Unboxed Function Closures

Roger Bosman

A thesis submitted for the Master of Science degree

Department of Information and Computing Sciences

Utrecht University

August 2020

Supervisors

dr. Wouter Swierstra	Utrecht University
dr. Richard Eisenberg	Tweag I/O

`ICA-6318908`

Abstract

In Haskell, both thunks and values are generally represented as a heap-allocated closure [18]. This introduces overhead, as the heap generally is much slower than the stack. To combat this inefficiency, programmers can use unboxed types [19]. These types are represented directly on the stack, and therefore do not carry such overhead.

So far, only *data values* such as `Int` and `Char` can be unboxed. In this thesis we explore the possibility of extending this notion, allowing for *function values* to be unboxed as well.

As functions can close over variables, they must be represented as a closure. Therefore, unboxing function values requires representing closures on the stack. This introduces a significant challenge, as variations in the set of closed over variables now affect the stack representation.

We propose an extension to function types, where the types of the closed over variables are annotated on the function arrow. These annotations make it possible to reason about the exact runtime representation of a closure at compile time. We do so by presenting two languages, $\mathcal{L}$ and $\mathcal{M}$, and a compilation function $\mathcal{L} \to \mathcal{M}$. Furthermore, we identify the key correctness criteria of $\mathcal{L} \to \mathcal{M}$, and proof that they hold.

Acknowledgments

First of all, I want to thank my supervisors, Wouter and Richard. Their knowledge and support have been paramount in the process which led to this thesis. Furthermore, I am grateful for their guidance on topics beyond this thesis, allowing me to explore the world of academia, and develop myself personally.

I would like to thank my parents, Anton and Els, for always supporting me throughout my education, even though my route was atypical. Finally, I want to thank my friend Koen for motivating me to peruse a master's degree, and my cat Donny for always being there for me, and for all the rubber duck debugging.

Contents

Chapter 1

Introduction

Parametric polymorphism is a powerful tool that allows for the declaration of generic functions and data types that abstract over concrete types. To illustrate this notion, consider the following functions `appInt` and `appFloat`:

```
appInt :: (Int → Bool) → Int → Bool
appInt f x = f x

appFloat :: (Float → Char) → Float → Char
appFloat f x = f x
```

While the concrete types for each function are different, a clear pattern exists. Both take a function of some type (`Int` or `Float`) to another (`Bool` or `Char`), and an argument of the first type. Both functions consist of applying the passed function to the passed argument, resulting in a return type equal to that of the passed function.

With parametric polymorphism, we can define a single function that generalizes both above definitions by abstracting over the concrete types through the usage of type variables, as shown below. We can reconstruct our original `appInt` function by instantiating `a` to `Int`, and `b` to `Bool`. We can recover `appFloat` in a similar fashion.

```
app :: ∀ a b. (a → b) → a → b
app f x = f x
```

By defining functions in this manner, we change our demands from the compiler. Instead of outputting code that can handle arguments of specific types, we now require this code to be able to handle *any* (valid) instantiation of its type variables. This is a significant change, as the concrete types offer crucial information about how to compile a function, which we do not have access to in the polymorphic case.

Consider the behaviour of functions `appInt` and `appFloat` in the situation where they are each passed their argument `x` via a register. As the registers for integers and floats are often split, the code for `appFloat` should fetch its argument from a floating-point register. In the case of `appInt`, `x` will be stored in a non-floating-point register, which means the code should fetch `x` elsewhere. Therefore, the type of an argument can change the interaction with that argument: the type of an argument influences its *calling convention.*

This discrepancy becomes an issue when both behaviours must to be captured by the same generic function, which is the case for `app`. On top of this, bit patterns may be represented on the stack, or even may not directly represent a value, as they could also encode a pointer to a heap-allocated object instead. Clearly, some kind of structure needs to be in place that deals with this issue.

One might wonder why a compiler does not simply expand polymorphic functions such as `app` into multiple versions, each instantiated to the needed concrete type. This process is called monomorphization, and is used in some form in several languages. However, this is not a solution for every language, including Haskell [6]. In such cases, a common solution for these problems is to implement a system where the calling convention is the same for all types. Such systems represent all types *uniformly* as a pointer to a heap-allocated object. While this solves the problem, it has drawbacks such as a significant speed penalty, as discussed in section 2.3.1.

To combat this speed penalty, some languages add the notion of unboxed types, which includes Haskell [19]. Unboxed types reintroduce the representation of variables as literal bit patterns on the stack and registers. At first glance, uniform representation and unboxed types seem mutually exclusive notions. However, given some constraints, the two can coexist in the same language, as discussed in section 2.3.3.

Currently there is a limitation on what kind of types can be unboxed. For example, Haskell allows for the unboxing of primitives such as integers and floats, but not of *functions*. Would it be possible to lift this restriction, allowing for function values to be unboxed? This thesis addresses this very issue.

Specifically, this thesis attempts to answer the question "Can we add unboxed function closures *to Haskell*?". Following the aforementioned preliminaries, we will make the following contributions:

- We elaborate further on what unboxed function closures are, what their intended behaviour is, and how they can be more efficient than boxed function closures. Furthermore, we discuss how unboxed function closures necessitate a change to conventional function types, and describe our approach for solving this issue (chapter 3).
- We present two languages, $\mathcal{L}$ (chapter 4) and $\mathcal{M}$ (chapter 5), each implementing unboxed function closures. As $\mathcal{L}$ is based on System F [9, 21, 22], it illustrates how unboxed function closures can be added to System F. Furthermore, as $\mathcal{M}$ is sufficiently close to a real machine, it illustrates the changes needed in the lower levels of Haskell's compilation stack, particularly cmm[1].
- We will present a compilation function $\mathcal{L} \to \mathcal{M}$ (chapter 6), and prove it correct (chapter 7).

---

[1] Cmm [24] is a language closely related to C-- [20].

Chapter 2

Background

2.1 Strictness

A language's evaluation strategy refers to the way a language evaluates expressions that are bound to variables, either as an explicit binding or when passed as a function argument. Languages with a strict evaluation strategy evaluate expressions as soon as they are bound. This means that further usages of the variable can work with the already-evaluated result of the expression. Languages with a non-strict evaluation strategy defer this evaluation: expressions are not evaluated as soon as they are bound to variables, but only upon the usage of that variable. Haskell implements the latter. More specifically, its evaluation strategy is lazy, which means it implements non-strict evaluation combined with *sharing*.

While discussing all ramifications of such semantics is out of scope for this thesis, we would like to examine what the effects of implementing non-strict semantics have in context of non-termination and the number of members of a type.

A problem arises when the expression being bound does not terminate. If we bind such a value, Haskell will happily bind the expression to the variable and continue on, given that it is well-typed. Only when the evaluation is forced upon usage, non-termination occurs.

To account for this, we must include a bottom $\bot$ in each type that represents this non-termination, which in Haskell is denoted as `undefined`. The levity of a type indicates the presence of a bottom: if it is lifted, it is lazy and its type contains $\bot$, if it is unlifted, it is strict, and its type does not contain $\bot$. We further discuss levity in section 2.4.

2.2 Closures

The term closure can be used to refer to various concepts, depending on the context. In this section, we define what we consider to be a closure.

In section 2.1 we described that, as Haskell is a language with a non-strict (lazy) evaluation strategy, expressions are not evaluated until the variable they bind to is used. This construct requires an additional way of storing variables: not only do we need to store values, but we also need to store suspensions, or *thunks*.

An important factor for storing closures is that the deferred expression can *close over* variables. That is, an expression can refer to variables it does not declare itself, but are in scope at the declaration of the expression. These closed over variables must be in scope when the expression is eventually evaluated. Therefore, we must store - along with the code representing the expression - an environment that stores these variables.

Consider `const'` below. It returns a (function) closure that mentions `x`, which is brought into scope by its surrounding function. As `f` does not declare `x` itself, it must be brought back into scope once `f` is eventually applied an argument.

```
const' :: a → (b → a)
const' x =
  let f = λy → x
  in  f
```

2.2.1 Values as closures

The above description motivates the need for closures in the case of as-yet unevaluated thunks. In Haskell, values are closures[1] as well. To understand this, we first observe that there are two kinds of values: *data values* and *function values*. Data values represent an atomic element of data (such as integers or booleans), whereas function values represent functions.

For data values, consider what happens when a thunk evaluates into a value. As Haskell is a lazy language, we need to *share* the result so that subsequent usages of this variable do not re-evaluate the thunk, but can reuse the previously found value.

Keeping track of which variable has already been evaluated gets complex quick, especially when considering parallelism. Therefore, Haskell implements a self-updating model [31].

In such a model, whenever a variable is encountered, it is always *forced*, regardless of its evaluation status. That is, evaluation is always switched to the variable, even if it already is a value. For thunks this works as expected, as the thunk is evaluated and the result is returned. For values a different approach must be taken. Instead of storing the raw value, a 'box' is stored, which is a function that upon evaluation simply returns the previously found value. This box is a closure that closes over a single variable: the value that the box stores.

In addition to the rules above, for functional values, an additional reason for representing it as a closure applies, as the contents of a function can close over. This means that functions that have been evaluated to a value (but not yet applied an argument) potentially *have* to store additional bindings, as otherwise these will be out of scope when the function body is evaluated.

2.2.2 Closure definition

We can now define a closure, which we consider to be a combination of the following two things:

- A pointer to the (static) closure code, representing the expression. This code may contain *free variables*. That is, in addition to variables bound locally trough function arguments or let-bindings, it can also refer to variables it does not define itself.
- An environment storing the bindings of exactly the free variables of the stored expression.

Note that while the closure code may contain free variables, the closure itself may not. That is, the closure code may be *open*, closures must be *closed*.

---

[1] Except for unlifted types, which we cover in section section 2.4.

2.2.3 Uniform representation

In the introduction, we described uniform representation as the notion where all types are represented uniformly as a pointer to a ‘heap-allocated object’. Now that we have defined closures and have shown that any type can be a closure, we can refine this heap-allocated object to be a heap-allocated closure instead.

2.3 Uniform representation & unboxed types

In this section we will show the implementation concerns regarding uniform representation. Specifically, we will first show how, in a naive implementation, performance can be severely affected. We then present unboxed types and show how they can be used to remedy this situation, with examples based on the tutorial by Peyton Jones and Launchbury [19]. Finally, we show how the two systems - that at first glance seem mutually exclusive - can coexist in the same language.

2.3.1 Naive uniform representation

Consider the function `add3` below:

```
add3 :: Int → Int → Int → Int
add3 x y z = x + (y + z)
```

When evaluating `add3`, a naive compiler for Haskell might output code performing the following steps:

1. First, the inner expression `(y + z)` needs to be evaluated. For this, the bit patterns of `y` and `z` are needed. These patterns are obtained by forcing `y` and `z`.
2. Now that the bit patterns for `y` and `z` are fetched, the inner addition can be performed. As all values are represented uniformly, a box must be allocated on the heap, which stores the resulting bit pattern of the addition.
3. Now the outer expression can be evaluated. In a similar fashion to the inner expression, the bit patterns of the arguments - `x` and the result of the inner expression `(y + z)` - are fetched by forcing their corresponding closures. Note that for the inner expression, the closure that is forced is the box that was just created.
4. Now that the bit patterns for both sides of the outer addition are fetched, the addition can be performed. As we implement sharing, the resulting bit pattern must be stored. Just like the result of the intermediate addition, a new closure (of the box form) is allocated, which stores the result.
5. The result is returned.

As is evident from the above description, adding three integers this way is quite involved, and requires many operations involving the heap. Fetching our simple, integer arguments requires heap access. Even worse, the intermediate result is stored on the heap, only to be retrieved in the very next step! This is horribly inefficient, especially when comparing to a language like C, which needs just a handful of instructions[2] to perform the additions, and does not access the heap once.

---

[2] Code included in appendix D.1.

2.3.2 Unboxed types

In the previous section, we have shown how a naive implementation of uniform representation can result in rather inefficient code. The reason that languages like C can implement `add3` much more efficiently is that they can work with literal bit patterns. The arguments for `x`, `y`, and `z` are not pointers to heap-allocated closures, but rather directly encode values, as does the return value. The only operations needed are the ones dealing with fetching the bit patterns, calculating the result, and returning the resulting pattern.

In this section, we show how unboxed types expose enough information such that the creation and subsequent forcing of the box for the intermediate result can be removed by correctness-preserving transformations. While we will not end up at code as efficient as languages like C will produce, we do show how, with further optimizations, further steps towards such an efficient solution can be taken.

`Int` and `Int#`

Unboxed types reintroduce the notion of literal bit patterns. Consider the following definition for `Int`.

```
data Int = Int Int#
```

As shown, Haskell's data types that would initially seem primitive are actually plain ADTs that wrap around their corresponding unboxed primitive. `Int` is just a normal ADT, that conforms with uniform representation. That is, it is always represented as a pointer to a heap-allocated closure that stores its contents, in this case `Int#`.

Here the identifier `Int#` represents the literal bit pattern for integers. We call these types unboxed. By convention, unboxed types are suffixed with `#`. Effectively, the constructor `Int` is one that promotes the unboxed type `Int#` to a type in uniform representation, which means we can pass to functions that expect variables to all be in this representation.

If we now rewrite the `add3` example from earlier to use this definition, and unfold the + operators, we get the following:

```
add3 :: Int → Int → Int → Int
add3 x y z = case x of
         Int x# → case ( case y of
                     Int y# → case z of
                            Int z# → case (y# +# z#) of
                                   t1# → Int t1#
                   ) of
                Int yz# → case (x# +# yz#) of
                       t2# → Int t2#
```

Case-of-case transformation

In the above example, case expressions are used to express the evaluation and unpacking of variables. Observe that we have a case statement that examines another case statement. That is, it has another case statement as *scrutinee*. Wherever such case-of-case expressions occur, we can apply the aptly named *case-of-case* transformation [19]. Applying this transformation moves the outer case expression into *each* of the alternatives of the inner statement. While this can cause duplication if the inner case expression has multiple alternatives, in the case of `add3`, it nicely merges into the following:

```
case x of
Int x# → case y of
          Int y# → case z of
                    Int z# → case (y# +# z#) of
                              yz1# → case (Int yz1#) of
                                      Int yz2# → case (x# +# yz2#) of
                                                  xyz# → Int xyz#
```

Factoring out the intermediate closure

Now that the case statements have been merged, we can clearly see the boxing and subsequent unboxing of the intermediate result. The result of `y# +# z#` is boxed inside an `Int`, only to be unboxed on the very next line! It is valid to remove this part, giving us a version that skips the (un)boxing of the intermediate result `y + z`.

```
case x of
Int x# → case y of
          Int y# → case z of
                    Int z# → case (y# +# z#) of
                              yz# → case (x# +# yz#) of
                                     xyz# → Int xyz#
```

Further optimizations

While we have gotten rid of the intermediate closure, we have not yet gotten the same efficient set of instructions that languages like C would emit. The reason for being able to remove the intermediate closure is that we are aware of its *entire* context: we know where it is created, and where it is used. If we want to further optimize `add3`, we thus need to know where it is called. In such case, we can inline the definition of `add3` (similar to how we have inlined the definition for +) and apply a similar set of transformations.

2.3.3 Combining the systems

In the introduction we presented the `app` function, which we rename `app`^{1} and repeat below:

```
app¹ :: ∀ a b. (a → b) → a → b
app¹ f x = f x
```

Furthermore, in the introduction, we described a problem with compiling such a function. As this definition has to be able to handle *any* data type, it somehow has to be able to handle many representations (and thus many calling conventions) at the same time, which it cannot. We solved this problem by introducing uniform representation, where every type is represented uniformly as a pointer to a heap-allocated closure. But directly after this, we reintroduced alternative representations in the form of unboxed types. Would this addition not reintroduce the problem?

No, it does not. The problem arises from the assumption that `app`^{1} is polymorphic over *all* types, which is not exactly true. Recall that the reason why uniform representation worked is that we always know the representation, even if we do not know the exact type. We achieved this by eliminating all other representations. We can get back the same guarantees in a system with multiple representations by restricting polymorphic functions to range over all types, *given* *a representation.* If unspecified, this representation defaults to boxed types. Therefore, `app`^{1} as specified ranges over all boxed types.

To specify representations other than boxed types, we need a notion of representation in the source language. For this, in Haskell, *kinds* are used, which classify types. For example, all *monotypes* (that is, nullary type constructors, or types that do not take any further type arguments) have the kind *TYPE r*, for some *r* :: *Rep* [6]. The data type *Rep* is an ordinary ADT (lifted to a kind [31]) which contains a constructor for every representation.

$$\begin{array}{ll} \mathit{Int} & :: \mathit{TYPE}\ \mathit{LiftedRep} \\ \mathit{Float} & :: \mathit{TYPE}\ \mathit{LiftedRep} \\ \mathit{Int\#} & :: \mathit{TYPE}\ \mathit{IntRep} \\ \mathit{Float\#} & :: \mathit{TYPE}\ \mathit{FloatRep} \\ & \vdots \end{array}$$

Note that the representation for boxed closures is `LiftedRep` instead of something like `BoxedRep`. We further discuss levity in section 2.4. For now, it suffices to know that a type being lifted implies that it is boxed as well, which is why it is named as such.

