigma_1) \vdash_{TU} e_2 : \sigma_2}{\Gamma \vdash_{U} \texttt{let } x^s : \sigma_1 = e_1 \texttt{ in } e_2 \texttt{ end} : \sigma_2}$$

$$\frac{\texttt{sizeOf}(\sigma) = S \quad S \in \{|1|, |2|\}}{\Gamma \vdash_{TU} S : |\sigma|}$$

$$\frac{\Gamma \vdash_{TU} e_i : |\sigma_i| \ (1 \le i \le n)}{\Gamma \vdash_{TU} [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|}$$

Figure 4.5: Typing Rules of λ^U

In this calculus, we distinguish between two kinds of function: ordinary function (represented by $\lambda\overline{x : \tau}.e$) and size function (represented by $\Lambda\overline{t}.\lambda\overline{x : |t|}.e$). A size function $\Lambda\overline{t}.\lambda\overline{x : |t|}.e'$ can be obtained from the compilation result of a type abstraction $\Lambda\overline{t}.e$ in λ^{ML} where $\overline{x}$ are the inserted bit tag parameters corresponding to $\overline{t}$.

We also distinguish two kinds of application: ordinary lambda application $(e_1\ \overline{e_2{}^{e_s}})$ and size application $(e\ \overline{\tau}\ \overline{e_s})$. In an ordinary lambda application $(e_1\ \overline{e_2{}^{e_s}})$, $\overline{e_s}$ represent the size terms of the actual arguments $\overline{e_2}$. This can be generated from types of $\overline{e_2}$. In the case of size application $(e\ \overline{\tau}\ \overline{e_s})$, $\overline{e_s}$ are actual size parameters. They are generated from the types $\overline{\tau}$.

We share the same definition of typing context, type well-formedness and context well-formedness of λ^U with those of Λ^U. The set of rules to derive typing judgments of the form $\Gamma \vdash_{TU} e : \sigma$ is given in Figure 4.5.

The typing rule for size function is similar to the second typing rule for function in Λ^U and the typing rule for bit tag application is similar to the second typing rule for application in Λ^U.

$$\Gamma \vdash_{TU} c^o \rightsquigarrow c^o$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_{TU} \sigma \rightsquigarrow s}{\Gamma \vdash_{TU} x \rightsquigarrow x^s}$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TU} e \rightsquigarrow e'}{\Gamma \vdash_{TU} \lambda \overline{x : \tau}.e \rightsquigarrow \lambda \overline{x : \tau}.e'}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{TU} e_1 \rightsquigarrow e_1' \quad \Gamma \vdash_{TU} \overline{e_2} \rightsquigarrow \overline{e_2'} \\ \Gamma \vdash_{TU} \overline{e_2} : \overline{\tau} \quad \Gamma \vdash_{TU} \overline{\tau} \rightsquigarrow \overline{e_s}\end{array}}{\Gamma \vdash_{TU} (e_1\ \overline{e_2}) \rightsquigarrow (e_1'\ \overline{e_2'^{e_s}})}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_{TU} e \rightsquigarrow e' \quad \overline{s} \text{ are fresh variables}}{\Gamma \vdash_{TU} \Lambda\overline{t}.e \rightsquigarrow \Lambda\overline{t}.\lambda\overline{s : |t|}.e'}$$

$$\frac{\Gamma \vdash_{TU} e \rightsquigarrow e' \quad \Gamma \vdash_{TU} \overline{\tau} \rightsquigarrow \overline{e_s}}{\Gamma \vdash_{TU} (e\ \overline{\tau}) \rightsquigarrow (e'\ \overline{\tau}\ \overline{e_s})}$$

$$\frac{\Gamma \vdash_{TU} e_i \rightsquigarrow g_i \quad \Gamma \vdash_{TU} g_i : \sigma_i \quad \Gamma \vdash_{TU} \sigma_i \rightsquigarrow s_i \quad (1 \le i \le n)}{\Gamma \vdash_{TU} (e_1, \ldots, e_n) \rightsquigarrow (g_1^{s_1}, \ldots, g_n^{s_n})}$$

$$\frac{\Gamma \vdash_{TU} e \rightsquigarrow e' \quad \Gamma \vdash_{TU} e' : \sigma_1 \times \cdots \times \sigma_n \quad \Gamma \vdash_{TU} \sigma_j \rightsquigarrow s_j \text{ for all } 1 \le j \le i}{\Gamma \vdash_{TU} \pi_i(e) \rightsquigarrow \pi^{s_i}_{[s_1 + \cdots + s_{i-1}]}(e')}$$

$$\frac{\Gamma \vdash_{TU} e_1 \rightsquigarrow e_1' \quad \Gamma \vdash_{TU} e_1 : \sigma' \quad \Gamma \vdash_{TU} \sigma' \rightsquigarrow s \quad \Gamma, local(x : \sigma) \vdash_{TU} e_2 \rightsquigarrow e_2'}{\Gamma \vdash_{TU} \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end} \rightsquigarrow \texttt{let}\ x^s : \sigma' = e_1'\ \texttt{in}\ e_2'\ \texttt{end}}$$

Figure 4.6: Unboxed Compilation

4.2.2 Compilation from λ^{ML} to λ^U

The compilation algorithm from λ^{ML} terms to λ^U terms is formulated as judgments of the forms $\Gamma \vdash_{TU} e \rightsquigarrow e'$ where Γ is a context in λ^U, e is a source expression in λ^{ML} and e' is a target expression in λ^U. The set of compilation rules is given in Figure 4.6.

In the compilation rules, size abstractions are inserted at each type abstraction and size applications are inserted at each type instantiation. The algorithm also inserts the necessary information (sizes and offsets) at appropriate places where they are needed: variables, lambda abstractions, records, projections, `let` expressions.

Sizes (and therefore offsets) are generated by inspecting the corresponding types. We formalize this by a *size creation algorithm* of the form $\Gamma \vdash_{TU} \sigma \rightsquigarrow e$. The set of rules to derive this judgment is given below.

$\Gamma \vdash_{TU} \sigma \rightsquigarrow \texttt{sizeOf}(\sigma)$	if σ has a proper outermost type constructor
$\Gamma \vdash_{TU} t \rightsquigarrow s$	where $s : \lvert t \rvert \in \Gamma$

Similar to the bit tag creation algorithm, the size creation algorithm will fails if $\sigma = t$ and there does not exist $s : |t| \in \Gamma$. Then for the soundness of the algorithm, we assume that the compile context Γ is constructed in such a way that any $tvar(\overline{t})$ assumption is followed by a $arg(\overline{s : |t|})$ assumption (this is guaranteed by the simulation relations defined later). The following lemma demonstrates this feature.

$$\tau \sim_U \tau$$

$$\frac{\sigma \sim_U \sigma'}{\overline{\tau} \to \sigma \sim_U \overline{\tau} \to \sigma'}$$

$$\frac{\sigma \sim_U \sigma'}{\forall \overline{t}.\sigma \sim_U \forall \overline{t}.\overline{|t|} \to \sigma'}$$

$$\frac{\sigma_i \sim_U \sigma_i' \text{ for each } 1 \leq i \leq n}{\sigma_1 \times \cdots \times \sigma_n \sim_U \sigma_1' \times \cdots \times \sigma_n'}$$

$$\emptyset \sim_U \emptyset$$

$$\frac{\Gamma \sim_B \Gamma' \quad \sigma \sim_U \sigma' \quad x \notin dom(\Gamma')}{(\Gamma, local(x : \sigma)) \sim_U (\Gamma', local(x : \sigma'))}$$

$$\frac{\Gamma \sim_U \Gamma' \quad \overline{s} \cap dom(\Gamma') = \emptyset}{(\Gamma, arg(\overline{x : \tau})) \sim_U (\Gamma', arg(\overline{x : \tau}))}$$

$$\frac{\Gamma \sim_U \Gamma' \quad \overline{s} \cap dom(\Gamma') = \emptyset}{(\Gamma, tvar(\overline{t})) \sim_U (\Gamma', tvar(\overline{t}), arg(\overline{s : |t|}))}$$

Figure 4.7: Simulation Relations on Types and Contexts

Lemma 4.6 *Let Γ be an well-formed context and each $tvar(\overline{t})$ assumption of Γ is followed by a $arg(\overline{s : |t|})$ assumption, σ be a type. If $\Gamma \vdash_U \sigma$ then $\Gamma \vdash_{TU} \sigma \rightsquigarrow s$ always succeeds and $\Gamma \vdash_{TU} s : |\sigma|$.*

PROOF. Similar to lemma 3.6. □

We also define simulations relations $\sigma \sim_U \sigma'$ between source and target types, and $\Gamma \sim_U \Gamma'$ between source and target contexts for unboxed compilation in Figure 4.7. We also have the following properties for the simulation relations.

Lemma 4.7 *Let Γ be a well-formed context in λ^{ML} and Γ' be a context in λ^U. If $\Gamma \sim_U \Gamma'$, then Γ' is well-formed and for any $tvar(\overline{t})$ assumption of Γ', there must be a $arg(\overline{s : |t|})$ follows.*

PROOF. Similar to lemma 3.7 □

Lemma 4.8 *Let Γ be a well-formed context in λ^{ML}, Γ' be a context in λ^U, σ be a type in λ^{ML}, and σ' be a type in λ^U. If $\Gamma \sim_U \Gamma'$, $\sigma \sim_U \sigma'$ and $\Gamma \vdash_{ML} \sigma$ then $\Gamma' \vdash_U \sigma'$.*

PROOF. Similar to lemma 3.8 □

Lemma 4.9 *If $\Gamma \sim_U \Gamma'$ then $\Gamma(x) \sim_U \Gamma'(x)$ for all $x \in dom(\Gamma)$*

PROOF. Similar to lemma 3.9

□

Lemma 4.10 *The simulation relations on types are stable under monomorphic substitution. If $\sigma \sim_U \sigma'$ then for any monomophic substitution $S = [\overline{\tau}/\overline{t}]$, $S(\sigma) \sim_U S(\sigma')$.*

PROOF. Similar to lemma 3.10 $\square$

The compilation from λ^{ML} terms to λ^{U} terms preserves types as shows in the following theorem

Theorem 4.2 *Suppose $\Gamma \vdash_{ML} e : \sigma$. For any Γ' so that $\Gamma \sim_U \Gamma'$ the compilation algorithm succeeds as $\Gamma' \vdash_{TU} e \rightsquigarrow e'$ with $\Gamma' \vdash_{TU} e' : \sigma'$ where $\sigma \sim_U \sigma'$*

PROOF. This is proved by induction on derivation of the typing rule. We proceed by do case analysis on e.
Case $e = c^o$**.** Straightforward.

Case $e = x$**.** Suppose $\Gamma \vdash_{ML} x : \sigma$, then by the source typing rule for variable, we have $\Gamma(x) = \sigma$. Since $\Gamma \sim_U \Gamma'$, by Lemma 4.9, $\Gamma(x) \sim_B \Gamma'(x)$. Let $\sigma' = \Gamma'(x)$. By Lemma 4.6, we have $\Gamma' \vdash_{TU} \sigma' \rightsquigarrow s$ succeeds and $\Gamma' \vdash_{TU} s : |\sigma'|$. Applying the compilation rule for variable, the compilation succeeds as $\Gamma' \vdash_{TU} x \rightsquigarrow x^s$. By the λ^U typing rule for variable, we have $\Gamma' \vdash_{TU} x^s : \Gamma'(x)$. Since $\Gamma(x) \sim_U \Gamma'(x)$, we have the expected result.

Case $e = \lambda\overline{x : \tau}.e_1$**.** Suppose $\Gamma \vdash_{ML} \lambda\overline{x : \tau}.e_1 : \overline{\tau} \rightarrow \sigma_1$. This is derived from

$$\Gamma, arg(\overline{x : \tau}) \vdash_{ML} e_1 : \sigma_1.$$

By the context simulation relation, we have

$$(\Gamma, arg(\overline{x : \tau})) \sim_U (\Gamma', arg(\overline{x : \tau})).$$

Applying induction hypothesis for e_1, we have $\Gamma', arg(\overline{x : \tau}) \vdash_{TU} e_1 \rightsquigarrow e'_1$ succeeds with $\Gamma', arg(\overline{x : \tau}) \vdash_{TU} e'_1 : \sigma'_1$ where $\sigma_1 \sim_U \sigma'_1$. Applying the compilation algorithm for function, we have

$$\Gamma' \vdash_{TU} \lambda\overline{x : \tau}.e_1 \rightsquigarrow \lambda\overline{x : \tau}.e'_1$$

Applying the λ^U typing rule for function, we have $\Gamma' \vdash_{TU} \lambda\overline{x : \tau}.e'_1 : \overline{\tau} \rightarrow \sigma'$. Since $\sigma_1 \sim_U \sigma'_1$, by the simulation relation for type, we have $(\overline{\tau} \rightarrow \sigma_1) \sim_U (\overline{\tau} \rightarrow \sigma'_1)$ as desired.

Case $e = (e_1\ \overline{e_2})$**.** Suppose $\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \overline{\tau} \rightarrow \sigma \qquad \Gamma \vdash_{ML} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma}$$

Applying induction hypothesis for e_1 and $\overline{e_2}$, we have

- $\Gamma' \vdash_{TU} e_1 \rightsquigarrow e'_1$ succeeds with $\Gamma' \vdash_{TU} e'_1 : \sigma_1$ where $(\overline{\tau} \rightarrow \sigma) \sim_U \sigma_1$
- $\Gamma' \vdash_{TU} \overline{e_2} \rightsquigarrow \overline{e'_2}$ succeeds with $\Gamma' \vdash_{TU} \overline{e'_2} : \overline{\sigma_2}$ where $\overline{\tau} \sim_U \overline{\sigma_2}$.

By simulation relations on types, we have $\sigma_1 \equiv \overline{\tau} \rightarrow \sigma'$ where $\sigma \sim_U \sigma'$, and $\overline{\sigma_2} \equiv \overline{\tau}$. Since σ_2 is well-formed under Γ', we have $\Gamma' \vdash_U \overline{\tau}$. By Lemma 4.6, we have $\Gamma' \vdash_{TU} \overline{\tau} \rightsquigarrow \overline{e_s}$ succeeds and $\Gamma' \vdash_{TU} \overline{e_s} : \overline{|\tau|}$. Applying the compilation rule for lambda application, we have $\Gamma' \vdash_{TU} (e_1\ \overline{e_2}) \rightsquigarrow (e'_1\ \overline{e_2'^{e_s}})$. Applying λ^U typing rule for application, we obtain $\Gamma' \vdash_{TU} (e'_1\ \overline{e_2'^{e_s}}) : \sigma'$ where $\sigma \sim_B \sigma'$ as desired.