If we want an alternative `app`^{2} that ranges over types with a representation other than boxed closures, we can use the *Rep* type to restrict the kind of accepted types by including a *kind constraint.* For example, we can imagine a function `app`^{2} that takes any type with an unboxed integer representation, and a function that turns this argument into some type with an unboxed floating-point representation. We can define it as follows:

```
app² :: ∀ (a :: TYPE IntRep) (b :: TYPE FloatRep). (a → b) → a → b
```

Note that `app`^{1} and `app`^{2} are both passed two bit patterns that encode their arguments. They differ exactly in how they *interpret* these patterns.

Now, given that Haskell has kind polymorphism [30, 31], one might expect to be able to formulate an alternative `app`^{3} that is polymorphic in its representation. Such definitions are called *levity polymorphic*[3], and can be defined as follows:

```
app³ :: ∀ (r :: Rep) (p :: Rep) (a :: TYPE r) (b :: TYPE p).
        a → (a → b) → b
```

While the specification for `app`^{3} correctly describes a levity-polymorphic function, it should be rejected, as it cannot be compiled[4]. If we do not know what the representation of `a` will be, we have no way of outputting the correct machine code. Therefore, there is the following principle concerning levity polymorphism: *Never move or store a levity-polymorphic value.* [6]. Based on this principle we can reject `app`^{3}.

---

[3] Something like *representation polymorphic* would be a more fitting description, as it is polymorphic in the representation of a type. However, for reasons similar to why the representation of boxed closures is called `LiftedRep` instead of `BoxedRep`, it is called *levity polymorphism* instead. See section 2.4.

[4] Note that there are levity polymorphic functions that *can* be compiled. Consider `($)`, which has type `∀ r a (b :: TYPE r). (a -> b) -> a -> b`. Eisenberg and Peyton Jones [6] refine the levity polymorphic principle, discussing what can be compiled, and what cannot.

2.4 Boxity & Levity

In section 2.3.2 we introduced the notion of boxity and described how types can be either boxed or unboxed. Then, in section 2.3.3, we mentioned the term levity, and how it implies boxity. This section explores these two definitions and describes how they relate.

Levity and boxity are different, but related terms that describe the representation of a type. Table 2.1 shows the four categories that arise.

	Boxed	Unboxed
Lifted	Int, Bool	
Unlifted	ByteArray$_{\#}$	Int$_{\#}$, Char$_{\#}$

Table 2.1: Boxity& Levity

The levity of a type refers to the strictness of the type. A lifted type is evaluated non-strictly. This means $\perp$ is an element of the type, and the type must be represented as closures to support thunks. Lifted types - at least for now - are always boxed, because closures cannot be represented on the stack, and therefore always are represented on the heap. Regular ADTs such as `Int` and `Bool` are examples of this category.

Unlifted types are evaluated strictly. This means that non-terminating terms are no longer a part of the type. As such non-terminating expressions will be evaluated eagerly, they will never evaluate into a value that can be bound. This means that unlifted types do not have to be represented as thunks. As unlifted types can be represented both on the stack on the heap, both categories have occupants.

The previously encountered unboxed types `Int#` and `Char#` are evaluated eagerly, and thus occupants of the unboxed, unlifted category. As of yet, we have not encountered the third category, which is boxed, unlifted types. One example of this is the `ByteArray#` type, which is a raw array of data values that lives on the heap.

The final category, which is unboxed, lifted types, is uninhabited, partially because of the aforementioned technical limitation (no support for closures on the stack). However, as established in section 2.2.1, unboxed closures will *have* to be represented as closures, which means that if we want to introduce unboxed closures, we have to introduce the ability to represent closures on the stack, removing the technical limitation.

Therefore, we ask the question: should unboxed closures be lifted, or unlifted?

Chapter 3

Problem statement

With the background covered, we can now formulate the problem we aim to solve.

3.1 Unboxing closures

Our main objective is to present a system that implements unboxed function closures. That is, we present a system that can represent function closures (as defined in section 2.2) on the stack.

In this section, we quantify what we mean with representing function closures on the stack, and discuss the biggest challenge of such functionality, which follows from the following observation:

Observation 3.1. With conventional function types, two unboxed closures with a different runtime representation can share the same type.

We focus on the biggest problem arising from this observation, which is that a closure's type does not indicate its runtime size. That is, closures of different runtime length can share the same type.

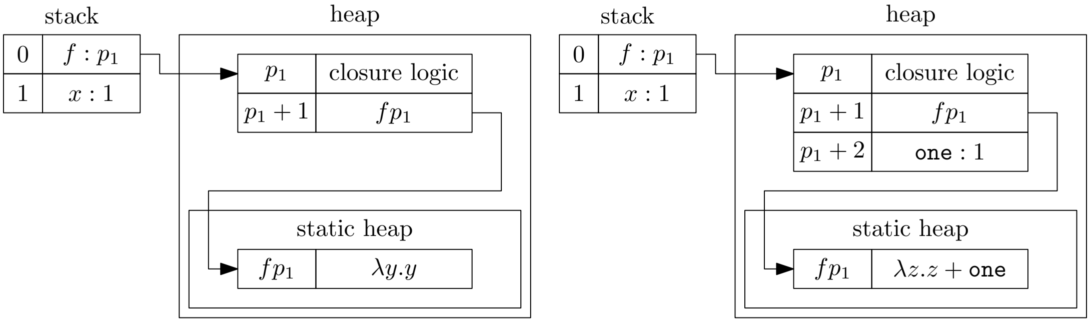
This can be a problem, as the stack - in contrast to the heap - is ill-equipped to deal with entities of unknown size. To understand this, consider the following examples. First, `app`^{4} is a version of the previously encountered `app` function, simplified to only range over `Int#`s. The examples `appID` and `appPlus` each apply `app`^{4} to a closure and the argument `1#`.

```
app^4 :: (Int# → Int#) → Int# → Int#
app^4 f x = f x
```

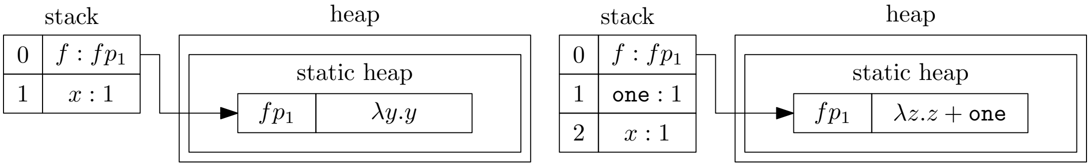
```
appID :: Int#
appID =
  let g y = y
  in app^4 g 1#
```

```
appPlus :: Int#
appPlus =
  let one = 1#
      h z = z +# one
  in app^4 h 1#
```

In the case of `appID`, the passed closure is the identity function, here named `g`. It does not close over any variables, so it can be represented as a pointer to `g`'s logic, in combination with an empty environment. In the case of `appPlus` however, the closure `h` closes over one variable, namely `one`. Therefore, along with a pointer to `h`'s logic, a binding for `one` needs to be stored as well. Note that closures store *a pointer to* their logic instead of the logic itself because it allows for the sharing of the static expression code across all dynamic instances of that closure [18].



(a) Layout stack and heap for boxed `appID`. (b) Layout stack and heap for boxed `appPlus`.



(c) Layout stack and heap for unboxed `appID`. (d) Layout stack and heap for unboxed `appPlus`.

Figure 3.1: Memory layouts of `appID` and `appPlus`, in boxed and unboxed case.

Along fig. 3.1 we will now examine the memory layout for the two applications to `app`^{4} in both the boxed and unboxed case.

3.1.1 The boxed case

Figures 3.1a and 3.1b display the memory layout once `app`^{4} has been applied to its arguments, in the case of the boxed alternatives of `appID` and `appPlus` respectively. On the left, we can see the stack, which in both cases stores a pointer to the closure at position 0, and the value for `x` (its second argument) on position 1.

On the right, we can see a representation of the heap. It is here where the two examples differ. As discussed, the heap representation in the case of `appID` consists of just some closure logic and the pointer fp_1. However, in the case of `appPlus`, a binding for `one` is stored as well.

Now we will examine what the logic for `app`^{4} must be, such that it can complete its operations in both cases. It is here where the functionality of the heap shines. The logic for `app`^{4} does not need to know the exact contents of the heap. It can simply force the pointer to the closure logic `f` with argument `x`. The closure logic takes care of calling the function logic by dereferencing fp_1, providing the bindings (if any) in the stored environment, and passing on the argument.

Note that here the distinction between *functions* and *function closures* becomes clear. While both are represented on the heap (in the boxed case), closures wrap function *pointers*, not function *logic*. It is the construct starting at p_1 that we want to unbox.

3.1.2 Unboxed case

In the unboxed case, one layer of indirection is removed. The pointer p_1 has been replaced by what in the boxed case was stored on the heap, with exception of the closure logic. We can see the effects of this in figs. 3.1c and 3.1d.

From these figures we can observe the following two issues:

1. The total length for the closure created by `appID` is 1, consisting of just the pointer to the expression logic fp_1. However, in the case of `appPlus`, the total length is 2, as it is increased by 1 due to the binding for `one`.
2. The logic stored in the heap-allocated closure - responsible for dereferencing fp_1 and passing any closed over variables and the argument - has disappeared from the closure representation.

Clearly, in the unboxed case, we must be able to differentiate between `appID` and `appPlus`. In section 4.2 we present our solution, which is an extension to the conventional types for functions such that function closures of varying representation have a varying type.

3.2 Motivation

This section discusses the motivation behind exploring the possibility of adding unboxed closures to Haskell.

3.2.1 Benefits

The motivation behind unboxed closures is twofold: they seem like a natural extension to the current system of unboxed types, and unboxed closures offer a speed benefit in certain situations.

Haskell is a language where functions (and therefore closures) are first-class citizens. However, the current unboxed types conflict with this idea, only allowing data values to be unboxed.

Furthermore, as the main bottleneck for most programs nowadays is memory access [5], the more efficient memory behaviour of unboxed closures (when compared to their boxed counterparts) can yield a performance gain. Consider again the examples in fig. 3.1.

In the unboxed case, the code for `app`[4] is more efficient, because it can skip a dereference. This saves instructions, and perhaps more importantly, reduces the interaction with the heap, which generally is much slower than the stack.

3.2.2 Drawbacks

Unfortunately, the solution we propose is not a free lunch. Firstly, in our solution, situations exist where unboxed closures require not only more stack space, but more memory in general. This is because of the way stacks operate when compared to heaps: stacks copy their values.

When allocating a new stack frame, all needed variables are copied into the new frame. This means that, if we pass a closure from frame to frame, each stack frame contains a copy of the stack representation of the closure. While this problem also exists in the boxed case, it is exacerbated in the unboxed case. In the boxed case, the copies are mere pointers. The actual closure lives on the heap, where there is only one copy. In the unboxed case, the entire closure is stored on the stack, which means that several copies of *the entire closure* can exist.

Furthermore, in some situations, we need some runtime metadata describing the contents of an unboxed closure, which carries a cost both in memory and instructions. We further describe this in section 8.1.

3.2.3 Trade-off

To conclude the motivation, we observe that in the solution we propose, unboxed closures - while worthwhile in some situations - are not strictly better than their boxed counterparts. Deciding in what situations unboxed closures are worthwhile depends on multiple factors, such as a willingness to sacrifice memory usage for a performance gain. That begin said, discovering where exactly this threshold lies is out of scope for this thesis and considered future work. We revisit the issue in section 9.3.

3.3 Approach

As described in the introduction, this thesis tries to answer the question “Can we add unboxed closures to Haskell”. However, presenting this functionality as a direct extension to the Haskell source language is infeasible. Therefore, following convention, we explore unboxed closures in a simpler lambda calculus.

A widely used approach is to use the reduced language that the full Haskell source is compiled[1] to: System F [9, 21, 22]. When presenting new language functionality, it is common to present this as a direct extension to (some variant of) System F [6, 26, 29, 30, 31], where later extensions often are based on previous extensions.

However, our situation differs from the above examples. Our main contribution is not the addition of unboxed closures to some high-level language, but rather a compilation stack below it that handle unboxed closures. Therefore, our high-level language can be fairly simple.

We follow the approach taken by Eisenberg and Peyton Jones [6], where we present two languages: $\mathcal{L}$ and $\mathcal{M}$.

$\mathcal{L}$ is a high-level language that contains the notion of unboxed closures. It is a mix between System F [26] and the STG machine [18]. The core of $\mathcal{L}$ is based on typed lambda calculus of System F. This alternative version of System F has been extended with some elements from the STG machine. Specifically, it adopts the STG representation of lambdas, where all lambdas are annotated with the set of closed over variables. $\mathcal{L}$ - just like System F - is typed, so the borrowed, untyped elements from the STG machine have been extended to their typed counterparts.

$\mathcal{M}$ is our lower-level language. The main goal of $\mathcal{M}$ is to show that our proposed system is implementable in a realistic compiler, by making it sufficiently close to a real machine. To do so we must be careful with the level of abstraction in $\mathcal{M}$. Setting the level of abstraction too low can be problematic, as this generally introduces noise in its presentation. However, if $\mathcal{M}$ is too high-level, it is no longer sufficiently close to a real machine, rendering our argument that unboxed closures as presented are implementable invalid.

Along with these languages, we present a compilation function $\mathcal{L} \to \mathcal{M}$, and prove it correct. By doing so we show that unboxed closures as presented in $\mathcal{L}$ can be expressed in terms of $\mathcal{M}$.

Scaling our solution to a realistic compiler like GHC[2] is beyond the scope of this thesis. However, as both $\mathcal{L}$ and $\mathcal{M}$ are approximations of existing components in the Haskell compilation process, an implementation strategy is implied.

---

[1] Or, more accurately, which Haskell is desugared in to.

[2] Glasgow Haskell Compiler [10].

Chapter 4

L

As introduced in section 3.3, $\mathcal{L}$ is our higher-level language. It is a relatively simple lambda calculus, based on System F, that has been extended to include unboxed function closures. In this section we present $\mathcal{L}$, by describing its grammar, typing rules, and operational semantics. We close with a section describing the type safety proof.

4.1 Grammar and typing

Figures 4.1 and 4.2 display the grammar and typing rules for $\mathcal{L}$.

$$\gamma \text{ Variables} \quad \alpha \text{ Type variables} \quad n \text{ Integer literals}$$

$$\begin{array}{rcll}
\nu & ::= & P\,A \mid U\,A & \text{Concrete reps.} \\
\kappa, \iota & ::= & \mathit{TYPE}\,\nu & \text{Kinds} \\
A & ::= & \Gamma \mid ? & \text{Annotations} \\
B & ::= & \mathit{Int} & \text{Base types} \\
\tau, \sigma & ::= & B \mid \tau_1 \xrightarrow{A} \tau_2 \mid \tau_1 \overset{A}{\rightsquigarrow} \tau_2 & \text{Types} \\
 & \mid & \alpha \mid \forall\,\alpha{:}\kappa.\,\tau & \\
e & ::= & \gamma \mid e\;\gamma \mid e\;\tau \mid \lambda\gamma{:}\tau.e & \text{Expressions} \\
 & \mid & \lambda_{\#}\gamma{:}\tau.e \mid n \mid \Lambda\alpha{:}\kappa.e & \\
 & \mid & \textbf{let}\ \gamma = e_1\ \textbf{in}\ e_2 & \\
 & \mid & \textbf{let}_{\#}\ \gamma = e_1\ \textbf{in}\ e_2 & \\
v & ::= & \lambda\gamma{:}\tau.e \mid \Lambda\alpha{:}\kappa.v \mid n & \text{Values} \\
\Gamma & ::= & \emptyset \mid \Gamma \bullet \gamma{:}\tau \mid \Gamma \bullet \alpha{:}\kappa & \text{Contexts}
\end{array}$$

Figure 4.1: $\mathcal{L}$ grammar

As stated, $\mathcal{L}$ is based on System F, the introduction of which we leave to existing literature [9, 21]. Instead, we focus on the particular language features added to support unboxed function closures.

$\boxed{\Gamma \vdash e : \tau}$ Term validity

$$\textsc{E\_Var}\ \frac{\gamma{:}\tau \in \Gamma}{\Gamma \vdash \gamma : \tau} \qquad \textsc{E\_IntLit}\ \frac{}{\Gamma \vdash n : \mathit{Int}}$$

$$\textsc{E\_App}\ \frac{\Gamma \vdash e : \tau_1 \xrightarrow{A} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2} \qquad \textsc{E\_App}_{\#}\ \frac{\Gamma \vdash e : \tau_1 \overset{A}{\rightsquigarrow} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2}$$

$$\textsc{E\_Lam}\ \frac{\Gamma \bullet \gamma{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\gamma{:}\tau_1.e : \tau_1 \xrightarrow{\Gamma} \tau_2} \qquad \textsc{E\_Lam}_{\#}\ \frac{\Gamma \bullet \gamma{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda_{\#}\gamma{:}\tau_1.e : \tau_1 \overset{\Gamma}{\rightsquigarrow} \tau_2}$$

$$\textsc{E\_Forget}\ \frac{\Gamma \vdash \gamma : \tau_1 \xrightarrow{A} \tau_2}{\Gamma \vdash \gamma : \tau_1 \xrightarrow{?} \tau_2} \qquad \textsc{E\_Forget}_{\#}\ \frac{\Gamma \vdash \gamma : \tau_1 \overset{A}{\rightsquigarrow} \tau_2}{\Gamma \vdash \gamma : \tau_1 \overset{?}{\rightsquigarrow} \tau_2}$$

$$\textsc{E\_TLam}\ \frac{\Gamma \bullet \alpha{:}\kappa \vdash e : \tau \qquad \Gamma \vdash_{\kappa} \kappa\,\mathrm{kind}}{\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\,\alpha{:}\kappa.\,\tau} \qquad \textsc{E\_TApp}\ \frac{\Gamma \vdash e : \forall\,\alpha{:}\kappa.\,\tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\textsc{E\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ P\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \textsc{E\_Let}_{\#}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ U\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$\boxed{\Gamma \vdash \tau : \kappa}$ Type validity

$$\textsc{T\_Int}\ \frac{}{\Gamma \vdash \mathit{Int} : \mathit{TYPE}\ P\ \emptyset} \qquad \textsc{T\_Var}\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

$$\textsc{T\_Arr}\ \frac{\begin{array}{c}\Gamma \vdash \tau_1 : \kappa_1 \\ \Gamma \vdash \tau_2 : \kappa_2\end{array}}{\Gamma \vdash \tau_1 \xrightarrow{A} \tau_2 : \mathit{TYPE}\ U\ A} \qquad \textsc{T\_Arr}_{\#}\ \frac{\begin{array}{c}\Gamma \vdash \tau_1 : \kappa_1 \\ \Gamma \vdash \tau_2 : \kappa_2\end{array}}{\Gamma \vdash \tau_1 \overset{A}{\rightsquigarrow} \tau_2 : \mathit{TYPE}\ U\ A}$$

$$\textsc{T\_AllTy}\ \frac{\Gamma \bullet \alpha{:}\kappa_1 \vdash \tau : \kappa_2 \qquad \Gamma \vdash_{\kappa} \kappa_1\,\mathrm{kind}}{\Gamma \vdash \forall\,\alpha{:}\kappa_1.\,\tau : \kappa_2}$$

$\boxed{\Gamma \vdash_{\kappa} \kappa\,\mathrm{kind}}$ Kind validity

$$\textsc{K\_Boxed}\ \frac{}{\Gamma \vdash_{\kappa} \mathit{TYPE}\ P\ A\ \ \mathrm{kind}} \qquad \textsc{K\_Unboxed}\ \frac{}{\Gamma \vdash_{\kappa} \mathit{TYPE}\ U\ A\ \ \mathrm{kind}}$$

$\boxed{\Gamma \vdash E}$ Environment validity

$$\textsc{EV\_Empty}\ \frac{}{\emptyset \vdash E} \qquad \textsc{EV\_Type}\ \frac{\Gamma \vdash E}{\Gamma \bullet \alpha{:}\kappa \vdash E} \qquad \textsc{EV\_Term}\ \frac{\begin{array}{c}\Gamma \vdash E \\ \Gamma \vdash e : \tau\end{array}}{\Gamma \bullet \gamma{:}\tau \vdash E, \gamma \mapsto e}$$

Figure 4.2: $\mathcal{L}$ typing

4.1.1 A-normal form

A language is in A-normal form (ANF) if all arguments to a function are trivial [7]. That is, intermediate results must be bound to a name before they can be used in any other context [1]. For $\mathcal{L}$ this means that arguments to functions are always variables γ instead of arbitrary expressions e.

The main reason $\mathcal{L}$ is in ANF is to match $\mathcal{M}$, which is in ANF because it allows for simpler evaluation. Because of this, we postpone discussing the motivation and consequences of ANF until section 5.2.6. Instead, here we only highlight the changes in the grammar of $\mathcal{L}$ that are necessitated by only allowing trivial function arguments.

We observe the restriction imposed by ANF in the grammar for expressions, listed in fig. 4.1. Here, application is of pattern $e\ \gamma$ instead of the more conventional pattern $e_1\ e_2$.

Setting just this restriction is not enough however, as it leaves us with a situation in which we can no longer bind expressions to variables. In a conventional lambda calculus, the only method of introducing such bindings is by applying functions to expressions, which is exactly what ANF prohibits. Therefore, $\mathcal{L}$ contains let expressions. As can be seen in fig. 4.1, we have two variants, namely **let** and $\textbf{let}_{\#}$. This duplication is a consequence of the addition of unboxed closures, which is discussed next.

4.1.2 Unboxed function closures

To add support for unboxed closures, we have added the unboxed alternatives for function expressions, denoted by $\lambda_{\#}$, function types, denoted by $\rightsquigarrow$, and let expressions, denoted by $\textbf{let}_{\#}$.

As described in section 3.1, our main challenge for implementing unboxed closures is that, with conventional function types, two closures with different runtime representation can have an equal type. Our solution to this problem is the annotation A on function arrows. We first discuss the grammar and typing, after which we discuss how this annotation allows us to statically differentiate between unboxed closures of varying representation.

As can be seen in fig. 4.1, both the boxed function arrow $\rightarrow$ and the unboxed function arrow $\rightsquigarrow$ feature an annotation A. Generally, this annotation is occupied by a typing environment Γ. As can be seen in rules E_Lam and E_Lam$_{\#}$ of fig. 4.2, this annotation is set during the typing of lambda expressions, and set to the Γ they are typed under. This annotation is then carried from the type level to the kind level, as shown by rules T_Arr and T_Arr$_{\#}$.

A consequence of annotating function arrows in such fashion is that functions have become less general. In conventional systems, a function with a closure as argument can accept *any* closure, as long as the argument and return type match. In our system, it requires that the closure's annotation matches the expected annotation. Therefore, all closures with a mismatched annotation are rejected.

To remedy this, annotations do not always have to be specified. Instead, they can be forgotten[1] to the annotation '?', as shown by rules E_Forget and E_Forget$_{\#}$ of fig. 4.2. The exact ramifications of this are discussed in section 8.1.3.

---

[1] Note that while we call this forgetting, no information is actually discarded, as the original type annotation is still part of the typing derivation.

4.2 Annotating arrows

The problem described in section section 3.1 is similar to the problem that adding existing unboxed types introduced: without further information, the same piece of logic is responsible for handling data of which the representation is not constant.

For the existing unboxed types (such as `Int#` and `Char#`), this problem is solved by restricting how functions can be polymorphic trough a kind constraint, as described in section 2.3. For unboxed closures, we propose a similar solution. However, as discussed, this solution cannot be applied to conventional function types, because closures of varying environments can have an equal type, and therefore equal kind. If we therefore want to encode representation information in the kind, we need to extend function types so that two closures with varying sets of closed over variable have different types. This then allows for setting kind constraints.

An important factor in this design is the granularity of the classification. While the conventional kind of function types is too coarse, we must be careful not to make the classification to fine. If we distinguish two closures of equal runtime representation we output the same code twice, thus causing unnecessary code duplication.

The principles we aim to satisfy are as follows:

1. Two items with varying representation must have a varying kind.
2. Two items with equal representation must have an equal kind.

It is clear that the current implementation of unboxed types adheres to these principles. `Int#` and `Char#` potentially[2] have a different representation. While both are represented as a non-floating-point word, `Int#` is signed, whereas `Char#` is unsigned, which means they are an instance of principle one. Correspondingly, they have a varying kind: `Int#` has kind *TYPE* `IntRep`, whereas `Char#` has kind *TYPE* `WordRep`[3].

For an example of principle two, consider `Word#`. While its type varies from `Char#`, its representation does not, as both are represented as an unsigned word-sized value. Correspondingly, the types share the same kind *TYPE* `WordRep`.

By annotating the set of closed over variables on the function type, we get a granularity that conforms with the two specified principles. To explain this, we imagine a version of $\mathcal{L}$ that has been extended with the (unboxed) base types `Int#`, `Word#`, and `Char#`.

For principle one, consider again the unboxed types `Int#` and `Char#`, but now occurring as the single closed over variable of two unboxed closures of unannotated type $\tau_1 \rightsquigarrow \tau_2$. Annotating the type of the closed over variable yields $\tau_1 \overset{\texttt{Int\#}}{\rightsquigarrow} \tau_2$ and $\tau_1 \overset{\texttt{Char\#}}{\rightsquigarrow} \tau_2$. The kinds corresponding to each type are *TYPE U* `IntRep` and *TYPE U* `WordRep`. These varying kinds allow us to correctly distinguish the two cases.

For principle two, consider an alternative to the above example with two closures where `Char#` and `Word#` occur as the single closed over variable instead. While their types will differ, their kinds will not, allowing us to catch both situations in the same constraint.

---

[2] Here we make no assumptions about a specific architecture. However, many architectures do not make this distinction. See section 8.1.

[3] `IntRep` and `WordRep` are constructors of `RuntimeRep`, see section 2.3.3.

4.2.1 Implementation in $\mathcal{L}$

One might observe that the solution as presented in this section does not fully match what is implemented in $\mathcal{L}$. This is true, as we have taken some following two liberties to simplify the design of $\mathcal{L}$.

Type list vs. Γ

As can be seen in fig. 4.1, types are not annotated with a list of types, but rather a full typing environment Γ. We have taken these liberties to simplify the compilation of $\mathcal{L}$. Therefore, we motivate this decision in section 6.3.

Closed over variables vs. entire Γ

Closures only need to store the variables closed over by the closure expression. Storing additional, unused bindings is inefficient, as they will never be used. Therefore, we can optimize for size, and include only the closed over variables in both the runtime closure and the annotation.

For $\mathcal{L}$, we make no such optimization, as our goal is to present the *possibility* of adding unboxed function closures, instead of an efficient implementation of them. Instead, as can be seen in rules E_Lam and $\text{E\_Lam}_{\#}$, the entire Γ is annotated.

4.3 Operational semantics

The operational semantics of $\mathcal{L}$ are displayed in fig. 4.3. The major differentiating factor between the semantics presented here and those of (variations of) System F is the way $\mathcal{L}$ deals with the binding and retrieving of variables. Whereas those languages usually implement a high-level approach for variable bindings (such as substitution semantics), $\mathcal{L}$ maintains an explicit environment E.

Such high-level approaches can work in systems where the semantics involving bindings are not the main subject of analysis. As substitution can be incredibly inefficient, any realistic implementation will opt to implement different semantics. This introduces a mismatch between the high-level language and the layers below, which may lead to problems. In our case, choosing substitution semantics for $\mathcal{L}$ means making mean some assumptions about the correctness of compilation. As these problems have already been studied in detail [4, 15, 23], systems that do not alter these semantics in any significant way can take this liberty, to simplify their design.

As we *are* introducing significant changes to the semantics resolving bindings, the last argument in our case does not apply. Therefore, we must be more explicit about the semantics involving bindings, even at our high-level language $\mathcal{L}$. Specifically, we maintain a set of bindings E that maps variables to expressions. Let bindings and applications introduce variables to this E, as shown in rules S_Let, $\text{S\_Let}_{\#b}$, S_Lam, and $\text{S\_Lam}_{\#}$. As we are not substituting away our variables, we need a rule that deals with them, as shown by rule S_Var. Variables are looked up in the environment E, such that the corresponding expression can be evaluated further.

We *do* maintain substitution semantics for typing abstractions, as shown by rule S_TBeta. As these abstractions are implemented as they are in conventional systems[4], we can abstract over their specifics following the same argument motivated above. Furthermore, as $\mathcal{L} \rightarrow \mathcal{M}$ is a type-erasing [21] compilation, typing abstractions and applications do not affect the operational

$$\text{S\_Var}\ \frac{\gamma \mapsto e \in E}{\langle \Gamma, E, \gamma \rangle \longrightarrow \langle \Gamma, E, e \rangle} \qquad \text{S\_App}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\langle \Gamma, E, e_1\ \gamma \rangle \longrightarrow \langle \Gamma', E', e_1'\ \gamma \rangle}$$

$$\text{S\_Lam}\ \frac{\begin{array}{c}\gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2\end{array}}{\langle \Gamma, E, (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle} \qquad \text{S\_Lam}_{\#}\ \frac{\begin{array}{c}\gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2\end{array}}{\langle \Gamma, E, (\lambda_{\#}\gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle}$$

$$\text{S\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto e_1\end{array}}{\langle \Gamma, E, \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle} \qquad \text{S\_Let}_{\#a}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\begin{array}{c}\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 \rangle \\ \longrightarrow \langle \Gamma', E', \mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2 \rangle\end{array}}$$

$$\text{S\_Let}_{\#b}\ \frac{\begin{array}{c}\Gamma \vdash v : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto v\end{array}}{\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = v\ \mathbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle} \qquad \text{S\_TLam}\ \frac{\langle \Gamma, E, e \rangle \longrightarrow \langle \Gamma', E', e' \rangle}{\langle \Gamma, E, \Lambda\alpha{:}\kappa.e \rangle \longrightarrow \langle \Gamma', E', \Lambda\alpha{:}\kappa.e' \rangle}$$

$$\text{S\_Tapp}\ \frac{\langle \Gamma, E, e \rangle \longrightarrow \langle \Gamma', E', e' \rangle}{\langle \Gamma, E, e\ \tau \rangle \longrightarrow \langle \Gamma', E', e'\ \tau \rangle} \qquad \text{S\_TBeta}\ \frac{\begin{array}{c}\Gamma_1 = \Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \\ \Gamma_2 = \Gamma \bullet \Gamma'\end{array}}{\begin{array}{c}\langle \Gamma_1, E, (\Lambda\alpha{:}\kappa.v)\ \tau \rangle \\ \longrightarrow \langle \Gamma_2[\tau/\alpha], E[\tau/\alpha], v[\tau/\alpha] \rangle\end{array}}$$

Figure 4.3: $\mathcal{L}$ operational semantics

semantics of $\mathcal{M}$, which means no assumptions of the correctness have to be made.

4.3.1 Let binding evaluation strategy

As we will further motivate in section 8.2, we have chosen to evaluate unboxed closures eagerly. As boxed closures remain lifted and therefore are evaluated non-strictly, $\mathcal{L}$ contains two alternatives for processing let bindings. Rule S_Let handles boxed closures, and stores the potentially non-value term e_1 in E, bound to γ. $\mathcal{L}$ allows for the unboxed let binding of arbitrary terms. Therefore, as can be seen in rules S_Let$_{\#a}$ and S_Let$_{\#b}$, non-value terms are stepped in-place. Only once a value has been found the let binding is fully processed.

4.4 Safety

We proof type safety by a combination of the following two properties, taken from Pierce and Benjamin [21]:

- Progress: *A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).*

---

[4] One could argue for the omission of typing abstractions from $\mathcal{L}$, as they do not influence unboxed function closures. However, they have been included to keep the presentation as close as possible to other works such as Eisenberg and Peyton Jones's presentation of levity polymorphism [6].

- Preservation: *If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.*

These properties are defined on *well-typed terms*. For $\mathcal{L}$, this condition is not strong enough, as the environment E influences how a term can step. For example, imagine trying to step a well-typed variable under an empty environment E. Such a state will fail, as S_VAR relies on the binding being present in E. We therefore extend our type safety theorems to hold on states $\langle \Gamma;\ E;\ e \rangle$ where $\Gamma \vdash e : \tau$ and $\Gamma \vdash E$. This way we eliminate the cases where E is malformed.

Theorem 4.1 (Progress)**.** *For any* $\langle \Gamma;\ E;\ e \rangle$*, if* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash E$*, then either* e *is a value, or there exists an* $\langle \Gamma';\ E';\ e' \rangle$ *such that* $\langle \Gamma;\ E;\ e \rangle \longrightarrow \langle \Gamma;\ E';\ e' \rangle$*.*

Theorem 4.2 (Preservation)**.** *If* $\langle \Gamma;\ E;\ e \rangle \longrightarrow \langle \Gamma';\ E';\ e' \rangle$*,* $\Gamma \vdash e : \tau$*, and* $\Gamma \vdash E$*, then* $\Gamma' \vdash e' : \tau$*, and* $\Gamma' \vdash E'$*.*

The proof for these theorems can be found in appendix A.

Chapter 5

M

As described in section 3.3, $\mathcal{M}$ is our lower-level language. The main goal of $\mathcal{M}$ is to show that our proposed system for unboxed closures is implementable in a realistic compiler, by making it sufficiently close to a real machine.

5.1 Grammar

Figure 5.1 displays the grammar for $\mathcal{M}$. For variables, y represents terms of boxed representation, and z represents terms of unboxed representation. Furthermore, x ranges over both representations.

For the most part, the expressions t of $\mathcal{M}$ correspond to the expressions e of $\mathcal{L}$, with two exceptions. First, as $\mathcal{M}$ is untyped, the $\mathcal{L}$ terms involving types do not have a counterpart in $\mathcal{M}$ (type abstractions $\Lambda\alpha{:}\kappa.e$ and type application $e\ \tau$), or have their type annotation removed (term abstractions $\lambda x.t$).