Case $e = \Lambda\overline{t}.e_1$**.** Suppose $\Gamma \vdash_{ML} \Lambda\overline{t}.e_1 : \forall\overline{t}.\sigma_1$ This is derived from $\Gamma, tvar(\overline{t}) \vdash_{ML} e_1 : \sigma_1$. Let $\overline{s}$ be a sequence of fresh variables, $\overline{s} \cap dom(\Gamma) = \emptyset$. By the context simulation relation, we have

$$(\Gamma, tvar(\overline{t})) \sim_U (\Gamma', tvar(\overline{t}), arg(\overline{s : |t|}))$$

Applying induction hypothesis for e_1, we have

$$\begin{array}{c} \Gamma', tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_{TU} e_1 \rightsquigarrow e_1' \text{ succeeds} \\ \Gamma', tvar(\overline{t}), arg(\overline{s : |t|}) \vdash_{TU} e_1 : \sigma_1' \\ \sigma_1 \sim_U \sigma_1' \end{array}$$

Applying the compilation algorithm for type abstraction, we obtain

$$\Gamma' \vdash_{TU} \Lambda\overline{t}.e_1 \rightsquigarrow \Lambda\overline{t}.\lambda\overline{s : |t|}.e_1'.$$

By the λ^U typing rule for type abstraction, we have

$$\Gamma' \vdash_{TU} \Lambda\overline{t}.\lambda\overline{s : |t|}.e_1' : \forall\overline{t}.\overline{|t|} \to \sigma_1'$$

Since $\sigma_1 \sim_U \sigma_1'$, by the simulation relation on type, we have $\forall\overline{t}.\sigma_1 \sim_U \forall\overline{t}.\overline{|t|} \to \sigma_1'$ as desired.

Case $e = (e_1\ \overline{\tau})$**.** Suppose $\Gamma \vdash_{ML} (e_1\ \overline{\tau}) : \sigma_1[\overline{\tau}/\overline{t}]$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \forall\overline{t}.\sigma_1 \quad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{\tau}) : \sigma_1[\overline{\tau}/\overline{t}]}$$

Applying induction hypothesis for e_1, we have $\Gamma' \vdash_{TU} e_1 \rightsquigarrow e_1'$ with $\Gamma' \vdash_{TU} e_1' : \sigma_2$ where $\forall\overline{t}.\sigma_1 \sim_U \sigma_2$. By the simulation relation for types, σ_2 must have form $\forall\overline{t}.\overline{|t|} \to \sigma_1'$ where $\sigma_1 \sim_U \sigma_1'$. Since $\Gamma \vdash_{ML} \overline{\tau}$, by Lemma 4.8, we have $\Gamma' \vdash_U \overline{\tau}$. By Lemma 4.7 and Lemma 4.6, $\Gamma' \vdash_{TU} \overline{\tau} \rightsquigarrow \overline{e_s}$ must succeeds and $\Gamma' \vdash_{TU} \overline{e_s} : \overline{|\tau|}$. Applying the compilation rule for type instantiation, we have $\Gamma' \vdash_{TU} (e_1\ \overline{\tau}) \rightsquigarrow (e_1'\ \overline{\tau}\ \overline{e_s})$. Applying the λ^U typing rule for type instantiation, we have $\Gamma' \vdash_{TU} (e_1'\ \overline{\tau}\ \overline{e_s}) : \sigma_1'[\overline{\tau}/\overline{t}]$. By Lemma 4.10, $\sigma_1[\overline{\tau}/\overline{t}] \sim_U \sigma_1'[\overline{\tau}/\overline{t}]$ as desired.

Case $e = (e_1, \ldots, e_n)$**.** Suppose $\Gamma \vdash_{ML} (e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n$. This is derived from $\Gamma \vdash_{ML} e_i : \sigma_i$, for all $1 \le i \le n$. Applying induction hypothesis for each e_i, we have $\Gamma \vdash_{TU} e_i \rightsquigarrow g_i$ succeeds with $\Gamma' \vdash_{TU} g_i : \sigma_i'$ where $\sigma_i \sim_U \sigma_i'$. Since σ_i is well-formed under Γ, by Lemma 4.8, we have $\Gamma' \vdash_U \sigma_i'$. By Lemma 4.7 and Lemma 4.6, $\Gamma' \vdash_{TU} \sigma_i' \rightsquigarrow s_i$ must succeeds and $\Gamma' \vdash_{TU} s_i : |\sigma_i'|$, for each i. Applying the compilation rule for record, we obtain $\Gamma' \vdash_{TU} (e_1, \ldots, e_n) \rightsquigarrow (g_1^{s_1}, \ldots, g_n^{s_n})$. Applying the λ^U typing rule for record, we have $\Gamma' \vdash_{TU} (g_1^{s_1}, \ldots, g_n^{s_n}) : \sigma_1' \times \cdots \times \sigma_n'$. By the simulation relation on types, we have $(\sigma_1 \times \cdots \times \sigma_n) \sim_U (\sigma_1' \times \cdots \times \sigma_n')$ as desired.

Case $e = \pi_i(e_1)$**.** Suppose $\Gamma \vdash_{ML} \pi_i(e_1) : \sigma_i$. This is derived from

$$\Gamma \vdash_{ML} e_1 : \sigma_1 \times \cdots \times \sigma_n$$

Applying the induction hypothesis for e_1, we have $\Gamma' \vdash_{TU} e_1 \rightsquigarrow e_1'$ with $\Gamma' \vdash_{TU} e_1' : \sigma'$ where $(\sigma_1 \times \cdots \times \sigma_n) \sim_U \sigma'$. Then σ' must have the form $\sigma_1' \times \cdots \times \sigma_n'$ where $\sigma_i \sim_U \sigma_j'$ for each j. Since σ_j is well-formed under Γ, by Lemma 4.8, we have $\Gamma' \vdash_U \sigma_j'$ for each j. By Lemma 4.7 and Lemma 4.6, $\Gamma' \vdash_{TU} \sigma_j' \rightsquigarrow s_j$ must succeed and $\Gamma' \vdash_{TU} s_j : |\sigma_j'|$, for each $1 \le j \le i$. Applying the compilation rule for projection, we have

$$\Gamma' \vdash_{TU} \pi_i(e_1) \rightsquigarrow \pi^{s_i}_{[s_1 + \cdots + s_{i-1}]}(e_1')$$

Applying the λ^U typing rule for offset, we have

$$\Gamma' \vdash_{TU} [s_1 + \cdots + s_{i-1}] : \| \sigma'_1, \ldots, \sigma'_{i-1} \|$$

Applying the λ^U typing rule for projection, we obtain

$$\Gamma' \vdash_{TU} \pi^{s_i}_{[s_1+\cdots+s_{i-1}]}(e'_1) : \sigma'_i$$

where $\sigma_i \sim_U \sigma'_i$ as desired.

Case $e = \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end}$**.** Suppose $\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2$. This is derived from

$$\frac{\Gamma \vdash_{ML} e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_{ML} e_2 : \sigma_2}{\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Applying induction hypothesis for e_1, we have $\Gamma' \vdash_{TU} e_1 \rightsquigarrow e'_1$ succeeds with $\Gamma' \vdash_{TU} e'_1 : \sigma'_1$ where $\sigma_1 \sim_U \sigma'_1$ By Lemma 4.7 and Lemma 4.6, $\Gamma' \vdash_{TU} \sigma'_1 \rightsquigarrow s$ must succeed and $\Gamma' \vdash_{TU} s : |\sigma'_1|$. By the simulation relation for context, we have

$$(\Gamma, local(x : \sigma_1)) \sim_U (\Gamma', local(x : \sigma'_1)).$$

Applying induction hypothesis for e_1, we obtain $\Gamma', local(x : \sigma'_1) \vdash_{TU} e_2 \rightsquigarrow e'_2$ succeeds with $\Gamma', local(x : \sigma'_1) \vdash_{TU} e'_2 : \sigma'_2$ where $\sigma_2 \sim_U \sigma'_2$. By the compilation algorithm, we have

$$\Gamma' \vdash_{TU} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} \rightsquigarrow \texttt{let}\ x^s : \sigma'_1 = e'_1\ \texttt{in}\ e'_2\ \texttt{end}.$$

Finally, by the λ^U typing rule for `let` expression, we have

$$\Gamma' \vdash_{TU} \texttt{let}\ x^s : \sigma'_1 = e'_1\ \texttt{in}\ e'_2\ \texttt{end} : \sigma'_2$$

where $\sigma_2 \sim_U \sigma'_2$ as desired. □

4.2.3 Transformation from λ^U terms to Λ^U terms

After obtaining the explicitly typed term in λ^U by the above compilation algorithm, we can easily get the target term in Λ^U by using the $erase_U(e)$ function which erases all type annotation in the given term e in λ^U. The $erase_U$ function can be inductively defined as follows

$$\begin{array}{rcl}
erase_U(c^0) & = & c^o \\
erase_U(x^s) & = & x^{erase_U(s)} \\
erase_U(\lambda \overline{x : \tau}.e) & = & \lambda \overline{x}.erase_U(e) \\
erase_U((e_1\ \overline{e_2{}^{e_s}})) & = & (erase_U(e_1)\ \overline{erase_U(e_2)^{erase_U(e_s)}}) \\
erase_U(\Lambda \overline{t}.\lambda \overline{x : \langle t \rangle}.e) & = & \lambda \overline{x}.erase_U(e) \\
erase_U((e\ \overline{\tau}\ \overline{e_b})) & = & (erase_U(e)\ \overline{erase_U(e_b)^{|1|}}) \\
erase_U((e_1^{s_1}, \ldots, e_n^{s_n})) & = & (erase_U(e_1)^{erase_U(s_1)}, \ldots, erase_U(e_n)^{erase_U(s_n)}) \\
erase_U(\pi^s_o(e)) & = & \pi^{erase_U(s)}_{erase_U(o)}(erase_U(e)) \\
erase_U(\texttt{let}\ x^s : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let}\ x^{erase_U(s)} = erase_U(e_1)\ \texttt{in}\ erase_U(e_2)\ \texttt{end} \\
erase_U([e_1 + \cdots + e_n]) & = & [erase_U(e_1) + \cdots + erase_U(e_n)] \\
erase_U(|B|) & = & |B|
\end{array}$$

The function $erase_U$ just simply eliminates type annotation in the given term. Typing derivation should be the same for the given term and for the result. This property is shown by the following theorem

Theorem 4.3 *Suppose* Γ, e, σ *is a well-formed context, a term and a well-formed type in* λ^U *so that* $\Gamma \vdash_{TU} e : \sigma$. *Then we also have* $\Gamma \vdash_U erase_U(e) : \sigma$.

PROOF. As the same as in Theorem 3.3, this can be proved straightforwardly by induction on the derivation of the typing rule $\Gamma \vdash_{TU} e : \sigma$. $\square$

This result together with the type soundness theorem of the target calculus (Theorem 4.1) and the type preservation theorem (Theorem 4.2) establishes that the type system of the source calculus is sound with respect to the operational semantics realized by λ^{ML} to λ^U to Λ^U compilation followed by evaluation of the compiled term.

Chapter 5

The Combined Algorithm

So far I have presented my compilation method in two separated directions for two related matters – generating bitmaps and supporting unboxed manipulation. Simplicity and readability are the reasons that made me choose this presentation strategy. In a practical implementation, we can not separate bitmap-passing compilation and unboxed compilation (e.g. implementing one after another) because of the mutual dependency among them and closure conversion.

Targeting to a practical implementation, in the rest of this chapter I present a combination of bitmap-passing compilation, unboxed compilation, and typed closure conversion. The combined method would be notationally complex, but it does not add much complexity to the resulting algorithm, and more important, it does not violate the required theoretical properties such as soundness of the target calculus and the correctness of the compilation algorithm.

In this chapter, I also present an extension of the combined compilation method for generating layout information (including bitmap) of stack frames.

5.1 Combination with Closure Conversion

Closure conversion is an important compilation step in a practical compiler. It plays the role of separating between data and code. A closure conversion algorithm transforms a function with free variables into a *closure* consisting of two parts: a *function's environment* encapsulating all free variables, and a *function's code* abstracted on the environment by replacing each occurrence of a free variable in the function with the corresponding reference to the environment. A typical run-time representation of a closure is a two-word heap block whose the first component is the pointer to the function's code and the second component is the pointer to environment block consisting of free variable's values.

As mentioned in Chapter 1, we encountered a mutual dependency problem between bitmap-passing compilation, unboxed compilation, and closure conversion. The major reasons of this problem are that extra abstractions (bit tag and size abstraction) inserted during bitmap-passing compilation and unboxed compilation require to be closure converted, and environment records of functions (generated by closure conversion) should be handled under bitmap-passing and unboxed compilation scheme. Thus these three compilation processes can not be implemented separately (i.e. one after another).

Due to these reasons, I develop a combined method that takes responsibility for doing the tasks of both three compilation methods. In this section, I present this combined

method in the following steps.

- introducing a target calculus, namely *bitmap-passing unboxed closure calculus* or Λ^{BUC} for short;
- establishing operational semantics for this calculus;
- developing the combined algorithm, namely *BUC transformation*, that transforms expressions in the source calculus (λ^{ML}) into expressions in Λ^{BUC}. Similar to bitmap-passing compilation and unboxed compilation, I also design an explicitly typed calculus, written λ^{BUC} which serves as the immediate language for the compilation method. The compilation process can be done in two phases: compilation from λ^{ML} terms into λ^{BUC} terms, and transformation from λ^{BUC} terms to Λ^{BUC} terms.

5.1.1 Bitmap-passing unboxed closure calculus – Λ^{BUC}

Syntax and Types

The set of terms of Λ^{BUC} is given by the following syntax.

e	$::=$	c^o	constant
	$\mid$	x^s	temporary variable (argument or local)
	$\mid$	$\mathtt{env}^e_e$	environment access (free variable)
	$\mid$	$\mathtt{code}(\overline{x}, e)$	function's code
	$\mid$	$\langle\!\langle e, e\rangle\!\rangle$	closure
	$\mid$	$(e\ \overline{e^e})$	application
	$\mid$	$(e; e^e, \ldots, e^e)$	record
	$\mid$	$\pi^e_e(e)$	projection
	$\mid$	$\mathtt{let}\ x^e = e\ \mathtt{in}\ e\ \mathtt{end}$	let binding
	$\mid$	$[e, \ldots, e]$	bitmap
	$\mid$	$[e + \cdots + e]$	offset
	$\mid$	$\langle B\rangle$	constant bit tag ($B \in \{0, 1, 00\}$)
	$\mid$	$\lvert S\rvert$	constant size ($S \in \{1, 2\}$)

Closure conversion generate code for a function by abstracting all free variables occurred in the function's body. This is done by replacing each occurrence of free variables with a reference to the function's environment. We consider a record representation for a function's environment consisting of free variables. Under the proposed unboxed approach, run-time presentation of the environment record is a heap block and consisting unboxed values. Similar to the case of projection in unboxed compilation, accessing to a free variable in the environment requires both offset and size of this variable. We formulate this by introducing environment access terms of the form $\mathtt{env}^s_o$ in syntax of the calculus where o and s are terms representing offset and size of the selected free variable. Other kinds of variables including arguments and local variables (introduced by `let` expression) are represented in the syntax by terms of form x^s as the same as in unboxed calculus.