Second, where $\mathcal{L}$ makes a distinction between boxed term abstraction $\lambda\gamma{:}\tau.e$ and unboxed term abstraction $\lambda_{\#}\gamma{:}\tau.e$, $\mathcal{M}$ does not. Instead, it contains a singular grammatical construct for all

$$x \text{ Variables} \quad y \text{ Pointer variables} \quad z \text{ Unboxed variables}$$

$$\begin{array}{rcll}
b & ::= & p \mid (w, \Delta) & \text{Bit patterns} \\
x & ::= & y \mid z & \text{Variables} \\
t & ::= & x \mid t\ x \mid \lambda x.t \mid n & \text{Expressions} \\
 & \mid & \textbf{let}\ y = t_1\ \textbf{in}\ t_2 \mid \textbf{let}_{\#}\ z = t_1\ \textbf{in}\ t_2 & \\
w & ::= & \lambda x.t \mid n & \text{Values} \\
S & ::= & \emptyset \mid \text{App}(b) \bullet S \mid \text{Let}(z, t, \Delta) \bullet S & \text{Continuation Stack} \\
\Delta & ::= & \emptyset \mid y \mapsto p \bullet \Delta \mid z \mapsto (t, \Delta) \bullet \Delta & \text{Environment} \\
H & ::= & \emptyset \mid p \mapsto (t, \Delta) \bullet H & \text{Heap} \\
i & ::= & t \mid b & \text{Work items} \\
\mu & ::= & \langle i;\ \Delta;\ S;\ H \rangle & \text{Machine states}
\end{array}$$

Figure 5.1: $\mathcal{M}$ grammar

abstractions, $\lambda x.t$, and instead represents them either as boxed or unboxed closures depending on what let binding has been used to introduce the closure: **let** introduces boxed closures, and $\textbf{let}_{\#}$ introduces unboxed closures.

It is at the point of binding storage where $\mathcal{L}$ and $\mathcal{M}$ differ significantly. Whereas $\mathcal{L}$ maintains a single environment E (containing bindings of patterns $\gamma \mapsto e$), $\mathcal{M}$ contains a split design. Here, the environment Δ maps variables to bit patterns, and the heap H maps pointers (which are bit patterns) to closures.

A key detail in this is that bit patterns b are not exclusively pointers, but instead can also represent closures directly. Therefore, if we want to extend some environment Δ and heap H with a binding of some variable x to some closure (t, Δ'), we can proceed in two ways, depending on the boxity of the variable. In the boxed case, $x = y$, and we create a new pointer p, map y to p on the environment Δ, and map p to the closure on the heap H, which yields $y \mapsto p \bullet \Delta$ and $p \mapsto (t, \Delta') \bullet H)$. In the unboxed case, $x = z$, and we map z directly to the closure on the environment Δ, which yields $y \mapsto (t, \Delta') \bullet \Delta$ and H.

Finally, we have our machine states μ, which is a quad consisting of a work item i (which is either a term t or a bit pattern b), an environment Δ, a (continuation) stack S, and a heap H.

5.2 Operational semantics

For the operational semantics, displayed in fig. 5.2, we first observe that where $\mathcal{L}$ exclusively deals with terms, $\mathcal{M}$ mostly deals with closures. In fact, the only times where a term occurs outside of a closure is when it is currently under evaluation, such that it can be converted[1] into a closure. We first examine the rule that does this conversion: rule LIFT.

5.2.1 Lifting values to closures

Whenever our current work item has been evaluated to a value w under environment Δ_1, LIFT converts the work item to a closure (w, Δ_2) such that Δ_2 contains the closed over variables of w. This rule makes two assumptions. First, it assumes knowledge of the closed over variables of w. Furthermore, we assume that all closed over variables of w are present in Δ_1, i.e. $\Delta_2 \subseteq \Delta_1$.

These assumptions do not hold for all valid $\mathcal{M}$ programs. However, as we are using $\mathcal{M}$ as a compilation target, we only need to consider the subset of programs that can be the output of compilation. In other words, we only need to consider the *image* of the compilation function presented in chapter 6.

The static knowledge of the closed over variables of w follows from the fact that in $\mathcal{L}$, its set of closed over variables is annotated. The subset constraint is proven to hold in the correctness proof of the compilation function, which is discussed in chapter 7.

Note that even though this set is labelled as 'free variables', the type annotations in $\mathcal{L}$ feature the *entire* environment at the time of encountering a lambda, as discussed in section 4.2.1. Therefore, this set may contain bindings not used by the closure.

---

[1] Note that while we use the term 'converted', we do not apply *closure conversion* [13]. That is, we assume a binding to exist in our current environment upon encountering a variable, instead of being passed an explicit environment (as does $\mathcal{L}$).

$$
\begin{array}{rcll}
\langle\ \ (w,\Delta_1);\ \Delta_2;\ \emptyset;\ H\rangle & \longrightarrow & \textbf{return}\ w & \text{RET}\\
\langle\ \ (w,\Delta_1);\ \Delta_2;\ \text{Let}(z,t,\Delta_3)\bullet S;\ H\rangle & \longrightarrow & \langle t;\ z\mapsto(w,\Delta_1)\bullet\Delta_3;\ S;\ H\rangle & \text{POP-L}\\
\langle(\lambda y.t,\Delta_1);\ \Delta_2;\ \text{App}(p)\bullet S;\ H\rangle & \longrightarrow & \langle t;\ y\mapsto p\bullet\Delta_1;\ S;\ H\rangle & \text{POP-A}\\
\langle(\lambda z.t,\Delta_1);\ \Delta_2;\ \text{App}(t,\Delta_3)\bullet S;\ H\rangle & \longrightarrow & \langle t;\ z\mapsto(t,\Delta_3)\bullet\Delta_1;\ S;\ H\rangle & \text{POP-A}_{\#}\\
\hline
\langle t\ x;\ \Delta[x]=b;\ S;\ H\rangle & \longrightarrow & \langle t;\ \Delta;\ \text{App}(b)\bullet S;\ H\rangle & \text{APP}\\
\\
\langle w;\ \Delta_1;\ S;\ H\rangle & \longrightarrow & \langle(w,\Delta_2);\ \Delta;\ S;\ H\rangle & \text{LIFT}\\
 & & \text{where } \Delta_2 = fv(w),\ \Delta_2\subseteq\Delta_1 & \\
\\
\langle\textbf{let}\ \ y=t_1\ \textbf{in}\ t_2;\ \Delta;\ S;\ H\rangle & \longrightarrow & \begin{array}{l}\langle t_2;\ y\mapsto p\bullet\Delta;\\ \quad S;\ p\mapsto(t_1,\Delta)\bullet H\rangle\end{array} & \text{LET}\\
\langle\textbf{let}_{\#}\ z=t_1\ \textbf{in}\ t_2;\ \Delta;\ S;\ H\rangle & \longrightarrow & \langle t_1;\ \Delta;\ \text{Let}(z,t_2,\Delta)\bullet S;\ H\rangle & \text{LET}_{\#}\\
\hline
\langle y;\ \Delta_1[y]=p\ \ ;\ S;\ H\rangle & \longrightarrow & \langle p;\ \Delta_1;\ S;\ H\rangle & \text{VAR-E}\\
\langle z;\ \Delta_1[z]=(w,\Delta_2);\ S;\ H\rangle & \longrightarrow & \langle(w,\Delta_2);\ \Delta_1;\ S;\ H\rangle & \text{VAR-E}_{\#}\\
\\
\langle p;\ \Delta_1;\ S;\ H[p]=(w,\Delta_2)\rangle & \longrightarrow & \langle(w,\Delta_2);\ \Delta_1;\ S;\ H\rangle & \text{VAR-H}\textsc{v}\\
\langle p;\ \Delta_1;\ S;\ H[p]=(t,\Delta_2)\rangle & \longrightarrow & \langle t;\ \Delta_2;\ S;\ H\rangle & \text{VAR-H}\textsc{t}
\end{array}
$$

Figure 5.2: $\mathcal{M}$ operational semantics

5.2.2 Variable lookup

For lookup, the goal is to find the closure the variable is mapped to, and evaluate its inner term to a value - if it is not already. The sequence of steps to achieve this differ on the representation of the closure the variable represents. If that closure is in boxed representation, $x = y$, and when looked up in the environment, a pointer p is found, as shown by rule VAR-E. This pointer is made the work item, after which it can be looked up on the heap, as shown by rules VAR-Hv and VAR-Ht. Here, the steps once again differ on whether the term stored by the closure is a value or not. If it is a value w, the entire closure is set as work item. If it is a non-value t, the stored term t is made the work item, and the environment stored in the closure is made the working environment. Evaluation of t can now proceed under its stored environment. After the term has been evaluated to a value, it is lifted back to a closure.

In the unboxed case, $x = z$. As unboxed closures are evaluated eagerly, the closure found in Δ is always already a value. Therefore, we proceed like the value case of boxed closures, by setting the entire closure as work item, as shown by rule VAR-E$_{\#}$.

Note that this is one of the places the efficiency of unboxed closures is visible. In the boxed case, we must first find a pointer p, and then resolve it to a closure. In the unboxed case we can skip this step, and find the closure on the stack immediately.

5.2.3 Introducing bindings

As described in section 5.1, $\mathcal{M}$ does not differentiate between boxed and unboxed term abstractions via the term itself: both are denoted as $\lambda x.t$. Instead, $\mathcal{M}$ represents term abstractions differently depending on what let construct is used: **let** and **let**$_{\#}$ denote bindings to be stored in boxed and unboxed representation respectively. Rules LET and LET$_{\#}$ process these expressions.

Rule LET processes boxed bindings. As boxed closures are not strictly evaluated, the bound term t_1 is combined with the working environment Δ, and stored on the environment and heap via the pointer p. As unboxed closures are evaluated strictly, the bound term t_1 is made the work item, such that it is evaluated into a closure.

As the operational semantics of $\mathcal{L}$ (fig. 4.3) uses inference rules, we can state that if $\langle \Gamma;\ E;\ e_1 \rangle \longrightarrow \langle \Gamma';\ E';\ e_1' \rangle$, then $\langle \Gamma;\ E;\ \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma';\ E';\ \mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2 \rangle$. In $\mathcal{M}$, we do not have this option: we can only modify the machine state, and do not have an assumption or conclusion by which we can "remember" that we are evaluating a subterm of a let expression. Therefore, $\mathcal{M}$ uses a continuation stack for cases where we need to evaluate a subterm while remembering the bigger context.

For unboxed let bindings the Let continuation is used, as shown by rule LET$_{\#}$. This continuation stores the variable bound to z, the inner term t_2, and the environment Δ at the time of encountering the let binding. As the let-bound term t_1 is made the work item, it will eventually be evaluated into a closure. By rule POP-L we can process the binding with the contextual information in the continuation, which involves extending the stored environment with a binding of the saved variable to the found closure, which is then set as working environment. The saved inner term t_2 can now be set as work item and evaluated under an environment containing the let bound term.

5.2.4 Processing applications

Rules APP, POP-A, and POP-A$_{\#}$ process applications. By the grammar of section 5.1, applications are always is of pattern $t\ x$, where t is a term. Rules POP-A and POP-A$_{\#}$ expect a *closure* rather than a term. Therefore, upon encountering an application $t\ x$, evaluation always switches to the subterm t, even in the case where it is of form $\lambda x.t$.

Like with let bindings, the subterm needs to be evaluated while saving the bigger context, which is done by the App continuation. As only the bit pattern corresponding to the variable applied to is needed, it is looked up in the environment and stored in the continuation. It would be possible for APP to skip the lookup and store the variable instead. However, in this case the environment would have to be stored as well, and rules POP-A and POP-A$_{\#}$ would have to switch to the saved context to fetch the bit pattern, only to immediately switch to the context stored by the closure. As looking up the bit pattern does not evaluate it in any way, the order does not matter, which is why we chose the simpler version.

When term t has been evaluated to a closure with an App continuation on the head of the continuation stack, rules POP-A and POP-A$_{\#}$ switch to the function's inner term, while extending the closure's environment with the saved bit pattern to the function's argument.

5.2.5 Terminal states

States of pattern $\langle (w, \Delta_1);\ \Delta_2;\ \emptyset;\ H \rangle$ are terminal, which means evaluation stops and the value w can be returned, as shown by rule RET. This rule could have been omitted, but has been included to make this process explicit.

5.2.6 A-normal form

Now that we have introduced the operational semantics, we can revisit the motivation for having $\mathcal{M}$ in ANF, as introduced in section 4.1.1. In short, disallowing non-trivial arguments allows for a simpler design, where let bindings introduce bindings, and applications apply functions to arguments.

Because arguments in $\mathcal{M}$ are trivial, rule APP can simply look up the bit pattern b on the environment, and proceed with the application. Allowing non-trivial arguments (that is, applications of patterns $e_1\ e_2$) burdens APP with the task of first converting the argument e_2 to a bit pattern b, before proceeding with processing the application.

As is shown by rules LET and $\text{LET}_{\#}$, processing the introduction of a new binding is a fairly complex affair. If $\mathcal{M}$ were to allow for non-trivial applications, it would either have to duplicate the logic of LET and $\text{LET}_{\#}$, or the rules would have to be merged, depending on whether let bindings are a part of this imaginary version of $\mathcal{M}$.

Furthermore, two let constructs are used, to indicate the intended representation of the bound term. Allowing non-trivial arguments would therefore necessitate two application operators, as the pattern $e_1\ e_2$ does not indicate the intended representation of e_2.

Chapter 6

Compilation

The compilation rules of $\mathcal{L}$ to $\mathcal{M}$ are displayed in fig. 6.1. Compilation is of pattern $[\![e]\!]^{\Gamma}$, where e is a $\mathcal{L}$ expression, and Γ the environment e is typed under.

6.1 Translating variables based on Γ

A crucial detail of our design is that the subscripted Γ is always representative of the runtime environment of $\mathcal{M}$. We can see this at work in rule C_VAR. The typing environment Γ is converted to a list of kinds κ via the unspecified (but trivial) operation $kindsOf$.

This list, along with the variable that is being translated (γ), is passed to an abstract operation called *lookup*. Because κ is representative of the runtime, the variable x that *lookup* outputs can be some static identifier, such as a De Bruijn level[1][3] or a stack offset.

We would have liked to express this property of *lookup* as a theorem, while continuing to abstract over the exact binding resolution strategy. However, we have not been able to find any literature that establishes these kinds of properties. Furthermore, our efforts to develop our own methodology for describing these properties have come up short, as any attempt at describing such theorem required us to assume a specific binding resolution strategy. We revisit this issue in our discussion of future work, section 9.2.

What we *can* do is give an overview of the high-level implementation of *lookup*. Its task is to output a variable inside the chosen binding resolution strategy based on the $\mathcal{L}$ variable γ and the list of kinds κ. Because this list is representative of the runtime situation, *lookup* can use this information while calculating the variable.

For example, if the chosen binding resolution strategy is a stack with stack pointer, and variables are represented as offsets to this pointer, *lookup* can determine the length of all variables stored before the variable in consideration to determine the offset. If γ is represented by a kind at position n in κ, then *lookup* can add the length of all kinds in positions 0 to $n-1$ to find the offset γ is stored at.

---

[1] Sometimes called reversed De Buijn's indexing [2], not to be confused with a (regular) De Bruijn index. With levels, n represents the nth item from the *top* of the stack, instead of the bottom.

$$\textsc{C\_Var}\ \frac{\kappa = \mathit{kindsOf}(\Gamma) \quad x = \mathit{lookup}(\kappa, \gamma)}{\llbracket \gamma \rrbracket^{\Gamma} = x} \qquad \textsc{C\_IntLit}\ \frac{}{\llbracket n \rrbracket^{\Gamma} = n} \qquad \textsc{C\_App}\ \frac{\llbracket e \rrbracket^{\Gamma} = t \quad \llbracket \gamma \rrbracket^{\Gamma} = x}{\llbracket e\ \gamma \rrbracket^{\Gamma} = t\ x}$$

$$\textsc{C\_TLam}\ \frac{\llbracket e \rrbracket^{\Gamma} = t}{\llbracket \Lambda \alpha{:}\kappa.e \rrbracket^{\Gamma} = t} \qquad \textsc{C\_TApp}\ \frac{\llbracket e \rrbracket^{\Gamma} = t}{\llbracket e\ \tau \rrbracket^{\Gamma} = t}$$

$$\textsc{C\_Let}\ \frac{\begin{array}{ll} \kappa = \mathit{kindsOf}(\Gamma) & \llbracket e_1 \rrbracket^{\Gamma} = t_1 \\ x = \mathit{fresh}(\kappa) & \llbracket e_2 \rrbracket^{\Gamma \bullet \gamma:\tau} = t_2 \end{array}}{\begin{array}{c} \llbracket \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 \rrbracket^{\Gamma} \\ = \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \end{array}} \qquad \textsc{C\_Let}_{\#}\ \frac{\begin{array}{ll} \kappa = \mathit{kindsOf}(\Gamma) & \llbracket e_1 \rrbracket^{\Gamma} = t_1 \\ x = \mathit{fresh}(\kappa) & \llbracket e_2 \rrbracket^{\Gamma \bullet \gamma:\tau} = t_2 \end{array}}{\begin{array}{c} \llbracket \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 \rrbracket^{\Gamma} \\ = \mathbf{let}_{\#}\ x = t_1\ \mathbf{in}\ t_2 \end{array}}$$

$$\textsc{C\_Lam}\ \frac{\begin{array}{ll} \multicolumn{2}{c}{\kappa = \mathit{kindsOf}(\Gamma_2)} \\ \Gamma_1 \vdash \lambda\gamma{:}\tau_1.e : \tau_2 & x = \mathit{fresh}(\kappa) \\ \Gamma_1 \vdash \tau_2 : \mathit{TYPE}\ P\ \Gamma_2 & \llbracket e \rrbracket^{\Gamma_2 \bullet \gamma:\tau} = t \end{array}}{\llbracket \lambda\gamma{:}\tau_1.e \rrbracket^{\Gamma_1} = \lambda x.t} \qquad \textsc{C\_Lam}_{\#}\ \frac{\begin{array}{ll} \multicolumn{2}{c}{\kappa = \mathit{kindsOf}(\Gamma_2)} \\ \Gamma_1 \vdash \lambda_{\#}\gamma{:}\tau_1.e : \tau_2 & x = \mathit{fresh}(\kappa) \\ \Gamma_1 \vdash \tau_2 : \mathit{TYPE}\ U\ \Gamma_2 & \llbracket e \rrbracket^{\Gamma_2 \bullet \gamma:\tau} = t \end{array}}{\llbracket \lambda_{\#}\gamma{:}\tau_1.e \rrbracket^{\Gamma_1} = \lambda x.t}$$

Figure 6.1: Compilation of $\mathcal{L}$ to $\mathcal{M}$

6.2 Maintaining a representative Γ

Rules C_IntLit, C_App, C_TLam, and C_TApp do not introduce any new bindings. The compilation of these rules therefore is relatively straightforward and therefore is not discussed further. Instead, we only discuss the rules that *do* deal with a change in environments, which are the rules for compiling let bindings and lambdas.

6.2.1 Introducing variables

Rules C_Let and C_Let$_{\#}$ compile boxed and unboxed let bindings. They use another abstract operation, fresh, that examines the list of kinds κ to generate an x representing the new closed over variable. The operation fresh enjoys the same guarantees about κ as *lookup*: based on κ, the runtime situation is known statically, which means x again can be some static identifier.

The let bound expression e_1 is compiled under the given Γ. However, expression e_2 is compiled under just Γ extended with a binding for e_1. In the unboxed case this differs with $\mathcal{L}$'s semantics, as all bindings introduced during the evaluation of e_1 are maintained (rule S_Let$_{\#a}$) and not removed once the binding is finally processed (rule S_Let$_{\#b}$). In contrast, $\mathcal{M}$ saves the environment upon encountering the binding in a Let continuation (rule LET), and restores this environment once the continuation is popped (rule POP-L). As the subscripted Γ needs to represent the runtime environment, we follow this behaviour by compiling e_2 under just $\Gamma \bullet \gamma : \tau$.

6.2.2 Entering lambdas

Finally, we discuss rules C_Lam and C_Lam$_{\#}$. The crucial detail of both these rules is that they compile their body under the environment stored in its type (Γ_2) instead of the environment that is passed (Γ_1).

The $\mathcal{M}$ operational semantics rules POP-A and POP-A$_{\#}$ switch to the environment stored in the closure. Therefore, compiling the body under Γ_1 will not work, as then it no longer is representative of the runtime. However, the environment annotated on the lambda's type (labelled as Γ_2) *is* representative, as it matches with the closures that $\mathcal{M}$ creates. In $\mathcal{M}$, this happens in rules LIFT and LET.

The closure rule LIFT creates is based on Γ_2, as discussed in section 5.2.1. Therefore, the closure is trivially represented by Γ_2.

Second is rule LET, which creates a closure (t, Δ). Here, Γ_2 and Δ each consist of the full environment of the time of encountering the term: rule E_LAM annotates the full environment at the time of encountering, as does LET. Therefore, Γ_2 represents Δ.

6.3 Type list vs. Γ

As motivated by section 4.2.1, only a list of types representing the closed over variables is needed as annotation on the function arrows. Instead, we have annotated an entire typing environment Γ.

This simplifies the design, as it allows us to retrieve the Γ from the kind of a lambda expression. If this was just a list of types or kinds, we would have to have additional functionality that relates lambda expressions to the environment they were defined in.

In the end, this change does not matter. As $kindsOf$ filters out everything except variable bindings, *lookup* is passed a list equal to the list it would be passed if we were annotating type lists instead of typing environments.

Chapter 7

Semantics preserving compilation

Now that we have presented $\mathcal{L}$, $\mathcal{M}$, and a compilation function $\mathcal{L} \to \mathcal{M}$, which we will denote as c, we want to proof the correctness of this function. In this section we discuss our approach to this proof, and elaborate on why the properties that we prove implies the compilation is correct.

In general, a compiler's goal is to take a program written in one language and to output a program in another language that "does the same thing". While compilers may implement various optimizations and other transformations, these changes (should) only affect *how* the end result is computed, and not the end result itself. Even changes in the computation must be carefully analysed, as they can influence the end result, particularly when termination is considered. If the input program does not terminate, then the output program should also not terminate (and vice versa).

Proving that, for any $l \in \mathcal{L}$, l "does the same thing" as its compiled result $c(l)$, will require us to further define what this relation is. What does it mean for a program in $\mathcal{L}$ and a program in $\mathcal{M}$ to do the same thing?

A naive approach could be to require that a program l and its compilation $c(l)$, when evaluated to completion, should both find values that on a bit level are equal. There is a major problem with such a definition: it assumes that the same bit patterns *encode* the same information, which may not be true. For example, the decimal 1 encoded in binary using little-endian is `00000001`. The same pattern in big-endian represents the decimal 128!

Clearly, we need to include the semantics of both languages into our definition of "doing the same thing". In an ideal world, such semantics preservation theorem [14, 17] would look something like fig. 7.1. Here, e, e' are terms in the source language, and t, t' are terms in the target language. The theorem states that if e steps to e', the compilation of e (which is t), steps to t' such that the decoding of t' (indicated by c^{-1}) yields e'.

In practice however, such theorem is hard to prove. The rest of this chapter explains why this is the case, and what alternate theorems we proof to yield a similar property.

7.1 Eventual correctness

As $\mathcal{L}$ is type-safe and $\mathcal{M}$ is in the image of $\mathcal{L}$, every $\mathcal{M}$ program that is the output of compiling an $\mathcal{L}$ program is expected to evaluate into a value.

$$\begin{tikzcd}
\text{Source} & \text{Target} \\
e \arrow[r, "c"] \arrow[d, "1"] & t \arrow[d, "1"] \\
e' & t' \arrow[l, "c^{-1}"']
\end{tikzcd}$$

Figure 7.1: Semantics preservation

We can utilize this fact by proving something that we dub eventual correctness. In general, we want to prove that, for any $\mathcal{L}$ program that evaluates to an observable value [9] (which in $\mathcal{L}$ are only integers i), we can compile the program, evaluate it to an observable value in $\mathcal{M}$ (also exclusively integers i), such that the value obtained is observationally equivalent to the value $\mathcal{L}$ finds.

Such theorem can easily be obtained from the semantics preservation theorem, as displayed in fig. 7.2 below.

$$\begin{tikzcd}
e \arrow[r, "c"] \arrow[d, "1"] & t \arrow[d, "1"] \\
e' \arrow[r, shift right, "c"'] \arrow[d, "*"] & t' \arrow[l, shift right, "c^{-1}"'] \arrow[d, "*"] \\
i_{\mathcal{L}} & i_{\mathcal{M}} \arrow[l, "c^{-1}"']
\end{tikzcd}$$

Figure 7.2: Full evaluation semantics preservation

Note that here $\xrightarrow{*}$ is used, which is the reflexive transitive closure on $\longrightarrow$. For $\mathcal{L}$ and $\mathcal{M}$ this relation has been defined in appendix B.2, definitions B.1 and B.2

7.1.1 Translating states over terms

The description as given ranges over terms e and t. However, as already discussed during the type safety proofs of $\mathcal{L}$ (section 4.4), the semantics of a term is coupled to the environment it is defined in. Therefore, we need to extend the compilation rules as presented in order to translate $\mathcal{L}$ states to $\mathcal{M}$ states, instead of $\mathcal{L}$ terms to $\mathcal{M}$ terms.

For closed terms this environment is empty. However, proving eventual correctness requires us to be able to translate open terms as well. Therefore, we formulate two new operations, namely the translation of environments E, and the compilation of $\mathcal{L}$ states $\langle \Gamma;\ E;\ e \rangle$.

Environment translating

The rules for translating environments E are given below. Since E does not store typing derivations, Γ is passed during translation as well, such that the type of e can be determined.

$$\textsc{Tr\_Empty}\ \frac{}{[\![\emptyset]\!]^{\Gamma} = (\emptyset, \emptyset)}$$

$$\textsc{Tr\_Boxed}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \Gamma \vdash \tau : \mathit{TYPE}\ P\ A \\ (\Delta, H) = [\![E]\!]^{\Gamma} \\ p = \mathit{fresh}(H) \\ [\![\gamma]\!]^{\Gamma} = y \\ [\![e]\!]^{\Gamma} = t \\ \Delta' = y \mapsto p \bullet \Delta \\ H' = p \mapsto (t, \Delta) \bullet H\end{array}}{[\![E, \gamma \mapsto e]\!]^{\Gamma} = (\Delta', H')} \qquad \textsc{Tr\_Unboxed}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \Gamma \vdash \tau : \mathit{TYPE}\ U\ A \\ (\Delta, H) = [\![E]\!]^{\Gamma} \\ \\ [\![\gamma]\!]^{\Gamma} = z \\ [\![v]\!]^{\Gamma} = w \\ \Delta' = z \mapsto (w, \Delta) \bullet \Delta \\ H' = H\end{array}}{[\![E, \gamma \mapsto v]\!]^{\Gamma} = (\Delta', H')}$$

State translating

The translation of states is defined as follows. $[\![\langle \Gamma;\ E;\ e\rangle]\!]^{S} = \langle [\![e]\!]^{\Gamma};\ \Delta;\ S;\ H\rangle$ where $[\![E]\!]^{\Gamma} = (\Delta, H)$. Note that the compilation takes an $\mathcal{M}$ stack S, which is needed for translating open terms.

7.1.2 Decoding value states

Defining a decode operation on arbitrary $\mathcal{M}$ terms is non-trivial. However, for eventual correctness, we are only interested in proving that the fully evaluated $\mathcal{M}$ value is observationally equivalent to the value $\mathcal{L}$ finds.

We do not need a full definition of observational equivalence [9] for our proof. Instead, we leave its definition abstract, and assume the following (in our opinion reasonable) property:

Assumption 7.1 (Compiled integers are observationally equivalent)**.** *For any $\mathcal{L}$ state $\langle \Gamma;\ E;\ i_{\mathcal{L}}\rangle$, if $\Gamma \vdash v : \tau$ and $\Gamma \vdash E$, then $[\![\langle \Gamma;\ E;\ v\rangle]\!]^{\emptyset} = \langle i_{\mathcal{M}};\ \Delta;\ \emptyset;\ H\rangle$, and $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.*

Note that here the integers i are subscripted with either $\mathcal{L}$ or $\mathcal{M}$, to indicate what language they are in.

7.1.3 Definition

We now have enough information to define our eventual correctness theorem:

Theorem 7.2 (Eventual correctness)**.** *If $\langle \emptyset;\ \emptyset;\ e\rangle \xrightarrow{*} \langle \Gamma;\ E;\ i_{\mathcal{L}}\rangle$ and $[\![\langle \emptyset;\ \emptyset;\ e\rangle]\!]^{\emptyset} = \langle t;\ \emptyset;\ \emptyset;\ \emptyset\rangle$, then there exists a $\langle i_{\mathcal{M}};\ \Delta;\ \emptyset;\ H\rangle$ such that $\langle t;\ \emptyset;\ \emptyset;\ \emptyset\rangle \xrightarrow{*} \langle i_{\mathcal{M}};\ \Delta;\ S;\ H\rangle$ and $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.*

Note that here we see that eventual correctness has been defined for closed terms only: the typing environment Γ, environment E, and stack S are all empty.

Here we only discuss the proof on a high level, as the full proof can be found in appendix C.3. Our proof relies heavily on the simulation theorem, which we discuss next. After this we present how we use simulation to prove eventual correctness.

7.2 Simulation

Our simulation theorem uses two new notions, which we discuss first, after which the simulation theorem is introduced.

7.2.1 Extension

First, we introduce the notion of extension, which is defined on states and its components. Its exact definition can be found in appendix B.2, definitions B.7 to B.9.

Let $Q_1 = \langle t_1;\ \Delta_1;\ S_1;\ H_1\rangle$ and $Q_2 = \langle t_2;\ \Delta_2;\ S_2;\ H_2\rangle$. On a high level, Q_1 is extended by Q_2, written $Q_1 \sqsubseteq Q_2$, if Q_2 contains at least the bindings Q_1 does. This can be thought of as a subset relation, although the specifics are slightly more involved due to the split nature of $\mathcal{M}$'s binding environment Δ and heap H.

7.2.2 $\mathcal{M}$ well-formedness

We define a well-formedness judgment on $\mathcal{M}$ states, written $\langle i;\ \Delta;\ S;\ H\rangle\ \mathsf{WF}$, such we can exclude malformed states. Its exact definition can be found in appendix B.2, definitions B.3 to B.6. On a high level, it can be compared to the $\mathcal{L}$ environment judgment $\Gamma \vdash E$, as it makes sure that a binding exists for every reachable variable in a state.

Its main usage is not for proving simulation, but for proving eventual correctness *based on* simulation, which we discuss in section 7.3.

7.2.3 Definition

We are now ready to introduce the simulation theorem, which has been given below.

Theorem 7.3 (Simulation)**.** *For all* $\langle\Gamma;\ E;\ e\rangle \longrightarrow \langle\Gamma';\ E';\ e'\rangle$ *and stacks* S_1 *and* S_1'*, let* $Q_1 = [\![\langle\Gamma;\ E;\ e\rangle]\!]^{S_1} = \langle t_1;\ \Delta_1;\ S_1;\ H_1\rangle$ *and* $Q_1' = [\![\langle\Gamma';\ E';\ e'\rangle]\!]^{S_1'} = \langle t_1';\ \Delta_1';\ S_1';\ H_1'\rangle$.

If $\Gamma \vdash e : \tau$, $\Gamma \vdash E$, $S_1 \sqsubseteq S_1'$, $H_1 \vdash S_1\ \mathsf{WF}$, *and* $H_1' \vdash S_1'\ \mathsf{WF}$, *there exists a* Q_2 *and a* Q_2' *such that* $Q_1 \xrightarrow{*} Q_2$, $Q_1' \xrightarrow{*} Q_2'$, $Q_2 \sqsubseteq Q_2'$, $Q_2\ \mathsf{WF}$, *and* $Q_2'\ \mathsf{WF}$.

We dissect the definition along the graphical representation in fig. 7.3.

$$\begin{tikzcd}
\langle \Gamma;\ \emptyset;\ e\rangle \arrow[rr, "c"] \arrow[d, "1"] & & Q_1 \arrow[drr, bend left, "*"] & & \\
\langle \Gamma';\ E;\ e'\rangle \arrow[rr, "c"'] & & Q_1' \arrow[r, "*"] & Q_2' \arrow[r, dash, "\sqsupseteq"] & Q_2
\end{tikzcd}$$

Figure 7.3: Simulation

Our simulation theorem takes a derivation for $\langle \Gamma;\ E;\ e \rangle \longrightarrow \langle \Gamma';\ E';\ e' \rangle$ along with two stacks S_1 and S'_1. We let Q_1 be the compilation of the unstepped state with stack S_1, and Q'_1 the compilation of the stepped state with stack S'_1.

Our precondition combines our well-formed $\mathcal{L}$ state condition (as we saw in the type safety proof, section 4.4) with the requirement that S_1 is extended by S'_1, and that both stacks are well-formed w.r.t. their corresponding heaps.

Given these conditions, we claim that there exist two states Q_2 and Q'_2, such that they are both well-formed, $Q_1 \xrightarrow{*} Q_2$, and $Q'_1 \xrightarrow{*} Q'_2$.

As the full proof can be found in appendix B.4, we do not discuss it here. In the rest of this section we describe the approach taken to arrive at our definition, by transforming the correctness theorem of fig. 7.1 into our definition in fig. 7.3.

Lockstep simulation

The first transformation reverses the bottom arrow, which yields us the situation as displayed in 7.4. Instead of going from the target to the source language trough the decompilation operation c^{-1}, we move from the source language to the target language by means of the same compilation operation c.

$$\begin{tikzcd}
\langle \Gamma;\ \emptyset;\ e \rangle \arrow[r, "c"] \arrow[d, "1"] & Q_1 \arrow[d, "1"] \\
\langle \Gamma';\ E;\ e' \rangle \arrow[r, "c"'] & Q_2
\end{tikzcd}$$

Figure 7.4: Lockstep simulation

Note that this transformation introduces the possibility for the target language (here $\mathcal{M}$) to be trivial, which is true for the next step (converging evaluation) and our final simulation as well. Because all of these have become "one-sided" (omitting a decode step), one can imagine a target language with just `unit` and a transition rule `unit` → `unit` that satisfies these simulation theorems. However, as simulation is used to proof eventual correctness, which reintroduces the decode step, this is not a problem.

Converging evaluation

Our second step is adjusting the constraint on the evaluation paths of Q_1 and Q_2. Instead of requiring Q_1 to directly step to Q_2, we instead require that their evaluation paths eventually converge. We do so by defining some third state Q_3 that both Q_1 and Q_2 step to in zero or more steps. This yields us the situation as displayed in fig. 7.5.

The benefit of this is that Q_2 is still allowed to step. We utilize this in the proof for variable lookup. As $\mathcal{M}$ stores closures, a variable resolves to a closure. In $\mathcal{L}$, variables resolve to terms, as $\mathcal{L}$ stores terms over closures. Compiling this $\mathcal{L}$ term yields a $\mathcal{M}$ term as work item of Q_2. Because now Q_2 is allowed to step, we can promote it to a closure trough rule LIFT to match the closure representation that Q_1 evaluates to.