Function's codes are represented by terms of the form $\mathtt{code}(\overline{x}, e)$. This term constructor can be used for representing the code generated from both lambda abstraction and type abstraction. In the case of type abstraction, bit tag and size parameters generated

from abstract type variables are uncurried and represented by the formal arguments of code.

A closure of the form $\langle\langle e_1, e_2\rangle\rangle$ is considered as a partial application of the code e_1 to the environment record e_2.

$(e_1\ \overline{e_2}^{\overline{e_s}})$ can represent the resulting term of either an ordinary lambda application or a type instantiation in the source calculus. In the case of lambda application $\overline{e_2}$ represent the actual parameters and $\overline{e_s}$ are size of them generated from their types. In the case of type instantiation, the transformation algorithm introduces actual bit tag and size parameters based on the instance type, they are uncurried and represented by $\overline{e_2}$. In this case we can discard $\overline{e_s}$ since they are alway $|1|$.

A records of the form $(e_{bm}; e_1^{s_1}, \ldots, e_2^{s_2})$ is a combined representation of record terms in bitmap-passing calculus and in unboxed calculus. e_{bm} is the bitmap of the record, $e_1, \ldots, e_n$ and $s_1, \ldots, s_n$ are fields and field sizes, respectively.

Projection, `let` expression, bitmap, and offset are represented as the same as in unboxed calculus and bitmap-passing calculus.

In the present of multi-word unboxed values, a "bit tag" information of a run-time object is no longer a machine bit. For unboxed floating point value which consists of two words, the corresponding bit tag is therefore two machine bits and represented in the syntax by $\langle 00\rangle$. For objects of single type, we use constant tag $\langle 0\rangle$, $\langle 1\rangle$ to denote bit tag of unboxed single values and pointers. Note that objects of double types are always unboxed, then we do not have a constant bit tag like $\langle 01\rangle$, $\langle 10\rangle$, or $\langle 11\rangle$. $|1|$ and $|2|$ denote sizes of single values and unboxed double values, respectively.

The set of types of the target calculus are defined as an union of the sets of bitmap-passing types, and unboxed types. In addition, for typing code which is actually an abstraction on free variables, we add a new category of types for code, i.e. $\sigma_e \to_C \sigma$ where σ_e stands for type of environment and σ represents type of the function.

We extend the function `tagOf`() for double unboxed base type o as `tagOf`$(o) = 00$ (two zero bits).

We also extend the function FTV for code types as

$$FTV(\sigma_1 \to_C \sigma_2) = FTV(\sigma_1) \cup FTV(\sigma_2)$$

Typing Environment

We define contexts in Λ^{BUC} slightly different from ones of previous calculus. A context in Λ^{BUC} is a tuple $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ where Γ_T is a sequence of type variables, and $\Gamma_T, \Gamma_A, \Gamma_L$ are sequences of type assumptions of the form $x : \sigma$ for recording type information of free variables, arguments and local variables, respectively.

We redefine the well-formednesses of types and contexts as follows.

- A type σ is well-formed under a context $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$,written $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} \sigma$, if all free type variables of σ are declared in Γ_T, i.e. $FTV(\sigma) \subseteq \Gamma_T$.
- A context $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ is well-formed, written $\vdash_{BUC} (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ if for any assumption $x : \sigma \in \Gamma_F$ or $x : \sigma \in \Gamma_A$ or $x : \sigma \in \Gamma_L$ then $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} \sigma$.

Typing Rules

A Λ^{BUC} term e has a type σ under a well-formed context $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$, written

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} c^o : o$$

$$\frac{x : \sigma \in \Gamma_A \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x^s : \sigma}$$

$$\frac{x : \sigma \in \Gamma_L \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x^s : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_i| \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} o : \| \sigma_1, \ldots, \sigma_{i-1} \| \\ \Gamma_F = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \quad i \leq n\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{env}^s_o : \sigma_i}$$

$$\frac{(\Gamma_T; (y_1 : \sigma_1, \ldots, y_n : \sigma_n); (\overline{x : \tau}); \emptyset) \vdash_{BUC} e : \sigma}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{code}(\overline{x}, e) : \sigma_1 \times \cdots \times \sigma_n \rightarrow_C \overline{\tau} \rightarrow \sigma}$$

$$\frac{((\Gamma_T, \overline{t}); (y_1 : \sigma_1, \ldots, y_n : \sigma_n); (\overline{b : \langle t \rangle}, \overline{s : |t|}); \emptyset) \vdash_{BUC} e : \sigma}{\begin{array}{l}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{code}(\overline{b, s}, e) : \\ \qquad \sigma_1 \times \cdots \times \sigma_n \rightarrow_C \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \rightarrow \sigma\end{array}}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \sigma_{env} \rightarrow_C \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_2 : \sigma_{env}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \langle\!\langle e_1, e_2 \rangle\!\rangle : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \overline{\tau} \rightarrow \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_2} : \overline{\tau} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_s} : \overline{|\tau|}\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e_1\ \overline{{e_2}^{e_s}}) : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \rightarrow \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{\tau} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_b} : \overline{\langle \tau \rangle} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_s} : \overline{|\tau|}\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e\ \overline{e_b, {e_s}^{|1|}}) : \sigma[\overline{\tau}/\overline{t}]}$$

Figure 5.1: Typing Rules of Λ^{BUC} (1)

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \sigma$$

This is derived from the set of rules given in Figure 5.1 and Figure 5.2.

In closure conversion we distinguish between free variables and temporary variables (by terms x^s and $\mathtt{env}^s_o$). Typing rules for these cases are therefore changed. For temporary variable, the typing rule is almost the same with the one in unboxed calculus, except that it only checks the appearance of x in Γ_A (for arguments), and in Γ_L (for local variables). The case for free variables (environment access) is different. Type assumption for free variables are recorded in Γ_F which models a run-time environment record. Thus we formulate the rule for free variable as the same as for projection (in which size and offset checking are involved). Variable names in Γ_F are not needed for type checking. We just keep this for technical development of the compilation algorithm presented later.

Since a code is a closed term, in the typing rules for it we discard all type assumptions of variables ($\Gamma_F, \Gamma_A, \Gamma_L$) when checking type of the body. Instead, we use type information of the abstracted environment and arguments given in the syntax.

We regard a closure as a partial application of a code of type $\sigma_{env} \rightarrow_C \sigma$ to an environment record of type σ_{env}. The typing rule for closure checks this type consistency.

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : \sigma_i \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s_i : |\sigma_i| \quad \text{for } 1 \leq i \leq n \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_{bm} : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e_{bm}; e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \sigma_1 \times \cdots \times \sigma_n \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} o : \| \sigma_1, \ldots, \sigma_{i-1} \| \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_i|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \pi_o^s(e) : \sigma_i}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \sigma_1 \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_1| \quad (\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x : \sigma_1)) \vdash_{BUC} e_2 : \sigma_2}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \texttt{let } x^s = e_1 \texttt{ in } e_2 \texttt{ end} : \sigma_2}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : \langle \sigma_i \rangle \ (1 \leq i \leq n)}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : |\sigma_i| \ (1 \leq i \leq n)}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|}$$

$$\frac{\texttt{tagOf}(\sigma) = B}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \langle B \rangle : \langle \sigma \rangle}$$

$$\frac{\texttt{sizeOf}(\sigma) = S}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} |S| : |\sigma|}$$

Figure 5.2: Typing Rules of Λ^{BUC} (2)

In the rule for records, bitmap of the record and sizes of fields are checked to ensure the consistency between record's fields and its layout information. For the rest of cases, typing rules are defined similarly to those of bitmap-passing calculus and unboxed calculus.

Type system of Λ^{BUC} is also stable under type substitution. In order to formalize this property, we re-define some related terms as follows.

- A substitution S respects to a well-formed context $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ if for all $t_1 \in dom(S) \cap \Gamma_T$ and for all $t_2 \in FTV(S(t_1))$ then t_2 must be located before t_1 in Γ_T
- An instantiation context $S((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L))$ is the context obtained from $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ by replacing each assumption $x : \sigma$ in $\Gamma_F, \Gamma_A, \Gamma_L$ with $x : S(\sigma)$.
- A ground context $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ is a context where $\Gamma_F, \Gamma_A, \Gamma_L$ only assign closed types to variables

Thus the substitution property holds as shown in the following lemma.

Lemma 5.1 *Suppose* $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \sigma$. *For any substitution S that respects* $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$, *we have* $S((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) \vdash_{BUC} e : S(\sigma)$

Semantics

Before establishing semantics of Λ^{BUC}, we define run-time objects which are evaluated from terms of the calculus. The sets of run-time values (ranged over by v) and run-time environments (ranged over by E) of Λ^{BUC} age given by the following grammar.

$$
\begin{array}{rcll}
v & ::= & c^o & \text{constant value} \\
& | & i & \text{integer} \\
& | & B & \text{bit tag} \\
& | & \mathtt{code}(\overline{x}, e) & \text{function code} \\
& | & \langle\!\langle v, v \rangle\!\rangle & \text{closure} \\
& | & (v; v, \ldots, v) & \text{record value} \\
& | & [v, \ldots, v] & \text{bitmap value} \\
& | & \mathit{wrong} & \text{runtime error} \\
B & ::= & \langle 0 \rangle \mid \langle 1 \rangle \mid \langle 00 \rangle & \text{bit tag value} \\
E_F & ::= & \emptyset \mid E, v & \text{environment for free variable} \\
E_A, E_L & ::= & \emptyset \mid E, x : v & \text{environment for temporary variable}
\end{array}
$$

Function's codes and type abstraction's codes are closed terms, they are first-class values in the runtime system. Closure is two word block whose components are pointers to the function's code and to the function's environment record. Bitmap of a closure must be constant $[1, 1]$, thus we omit this in the syntax of values. Record's values have the same layout information as in bitmap-passing calculus. The first value (first word) v_0 in a record $(v_0; v_1, \ldots, v_2)$ is the layout bitmap of the record. This is a sequence of tag bits of the forms $\{\langle 0 \rangle \mid \langle 1 \rangle \mid \langle 00 \rangle\}$. In our implementation, a bitmap value is computed by concatenating all bit tags (using logical bitwise operators). Offsets and sizes are evaluated to integer (denoted by i).

Terms are evaluated to values under a run-time environment of the form $(E_F; E_A; E_L)$ where E_F is an usual heap block (which may consist of unboxed values) achieved from function's environment, E_A, E_L is a special structure (stack frame) where variable's values are allocated at fixed positions. Accessing to a value in E_F, run-time system needs to know both the offset and the size of the value, but accessing to E_A and E_L, only size are required. For the sake of technical development, we represent E_F as a sequence of run-time values, omitting the layout bitmap. We also present $, E_A, E_L$ as a mapping that map from variable names to value of values.

The set of evaluation rules is given in Figure 5.3 and Figure 5.4.

In evaluation rule for arguments and local variables, we lookup the expected value in E_A and E_L, respectively. Since we assume E_A and E_L have a structure that assign variable's values in fix positions, we can find the expected value by variable name as key.

The case of $\mathtt{env}_o^s$ is different. In this case E_F is a heap block consisting of unboxed value. Thus we can get the expected value by using offset and size evaluated from o and s.

In the evaluation rule for application $(e_1\ \overline{e_2{}^{e_s}})$, the value of e_1 should be a closure. Function's environment is evaluated and the resulting value is used as the run-time environment of free variables (E_F) for running the function's code. We check the consistency of actual argument size and the values of size term by $\overline{v_s = \mathtt{sizeOf}(v_2)}$.

For records, we evaluate the bit tag of given fields to get the necessary portion of the layout bitmap. We also evaluate all field sizes and using the resulting values for checking the size consistency. This process is just a combination of the evaluation rules for records in bitmap-passing calculus and unboxed calculus.

Since we have passed the presentations of bitmap-passing compilation and unboxed compilation, the remain cases are intuitively understandable.

Note that in a practical implementation, all checking operations can be omitted. The soundness property of the target calculus can guarantee that such size conditions are

$$(E_F; E_A; E_L) \vdash_{BUC} c^o \Downarrow c^o$$

$$\frac{(E_F; E_A; E_L) \vdash_{BUC} s \Downarrow v_s \quad x \in dom(E_A) \quad v_s = \mathtt{sizeOf}(E_A(x))}{(E_F; E_A; E_L) \vdash_{BUC} x^s \Downarrow E_A(x)}$$

$$\frac{(E_F; E_A; E_L) \vdash_{BUC} s \Downarrow v_s \quad x \in dom(E_L) \quad v_s = \mathtt{sizeOf}(E_L(x))}{(E_F; E_A; E_L) \vdash_{BUC} x^s \Downarrow E_L(x)}$$

$$\frac{\begin{array}{c} E_F = \{v_1, \ldots, v_n\} \quad (E_F; E_A; E_L) \vdash_{BUC} o \Downarrow v_o \quad (E_F; E_A; E_L) \vdash_{BUC} s \Downarrow v_s \\ v_s = \mathtt{sizeOf}(v_i) \quad v_o = \mathtt{sizeOf}(v_1) + \cdots + \mathtt{sizeOf}(v_{i-1}) \end{array}}{(E_F; E_A; E_L) \vdash_{BUC} \mathtt{env}_o^s \Downarrow v_i}$$

$$(E_F; E_A; E_L) \vdash_{BUC} \mathtt{code}(\overline{x}, e) \Downarrow \mathtt{code}(\overline{x}, e)$$

$$\frac{(E_F; E_A; E_L) \vdash_{BUC} e_1 \Downarrow v_1 \quad (E_F; E_A; E_L) \vdash e_2 \Downarrow v_2}{(E_F; E_A; E_L) \vdash_{BUC} \langle\langle e_1, e_2 \rangle\rangle \Downarrow \langle\langle v_1, v_2 \rangle\rangle}$$

$$\frac{\begin{array}{c} (E_F; E_A; E_L) \vdash_{BUC} e_1 \Downarrow \langle\langle \mathtt{code}(\overline{x}, e), (v_{bm}; v_1^e, \ldots, v_n^e) \rangle\rangle \\ \overline{(E_F; E_A; E_L) \vdash_{BUC} \overline{e_2} \Downarrow \overline{v_2}} \quad \overline{(E_F; E_A; E_L) \vdash_{BUC} \overline{e_s} \Downarrow \overline{v_s}} \\ \overline{v_s = \mathtt{sizeOf}(v_2)} \quad ((v_1^e, \ldots, v_n^e); \overline{x : v_2}; \emptyset) \vdash_{BUC} e_0 \Downarrow v_0 \end{array}}{(E_F; E_A; E_L) \vdash_{BUC} (e_1 \ \overline{e_2{}^{e_s}}) \Downarrow v_0}$$

Figure 5.3: Operational Semantics of Λ^{BUC} (1)

always correct for a given well-typed term. Before giving this desired property, we define a set of typing rules for run-time values and run-time environments in Figure 5.6 and Figure 5.5.