Note that lockstep simulation is an instance of converging simulation, where $Q_1 \xrightarrow{1} Q_3$, and $Q_2 = Q_3$.

$$\begin{tikzcd}
\langle \Gamma;\ \emptyset;\ e \rangle \arrow[r, "c"] \arrow[d, "1"] & Q_1 \arrow[dr, bend left, "*"] & \\
\langle \Gamma';\ E;\ e' \rangle \arrow[r, "c"'] & Q_2 \arrow[r, "*"] & Q_3
\end{tikzcd}$$

Figure 7.5: Converging simulation

State extension

Our final adjustment involves the state which Q_1 and Q_2 converge, labeled Q_3 in fig. 7.5. In our previous version, we required this to be the same state. However, in our final simulation, we have relaxed this constraint such that the state Q_1 steps to does not have to be equal to the state Q_2 steps to, but only has to be extended by it. This yields us our final simulation diagram, as displayed in fig. 7.3.

This change is necessitated by the discrepancy in how $\mathcal{L}$ and $\mathcal{M}$ handle their contexts. For $\mathcal{L}$, its environment E never shrinks. While this is true for $\mathcal{M}$'s heap H as well, it is not for its binding environment Δ.

Consider the case where an application is being processed. Rule POP-A switches to the closure's stored environment, which may contain less bindings than the current environment. This leaves us with a problem when compiling the processed application. As $\mathcal{L}$ does not contain closures, it does not have a way of retrieving the specific environment in the closure. It only has access to E, which stores everything.

Our solution to this problem is to include all bindings in E during the compilation of Q_2. This may yield additional, unused bindings. As these bindings can leak into closures trough rules LET and $\text{LET}_{\#}$, the closures that are stored during the evaluation of Q_2 may be bigger than those stored during the evaluation of Q_1, which is precisely what our definition of state extension accounts for.

7.3 Proving eventual correctness

The proof for eventual correctness heavily relies on the simulation theorem. The full proof can be found in appendix C. Here we describe the approach of the inductive case on a high level, along the visual representation in fig. 7.6.

The solid lines represent the information we gain by induction. The dotted lines represent information we gain by applying the simulation theorem. Crucial for the proof are the two bold lines.

The first line, $Q_2' \xrightarrow{*} Q_i'$ follows from the observation that $Q_1' \longrightarrow Q_2'$ and $Q_1' \longrightarrow Q_i'$. As Q_i' is a state that does not step and $\xrightarrow{*}$ for $\mathcal{M}$ is deterministic, it follows that $Q_2' \xrightarrow{*} Q_i'$.

The second line, $Q_2 \xrightarrow{*} Q_i''$, is where our well-formedness comes in. A consequence of relaxing our simulation theorem to converge on extending states instead of equal states is that we lose

$$\begin{tikzcd}
\langle \Gamma;\ \emptyset;\ e\rangle \arrow[r, "c"] \arrow[d, "1"] & Q_1 \arrow[d, "1"] \arrow[rrd, dashed, bend left, "*"] & & \\
\langle \Gamma';\ E;\ e'\rangle \arrow[r, "c"'] \arrow[d, "*"] & Q'_1 \arrow[r, dashed, "*"] \arrow[rd, bend right, "*"] & Q'_2 \arrow[r, dash, "\sqsupseteq"] \arrow[d, thick, "*"] & Q_2 \arrow[d, thick, "*"] \\
\langle \Gamma'';\ E';\ i_{\mathcal{L}}\rangle \arrow[r, shift left, leftarrow, "c^{-1}"] \arrow[r, shift right, "c"'] & Q_i \arrow[r, dash, "\sqsupseteq"] & Q'_i \arrow[r, dash, "\sqsupseteq"] & Q''_i
\end{tikzcd}$$

Figure 7.6: Proving eventual correctness

transitivity. That is, the fact that $Q'_2 \xrightarrow{*} Q'_i$ does not imply that $Q_2 \xrightarrow{*} Q''_i$. We can get back such implication trough the lemma below, which completes our proof.

Lemma 7.4 (Equivalent states step to equivalent states)**.** *Let* $Q_1 = \langle t_1;\ \Delta_1;\ S_1;\ H_1\rangle$, $Q'_1 = \langle t'_1;\ \Delta'_1;\ S'_1;\ H'_1\rangle$, $Q_2 = \langle t_2;\ \Delta_2;\ S_2;\ H_2\rangle$, *and* $Q'_2 = \langle t'_2;\ \Delta'_2;\ S'_2;\ H'_2\rangle$.

If $Q_1 \sqsubseteq Q'_1$, Q_1 WF, Q'_1 WF, *and* $Q'_1 \xrightarrow{*} Q'_2$, *then there exists some* Q_2 *such that* $Q_1 \xrightarrow{*} Q_2$ *and* $Q_2 \sqsubseteq Q'_2$.

Chapter 8

Unboxed closures & Memory

Now that we have presented our solution, we can elaborate on its design, and justify why we have made certain choices. Specifically, in this chapter we will motivate the following two aspects:

- The need for the possibility of annotations to be forgotten
- Why we have opted to let unboxed closures be unlifted

Both these discussions require us to discuss the low-level interaction with memory, which is why this discussion has been postponed until now.

8.1 Generalizing the unboxed function closure type

As described in section 4.1.2, a logical consequence of differentiating unboxed closures by their representation is that functions accepting closures are now less general. While there are "only" 18 constructors for `RuntimeRep`[1] [28], in theory there can be an infinite number of closed over variables. Therefore, in theory, an infinite number of alternatives is needed, each set up to handle a closure with a specific set of closed over variables.

As at most one alternative per call site is needed, the number of alternatives needed in practice will be far less than infinite. Nevertheless, if we can reduce the number of alternatives needed, we can avoid unnecessary code duplication.

One solution would be to *wrap* unboxed closures as boxed closures [11], but that would defeat the entire purpose of having closures begin unboxed. Instead, we can generalize the unboxed function closure type in two ways:

1. By classifying types by their *concrete* representation instead of their *abstract* representation.
2. By opting out of passing the set of closed over variables via registers, thus only passing them over the stack.

All generalizations discussed in this chapter have been displayed in fig. 8.1. The layers 'Types' and 'Kinds' have been discussed in section 4.2. Layer 'Registers' is the subject of optimization 1. Layers 'Stack known' and 'Stack unknown' are the subject of optimization 2.

---

[1] For an introduction to `RuntimeRep`, see section 2.3.

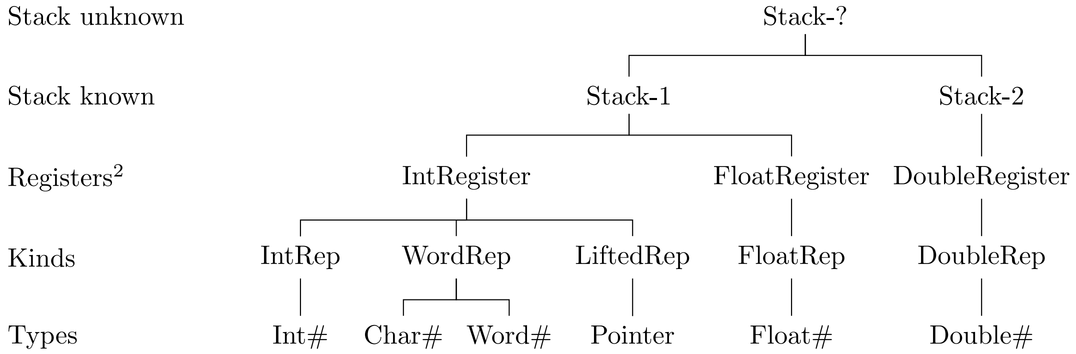


Figure 8.1: Generalizations of closed over variables classification

8.1.1 Classification by concrete representation

So far, we have been using the *RuntimeRep* data type as classifier. This classification makes no assumptions about the underlying architecture. This means that if one type *potentially* could be represented differently than another, then they must have a varying *RuntimeRep* constructor. For example, a distinction between `IntRep` and `WordRep` exists because architectures *may* not represent them equally.

However, in practice, not all architectures do this for every constructor of *RuntimeRep*. This allows for platform-specific optimizations. For example, an architecture may not distinguish between `IntRep`, `WordRep`, and `LiftedRep`, and instead represent them all as non-floating-point words. In such case, code set up to handle a closure with a closed over variable description of *TYPE U* `IntRep` can also accept closures that close over a single variable in `WordRep` or `LiftedRep`, and vice versa. Therefore, those specifications could be merged.

8.1.2 Opting out of registers

The second generalization we can apply is opting out of registers. We can do so in two ways: on a variable-by-variable basis, and by representing all closed over variables as a single block.

Variable-by-variable

The reason that the code set up to accept a closure with a closed over variable in `IntRep` cannot accept a closure with its closed over variable in `FloatRep` is that they may live in a different kind of register. However, if both are passed via the stack, both are represented equally, namely as a single word on the stack.[3] Figure 8.1 captures this idea by the 'Stack-1' construct on the 'Stack known' layer. Similarly, variables of length 2 can be represented by 'Stack-2', and variables of length n by 'Stack-n'.

Single stack-allocated block

For the last possible optimization, the 'Stack unknown' layer of fig. 8.1, we need to take a step back and review our previous solutions. Here, each classification still ranges over a single variable: classifications describe where a single variable is located at runtime.

---

[2] Archtecture specific.

[3] Assuming the architecture in question represents both as a single word.

None of the above solutions apply for cases where the number of variables differs, or where the variables are of unequal length. Examples of these situations are $\tau_1 \overset{\texttt{[Int\#]}}{\rightsquigarrow} \tau_2$ vs. $\tau_1 \overset{\texttt{[Int\#,Int\#]}}{\rightsquigarrow} \tau_2$ and $\tau_1 \overset{\texttt{[Int\#]}}{\rightsquigarrow} \tau_2$ vs. $\tau_1 \overset{\texttt{[Double\#]}}{\rightsquigarrow} \tau_2$, respectively.

For a solution to these situations, we observe that code *handling* an unboxed closure never has to individually address the variables. Instead, it merely has to copy all the variables into a new stack frame, such that the next function can access them. If the variables are scattered across (different kinds) of registers and the stack, the closure needs to be pieced together variable by variable. However, if all closed over variables are stored as a single block on the stack, only the beginning and end indices of this block are needed, as the block can be copied whole.

Essentially, we are proposing a solution similar to ad hoc polymorphism [25]. However, instead of outputting multiple functions and deciding what alternative to pick, we can output code that examines the passed closure, as displayed in the following snippet of pseudocode:

```
$app^5$ :: (a $\overset{?}{\rightsquigarrow}$ b) $\rightarrow$ a $\rightarrow$ b
$app^5$ f x =
  let start = startOf f
      end   = widthOf f + start
  in -- copy words between start and end to new frame
     -- call f with x
```

This is what our 'Stack unknown' layer indicates with the 'Stack-?' construct. This construct is not meant to be used as the indicator of a single variable, but rather of the entire set of closed over variables, yielding '*TYPE U* ?'.

While such runtime casing might seem infeasible at first, these solutions are not uncommon, and are actually used in realistic compilers. For example, GHC uses pointer tagging, such that type information can be encoded into pointers, and cased upon during runtime [12].

Furthermore, such runtime switched can be optimized away in cases where the information is known statically[4]. This applies to our situation as well: if the length of the set of closed over variables of all considered closures is equal, the inspections `startOf` and `widthOf` can be optimized away, and substituted for the statically known locations.

8.1.3 Implementation in $\mathcal{L}$

To simplify $\mathcal{L}$, most of these optimizations have been omitted. Instead, $\mathcal{L}$ contains the two extremes.

As discussed in section 4.2, by default the annotation A on function arrows consists of a typing environment Γ, which per rule C_Var (fig. 6.1) is converted to a list of kinds κ. This implements the 'Kinds' layer of fig. 8.1.

Furthermore, annotations can be 'forgotten' to '?' via rules E_Forget and E_Forget$_\#$ (fig. 4.2), which needs the runtime metadata as described in section 8.1.2. Therefore, this implements the 'Stack unknown' layer of fig. 8.1.

---

[4] Tarditi et al. [27] apply this technique to a similar construct they call intensional polymorphism. Vytiniotis, Peyton Jones, and Magalhães [29] apply a similar approach in in their approach to support *deferred type errors* (runtime type errors).

Effects on compilation

One might wonder if ‘forgetting’ the type annotation does not interfere with the compilation as presented in fig. 6.1 (page 27). In rules C_LAM and C_LAM$_\#$ we require the annotation Γ_2 to be present, as we compile e under Γ_2. Would annotating ‘?’ not introduce problems?

No, it does not. Per rules E_LAM and E_LAM$_\#$ of fig. 4.2 (page 15), the annotation of a *singular* lambda expression is always known. Rules E_FORGET and E_FORGET$_\#$ allow for the forgetting of this annotation, but cannot introduce annotated types. That is, only rules E_LAM and E_LAM$_\#$ can introduce function types, which means the actual annotation is always available further down the typing derivation.

It is only when functions that *handle* closures that the exact derivation might not be known, as they can be passed closures from multiple locations. However, when wen compiling lambdas, we always know what exact lambda is being compiled, and therefore have access to its annotation Γ.

8.2 Unboxed closures must be unlifted

In section 2.4 we observed that all current unboxed types are unlifted. One of the reasons for this is that currently it is not possible for unboxed types to be lifted, because that would require the ability to store closures on the stack, which before was not possible. As one of the major contributions of this thesis is the presentation of this exact functionality, we potentially could let unboxed closures be lifted, given that it makes sense to do so and no other technical limitations apply. However, as we will describe in this section, certain limitations *do* apply, which makes lifting unboxed closures unfeasible. Specifically, the rest of this section motivates the following to observations:

Observation 8.1. In order to efficiently implement lifted closures, we need to be able to update thunks with their values.

Observation 8.2. Because of limitations of the stack, we cannot update unboxed closures.

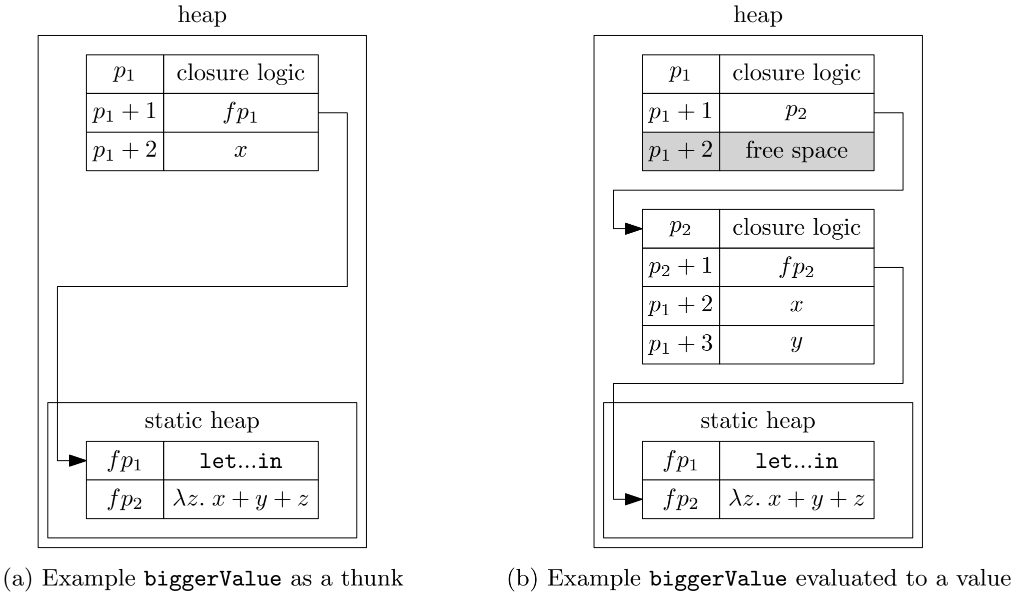
8.2.1 Updating boxed closures

Non-strict semantics can sometimes cause a significant performance penalty, as the term bound to some variable is re-evaluated upon every usage of said variable. For Haskell, the majority of this performance penalty is avoided by implementing sharing, such that subsequent usages do not require re-evaluation (as discussed in section 2.1). This necessitates the ability to update a closure.

Updating boxed closures is possible because the heap allows for the updating of thunks with values bigger than the thunk through *indirections*. To further understand this, consider the following example.

```
biggerValue = let y = 0
              in λz → x + y + z
```

`biggerValue` is a thunk that upon evaluation yields a value bigger than its thunk. We observe that the thunk of `biggerValue` closes over one variable, namely `x`. Therefore, along with the closure logic and a pointer to the static function logic, the thunk needs to store a (pointer to) `x`, as shown in fig. 8.2a.



(a) Example `biggerValue` as a thunk (b) Example `biggerValue` evaluated to a value

Figure 8.2: Example `biggerValue` in thunk and value representation

During evaluation, a binding for `y` is created, which means that the closure containing the value must store (pointers to) `x` as well as `y`. This does not fit inside the original space. While in some situations the space right after the end of the thunk ($p_1 + 3$ in this case) might be free, this is not true in the general case. Therefore, a new closure is created at location p_2, as shown in fig. 8.2b. The closure at p_1 is updated with an *indirection.* That is, upon forcing the updated closure starting at p_1, the evaluation process is redirected to the closure starting at p_2.