The type system of the bitmap-passing closure calculus is sound with respects to the operational semantics. This property is shown by the following theorem.

Theorem 5.1 *Let $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ be a well-formed context, e be an expression, and σ be a type in Λ^{BUC}. Let S be a ground substitution and $\Gamma_T \subseteq dom(S)$. Let $(E_F; E_A; E_L)$ be a run-time environment so that $(E_F; E_A; E_L) \vdash_{BUC} S((\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L))$.*

If $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} e : \sigma$ and $(E_F; E_A; E_L) \vdash_{BUC} e \Downarrow v$ then $\models_{BUC} v : S(\sigma)$.

5.1.2 Explicitly Typed BUC calculus – λ^{BUC}

As the same as in bitmap-passing compilation and unboxed compilation, I define an explicitly typed calculus, written λ^{BUC}, which serves as the immediate language for the compilation algorithm. The set of term of this calculus is given in the following syntax.

e	$::=$	c^o	constant
	$\mid$	x^s	temporary variable (argument or local)
	$\mid$	$\mathtt{env}_e^e$	environment access (free variable)
	$\mid$	$\mathtt{fcode}(\sigma, \overline{x : \tau}, e)$	monomorphic function's code
	$\mid$	$\mathtt{tcode}(\sigma, \overline{t}, \overline{b : \langle t \rangle}, \overline{s : \lvert t \rvert}, e)$	polymorphic function's code
	$\mid$	$\langle\langle e, e \rangle\rangle$	closure
	$\mid$	$(e \ \overline{e^e})$	application
	$\mid$	$(e \ \overline{\tau} \ \overline{e} \ \overline{e})$	type instantiation and bit tag/size application

$$\frac{\begin{array}{c}(E_F;E_A;E_L)\vdash_{BUC} e_{bm}\Downarrow[B_1,\ldots,B_n]\\ (E_F;E_A;E_L)\vdash_{BUC} e_i\Downarrow v_i \quad (E_F;E_A;E_L)\vdash_{BUC} s_i\Downarrow v_i^s\\ \texttt{tagOf}(v_i)=B_i \text{ and } \texttt{sizeOf}(v_i)=v_i^s \text{ for all } 1\le i\le n\end{array}}{(E_F;E_A;E_L)\vdash_{BUC}(e_{bm};e_1^{s_1},\ldots,e_n^{s_n})\Downarrow([B_1,\ldots,B_n];v_1,\ldots,v_n)}$$

$$\frac{\begin{array}{c}(E_F;E_A;E_L)\vdash_{BUC} e\Downarrow(v_0;v_1,\ldots,v_n)\\ (E_F;E_A;E_L)\vdash_{BUC} o\Downarrow v_o \quad (E_F;E_A;E_L)\vdash_{BUC} s\Downarrow v_s\\ v_s=\texttt{sizeOf}(v_i) \quad v_o=\texttt{sizeOf}(v_1)+\cdots+\texttt{sizeOf}(v_{i-1})\end{array}}{(E_F;E_A;E_L)\vdash_{BUC}\pi_o^s(e)\Downarrow v_i}$$

$$\frac{\begin{array}{c}(E_F;E_A;E_L)\vdash_{BUC} e_1\Downarrow v_1 \quad (E_F;E_A;E_L)\vdash_{BUC} s\Downarrow v_s \quad v_s=\texttt{sizeOf}(v_1)\\ (E_F;E_A;(E_L,x:v_1))\vdash_{BUC} e_2\Downarrow v_2\end{array}}{(E_F;E_A;E_L)\vdash_{BUC}\texttt{let } x^s=e_1 \texttt{ in } e_2 \texttt{ end}\Downarrow v_2}$$

$$\frac{(E_F;E_A;E_L)\vdash e_i\Downarrow B_i \quad (1\le i\le n)}{(E_F;E_A;E_L)\vdash[e_1,\ldots,e_n]\Downarrow[B_1,\ldots,B_n]}$$

$$\frac{(E_F;E_A;E_L)\vdash e_j\Downarrow i_j \quad (1\le j\le n)}{(E_F;E_A;E_L)\vdash[e_1+\cdots+e_n]\Downarrow i_1+\cdots+i_n}$$

$$(E_F;E_A;E_L)\vdash B\Downarrow B \quad \text{If } B\in\{\langle 0\rangle,\langle 1\rangle,\langle 00\rangle\}$$

$$(E_F;E_A;E_L)\vdash |i|\Downarrow i$$

Figure 5.4: Operational Semantics of Λ^{BUC} (2)

e	$::=$	$(e;e^e,\ldots,e^e)$	record
	$\mid$	$\pi_e^e(e)$	projection
	$\mid$	$\texttt{let } x^e:\sigma=e \texttt{ in } e \texttt{ end}$	let binding
	$\mid$	$[e,\ldots,e]$	bitmap
	$\mid$	$[e+\cdots+e]$	offset
	$\mid$	$\langle B\rangle$	constant bit tag ($B\in\{0,1,00\}$)
	$\mid$	$\lvert S\rvert$	constant size ($S\in\{1,2\}$)

In this syntax, we distinguish two different kinds of code: code for ordinary application $\texttt{fcode}(\sigma,\overline{x:\tau},e)$, and code for type abstraction (together with bit tag/size abstraction) $\texttt{tcode}(\sigma,\overline{t},\overline{b:\langle t\rangle},\overline{s:|t|},e)$. By applying type erasure, we will obtain a single form of code in Λ^{BUC}.

We also distinguish two kinds of application: one for ordinary application $(e_1\ \overline{e_2{}^{e_s}})$ where $\overline{e_s}$ are size terms of $\overline{e_2}$, and other for type instantiation (together with bit tag/size application) $(e\ \overline{\tau}\ \overline{e_b}\ \overline{e_s})$ where $\overline{e_b}$ and $\overline{e_s}$ are bit tag and size arguments generated from $\overline{\tau}$.

Typing rules for λ^{BUC} are defined in Figure 5.7 and Figure 5.8.

5.1.3 λ^{ML} to λ^{BUC} transformation

The major key idea of closure conversion is to replace all occurrences of free variables in a function by the corresponding references to function's environment. In order to do this, when transforming a function, a conventional closure conversion first computes the set of all free variables, then constructs the function's environment, and finally performs the

$$\frac{\models_{BUC} E_F : \Gamma_F \quad \models_{BUC} E_A : \Gamma_A \quad \models_{BUC} E_L : \Gamma_L}{\models_{BUC} (E_F; E_A; E_L) : (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)}$$

$$\frac{\models_{BUC} v_i : \sigma_i \text{ for each } 1 \leq i \leq n}{\models_{BUC} (v_1, \ldots, v_n) : (x_1 : \sigma_1, \ldots, x_n : \sigma_n)}$$

$$\frac{\models_{BUC} v_i : \sigma_i \text{ for each } 1 \leq i \leq n}{\models_{BUC} (x_1 : v_1, \ldots, x_n : v_n) : (x_1 : \sigma_1, \ldots, x_n : \sigma_n)}$$

Figure 5.5: Typing rules on runtime environments

replacement process under a compile context that takes free variable assumptions from the the function's environment.

In the presence of bitmap-compilation and unboxed compilation, this ordinary transformation process is no longer correct due to the soundness of bit tag and size creation algorithms. As we have seen in previous two chapters, these two algorithms generate a bit tag/size from a type variable by looking up the corresponding bit tag/size parameter recorded in the compile environment. If we do not carefully construct the compile context when transforming the body of the function, these two algorithm may fail.

Let us recall the conventional free variable collection algorithm $FV(e)$ where e is a source expression.

$$\begin{array}{rcl}
FV(c^o) & = & \emptyset \\
FV(x) & = & \{x\} \\
FV(\lambda \overline{x : \tau}.e) & = & FV(e) - \overline{x} \\
FV((e_1\ \overline{e_2})) & = & FV(e_1) \cup FV(\overline{e_2}) \\
FV(\Lambda \overline{t}.e) & = & FV(e) \\
FV((e\ \overline{\tau})) & = & FV(e) \\
FV((e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} FV(e_i) \\
FV(\pi_i(e)) & = & FV(e) \\
FV(\texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & FV(e_1) \cup (FV(e_2) - \{x\})
\end{array}$$

This algorithm just returns the set of ordinary variables. In order to achieve the full set of free variables including bit tag and size ones, I define a new function FV^{BUC} as follows.

$$\begin{array}{rcl}
FV^{BUC}(e, (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)) & = & FV(e) \cup \{lookuptag(t, (\Gamma_F; \Gamma_A; \Gamma_L)) | \forall t \in \Gamma_T\} \\
& & \cup \{lookupsize(t, (\Gamma_F; \Gamma_A; \Gamma_L)) | \forall t \in \Gamma_T\} \\
lookuptag(t, \Gamma_F; \Gamma_A; \Gamma_L) & = & b \quad If \ \ (b : \langle t \rangle) \in \Gamma_F \ or \ (b : \langle t \rangle) \in \Gamma_A \ or \ (b : \langle t \rangle) \in \Gamma_L \\
lookupsize(t, \Gamma_F; \Gamma_A; \Gamma_L) & = & s \quad If \ \ (s : |t|) \in \Gamma_F \ or \ (s : |t|) \in \Gamma_A \ or \ (s : |t|) \in \Gamma_L
\end{array}$$

Similarly to the bit tag/size creation algorithms, FV^{BUC} will fail if $(b : \langle t \rangle)$ or $(s : |t|)$ is not recorded in the given context for some $t \in \Gamma_T$. Fortunately, the simulation relations over contexts can guarantee that this case never happens.

$$\models_{BUC} c^o : o$$

$$\models_{BUC} i : |\sigma| \quad \text{if } i = \texttt{sizeOf}(\sigma)$$

$$\models_{BUC} i : \| \sigma_1, \ldots, \sigma_n \| \quad \text{if } i = \texttt{sizeOf}(\sigma_1) + \cdots + \texttt{sizeOf}(\sigma_n)$$

$$\models_{BUC} \langle B \rangle : \langle \sigma \rangle \quad \text{if } B = \texttt{tagOf}(\sigma)$$

$$\frac{\models_{BUC} v_i : \langle \sigma_i \rangle \quad \text{for all } 1 \leq i \leq n}{\models_{BUC} [v_1, \ldots, v_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\{\sigma_1, \ldots, \sigma_n\} \text{ and } \overline{\tau} \text{ are closed types} \\ (\emptyset; (y_1 : \sigma_1, \ldots, y_n : \sigma_n); \overline{x : \tau}; \emptyset) \vdash_{BUC} e : \sigma\end{array}}{\models_{BUC} \texttt{code}(\overline{x}, e) : \sigma_1 \times \cdots \times \sigma_n \to_C \overline{\tau} \to \sigma}$$

$$\frac{\begin{array}{c}\{\sigma_1, \ldots, \sigma_n\} \text{ are closed types} \\ (\overline{t}; (y_1 : \sigma_1, \ldots, y_n : \sigma_n); (\overline{b : \langle t \rangle}, \overline{s : |t|}); \emptyset) \vdash_{BUC} e : \sigma\end{array}}{\models_{BUC} \texttt{code}(\overline{b}, \overline{s}, e) : \sigma_1 \times \cdots \times \sigma_n \to_C \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \to \sigma}$$

$$\frac{\models_{BUC} v_1 : \sigma_{env} \to_c \sigma \qquad \models_{BUC} v_2 : \sigma_{env}}{\models_{BUC} \langle\!\langle v_1, v_2 \rangle\!\rangle : \sigma}$$

$$\frac{\models_{BUC} v_i : \sigma_i \text{ for all } 1 \leq i \leq n \qquad \models_{BUC} v_{bm} : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\models_{BUC} (v_{bm}; v_1, \ldots, v_n) : \sigma_1 \times \cdots \times \sigma_n}$$

Figure 5.6: Typing rules on runtime values

Using this new free variable collection function, I develop the combined algorithm, namely *BUC transformation*, as a deduction system that derive compilation judgments of the form $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} e \rightsquigarrow e'$ where $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ is a context in λ^{BUC}, e is a source expression (in λ^{ML}), and e' is the target expression (in λ^{BUC}). The set of rules of this system is given in Figure 5.9 and Figure 5.10.

In the compilation rules for a variables, whether it will be transformed into a variable or an environment access in the target calculus is decided by looking up the appearance of this variable in Γ_A, Γ_L or in Γ_F.

In the rule for generating code (from lambda abstraction or type abstraction), FV^{BUC} collects all free variables including bit tag and size ones. The environment record is constructed based on this information. Body of the function's code will be generated under a context containing all free variable assumptions.

In the rules for type abstractions, bit tag and size arguments are generated and inserted based on the instance types by applying bit tag/size creation algorithms. In the rule for records, layout information and size information are also generated from types of the fields. For projection, offset and size of the selected field are generated from type of the record. Other cases are easily understandable.

The compilation algorithm preserves typing. Before showing this desired properties we redefine the simulation relations between source and target contexts as follows (simulation relations over types are similarly defined as in previous two chapters).

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} c^o : o$$

$$\frac{x:\sigma \in \Gamma_A \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x^s : \sigma}$$

$$\frac{x:\sigma \in \Gamma_L \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x^s : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_i| \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} o : \|\sigma_1,\ldots,\sigma_{i-1}\| \\ \Gamma_F = \{x_1:\sigma_1,\ldots,x_n:\sigma_n\} \quad i \le n\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{env}_o^s : \sigma_i}$$

$$\frac{(\Gamma_T;(y_1:\sigma_1,\ldots,y_n:\sigma_n);(\overline{x:\tau});\emptyset) \vdash_{BUC} e : \sigma}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{fcode}(\sigma_1\times\cdots\times\sigma_n, \overline{x:\tau}, e) : \sigma_1\times\cdots\times\sigma_n \to_C \overline{\tau} \to \sigma}$$

$$\frac{((\Gamma_T,\overline{t});(y_1:\sigma_1,\ldots,y_n:\sigma_n);(\overline{b:\langle t\rangle},\overline{s:|t|});\emptyset) \vdash_{BUC} e : \sigma}{\begin{array}{l}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{tcode}(\sigma_1\times\cdots\times\sigma_n, \overline{t}, \overline{b:\langle t\rangle}, \overline{s:|t|}, e) : \\ \qquad\qquad \sigma_1\times\cdots\times\sigma_n \to_C \forall\overline{t}.\{\overline{\langle t\rangle},\overline{|t|}\} \to \sigma\end{array}}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \sigma_{env} \to_C \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_2 : \sigma_{env}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \langle\!\langle e_1, e_2\rangle\!\rangle : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \overline{\tau} \to \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_2} : \overline{\tau} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_s} : \overline{|\tau|}\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e_1\ \overline{e_2}^{\overline{e_s}}) : \sigma}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \forall\overline{t}.\{\overline{\langle t\rangle},\overline{|t|}\} \to \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{\tau} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_b} : \overline{\langle\tau\rangle} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \overline{e_s} : \overline{|\tau|}\end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e\ \overline{\tau}\ \overline{e_b}\ \overline{e_s}) : \sigma[\overline{\tau}/\overline{t}]}$$

Figure 5.7: Typing Rules of λ^{BUC} (1)

$$\emptyset \sim_{BUC} (\emptyset;\emptyset;\emptyset;\emptyset)$$

$$\frac{\Gamma \sim_{BUC} (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \quad \overline{x} \cap dom(\Gamma_F \oplus \Gamma_A \oplus \Gamma_L) = \emptyset}{(\Gamma, arg(\overline{x:\tau})) \sim_{BUC} (\Gamma_T;\Gamma_F \oplus \Gamma_A \oplus \Gamma_L;\overline{x:\tau};\emptyset)}$$

$$\frac{\Gamma \sim_{BUC} (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \quad x \notin dom(\Gamma_F \oplus \Gamma_A \oplus \Gamma_L) \quad \sigma \sim_{BUC} \sigma'}{(\Gamma, local(x:\sigma)) \sim_{BUC} (\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x:\sigma'))}$$

$$\frac{\Gamma \sim_{BUC} (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \quad \overline{t} \cap TV(\Gamma_T) = \emptyset \quad \overline{b} \cap dom(\Gamma_F \oplus \Gamma_A \oplus \Gamma_L) = \emptyset}{(\Gamma, tvar(\overline{t})) \sim_{BUC} ((\Gamma_T,\overline{t});\Gamma_F \oplus \Gamma_A \oplus \Gamma_L;\overline{b:\langle t\rangle};\emptyset)}$$

The binary operator $\oplus$ denote concatenations of two type assumptions. In this case for $arg()$ and $tvar()$ (corresponding to the case of generating code in the compilation algorithm) we choose a maximum set of assumptions for Γ_F, this must be a superset of any free variable set of well-typed term under the target context. This guarantee the compilation algorithm never fail.

Type preservation property of the compilation algorithm is given as the following theorem.

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : \sigma_i \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s_i : |\sigma_i| \quad \text{for } 1 \le i \le n \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_{bm} : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e_{bm}; e_1^{s_1}, \ldots, e_n^{s_n}) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e : \sigma_1 \times \cdots \times \sigma_n \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} o : \| \sigma_1, \ldots, \sigma_{i-1} \| \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_i|}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \pi_o^s(e) : \sigma_i}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_1 : \sigma_1 \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} s : |\sigma_1| \quad (\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x : \sigma_1)) \vdash_{BUC} e_2 : \sigma_2}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \texttt{let } x^s : \sigma_1 = e_1 \texttt{ in } e_2 \texttt{ end} : \sigma_2}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : \langle \sigma_i \rangle \ (1 \le i \le n)}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e_i : |\sigma_i| \ (1 \le i \le n)}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} [e_1 + \cdots + e_n] : \| \sigma_1, \ldots, \sigma_n \|}$$

$$\frac{\texttt{tagOf}(\sigma) = B}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \langle B \rangle : \langle \sigma \rangle}$$

$$\frac{\texttt{sizeOf}(\sigma) = S}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} |S| : |\sigma|}$$

Figure 5.8: Typing Rules of λ^{BUC} (2)

Theorem 5.2 *Suppose* $\Gamma \vdash_{ML} e : \sigma$, *for any* $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ *so that* $\Gamma \sim_{BUC} (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ *then* $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} e \rightsquigarrow e'$ *succeeds, and* $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{TBUC} e' : \sigma'$ *where* $\sigma \sim_{BUC} \sigma'$

5.1.4 λ^{BUC} to Λ^{BUC} transformation

We define a type erasure function that eliminates all type annotation in λ^{BUC} terms to obtain Λ^{BUC} terms as follows.

$$\begin{array}{rcl}
erase_{BUC}(c^0) & = & c^o \\
erase_{BUC}(x^s) & = & x^{erase_{BUC}(s)} \\
erase_{BUC}(\texttt{env}_o^s) & = & \texttt{env}_{erase_{BUC}(o)}^{erase_{BUC}(s)} \\
erase_{BUC}(\texttt{fcode}(\sigma_{env}, \overline{x : \tau}, e)) & = & \texttt{code}(\overline{x}, erase_{BUC}(e)) \\
erase_{BUC}(\texttt{tcode}(\sigma_{env}, \overline{t}, \overline{s : \langle t \rangle}, \overline{b : |t|}, e)) & = & \texttt{code}(\overline{(s,b)}, erase_{BUC}(e)) \\
erase_{BUC}((e_1\ \overline{e_2{}^{e_s}})) & = & (erase_{BUC}(e_1)\ \overline{erase_{BUC}(e_2)^{erase_{BUC}(e_s)}}) \\
erase_{BUC}((e\ \overline{\tau}\ \overline{e_b}\ \overline{e_s})) & = & (erase_{BUC}(e)\ \overline{(erase_{BUC}(e_b), erase_{BUC}(e_s))^{|1|}}) \\
erase_{BUC}(\langle\!\langle e_1, e_2 \rangle\!\rangle) & = & \langle\!\langle erase_{BUC}(e_1), erase_{BUC}(e_2) \rangle\!\rangle \\
erase_{BUC}((e_{bm}; e_1^{s_1}, \ldots, e_n^{s_n})) & = & (e'_{bm}; e_1'^{s'_1}, \ldots, e_2'^{s'_2}) \\
 & & e'_{bm} = erase_{BUC}(e_{bm}) \\
 & & e'_i = erase_{BUC}(e_i), s'_i = erase_{BUC}(s_i)
\end{array}$$

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} c^o \leadsto c^o$$

$$\frac{x:\sigma \in \Gamma_A \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash^S_{BUC} \Gamma_A(x) \leadsto s}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x \leadsto x^s}$$

$$\frac{x:\sigma \in \Gamma_L \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash^S_{BUC} \Gamma_L(x) \leadsto s}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x \leadsto x^s}$$

$$\frac{\begin{array}{c} x:\sigma \text{ is the } i^{th} \text{ element of } \Gamma_F \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash^S_{BUC} \Gamma_F(j) \leadsto s_j \text{ for each } 1 \le j \le i \end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x \leadsto \mathtt{env}^{s_i}_{[s_1+\cdots+s_{i-1}]}}$$

$$\frac{\begin{array}{c} FV^{BUC}(e,(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) = \{y_1,\ldots,y_n\} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (y_1,\ldots,y_n) \leadsto e_{env} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{TBUC} (y_1,\ldots,y_n) : \sigma_1 \times \cdots \times \sigma_n \\ (\Gamma_T;(\{y_1:\sigma_1,\ldots,y_n:\sigma_n\});\overline{x:\tau};\emptyset) \vdash_{BUC} e \leadsto e' \end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \lambda\overline{x:\tau}.e \leadsto \langle\!\langle \mathtt{fcode}(\sigma_1 \times \cdots \times \sigma_n, \overline{x:\tau}, e'), e_{env} \rangle\!\rangle}$$

$$\frac{\begin{array}{c} FV^{BUC}(e,(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) = \{y_1,\ldots,y_n\} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (y_1,\ldots,y_n) \leadsto e_{env} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{TBUC} (y_1,\ldots,y_n) : \sigma_1 \times \cdots \times \sigma_n \\ \overline{b} \text{ and } \overline{s} \text{ are fresh variables} \\ ((\Gamma_T,\overline{t});(\{y_1:\sigma_1,\ldots,y_n:\sigma_n\}),(\overline{b:\langle t\rangle},\overline{s:|t|}),\emptyset) \vdash_{BUC} e \leadsto e' \end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \Lambda\overline{t}.e \leadsto \langle\!\langle \mathtt{tcode}(\sigma_1 \times \cdots \times \sigma_n, \overline{t}, \overline{b:\langle t\rangle}, \overline{s:|t|}, e'), e_{env} \rangle\!\rangle}$$

Figure 5.9: BUC transformation (1)

$$\begin{array}{rcl}
erase_{BUC}(\pi^s_o(e)) & = & \pi^{erase_{BUC}(s)}_{erase_{BUC}(o)}(erase_{BUC}(e)) \\
erase_{BUC}(\mathtt{let}\ x^s:\sigma = e_1\ \mathtt{in}\ e_2\ \mathtt{end}) & = & \mathtt{let}\ x^{s'} = e'_1\ \mathtt{in}\ e'_2\ \mathtt{end} \\
& & s' = erase_{BUC}(s) \\
& & e'_1 = erase_{BUC}(e_1), e'_2 = erase_{BUC}(e_2) \\
erase_{BUC}([e_1+\cdots+e_n]) & = & [erase_{BUC}(e_1)+\cdots+erase_{BUC}(e_n)] \\
erase_{BUC}([e_1,\ldots,e_n]) & = & [erase_{BUC}(e_1),\ldots,erase_{BUC}(e_n)] \\
erase_{BUC}(|S|) & = & |S| \\
erase_{BUC}(\langle B\rangle) & = & \langle B\rangle
\end{array}$$

The function $erase_{BUC}$ just simply eliminates type annotation in the given term. Typing derivation should be the same for the given term and for the result. This property is shown by the following theorem

Theorem 5.3 *Suppose Γ, e, σ is a well-formed context, a term and a well-formed type in λ^{BUC} so that $\Gamma \vdash_{TBUC} e : \sigma$. Then we also have $\Gamma \vdash_{BUC} erase_{BUC}(e) : \sigma$.*

This result together with the type soundness theorem of the target calculus (Theorem 5.1) and the type preservation theorem (Theorem 5.2) establishes that the type system of the source calculus is sound with respect to the operational semantics realized by λ^{ML} to λ^{BUC} to Λ^{BUC} compilation followed by evaluation of the compiled term.

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} e_1 \leadsto e_1' \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} \overline{e_2} \leadsto \overline{e_2'} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{TBUC} \overline{e_2} : \overline{\tau} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} \overline{\tau} \leadsto \overline{e_s}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} (e_1\ \overline{e_2}) \leadsto (e_1'\ \overline{e_2'^{e_s}})}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} e \leadsto e' \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{B}_{BUC} \overline{\tau} \leadsto \overline{e_b} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{S}_{BUC} \overline{\tau} \leadsto \overline{e_s}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} (e\ \overline{\tau}) \leadsto (e'\ \overline{\tau}\ \overline{e_b}\ \overline{e_s})}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} e_i \leadsto g_i \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{TBUC} g_i : \sigma_i \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{B}_{BUC} \sigma_i \leadsto b_i \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{S}_{BUC} \sigma_i \leadsto s_i \quad (1 \le i \le n)}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} (e_1,\ldots,e_n) \leadsto ([b_1,\ldots,b_n]; g_1^{s_1},\ldots,g_n^{s_n})}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} e \leadsto e' \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{TBUC} e : \sigma_1 \times \cdots \times \sigma_n \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{S}_{BUC} \sigma_j \leadsto s_j \text{ for } 1 \le j \le i}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} \pi_i(e) \leadsto \pi^{s_i}_{[s_1+\cdots+s_{i-1}]}(e')}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} e_1 \leadsto e_1' \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{TBUC} e_1 : \sigma' \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash^{S}_{BUC} \sigma' \leadsto s \quad (\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x:\sigma'))\vdash_{TB} e_2 \leadsto e_2'}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)\vdash_{BUC} \texttt{let}\ x:\sigma = e_1\ \texttt{in}\ e_2\ \texttt{end} \leadsto \texttt{let}\ x^s:\sigma' = e_1'\ \texttt{in}\ e_2'\ \texttt{end}}$$

Figure 5.10: BUC tranformation (2)

5.2 Extension for Generating Stack Frame Layout

We have seen how bitmap information for heap-allocated objects are generated in Chapter 3 and in previous section. For implementing a correct bitmap-passing garbage collector, bitmap information of temporary variables used in each function also need to be computed.

We consider a stack-based implementation, where a stack frame is allocated for each function, and all the temporary variables used by the function are allocated in the stack frame. The function code is implemented by a sequence of instructions in the format $x = op(x_1, \ldots, x_n)$ where $x, x_1, \ldots, x_n$ are locations in a stack frame relative to the frame pointer. Our strategy of compiling a source expression into an efficient target code of the above format is outlined as follows.

1. compiling the source expression into a Λ^{BUC} expression.
2. A-normalizing the Λ^{BUC} expression by adopting the algorithm presented in [FSDF93].
3. minimizing the number of stack frame slots and eliminating dead code fragments by applying a liveness analysis.
4. generating instructions from the A-normalized Λ^{BUC} expression achieved from the above steps.

Since bitmap information of temporary variables are required, we need to refine BUC transformation for generating such information. In the rest of this section, I will present a refined algorithm that can predict the stack frame layout without performing A-normalization.

As we have mentioned, bit tag information of temporary variables (arguments and local variables) of each function code are needed for garbage collection. These information can be computed from the types of temporary variables by the following strategy.

Firstly, we type each temporary variables with one of the following types.

$$\tau ::= \textit{boxed} \mid \textit{unboxed} \mid t$$

where t stands for type variables. Let $\{t_1, \ldots, t_n\}$ be the set of type variables appearing as types of temporary variables located in a stack frame (we shortly write "appearing in the stack frame"). We represent the layout information of the stack frame by the following data:

1. the number of slots of type *unboxed*,
2. the number of slots of type *boxed*,
3. a bitmap of type $\langle\langle t_1, \ldots, t_n \rangle\rangle$, and
4. the number of slots of each type t_i.

Secondly, we refine the bitmap-passing closure conversion algorithm to compute the necessary layout information. Among the described items, only the bitmap of type $\langle\langle t_1, \ldots, t_n \rangle\rangle$ needs to be concerned. Other information can be computed statically. The syntaxes of λ^{BUC} and Λ^{BUC} are refined so that a code expression includes its layout information. We re-define the syntax of function code and type abstraction codes in λ^{BUC} as $\texttt{fcode}(e_b, \sigma, \overline{x : \tau}, e)$ and $\texttt{tcode}(e_b, \sigma, \overline{t}, \overline{b : \langle t \rangle}, \overline{s : |t|}, e)$, respectively. The syntax of code in Λ^{BUC} is also refined as $\texttt{code}(\overline{e_b, x}, e)$. Addition item e_b in these syntaxes represents layout information of stack frame of the code. Here, this is the bitmap of type $\langle\langle t_1, \ldots, t_n \rangle\rangle$ for $\{t_1, \ldots, t_n\}$ are the set of type variable appearing in the stack frame. We refine BUC transformation algorithm (from λ^{ML} to λ^{BUC}) for generating e_b by the following steps.

1. collecting all type variables $\{t_1, \ldots, t_n\}$ appearing in the stack frame;
2. translating these type variables into a bit tag terms $\{e^b_1, \ldots, e^b_n\}$ by using bit tag creation algorithm.
3. composing the expected bitmap, i.e. $[e^b_1, \ldots, e^b_n]$.