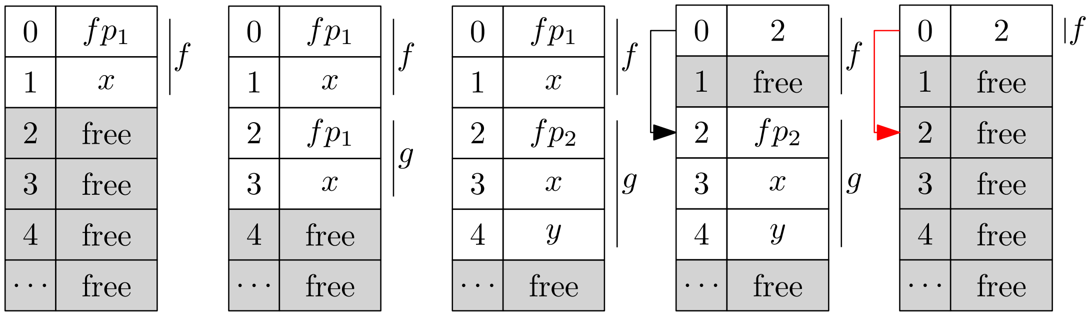
8.2.2 Updating unboxed closures

Updating unboxed closures is a challenge, because the stack does not support indirections. To illustrate this, we consider the `manyBiggerValue` example below.

```
manyBiggerValue = let c = biggerValue
                  in map (g c) [1..10]
```

Here, `g` is some function that takes some closure (`biggerValue` in this case) and an integer, evaluates the closure, and returns some result. Figure 8.3 shows the stack layouts that occur during the evaluation of `manyBiggerValue`. This overview has been simplified to only show the elements of `biggerValue`. Furthermore, fp_1 and fp_2 refer to the static heap as defined in figs. 8.2a and 8.2b.

In fig. 8.3a we can see the stack frame for f, with the thunk of `biggerValue` of fig. 8.2a, but in unboxed representation. The call to g proceeds by creating a new stack frame and copying the closure into it, as shown in fig. 8.3b. Now, when g evaluates `biggerValue`, we get the unboxed version of the closure as shown in fig. 8.2b, which needs one more word than the thunk to store the binding for y. In this case, as index 4 is free, we can update the closure in g's frame, as shown in fig. 8.3c. Note that this is not possible in the general case, as index 4 will not always be free.



(a) Before call to g

(b) After call to g

(c) Updating only in bottom frame

(d) Trying to create stack indirection

(e) Stack indirection does not work

Figure 8.3: Stack layouts unboxed `biggerValue`

A bigger problem is formed by all copies in frames above g, as here we never have this free word to expand in, as that memory is always occupied by further frames. In this case, g's frame blocks us from storing the expanded value in f's frame. Therefore, when g returns, the situation will once again be that of fig. 8.3a, which means `biggerValue` needs to be re-evaluated.

One might try to implement some stack-based indirection, as shown in fig. 8.3d. Here, we update g as before, and let the closure in f point to the one in g. Ignoring the fact that such an approach requires significant bookkeeping, it would not be a valid approach. As soon as g returns, its frame is popped. This means that f now contains a (stack)pointer to a location that is considered to be free, as shown by fig. 8.3e.

8.2.3 Conclusion

As demonstrated, it is not possible to implement sharing. Therefore, we must choose between 'pure' call-by-name or strict evaluation semantics. As the potential performance benefit of unboxing closures is the main motivational factor behind exploring them, we have opted to strictly evaluate unboxed closures. Therefore, they fall into the same unboxed, unlifted category as existing unboxed types such as `Int#` and `Char#`.

Chapter 9

Conclusion and future work

9.1 Conclusion

In this thesis, we have explored the possibility of adding unboxed function closures to Haskell. We first motivated the usefulness of this extension, describing how they are a natural fit to a language where functions are first-class citizens. Furthermore, we described how unboxed function closures can be more efficient, by reducing the (expensive) interaction with the heap.

We then established how, in conventional type systems, closures of equal type can have a varying runtime representation. We then presented our solution, which involves annotating the function type with the list of the types of the closed over variables. We presented this type system in $\mathcal{L}$, which is our high-level language.

We then presented $\mathcal{M}$ and a compilation function $\mathcal{L} \to \mathcal{M}$. During compilation we maintain a typing environment Γ such that it is representative of the runtime environment. Therefore, the abstract functions *lookup* and $fresh$ can output static identifiers such as stack offsets, based on this Γ. The annotation on the function arrows are critical for this process.

By making $\mathcal{M}$ sufficiently close to a real machine, we have achieved our main goal, which was to demonstrate the possibility of adding unboxed closures to Haskell. However, more work is needed before adding unboxed closures to Haskell can be seriously considered.

Specifically, we recognize two categories of future work: improvements to our presentation in the form of a formalism for the properties of *lookup*, and work based on this presentation in the form of a proof of concept.

9.2 Properties of *lookup*

As described in section 6.1, we came up short when trying to formally define the properties of the *lookup* function. Specifically, we had problems defining properties without assuming a specific binding resolution strategy. The proposition we would have liked to proof can, on a high level, be described as follows.

Proposition 9.1. The list of kinds passed to *lookup* during the compilation of $\mathcal{L}$ is representative of the runtime environment of $\mathcal{M}$.

The problem with such a proposition is the notion of 'representative of'. We want to formulate that, for any variable resolution strategy, its runtime behaviour can be emulated at compile time by examining the list of kinds. We feel this notion should be expressible whilst abstracting over the concrete strategy, but have not found a way.

Luckily, when a specific variable resolution strategy is used, this notion *can* be formulated and proven. Therefore, work towards this area will mostly benefit "pure" works such as this one, as more concrete proposals (such as GHC proposals [8]), can or even have to consider a concrete strategy.

9.3 Proof of concept

While we have taken care to make $\mathcal{M}$ sufficiently close to a real machine, this thesis is not a proof of concept. Therefore, the next step for assessing the potential of unboxed closures is to create an implementation of the system proposed.

The main purpose of this proof of concept is not to show that the system as presented is implementable, but to determine in what situations it is worthwhile to use unboxed function closures over their boxed counterparts. As show, situations exist where unboxed closures are strictly better than boxed closures, as they allow for a reduction in interaction with the heap while consuming the same amount of memory and requiring no runtime metadata.

However, not all situations are this ideal, as depending on the amount of copies, unboxed closures require more memory. Furthermore, depending on what generalization of the closure shape has been applied (section 8.1), registers cannot be used, or even some runtime metadata is required.

As speed is concerned, a realistic implementation must be made.

The system as proposed in this thesis optimizes for unboxed closures, but makes no attempt at efficiently evaluating boxed closures. Therefore, the proof of concept needs to go beyond what we did here, and apply optimizations for both the boxed and unboxed case.

Once the proof of concept has been made, a benchmark suite like `nofib` [16] can be translated into the (equivalent of) $\mathcal{L}$, to get a good idea of the situations in which unboxing closures makes sense, and in which situations they do not.

Bibliography

[1] M. C. Bolingbroke and S. L. Peyton Jones. “Types Are Calling Conventions”. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. 2009, pp. 1–12.

[2] P. Crégut. “An Abstract Machine for the Normalization of Lambda-Calculus”. In: Proc. Conf. on Lisp and Functional Programming, pp. 333–340.

[3] N. G. De Bruijn. “Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem”. In: 75.5 (1972), pp. 381–392. ISSN: 1385-7258.

[4] A. de la Encina and R. Pena. “Formally Deriving an STG Machine”. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. 2003, pp. 102–112.

[5] U. Drepper. “What Every Programmer Should Know about Memory”. In: *Red Hat, Inc* 11 (2007), p. 2007.

[6] R. A. Eisenberg and S. Peyton Jones. “Levity Polymorphism”. In: *ACM SIGPLAN Notices*. Vol. 52. ACM, 2017, pp. 525–539.

[7] C. Flanagan et al. “The Essence of Compiling with Continuations”. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. 1993, pp. 237–247.

[8] *GHC Proposals - Github*. URL: `https://github.com/ghc-proposals/ghc-proposals` (visited on 08/10/2020).

[9] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016. ISBN: 1-107-15030-2.

[10] S. P. Jones et al. “The Glasgow Haskell Compiler: A Technical Overview”. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference. Vol. 93. 1993.

[11] X. Leroy. “Unboxed Objects and Polymorphic Typing”. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992, pp. 177–188.

[12] S. Marlow and S. Peyton Jones. “Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-Order Languages”. In: ACM SIGPLAN Notices. Vol. 39. ACM, 2004, pp. 4–15. ISBN: 1-58113-905-5.

[13] Y. Minamide, G. Morrisett, and R. Harper. “Typed Closure Conversion”. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1996, pp. 271–283.

[14] F. L. Morris. “Advice on Structuring Compilers and Proving Them Correct”. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1973, pp. 144–152.

[15] J. Mountjoy. “The Spineless Tagless G-Machine, Naturally”. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. 1998, pp. 163–173.

[16] W. Partain. “The Nofib Benchmark Suite of Haskell Programs”. In: *Functional Programming, Glasgow 1992.* Springer, 1993, pp. 195–202.

[17] D. Patterson and A. Ahmed. “The next 700 Compiler Correctness Theorems (Functional Pearl)”. In: *Proceedings of the ACM on Programming Languages* 3 (ICFP 2019), pp. 1–29. ISSN: 2475-1421.

[18] S. Peyton Jones. “Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine”. In: *Journal of functional programming* 2.2 (1992), pp. 127–202.

[19] S. Peyton Jones and J. Launchbury. “Unboxed Values as First Class Citizens in a Non-Strict Functional Language”. In: Conference on Functional Programming Languages and Computer Architecture. Springer, 1991, pp. 636–666.

[20] S. Peyton Jones, N. Ramsey, and F. Reig. “C-—: A Portable Assembly Language That Supports Garbage Collection”. In: *International Conference on Principles and Practice of Declarative Programming.* Springer, 1999, pp. 1–28.

[21] B. C. Pierce and C. Benjamin. *Types and Programming Languages.* 2002. ISBN: 0-262-16209-1.

[22] J. C. Reynolds. “Towards a Theory of Type Structure”. In: Programming Symposium. Springer, 1974, pp. 408–425.

[23] P. Sestoft. “Deriving a Lazy Abstract Machine”. In: *Journal of Functional Programming* 7.3 (1997), pp. 231–264. ISSN: 0956-7968.

[24] Sri, Baskaran. *GHC Commentary: What the Hell Is a .Cmm File?* Apr. 26, 2020. URL: `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/cmm` (visited on 07/22/2020).

[25] C. Strachey. “Fundamental Concepts in Programming Languages”. In: *Higher-order and symbolic computation* 13.1-2 (2000), pp. 11–49. ISSN: 1388-3690.

[26] M. Sulzmann et al. “System F with Type Equality Coercions”. In: *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation.* ACM, 2007, pp. 53–66.

[27] D. Tarditi et al. “TIL: A Type-Directed Optimizing Compiler for ML”. In: *ACM Sigplan Notices* 31.5 (1996), pp. 181–192. ISSN: 0362-1340.

[28] The University of Glasgow. *GHC.Exts - Hackage.* URL: `https://hackage.haskell.org/package/base-4.14.0.0/docs/GHC-Exts.html#t:RuntimeRep` (visited on 07/23/2020).

[29] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. “Equality Proofs and Deferred Type Errors: A Compiler Pearl”. In: *ACM SIGPLAN Notices* 47.9 (2012), pp. 341–352.

[30] S. Weirich, J. Hsu, and R. A. Eisenberg. “System FC with Explicit Kind Equality”. In: *ACM SIGPLAN Notices.* Vol. 48. ACM, 2013, pp. 275–286.

[31] B. A. Yorgey et al. “Giving Haskell a Promotion”. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation.* ACM, 2012, pp. 53–66.

Appendix A

L type safety

A.1 Lemmas

Lemma A.1 (Term substitution). *If* $\Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash \tau' : \kappa$, *then* $\Gamma \bullet \Gamma'[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau$.

Proof. Straightforward induction on the typing derivation. ☐

Lemma A.2 (Environment substitution). *If* $\Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \vdash e : \tau$, $\Gamma \vdash \tau' : \kappa$, *and* $\Gamma \vdash E$, *then* $\Gamma \bullet \Gamma'[\tau'/\alpha] \vdash E[\tau'/\alpha]$.

Proof. Straightforward induction on the well-formedness derivation for environments. ☐

Lemma A.3 (Stepping does not shrink environments). *If* $\langle \Gamma;\ E;\ e\rangle \longrightarrow \langle \Gamma';\ E';\ e'\rangle$, *then* $\Gamma \subseteq \Gamma'$ *and* $E \subseteq E'$.

Proof. Straightforward induction on the derivation of $\langle \Gamma;\ E;\ e\rangle \longrightarrow \langle \Gamma';\ E';\ e'\rangle$. ☐

A.2 Progress

Theorem 4.1 (Progress). *For any* $\langle \Gamma;\ E;\ e\rangle$, *if* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash E$, *then either* e *is a value, or there exists an* $\langle \Gamma';\ E';\ e'\rangle$ *such that* $\langle \Gamma;\ E;\ e\rangle \longrightarrow \langle \Gamma;\ E';\ e'\rangle$.

Proof. By induction over the typing derivation of e.

E_Var

$$\textsc{E\_Var}\ \frac{\gamma{:}\tau \in \Gamma}{\Gamma \vdash \gamma : \tau} \qquad \textsc{S\_Var}\ \frac{\gamma \mapsto e \in E}{\langle \Gamma, E, \gamma\rangle \longrightarrow \langle \Gamma, E, e\rangle}$$

As $\gamma : \tau \in \Gamma$ and $\Gamma \vdash E$, by rule EV_Term there exists an e_1 such that $\Gamma \vdash e_1 : \tau$ and $\gamma \mapsto e_1 \in E$. Therefore we can step e by rule S_Var.

E_App

$$\textsc{E\_App}\ \frac{\Gamma \vdash e : \tau_1 \overset{A}{\rightarrow} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2} \qquad \textsc{S\_App}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\langle \Gamma, E, e_1\ \gamma \rangle \longrightarrow \langle \Gamma', E', e_1'\ \gamma \rangle}$$

$$\textsc{S\_Lam}\ \frac{\begin{array}{c} \gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2 \end{array}}{\langle \Gamma, E, (\lambda \gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle}$$

By induction we know that e_1 is either a value, or it can take a step.

By rule E_App $\Gamma \vdash e_1 : \tau_1 \overset{A}{\rightarrow} \tau_2$. Therefore, in the case where e_1 is a value, e_1 must be of form $\lambda\gamma_1{:}\tau.e_2$, as that is the only value of type $\tau_1 \overset{A}{\rightarrow} \tau_2$. We observe that $\Gamma \vdash E$, and by rule E_App $\Gamma \vdash \gamma_2 : \tau_1$. Therefore, by rules E_Var and EV_Term it follows that there exists an e_3 such that $\Gamma \vdash e_3 : \tau_1$, and $\gamma_2 \mapsto e_3 \in E$. Therefore, we can step e by S_Lam.

If e_1 is not a value, then we know it can step. Therefore, we can step e by S_App.

E_App$_{\#}$

$$\textsc{E\_App}_{\#}\ \frac{\Gamma \vdash e : \tau_1 \overset{A}{\rightsquigarrow} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2} \qquad \textsc{S\_App}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\langle \Gamma, E, e_1\ \gamma \rangle \longrightarrow \langle \Gamma', E', e_1'\ \gamma \rangle}$$

$$\textsc{S\_Lam}_{\#}\ \frac{\begin{array}{c} \gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2 \end{array}}{\langle \Gamma, E, (\lambda_{\#} \gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle}$$

Identical to the case for E_App, but with the unboxed version of the lambda rule S_Lam$_{\#}$.

E_TApp

$$\textsc{E\_TApp}\ \frac{\Gamma \vdash e : \forall\, \alpha{:}\kappa.\ \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\textsc{S\_Tapp}\ \frac{\langle \Gamma, E, e \rangle \longrightarrow \langle \Gamma', E', e' \rangle}{\langle \Gamma, E, e\ \tau \rangle \longrightarrow \langle \Gamma', E', e'\ \tau \rangle} \qquad \textsc{S\_TBeta}\ \frac{\begin{array}{c} \Gamma_1 = \Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \\ \Gamma_2 = \Gamma \bullet \Gamma' \end{array}}{\begin{array}{c} \langle \Gamma_1, E, (\Lambda\alpha{:}\kappa.v)\ \tau \rangle \\ \longrightarrow \langle \Gamma_2[\tau/\alpha], E[\tau/\alpha], v[\tau/\alpha] \rangle \end{array}}$$

By induction, we know that e_1 either is a value, or can take a step. If it is a value, we can step e with S_TBeta. If e_1 can step, we can step e with S_TApp.

E_Let

$$\textsc{E\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ P\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{let}\ \gamma = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \textsc{S\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto e_1\end{array}}{\langle \Gamma, E, \textbf{let}\ \gamma = e_1\ \textbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle}$$

By rule E_Let we know $\Gamma \vdash e_1 : \tau_1$, which means we can step e with rule S_Let.