All these steps need to be done at BUC transformation (before A-normalization). However, step (1) requires information of temporary variables residing in stack frame which is only known after A-normalization. We seems to have another mutual dependency problem. Fortunately, we do not need to combine BUC transformation and A-normalization, just an extension is enough to solve this problem.

We know that A-normalization algorithm translates terms in tree structure into terms in sequential structure and each node of the tree term will be assigned to a temporary variable. Thus we can predict the set of type variables appearing in the stack frame by examining types of function's arguments (which also reside in stack frame) and types of all sub-terms of the function body.

We define two functions $TVType(\sigma)$ and $TVTerm(\Gamma, e)$ for predicting layout information. The former returns a singleton set of σ if σ is a type variable. The latter collects

the set of type variables appearing as types of sub-terms of e including itshelf. Figure 5.11 gives the set of rules for implementing $TVTerm$.

In these rules, we do not inspect structure of bit tag, size, bitmap, offset expressions. These expressions do not contain any sub-expression which has type variable type because they are generated as a composition of operations on objects of single unboxed types, i.e. bit tag, size, bitmap, offset types.

In the rules for codes (`fcode` and `tcode`), we do not inspect the body of the code since the sub-expressions of the body will be only bound to temporary variables of this code's strack frame, but not bound to any temporary variable of current code's stack frame.

By using these functions, the two compilation rules for generating codes in BUC transformation would be re-defined for predicting layout information as follows.

$$\frac{\begin{array}{c} FV^{BUC}(e, (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)) = \{y_1, \ldots, y_n\} \\ (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} (y_1, \ldots, y_n) \rightsquigarrow e_{env} \\ (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} (y_1, \ldots, y_n) : \sigma_1 \times \cdots \times \sigma_n \\ \Gamma = (\Gamma_T; (\{y_1 : \sigma_1, \ldots, y_n : \sigma_n\}); \overline{x : \tau}; \emptyset) \quad \Gamma \vdash_{BUC} e \rightsquigarrow e' \\ T_1 = TVTerm(\Gamma, e') \quad T_2 = \cup\overline{TVType(\tau)} \\ \Gamma \vdash_{BUC} \langle\!\langle t_1, \ldots, t_m \rangle\!\rangle \rightsquigarrow e_b \text{ for } \{t_1, \ldots, t_n\} = T_1 \cup T_2 \end{array}}{(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} \lambda\overline{x : \tau}.e \rightsquigarrow \langle\!\langle \mathtt{fcode}(e_b, \sigma_1 \times \cdots \times \sigma_n, \overline{x : \tau}, e'), e_{env} \rangle\!\rangle}$$

$$\frac{\begin{array}{c} FV^{BUC}(e, (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)) = \{y_1, \ldots, y_n\} \\ (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} (y_1, \ldots, y_n) \rightsquigarrow e_{env} \\ (\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} (y_1, \ldots, y_n) : \sigma_1 \times \cdots \times \sigma_n \\ \Gamma = ((\Gamma_T, \overline{t}); (\{y_1 : \sigma_1, \ldots, y_n : \sigma_n\}), (\overline{b : \langle t \rangle}, \overline{s : |t|}), \emptyset) \quad \overline{b} \text{ and } \overline{s} \text{ are fresh variables} \\ \Gamma \vdash_{BUC} e \rightsquigarrow e' \\ \{t_1, \ldots, t_n\} = TVTerm(\Gamma, e') \quad \Gamma \vdash_{BUC} \langle\!\langle t_1, \ldots, t_m \rangle\!\rangle \rightsquigarrow e_b \end{array}}{(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L) \vdash_{BUC} \Lambda\overline{t}.e \rightsquigarrow \langle\!\langle \mathtt{tcode}(e_b, \sigma_1 \times \cdots \times \sigma_n, \overline{t}, \overline{b : \langle t \rangle}, \overline{s : |t|}, e'), e_{env} \rangle\!\rangle}$$

The refinement does not put any new constraint on typing system of BUC. This implies that the type preservation property still holds for the refined algorithm. In addition, if the compilation phases after A-normalization do not introduce any new temporary variables whose bit tag information are not described by the generated frame layout, the refined algorithm should be correct in the sense of stack frame layout. In this case we can safely say that the type system of the source calculus is sound with respect to an operational semantics implementing by a series of type-preserving compilation phases (including the refined bitmap-passing closure conversion and A-normalization) and following by an execution of generated instructions.

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), c^0) = \emptyset$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} x^s : \sigma}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), x^s) = TVType(\sigma)}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{env}_o^s : \sigma}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \mathtt{env}_o^s) = TVType(\sigma)}$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \mathtt{fcode}(e_b, \sigma, \overline{x:\tau}, e) = \emptyset$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \mathtt{tcode}(e_b, \sigma, \bar{t}, \overline{b:\langle t\rangle}, \overline{s:|t|}, e) = \emptyset$$

$$\frac{T_1 = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_1) \quad T_2 = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_2)}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \langle\!\langle e_1, e_2\rangle\!\rangle) = T_1 \cup T_2}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e_1\ \overline{e_2}) : \sigma \\ T_1 = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_1) \quad T_2 = \bigcup \overline{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_2)}\end{array}}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), ((e_1\ \overline{e_2}^{e_s})) = T_1 \cup T_2 \cup TVType(\sigma)}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} (e\ \overline{\tau}\ \overline{e_b}\ \overline{e_s}) : \sigma \quad T = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e)}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), ((e\ \overline{\tau}\ \overline{e_b}\ \overline{e_s})) = T \cup TVType(\sigma)}$$

$$\frac{T_i = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_i) \text{ for all } 1 \le i \le n}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), (e_{bm}; e_1^{s_1}, \ldots, e_2^{s_2})) = T_1 \cup \cdots \cup T_n}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \pi_o^s(e) : \sigma \quad T = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e)}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \pi_o^s(e)) = T \cup TVType(\sigma)}$$

$$\frac{\begin{array}{c}(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BUC} \mathtt{let}\ x^s : \sigma_1 = e_1\ \mathtt{in}\ e_2\ \mathtt{end} : \sigma_2 \\ T_1 = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e_1) \quad T_2 = TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x:\sigma_1)), e_2)\end{array}}{TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \mathtt{let}\ x^s : \sigma_1 = e_1\ \mathtt{in}\ e_2\ \mathtt{end}) = T_1 \cup T_2 \cup TVType(\sigma_2)}$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), \langle B\rangle) = \emptyset$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), |S|) = \emptyset$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), [e_1, \ldots, e_n]) = \emptyset$$

$$TVTerm((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), [e_1 + \cdots + e_n]) = \emptyset$$

Figure 5.11: Algorithm for predicting stack frame layout

Chapter 6

Separate Compilation and Module Language

In previous chapters, I have shown how the proposed compilation method can generate code with layout bitmap and information for manipulating unboxed values. We have considered a mini ML-style calculus with rank-1 polymorphism as the source calculus in the compilation scheme. Extending this to the full set of ML Core language would be technically easy. However, for ML module language with functors and for separate compilation where user-define type variables may appear free in a separate module, we need to make more efforts for compiling the language.

In order to see the problems related to separate compilation and module language, let me first recall the bit tag creation algorithm and the size creation algorithm presented in Chapter 3, Chapter 4, and Chapter 5. Bit tag creation and size creation algorithms have forms $\Gamma \vdash^{B}_{BUC} \sigma \rightsquigarrow b$ and $\Gamma \vdash^{S}_{BUC} \sigma \rightsquigarrow s$, respectively. They generate bit tag/size term corresponding to the given type σ. In the case σ has a proper outermost type constructor, they will return a constant bit tag/size corresponding to this type. If σ is an abstract type variable, i.e. t, the algorithms will looking for the corresponding bit tag/size variable recorded in the context Γ. For ML Core language, where a given source term to be compiled is closed in sense of type, the proposed compilation algorithm sould be sound, i.e. it should not fail for a given well-typed source term. The reason that guarantee the soundness of the proposed algorithm is that when compiling a type abstraction term, the algorithms always assume a set of bit tag/size parameters (by recording them in the context) for compiling the body of the type abstraction. Thus we can always find a bit tag/size parameter corresponding to a type variable introduced by type abstractions, the bit tag/size creation algorithm therefore always succeeds.

In the presence of functors and separate compilation, we face a subtle problem, type variables are not only introduced by type abstractions, but also introduced by user-define abstract type definitions. Let's consider the following example of two separated module `A` and `B`.

Module A
```
type t = int
val x : t = 1
...
```

Module B
```
type t = A.t
val x : t = A.x
val r : t * t = (x,x)
```

Module `B` generate a record `(x,x)` where `x` is imported from module `A`. In `B`,`x` has an

abstract type `t` which is specialized as `int` in `A`. Under the compilation scheme, generating code for `(x,x)` requires us to create size and offset terms corresponding to the type variable `t`. In separate compilation, `A` and `B` are compiled separately. At the time of compiling `B`, the compiler does not know the actual implementation of the type variable `t`, then the bit tag/size creation algorithm can not decide what bit tag/size term corresponding to this type variable. The whole compilation process, therefore, fails.

To overcome this difficulty, I propose a simple method by which bit tags and sizes of any user-define abstract type variables are "abstracted". The following code shows a resulting pseudo codes of `A` and `B` by using this method.

Module A	Module B
`type t = int`	`type t = A.t`
`val tag_t : <t> = 0`	`val tag_t : <t> = A.tag_t`
`val size_t : |t| = 1`	`val size_t : |t| = A.size_t`
`val x : t = 1`	`val x : t = A.x`
`...`	`val r : t * t =`
	`  ([tag_t,tag_t],x^size_t,x^size_t)`

As shown in the example, bit tag and size of each user-define type variable `t` are abstracted by introducing top-level variables `tag_t` and `size_t` which are bound to corresponding terms of bit tag and size of `t`. We call them "top-level bit tag" and "top-level size", correspondingly.

If `t` is specialized in the current module with a proper outermost type constructor, top-level bit tag `tag_t` and top-level size `size_t` will be bound to the corresponding constant bit tag and constant size. Otherwise, `t` must be bound to another abstracted type defined in other module, e.g. `A.t` In this case, top-level bit tag and top-level size of `t` will refer to the corresponding top-level bit tag and top-level size in the other module, e.g `tag_t = A.tag_t` and `size_t = A.size_t`

To realizing this idea in proposed compilation algorithm (BUC transformation), we first have to extend the source calculus for module language (which I do not present here because this is out of the thesis's concerns). We also maintain two global mappings that map from the set of user-define type variables to the set top-level bit tag and to the set of top-level size. Let's call these mappings $\mathcal{B}$ and $\mathcal{S}$, respectively. $\mathcal{B}(t)$ returns a variable representing top-level bit tag of t, and $\mathcal{S}(t)$ returns a variable representing top-level size of t. The formalism of these global mapping do not violate to the principles of separate compilation since they just syntactically map from type variable's names to variable's names.

Then, I refine the compilation method by using global mapping $\mathcal{B}$ and $\mathcal{S}$. Important keys of this refinement are outlined as follows.

- When elaborating an user-define type definition `type` $t = \sigma$, the compiler inserts two top-level bindings `val` $tag_t = e^b_t$ and `val` $size_t = e^s_t$ where $tag_t = \mathcal{B}(t)$ and $size_t = \mathcal{S}(t)$. e^b_t, e^s_t are generated from σ by the following rules.
  - $e^b_t = \texttt{tagOf}(\sigma)$ and $e^s_t = \texttt{sizeOf}(\sigma)$ if σ has a proper outermost constructor.
  - $e^b_t = \mathcal{B}(t')$ and $e^s_t = \mathcal{S}(t')$ if $\sigma = t'$ (t' may be an user-defined type variable in another module such as $A.t$)
- Compilation rules are extended for the cases of top-level bit tag variables and top-level size variables. We do not treat these variables as the same as other usual

variables (usual variables may be enclosed in function's environment records by closure conversion – see Chapter 5). Instead, we generate a reference for each occurrence of top-level bit tag/size variable. This reference will be resolved at link time by using an usual reference resolution technique. This strategy would be helpful for reducing the burden of bit tag and size passing.

- The bit tag/size creation rules are extended with the following rules.
  - $\Gamma \vdash^{B}_{BUC} t \rightsquigarrow e^{b}_{t}$ where t is an user-define type variable and e^{b}_{t} is a reference resulted from the transformation $\Gamma \vdash_{BUC} \mathcal{B}(t) \rightsquigarrow e^{b}_{t}$ as described above.
  - $\Gamma \vdash^{S}_{BUC} t \rightsquigarrow e^{s}_{t}$ where t is an user-define type variable and e^{s}_{t} is a reference resulted from the transformation $\Gamma \vdash_{BUC} \mathcal{S}(t) \rightsquigarrow e^{s}_{t}$.

We meet a very similar situation when dealing with functors. Let us consider the following example.

```
functor F(X : sig type t val x : t end) =
struct
  val r : t * t = (X.x, X.x)
end
```

The functor `F` takes a structure `S` that matches the signature

```
sig type t val x : t end
```

to produce another structure containing a record of type $t \times t$. In order to generating code for such record, the standard compiler (as presented in previous chapters) must know the instance type of t, which is not available before functor applications, for creating the corresponding bitmap and size.

Adopting the compilation scheme for separate compilation, we solve this problem by introducing top-level bit tag and top-level size for each abstract type in a structure. The resulting pseudo code for this example would be

```
functor F(X :
          sig
            type t
            val tag_t : <t>
            val size_t : |t|
            val x : t
          end) =
struct
  val r : t * t = ([X.tag_t, X.tag_t], X.x^X.size_t, X.x^X.size_t)
end
```

Similar to the compilation scheme for separate compilation, we present `tag_t` and `size_t` in the target code by references to top-level bit tag and size variable abstracted in the formal structure parameter `X`. This will be resolved at the time of functor application. To demonstrate this process, we consider the following structure `S`.

```
structure S =
  struct
    type t = int
    val x : t = 1
  end
```

When compiling this structure, we also include information of top-level bit tag and top-level size in the resulting code.

```
structure S =
  struct
    type t = int
    val tag_t : <t> = 0
    val size_t : |t| = 1
    val x : t = 1
  end
```

When applying functor `F` to the structure `S`, we generate code for the resulting structure by

- duplicating code of the functor
- linking the code of `S` to the formal structure `X` in the duplicated code of `F`
- resolving all necessary references including top-level bit tags and top-level sizes

Due to the complication of module language and separate compilation implementation (which are out of my concerns in this thesis), I just intuitively describe the idea of how to incorporate them with my compilation scheme, omitting all technical details.

Chapter 7

Implementation and Optimizations

We have integrated almost all the compilation steps described (except separate compilation) in our type-directed compiler (SML#) for the full set of Standard ML Language extended with rank-1 polymorphism and record polymorphism. The compiler consists of several type-directed compilation steps, including the following:

1. rank-1 type reconstruction,
2. flatten module compilation,
3. polymorphic record compilation,
4. source-level optimizations,
5. BUC transformation,
6. A-normalization,
7. code-level optimization, and
8. code generation.

We have also implemented an abstract machine that executes three address code working with unboxed data and a bitmap-inspecting copying collection, and have successfully tested for the compiled code.

In SML#, we have considered several implementation issues related to bitmap creation and unboxed manipulation, and several optimizations to minimize runtime overhead arising by bit tag/size passing and bitmap/offset computation, and have implemented some of them.

The rest of this chapter gives a quick glance at these implementation issues and optimization.

7.1 Mutually Recursive Function Definition

For a practical implementation, the very first thing we have to do is to extend the type-directed compilation method for dealing with recursion. The extension is not so difficult for monomorphic recursion (where recursive function are monomorphic). In this thesis, I

only introduce extension for polymorphic recursion since type abstraction in the recursion directly affects my compilation scheme.

Let's define polymorphic recursion by the form of mutually recursive function definition represented in the source language as follows.

$$\begin{array}{l}\texttt{let rec } \forall\overline{t}.f_1 : \overline{\tau_1} \rightarrow \sigma_1 \texttt{=} \ \lambda\overline{x_1 : \tau_1}.e_1 \\ \qquad\qquad \cdots \\ \qquad\qquad f_n : \overline{\tau_n} \rightarrow \sigma_n \texttt{=} \ \lambda\overline{x_1 : \tau_1}.e_1 \\ \texttt{in e end}\end{array}$$

This term defines n polymorphic functions $f_1, \ldots, f_n$ which share the same set of type abstraction $\overline{t}$. At runtime, these functions can interoperate from one to another, forming a recursion.

We meet a subtle problem when compiling such polymorphic functions under our type-directed compilation method. The standard algorithm transforms polymorphic expression by inserting extra bit tag/size abstraction. It therefore transforms each polymorphic function into a target term of the type $\forall\overline{t}.\{\overline{\langle t\rangle}, \overline{|t|}\} \rightarrow \overline{\tau_i} \rightarrow \sigma_i$. However, in the mutually recursive function definition, type abstractions $\forall\overline{t}$. are generally shared among the functions. This is guaranteed in type reconstruction phase by regarding each occurrence of f_i in $e_1, \ldots, e_n$ as a monomorphic function. The standard algorithm therefore does not generate actual bit tag and size parameters for the occurrence. Obviously, this causes the disagreement between the type of f_i and the arguments of its applications.

A simple strategy to overcome this problem is to insert the same set of bit tag/size abstractions to each function definition as follows.

$$\begin{array}{l}\texttt{let rec } \forall\overline{t}.f_1 : \overline{\{\overline{\langle t\rangle}, \overline{|t|}\}} \rightarrow \overline{\tau_1} \rightarrow \sigma_1 \ \texttt{=} \ \lambda\{\overline{b : \langle t\rangle}, \overline{s : |t|}\}\lambda\overline{x_1^{s_1} : \tau_1 : e_1'}. \\ \qquad\qquad \cdots \\ \qquad\qquad f_n : \overline{\{\overline{\langle t\rangle}, \overline{|t|}\}} \rightarrow \overline{\tau_n} \rightarrow \sigma_n \ \texttt{=} \ \lambda\{\overline{b : \langle t\rangle}, \overline{s : |t|}\}\lambda\overline{x_n^{s_n} : \tau_n : e_n'}. \\ \texttt{in } e' \texttt{ end}\end{array}$$

where e_i is the transformation result of e_i for each i, and e' is the transformation result of e. In each e_i', we solve the problem of disagreement by passing the same bit tag/size variables $\{\overline{b}, \overline{s}\}$ to each occurrence f_j. This is done by substituting $(f_j\ \overline{t}\ \overline{b}\ \overline{s})$ for each f_j in the resulting body e_i' achieved from applying the standard transformation algorithm.

One weakness of this compilation scheme, which may become serious for heavily recursive invocations, is the performance penalty for passing around the same set of bit tag/size arguments. This may become more serious when we incorporate this compilation strategy with closure conversion. Suppose a mutual recursive code is defined in λ^{BUC} as follows.

$$\begin{array}{l}\texttt{let rec } f_1 : \sigma_{env} \rightarrow_C \sigma_1 \ \texttt{=} \ c_1 \\ \qquad\qquad \cdots \\ \qquad\qquad f_n : \sigma_{env} \rightarrow_C \sigma_n \ \texttt{=} \ c_n \\ \texttt{in } e \texttt{ end}\end{array}$$

where $c_1, \ldots, c_n$ are codes of form $\texttt{fcode}(\sigma_{env}, \overline{x_i : \tau_i}, e_i)$ or $\texttt{tcode}(\sigma_{env}, \overline{t}, \overline{b : \langle t\rangle}, \overline{s : |t|}, e_i)$, σ_{env} is the type of the common environment.

Since the bit tag/size abstractions are inserted for each function f_i, a naive application of the BUC transformation for mutual recursive functions would be

$$
\begin{array}{l}
\texttt{let rec } f_1^{code} : \sigma_{env}^1 \to_C \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \to \overline{\tau_1} \to \sigma_1 \\
\qquad\qquad \texttt{= tcode}(\sigma_{env}^1, \overline{t}, \overline{b : \langle t \rangle}, \overline{s : |t|}, \langle\!\langle \texttt{fcode}({\sigma_{env}^1}', \overline{x_1^{s_1} : \tau_1}, e_1'), {e_{env}^1}' \rangle\!\rangle) \\
\qquad\quad \cdots \\
\qquad\quad f_n^{code} : \sigma_{env}^n \to_C \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \to \overline{\tau_n} \to \sigma_n \\
\qquad\qquad \texttt{= tcode}(\sigma_{env}^n, \overline{t}, \overline{b : \langle t \rangle}, \overline{s : |t|}, \langle\!\langle \texttt{fcode}({\sigma_{env}^n}', \overline{x_n^{s_n} : \tau_n}, e_n'), {e_{env}^n}' \rangle\!\rangle) \\
\texttt{in} \\
\quad \texttt{let } f_1 : \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \to \overline{\tau_1} \to \sigma_1 = \langle\!\langle f_1^{code}, e_{env}^1 \rangle\!\rangle \texttt{ in} \\
\qquad \cdots \\
\quad \texttt{let } f_n : \forall \overline{t}.\{\overline{\langle t \rangle}, \overline{|t|}\} \to \overline{\tau_n} \to \sigma_n = \langle\!\langle f_n^{code}, e_{env}^n \rangle\!\rangle \\
\texttt{in } e' \texttt{ end } \cdots \texttt{ end end}
\end{array}
$$

Two closures are generated for each function f_i. The only different portion between the environment record e_i^{env} from $e_i^{env\prime}$ is the abstracted bit tags/sizes $\overline{b}, \overline{s}$ which are used in the body e_i'. At running time, a closure $\langle\!\langle \texttt{fcode}({\sigma_{env}^i}', \overline{x_i : \tau_i}, e_i'), {e_{env}^i}' \rangle\!\rangle$ may have to be created many time depending on the recursion. Extra run-time costs for duplicate part of e_i^{env} in $e_i^{env\prime}$, creation of closure, and application to the second closure are other weaknesses of this naive compilation.

In order to avoid the described runtime overhead, we consider an alternative strategy for compiling mutually recursive function definition by assuming that mutual recursive functions are always monomorphic. By this assumption, there's no bit tag/size abstractions and bit tag/size applications need to be inserted. As the consequence, there's no bit tag/size arguments to pass around and no more closures to be created. This assumption can be fulfilled by creating a wrapper polymorphic function f_i' for each f_i:

$$
\begin{array}{l}
f_i' = \Lambda \overline{t}.\lambda \overline{x_i : \tau_i}. \\
\qquad \texttt{let rec } f_1 : \overline{\tau_1} \to \sigma_1 \texttt{=} \ \lambda \overline{x_1 : \tau_1}.e_1 \\
\qquad\qquad \cdots \\
\qquad\qquad f_n : \overline{\tau_n} \to \sigma_n \texttt{=} \ \lambda \overline{x_1 : \tau_1}.e_1 \\
\qquad \texttt{in } (f_i \ \overline{x_i}) \texttt{ end}
\end{array}
$$

The code part of the wrapper function's closure is generated as follows

$$
\begin{array}{l}
f_i^{code\prime} = \texttt{tcode}(\sigma_{env}, \overline{t}, \{\overline{b : \langle t \rangle}, \overline{s : |t|}, \overline{x^s : \tau}\}, \\
\qquad \texttt{let rec } f_1^{code} : \sigma_{env}^1 \to_C \overline{\tau_1} \to \sigma_1 \texttt{= fcode}(\sigma_{env}^1, \overline{x_1^{s_1} : \tau_1}, e_1') \\
\qquad\qquad \cdots \\
\qquad\qquad f_n^{code} : \sigma_{env}^n \to_C \overline{\tau_n} \to \sigma_n \texttt{= fcode}(\sigma_{env}^n, \overline{x_n^{s_n} : \tau_n}, e_n') \\
\qquad \texttt{in } (\langle\!\langle f_i^{code}, e_{env}^i \rangle\!\rangle \ \overline{x_i}) \texttt{ end)}
\end{array}
$$

Here we apply an uncurrying optimization (described later) for the wrapper function which uncurries the bit tag, size and lambda arguments together. Although a code for monomorphic mutual recursive function declaration are generated for each wrapper, they will be finally shared among wrappers because they are identical.

By taking this compilation strategy, we still have to pay some performance costs for manipulating wrappers, i.e. costs for creating wrapper closures and for passing arguments from wrapper to the original function. But I believe this is acceptable because we only need to invoke a wrapper once for an entire recursion loop. Furthermore, separating polymorphic part from the mutual recursive function definition, we will have a chance for optimization by lifting bitmap/offset computation inside the monomorphic recursive functions to the wrapper, ensuring they are never performed twice. I will present this optimization in Section 7.3.

7.2 Double-Word Alignment

The major intention of this thesis is to compile ML so that it can seamlessly interoperate with other languages such as C. Supporting a natural representations of data is the first condition to achieve this goal. However, in some specific architecture, natural representations is not sufficient. For example, a conventional assumption for allocating data in SPARC architecture is that addresses of floating point values must be double-word aligned. In this situation, the expected compiler must generate codes that conform to this assumption.

There are two places where exchanging floating point values may reside: in heap blocks or in stack frames. Since we assume variable's values are allocated in fixed positions in stack frames, this would be easy to align floating point values of stack frames to double-word aligned positions. To double-word align floating point values in heap blocks in the presence of polymorphism is a subtle problem.

Assuming that the value of the first field in a heap block is always aligned, a simple strategy for aligning other floating point values in the block is to insert a dummy word before each floating point value if this value located at an odd offset.

For example, in the record $(1, 1.2)$, the floating point value is located at offset 1. A dummy word is inserted before this value to produce the record $(1, 0w0, 1.2)$ that conforms with the double-word alignment assumption.

In the presence of polymorphism, doble-word alignment is not so easy. Consider a record (x, y) of type $t_1 \times t_2$, if t_1 is instantiated to a single type (e.g. *int*, *string*), and t_2 is instantiated to *real*, the compiler needs to insert a dummy word before y, i.e. $(x, 0w0, y)$. In other cases of instantiation (t_1 is instantiated to a double type or t_2 is instantiated to a single type), compiler does not need to align the record.

For a uniform treatment of aligning process, I introduce a representation of records so that special fields are inserted at positions where alignment may occur. For example, compiler transform the above record into (x, d, y) where d is the special field. This field is typed specially so that we can compute the size of this field at run-time. If this size is zero, no dummy word is inserted. Otherwise, run-time system must allocate the record with a dummy word inserted at the position of d.

Now we consider how to type special fields. Let consider a record $(x_1, \ldots, x_n)$ of type $\sigma_1 \times \cdots \times \sigma_n$. There are four cases that we may have to insert a dummy field (special field) before an ordinary field x_i:

- If σ_i is a type variable t, and $\{\sigma_1, \ldots, \sigma_i\}$ contains an arbitrary type (i.e. type variable), the compiler inserts a dummy field d_i of type $t \Rightarrow \sigma_1 \times \cdots \times \sigma_{i-1}$ For example, consider record (x, y) of type $t_1 \times t_2$, the compiler constructs (x, d, y) where d has type $t_2 \Rightarrow t_1$.

- If σ_i is a type variable t, $\{\sigma_1, \ldots, \sigma_i\}$ does not contain any arbitrary type, and the total size of these types (which can be statically computed) is an odd number, the compiler insert a dummy field d_i of type $t \Rightarrow int$. For example, consider record (x, y) of type $string \times t_2$, the compiler constructs (x, d, y) where d has type $t_2 \Rightarrow int$.

- If σ_i is *real* type, and $\{\sigma_1, \ldots, \sigma_i\}$ contains an arbitrary type (i.e. type variable), the compiler inserts a dummy field d_i of type $real \Rightarrow \sigma_1 \times \cdots \times \sigma_{i-1}$ For example, consider record (x, y) of type $t_1 \times real$, the compiler constructs (x, d, y) where d has type $real \Rightarrow t_1$.

- If σ_i is *real* type, $\{\sigma_1, \ldots, \sigma_i\}$ does not contain any arbitrary type, and the total size of these types (which can be statically computed) is an odd number, the compiler alway insert a dummy field $0w0$ (of *word* type). For example, consider record (x, y) of type $string \times real$, the compiler constructs $(x, 0w0, y)$.

After performing this transformation process for a source language, we can apply BUC transformation to compile the target term in λ^{BUC} with an extension of the size creation rule for dummy field types.

$$\texttt{sizeOf}(\sigma \Rightarrow \sigma_1 \times \cdots \times \sigma_n) = \\ \quad \texttt{if sizeOf}(\sigma) \texttt{ = 2 and ((sizeOf}(\sigma_1) + \cdots + \texttt{sizeOf}(\sigma_n)\texttt{) mod 2) = 1} \\ \quad \texttt{then 1 else 0}$$

The extra computation for this size computation may be large, but we can achieve a full compatible data representations with the target SPARC architecture. Moreover, this computation can be optimized by extending the arithmetic optimization – the technique I present in next section.

7.3 Optimizations

A common scenario of type-directed compilation implementation is that source level optimizations such as uncurrying, in-lining, common expression elimination, invariant removal, etc. are performed before closure conversion. Our compilation strategy, however, combines bitmap-passing compilation, unboxed compilation and closure conversion into a single phase. For polymorphic language, this introduces significant extra codes for passing bit tags/sizes, and for computing bitmap/offset. Minimizing the runtime overhead arising by these extra codes is an important task in developing a practical compiler. Such optimizations should be done in or after BUC transformation. I realize that BUC transformation is the best place for integrating them, though they would make the algorithm much more complex.

In this section, I introduce three important optimizations which have been implemented in our SML# compiler.

7.3.1 Uncurrying optimization

The common sense of uncurrying optimization is that it transforms curried functions in the source language into multi-argument function in the target language. In our implementation, we also consider this kind of optimization in the source level optimization phase implemented before the BUC transformation.

However, standard BUC transformation may introduces a special form of curried function: an ordinary lambda abstraction is followed by bit tag/size abstractions generated by BUC transformation. Uncurrying them, obviously, would help us to obtain better target codes. Let me first analyze this situation and then propose the solution for this optimization.

Rank-1 type reconstruction mostly generates type abstraction at ordinary lambda abstractions to form polymorphic functions (the other cases are that type abstractions are generated for constructing polymorphic datatype). Let's consider a polymorphic function in the source language $\Lambda\bar{t}.\lambda\overline{x:\tau}.e$. Bitmap-passing compilation and unboxed compilation

insert bit tag abstractions and size abstractions for each type variable of $\overline{t}$. The resulting code of these two phases would therefore be $\Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.\lambda\overline{s : |t|}.\lambda\overline{x : \tau}.e'$ where $\overline{b}$ and $\overline{s}$ are the inserted bit tag parameters and size parameters, e' is the target term of e.

In the standard BUC transformation, we already have $\overline{b}$ and $\overline{s}$ uncurried since they are always come together at both abstraction and application. However, BUC transformation does not consider about uncurrying them with the ordinary lambda parameters $\overline{x}$. As the consequence, the resulting term in λ^{BUC} consists of two closures representing the bit tag/size abstractions and the ordinary lambda abstaction. This is obviously an inefficient code. Uncurrying them would helps us to obtain a better one. The resulting code in λ^{BUC} with uncurrying optimization would be something like

$$\langle\!\langle \Lambda\overline{t}.\lambda\{\overline{b : \langle t \rangle}, \overline{s : |t|}, \overline{x : \tau}\}.e', e_{env} \rangle\!\rangle \colon \forall\overline{t}.\{\overline{\langle t \rangle}, \overline{|s|}, \overline{\tau}\} \to \sigma'$$

The next step of such uncurrying optimization is to solve the disagreement between the type of the resulting uncurried function and its applications. After uncurrying, type of the above function is recorded as $\forall\overline{t}.\{\overline{\langle t \rangle}, \overline{|s|}, \overline{\tau}\} \to \sigma'$. At a type instantiation $(f\ \overline{\tau}\ \overline{e_b}\ \overline{e_s})$, actual bit tag parameters $\overline{b}$ and size parameters $\overline{s}$ are given. Applying the function f at this time would cause the disagreement problem between the type of f and the number of actual arguments. We solve this by compiling type abstraction and type application together. For example, $((f\ \overline{\tau})\ \overline{e_x})$ would be compiled into $(f\ \overline{\tau}\ \{\overline{e_b}, \overline{e_s}, \overline{e_x}\})$ where $\overline{e_b}$ and $\overline{e_s}$ are the bit tag terms and size terms generated from $\overline{\tau}$. If such function is instantiated without its argument, i.e. type instantiation without application followed, the compiler first performs eta-expansion, and then merge the bit tag values and the variables introduced by the eta-expansion. For example, a type instantiation in the source calculus $(f\ \overline{\tau})$ can be eta-expanded and uncurried as $\lambda\overline{x : \tau}.(f\ \overline{\tau}\ \overline{x})$. This term will be compiled into BUC with unccurying optimization as

$$\langle\!\langle \mathtt{fcode}(\sigma_{env}, \overline{x^{e_s} : \tau}, (f\ \overline{\tau}\ \{\overline{e_b}, \overline{e_s}, \overline{x}\})), e_{env} \rangle\!\rangle$$

where $\overline{e_b}$ and $\overline{e_s}$ are bit tag terms and size terms generated from τ, e_{env} is the environment record generated by closure conversion in BUC transformation.

In this case, we need to pay extra performance cost for manipulating with the eta-expanded function. How ever, I believe this cost is not too much since type instantiation usually come together with lambda application under the rank-1 type reconstruction.

7.3.2 Sharing bitmap/offset computation

This optimization has the same spirit with common expression elimination which has been implemented in many type directed compiler: the same value should not be computed twice. Different from the well-known common expression elimination technique, sharing bitmap/offset computation should be performed at BUC transformation where bitmap and offset are generated.

I developed this optimization by taking the following strategy

1. lifting bitmap/offset to the best place where their computation can be share
2. comparing bitmaps or offsets to eliminate redundant computations.

For the first one, my strategy is to lift bitmaps/offsets to the furthermost place whe we can compute them. This would be the place where all required tag/size are abstracted. For example, in the code

$$\Lambda t_1.\lambda\{b_1 : \langle t_1\rangle\}\cdots\Lambda t_2.\lambda\{b_2 : \langle t_2\rangle\}\cdots\Lambda t_3.\lambda\{b_3 : \langle t_3\rangle\}.(\cdots[b_1, b_2]\cdots)$$

the bitmap $[b_1, b_2]$ can be composed based on bit tags b_1, b_2, therefore the compiler lift this bitmap to the second abstraction as in the following pseudo code.

$$\begin{aligned}&\Lambda t_1.\lambda\{b_1 : \langle t_1\rangle\}\cdots\Lambda t_2.\lambda\{b_2 : \langle t_2\rangle\}\\ &\qquad\qquad\qquad \texttt{let bm = } [b_1, b_2]\\ &\qquad\qquad\qquad \texttt{in } \cdots\Lambda t_3.\lambda\{b_3 : \langle t_3\rangle\}.(\cdots\texttt{bm}\cdots)\texttt{ end}\end{aligned}$$

Of course, under closure conversion, lifted bitmaps/offset may need to be enclosed in environment records to pass to the corresponding places. This would cost another run-time performance, but I believe that the benefit of lifting is more larger than the performance penalty it makes.

After lifting bitmaps, and offsets to appropriate place, the second matter is how to identify identical bitmaps and offsets to eliminate the redundant ones. This would be done based on types of the given bitmaps and offsets. Given two bitmaps of types $\langle\!\langle\sigma^1_1, \ldots, \sigma^1_n\rangle\!\rangle$ and $\langle\!\langle\sigma^2_1, \ldots, \sigma^2_m\rangle\!\rangle$, in order to compare them, I first compact each type σ^j_i into one of the followings $\{u, b, d, t_i\}$ where u denotes single unboxed types, b denotes single boxed types, d denotes double unboxed types, and t_i represents a type variable. The comparison algorithm for two bitmap types is therefore given as the following relations on two list of types .

$$\begin{array}{rcll} [\,] & =_B & [\,] & \\ u :: L_1 & =_B & u :: L_2 & \texttt{If } L_1 =_B L_2 \\ b :: L_1 & =_B & b :: L_2 & \texttt{If } L_1 =_B L_2 \\ d :: L_1 & =_B & d :: L_2 & \texttt{If } L_1 =_B L_2 \\ t_1 :: L_1 & =_B & t_2 :: L_2 & \texttt{If } L_1 =_B L_2 \texttt{ and } t_1 = t_2 \\ u :: L_1 & =_B & d :: L_2 & \texttt{If } L_1 =_B u :: L_2 \\ d :: L_1 & =_B & u :: L_2 & \texttt{If } u :: L_1 =_B L_2 \end{array}$$

For comparing offsets, we define a function $Ofs(L)$ that takes a list of types and return the total sizes of layout-fixed types (u, b, d) and an ordered list of type variables. This function is defined as follows

$$\begin{array}{rcll} Ofs([]) & = & (0, \emptyset) & \\ Ofs(u :: L) & = & (n+1, S) & \texttt{If } Ofs(L) = (n, S) \\ Ofs(b :: L) & = & (n+1, S) & \texttt{If } Ofs(L) = (n, S) \\ Ofs(d :: L) & = & (n+2, S) & \texttt{If } Ofs(L) = (n, S) \\ Ofs(t :: L) & = & (n, S \uplus t) & \texttt{If } Ofs(L) = (n, S) \end{array}$$

where $S \uplus t$ is the operation that inserts t into the ordered list S. Given two offsets of types $\parallel \sigma^1_1, \ldots, \sigma^1_n \parallel$ and $\parallel \sigma^2_1, \ldots, \sigma^2_m \parallel$, for comparing them we first compute $(n_1, S_1) = Ofs([\|, \ldots, \ldots\|, \sigma^1_1\sigma^1_n])$ and $(n_2, S_2) = Ofs([\|, \ldots, \ldots\|, \sigma^2_1\sigma^1_m])$. Two offsets is identical if $n_1 = n_2$ and $S_1 \equiv S_2$.

With the comparison methods of bitmap and offsets, and the lifting strategy, we can refine BUC transformation for sharing bitmap/offset computations.

Together with the presented compilation method for mutually recursive function definition, this optimization is extremely efficient for the case in which bitmaps/offsets may

be lifted out to the wrapper function (or outside the wrapper function). By this way, at the runtime, these bitmaps/offsets are only computed once for an entire recursive call since they are already out of the computation loop. Obviously, this would help us reducing lot of runtime overhead arising by recomputing bitmaps and offsets.

7.3.3 Arithmetic Optimization

Arithmetic optimization is another important optimization for reducing run-time overhead arising by bitmap/offset computations. In our implementation, bitmap and offset computations are broken down into simple arithmetic operation such as "+", "`and`", "`or`", "≪", and "≫" (the last four operators are logical and, logical or, shift left, and shift right). When computing a set of bitmaps/offsets, some parts of the computations can be shared. For example, the bitmap $[b_1, b_2, b_3]$ and the bitmap $[b_1, b_2, b_4]$ can share the same arithmetic computation for the first two bits, i.e. $[b_1, b_2]$.

In order to finding and removing the identical computation, let us consider first a simple algebra where primitives are integers and variables and operators are "+", "`and`", "`or`", "≪", and "≫". We define functions *ArithB*, *ArithO*, *ArithT*, and *ArithS* that transform bitmap types, offset types, bit tag types, and size types into expressions in the given algebra.

$$
\begin{array}{rcll}
ArithO([\,]) & = & 0 & \\
ArithO(\sigma :: L) & = & ArithS(\sigma) \ + \ ArithO(L) & \\
ArithB([\,]) & = & 0 & \\
ArithB(\sigma :: L) & = & ArithT(\sigma) \ \texttt{and} \ (ArithB(L) \ \ll \ ArithS(\sigma)) & \\
ArithT(\sigma) & = & 0 & \texttt{If } \sigma \texttt{ is an unboxed type} \\
ArithT(\sigma) & = & 1 & \texttt{If } \sigma \texttt{ is a boxed type} \\
ArithT(t) & = & b & \texttt{If } b : \langle t \rangle \\
ArithS(\sigma) & = & 1 & \texttt{If } \sigma \texttt{ is a single type} \\
ArithT(\sigma) & = & 2 & \texttt{If } \sigma \texttt{ is a double type} \\
ArithT(t) & = & s & \texttt{If } s : |t|
\end{array}
$$

Note that the cases $ArithT(t)$ and $ArithS(t)$ are given by the bit tag and size creation algorithms which generate the corresponding bit tag/size variable.

Using these transformations, each bitmap or offset can be transformed into an expression of the algebra. Each expression represents a computation tree where the outer nodes are either constants or variables, the inner nodes are operators, the root represent the result. Given a set of bitmaps and offsets, we can transform them into a set of expressions which are also represented by a *forest* of computation trees.

We can optimize the whole computation by transforming the computation forest into an acyclic directed graph, by unifying identical sub-trees and replacing some sub-trees to the simpler one. (for example. the tree of expression 0 `and` e can be replaced by the tree of 0).

In this thesis, I do not give the full implementation of this optimization in details.

Chapter 8

Conclusion and Future Directions

So far in this thesis, I have presented a type-directed compilation method that support an interoperable memory management where integers, floating point values and other atomic data are naturally represented, and each run-time object includes a layout bitmap for tracing garbage collection.

The key idea of this method is to pass bit tag and size information of types to run-time for composing the necessary bitmap , and for manipulating unboxed values.

To achieve this, the type-directed compilation statically computes bit tags/sizes whenever layout of objects of types are fixed, introduces bit tag/size parameters for each abstract type variable, and passes them to corresponding computation codes for bitmap and offsets (which are using for manipulating unboxed values).

I have solved the problem of mutual dependency among the type-directed compilation, closure conversion, and A-normalization. The compilation method has been further refined for separate compilation and module language.

Beside the theoretical development of this method, I have also presented several implementation issues and optimizations which are successfully integrated in our SML# compiler.

There are few further issues to be consider in the near future:

Implementation for separate compilation. Although I have proposed the solution for separate compilation, implementing this solution would make much more efforts and much more problems to be solved. As far as I know, there does not exists any true separate compilation for ML in which programs can be separately compiled in to codes and link together or to other language's binary objects. This would be a big issue for considering.

Optimization. Current optimizations are strong enough to make a significant performance advantage in comparison to other ML dialects such as SML/NJ. There is still room for further optimizations for reducing performance cost arising by bitmap/offset computation. The biggest one might be a source-level optimization for reducing polymorphism whenever they are not required. Other state-of-the-art optimizations are also included in the to-do list of my future research.

Register Allocation. Until now, our SML# does not have register allocations yet. Considering this in the presence of bitmap-inspecting garbage collection is another interesting issue for my future research.

Bibliography

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[CWM98] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312, 1998.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[Fra94] Pascal Fradet. Collecting more garbage. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 24–33, New York, NY, USA, 1994. ACM Press.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, 1991. ACM Press.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[HJ94] Fritz Henglein and Jesper Jorgensen. Formally optimal boxing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–226, New York, NY, USA, 1994. ACM Press.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[IK00] A. Igarashi and N. Kobayashi. Garbage collection based on a linear type system, 2000.

[JL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Kah87] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order lambda-calculus. *Inf. Comput.*, 98(2):228–257, 1992.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, 1992.

[MDCB91] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, July 1991.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.

[MMH96] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[Oho99] Atsushi Ohori. A curry-howard isomorphism for compilation and program execution. In *Typed Lambda Calculus and Applications*, pages 280–294, 1999.

[Oho04] Atsushi Ohori. Register allocation by proof transformation. *Sci. Comput. Program.*, 50(1-3):161–187, 2004.

[OT97] Atsushi Ohori and Tomonobu Takamizawa. An unboxed operational semantics for ml polymorphism. *Lisp Symb. Comput.*, 10(1):61–91, 1997.

[OY99] Atsushi Ohori and Nobuaki Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. In *International Conference on Functional Programming*, pages 160–171, 1999.

[PJ93] John Peterson and Mark Jones. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 227–236, New York, NY, USA, 1993. ACM Press.

[SS98] Bratin Saha and Zhong Shao. Optimal type lifting. In *TIC '98: Proceedings of the Second International Workshop on Types in Compilation*, pages 156–177, London, UK, 1998. Springer-Verlag.

[Thi95] Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 24–35, New York, NY, USA, 1995. ACM Press.

[TMC+96] 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 '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[Tol94] Andrew P. Tolmach. Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming*, pages 1–11, 1994.