E_Let$_\#$

$$\textsc{E\_Let}_\#\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ U\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{let}_\#\ \gamma = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\textsc{S\_Let}_{\#a}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\begin{array}{c}\langle \Gamma, E, \textbf{let}_\#\ \gamma = e_1\ \textbf{in}\ e_2 \rangle \\ \longrightarrow \langle \Gamma', E', \textbf{let}_\#\ \gamma = e_1'\ \textbf{in}\ e_2 \rangle\end{array}} \qquad \textsc{S\_Let}_{\#b}\ \frac{\begin{array}{c}\Gamma \vdash v : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto v\end{array}}{\langle \Gamma, E, \textbf{let}_\#\ \gamma = v\ \textbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle}$$

By induction we know that e_1 is either a value, or it can take a step. If e_1 can take a step, we can step e with rule S_Let$_{\#a}$. If it is a value, we can step e with rule S_Let$_{\#b}$.

E_Lam, E_Lam$_\#$, E_Forget, E_Forget$_\#$, E_TLam, E_IntLit

In all these cases, e is a value. ☐

A.3 Preservation

Theorem 4.2 (Preservation)**.** *If* $\langle \Gamma;\ E;\ e \rangle \longrightarrow \langle \Gamma';\ E';\ e' \rangle$*,* $\Gamma \vdash e : \tau$*, and* $\Gamma \vdash E$*, then* $\Gamma' \vdash e' : \tau$*, and* $\Gamma' \vdash E'$*.*

Proof. by induction on the typing derivation of e.

E_Var

$$\textsc{E\_Var}\ \frac{\gamma{:}\tau \in \Gamma}{\Gamma \vdash \gamma : \tau} \qquad \textsc{S\_Var}\ \frac{\gamma \mapsto e \in E}{\langle \Gamma, E, \gamma \rangle \longrightarrow \langle \Gamma, E, e \rangle}$$

As by rule E_Var $\gamma : \tau \in \Gamma$ and $\Gamma \vdash E$, by rule EV_Term and the fact that all γ are fresh, we get that $\Gamma \vdash e_1 : \tau$. Furthermore, as E is unchanged, it is trivially well-formed.

E_App

$$\textsc{E\_App}\ \frac{\Gamma \vdash e : \tau_1 \overset{A}{\to} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2} \qquad\qquad \textsc{E\_Lam}\ \frac{\Gamma \bullet \gamma{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\gamma{:}\tau_1.e : \tau_1 \overset{\Gamma}{\to} \tau_2}$$

$$\textsc{S\_Lam}\ \frac{\begin{array}{c}\gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2\end{array}}{\langle \Gamma, E, (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle} \quad \textsc{S\_App}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\langle \Gamma, E, e_1\ \gamma \rangle \longrightarrow \langle \Gamma', E', e_1'\ \gamma \rangle}$$

S_Lam By rule E_App we know that $\Gamma \vdash \lambda\gamma_1{:}\tau.e_1 : \tau_1 \overset{A}{\to} \tau_2$, which by rule E_Lam means that $\Gamma, \gamma_1{:}\tau_1 \vdash e_1 : \tau_2$.

As $\Gamma \vdash E$, $\Gamma \vdash \gamma_2 : \tau_1$, and $\gamma_2 \mapsto e_2 \in E$, by rule EV_Term $\Gamma \vdash e_2 : \tau_1$. As every γ is fresh, we know that the binding of γ_1 in $E, \gamma_1 \mapsto e_2$ is unique, so by rule EV_Term $\Gamma, \gamma_1{:}\tau_1 \vdash E, \gamma_1 \mapsto e_2$.

S_App By rule E_App, $\Gamma \vdash e_1 : \tau_1 \overset{A}{\to} \tau_2$ and $\Gamma \vdash \gamma : \tau_1$. From lemma A.3 and the fact that every γ is fresh we gather that $\Gamma' \vdash \gamma : \tau_1$. By induction, $\Gamma' \vdash e_1' : \tau_1 \overset{A}{\to} \tau_2$, which by E_App means that $\Gamma' \vdash e_1'\ \gamma : \tau_2$. Furthermore, by induction we know that $\Gamma' \vdash E'$, so we have proven this case.

E_App$_\#$

$$\textsc{E\_App}_\#\ \frac{\Gamma \vdash e : \tau_1 \overset{A}{\rightsquigarrow} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2}$$

$$\textsc{S\_Lam}_\#\ \frac{\begin{array}{c}\gamma_2 \mapsto e_2 \in E \\ \Gamma \vdash e_2 : \tau \\ \Gamma' = \Gamma \bullet \gamma_1{:}\tau \\ E' = E, \gamma_1 \mapsto e_2\end{array}}{\langle \Gamma, E, (\lambda_\#\gamma_1{:}\tau.e_1)\ \gamma_2 \rangle \longrightarrow \langle \Gamma', E', e_1 \rangle} \quad \textsc{S\_App}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\langle \Gamma, E, e_1\ \gamma \rangle \longrightarrow \langle \Gamma', E', e_1'\ \gamma \rangle}$$

Identical to the case for E_App, but with the unboxed version of the lambda rule S_Lam$_\#$.

E_TApp

$$\textsc{E\_TApp}\ \frac{\Gamma \vdash e : \forall\, \alpha{:}\kappa.\ \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\textsc{S\_Tapp}\ \frac{\langle \Gamma, E, e \rangle \longrightarrow \langle \Gamma', E', e' \rangle}{\langle \Gamma, E, e\ \tau \rangle \longrightarrow \langle \Gamma', E', e'\ \tau \rangle} \quad \textsc{S\_TBeta}\ \frac{\begin{array}{c}\Gamma_1 = \Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \\ \Gamma_2 = \Gamma \bullet \Gamma'\end{array}}{\begin{array}{l}\langle \Gamma_1, E, (\Lambda\alpha{:}\kappa.v)\ \tau \rangle \\ \quad \longrightarrow \langle \Gamma_2[\tau/\alpha], E[\tau/\alpha], v[\tau/\alpha] \rangle\end{array}}$$

Case follows from lemmas A.1 and A.2.

E_Let

$$\textsc{E\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ P\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \textsc{S\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto e_1\end{array}}{\langle \Gamma, E, \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle}$$

From E_Let we immediately get that $\Gamma, \gamma{:}\tau_1 \vdash e_2 : \tau_2$. Furthermore, as $\Gamma \vdash e_1 : \tau_1$, by rule EV_Term, we get that $\Gamma, \gamma{:}\tau_1 \vdash E, \gamma \mapsto e_1$, so we have proven this case.

E_Let$_{\#}$

$$\textsc{E\_Let}_{\#}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ U\ A \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\textsc{S\_Let}_{\#a}\ \frac{\langle \Gamma, E, e_1 \rangle \longrightarrow \langle \Gamma', E', e_1' \rangle}{\begin{array}{c}\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 \rangle \\ \longrightarrow \langle \Gamma', E', \mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2 \rangle\end{array}} \qquad \textsc{S\_Let}_{\#b}\ \frac{\begin{array}{c}\Gamma \vdash v : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto v\end{array}}{\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = v\ \mathbf{in}\ e_2 \rangle \longrightarrow \langle \Gamma', E', e_2 \rangle}$$

S_Let$_{\#a}$

By rule E_Let$_{\#}$ we know that $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, \gamma{:}\tau_1 \vdash e_2 : \tau_2$. From lemma A.3 and the fact that every γ is fresh we gather that $\Gamma', \tau_1 \vdash e_2 : \tau_2$. By induction we know that stepping e_1 maintains its type, which means $\Gamma' \vdash e_1 : \tau_1$. From this we gather that $\Gamma' \vdash \mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2 : \tau_2$. Furthermore, by induction we get that $\Gamma' \vdash E'$, so we have proven this case.

S_Let$_{\#b}$

From E_Let$_{\#}$ we immediately get that $\Gamma, \gamma{:}\tau_1 \vdash e_2 : \tau_2$. Furthermore, as $\Gamma \vdash v : \tau_1$, by rule EV_Term, we get that $\Gamma, \gamma{:}\tau_1 \vdash E, \gamma \mapsto v$, so we have proven this case.

E_Lam, E_Lam$_{\#}$, E_Forget, E_Forget$_{\#}$, E_TLam, E_IntLit

In all these cases, e is a value, which do not step. $\square$

Appendix B

Simulation

B.1 Notation

- $[\![e]\!]^{\Gamma} = t$: translation function for terms
- $[\![E]\!]^{\Gamma} = (\Delta, H)$: translation function for envs
- $[\![\langle \Gamma;\ E;\ e \rangle]\!]^{S} = \langle [\![e]\!]^{\Gamma};\ \Delta;\ S;\ H \rangle$ where $[\![E]\!]^{\Gamma} = (\Delta, H)$

B.2 Definitions

Definition B.1. In $\mathcal{L}$, $\xrightarrow{*}$ is the reflexive transitive closure on $\longrightarrow$ as defined in fig. 4.3 (page 19). That is, a $\mathcal{L}$ state steps to another in zero or more steps, written $\langle \Gamma_1;\ E_1;\ e_1 \rangle \xrightarrow{*} \langle \Gamma_2;\ E_2;\ e_2 \rangle$, if $\langle \Gamma_1;\ E_1;\ e_1 \rangle = \langle \Gamma_2;\ E_2;\ e_2 \rangle$, or if there exists some $\langle \Gamma_1';\ E_1';\ e' \rangle$ such that $\langle \Gamma_1;\ E_1;\ e_1 \rangle \longrightarrow \langle \Gamma_1';\ E_1';\ e_1' \rangle$ and $\langle \Gamma_1';\ E_1';\ e_1' \rangle \xrightarrow{*} \langle \Gamma_2;\ E_2;\ e_2 \rangle$. ☐

Definition B.2. In $\mathcal{M}$, $\xrightarrow{*}$ is the reflexive transitive closure on $\longrightarrow$ as defined in fig. 5.2 (page 23). That is, a $\mathcal{M}$ state steps to another in zero or more steps, written $\langle t_1;\ \Delta_1;\ S_1;\ H_1 \rangle \xrightarrow{*} \langle t_2;\ \Delta_2;\ S_2;\ H_2 \rangle$, if $\langle t_1;\ \Delta_1;\ S_1;\ H_1 \rangle = \langle t_2;\ \Delta_2;\ S_2;\ H_2 \rangle$, or if there exists some $\langle t_1';\ \Delta_1';\ S_1';\ H_1' \rangle$ such that $\langle t_1;\ \Delta_1;\ S_1;\ H_1 \rangle \longrightarrow \langle t_1';\ \Delta_1';\ S_1';\ H_1' \rangle$ and $\langle t_1';\ \Delta_1';\ S_1';\ H_1' \rangle \xrightarrow{*} \langle t_2;\ \Delta_2;\ S_2;\ H_2 \rangle$. ☐

Definition B.3 ($\mathcal{M}$ closure well-formedness)**.** A $\mathcal{M}$ closure (t, Δ) is well-formed, written $\Delta \vdash t\ \mathsf{WF}$, if Δ contains a binding for all closed over variables of t. That is, $\forall x.\ x \in fv(t) \implies \exists b.\ x \mapsto b \in \Delta$. ☐

Definition B.4 ($\mathcal{M}$ binding storage well-formedness)**.** A $\mathcal{M}$ environment Δ and heap H are well-formed, written $H \vdash \Delta\ \mathsf{WF}$, if any stored closure (t, Δ') they store is well-formed, and the stored environment with the given heap is well-formed as well.

$$\frac{\begin{array}{c} p \mapsto (t, \Delta') \in H \\ \Delta' \vdash t\ \mathsf{WF} \\ H \vdash \Delta'\ \mathsf{WF} \end{array}}{H \vdash y \mapsto p \bullet \Delta\ \mathsf{WF}} \qquad \frac{\begin{array}{c} \Delta' \vdash t\ \mathsf{WF} \\ H \vdash \Delta'\ \mathsf{WF} \end{array}}{H \vdash z \mapsto (t, \Delta') \bullet \Delta\ \mathsf{WF}}$$

☐

Definition B.5 ($\mathcal{M}$ stack well-formedness)**.** A $\mathcal{M}$ stack S is well-formed w.r.t. a heap H, written $H \vdash S$ WF, in the following cases:

$$\frac{}{H \vdash \emptyset \ \mathsf{WF}} \qquad \frac{H \vdash S \ \mathsf{WF}}{H \vdash \mathrm{App}(b) \bullet S \ \mathsf{WF}} \qquad \frac{\begin{array}{c}\Delta \vdash t \ \mathsf{WF}\\ H \vdash \Delta \ \mathsf{WF}\\ H \vdash S \ \mathsf{WF}\end{array}}{H \vdash \mathrm{Let}(z, t, \Delta) \bullet S \ \mathsf{WF}} \qquad \square$$

Definition B.6 ($\mathcal{M}$ state well-formedness)**.** A $\mathcal{M}$ state $\langle i;\ \Delta;\ S;\ H\rangle$ is well-formed, written $\langle i;\ \Delta;\ S;\ H\rangle$ WF, in the following cases:

$$\frac{\begin{array}{c}p \mapsto (t, \Delta') \in H\\ \Delta' \vdash t \ \mathsf{WF}\\ H \vdash \Delta' \ \mathsf{WF}\\ H \vdash S \ \mathsf{WF}\\ H \vdash \Delta \ \mathsf{WF}\end{array}}{\langle p;\ \Delta;\ S;\ H\rangle \ \mathsf{WF}} \qquad \frac{\begin{array}{c}\Delta' \vdash t \ \mathsf{WF}\\ H \vdash \Delta' \ \mathsf{WF}\\ H \vdash S \ \mathsf{WF}\\ H \vdash \Delta \ \mathsf{WF}\end{array}}{\langle (t, \Delta');\ \Delta;\ S;\ H\rangle \ \mathsf{WF}} \qquad \frac{\begin{array}{c}\Delta \vdash t \ \mathsf{WF}\\ H \vdash S \ \mathsf{WF}\\ H \vdash \Delta \ \mathsf{WF}\end{array}}{\langle t;\ \Delta;\ S;\ H\rangle \ \mathsf{WF}} \qquad \square$$

Definition B.7 (Environment extension)**.** An environment (Δ_1, H_1) is extended by another environment (Δ_2, H_2), written $(\Delta_1, H_1) \sqsubseteq (\Delta_2, H_2)$ or $(\Delta_2, H_2) \sqsupseteq (\Delta_1, H_1)$ in the following cases:

$$\frac{\begin{array}{c}H_1[p] = (t_1, \Delta_3)\\ \Delta_2[y] = p'\\ H_2[p'] = (t_2, \Delta_4)\\ \Delta_3 \subseteq \Delta_4\\ (\Delta_1, H_1) \sqsubseteq (\Delta_2, H_2)\end{array}}{(y \mapsto p \bullet \Delta_1, H_1) \sqsubseteq (\Delta_2, H_2)} \qquad \frac{\begin{array}{c}\Delta_2[z] = (t_2, \Delta_4)\\ \Delta_3 \subseteq \Delta_4\\ (\Delta_1, H_1) \sqsubseteq (\Delta_2, H_2)\end{array}}{(z \mapsto (t_1, \Delta_3) \bullet \Delta_1, H_1) \sqsubseteq (\Delta_2, H_2)} \qquad \square$$

Definition B.8 (Stack extension)**.** A stack S_1 is extended by another stack S_2, written $S_1 \sqsubseteq S_2$ or $S_2 \sqsupseteq S_1$, if they are equal, modulo let continuations. For these continuations, the stored variable and term are required to be equal. For the stored environment, S_2 may store a superset of the environment stored by S_1.

$$\frac{}{\emptyset \sqsubseteq \emptyset} \qquad \frac{S_1 \sqsubseteq S_2}{\mathrm{App}(b) \bullet S_1 \sqsubseteq \mathrm{App}(b) \bullet S_2} \qquad \frac{\begin{array}{c}\Delta_1 \subseteq \Delta_2\\ S_1 \sqsubseteq S_2\end{array}}{\mathrm{Let}(z, t, \Delta_1) \bullet S_1 \sqsubseteq \mathrm{Let}(z, t, \Delta_2) \bullet S_2} \qquad \square$$

Definition B.9 (State extension)**.** One state $\langle t_1;\ \Gamma_1;\ S_1;\ H_1\rangle$ is extended by another state $\langle t_2;\ \Gamma_2;\ S_2;\ H_2\rangle$, written $\langle t_1;\ \Gamma_1;\ S_1;\ H_1\rangle \sqsubseteq \langle t_2;\ \Gamma_2;\ S_2;\ H_2\rangle$ or $\langle t_2;\ \Gamma_2;\ S_2;\ H_2\rangle \sqsupseteq \langle t_1;\ \Gamma_1;\ S_1;\ H_1\rangle$, if $t_1 = t_2$, $S_1 \sqsubseteq S_2$, and $(\Gamma_1, H_1) \sqsubseteq (\Gamma_2, H_2)$. ☐

B.3 Lemmas

Lemma B.10. *For all* Γ, E, *if* $[\![E]\!]^\Gamma = (\Delta_1, H)$, *then for all patterns* $p \mapsto (t, \Delta_2) \in H$, $\Delta_2 \subseteq \Delta_1$.

Proof. Straightforward induction on the rules of translating environments. For rules TR_EMPTY and TR_UNBOXED, the condition holds trivially, as H is either $\emptyset$ or unchanged. For TR_BOXED, H is extended with Δ, which is a subset of Δ'. ☐

Lemma B.11 (Δ & H uniqueness)**.** *For all Γ and E, if $\Gamma \vdash E$ and $[\![E]\!]^\Gamma = (\Delta, H)$, then all mappings bound in Δ and H are unique. That is, the following holds:*

- *For all x, b, and b', if $x \mapsto b \in \Delta$ and $x \mapsto b' \in \Delta$, then $b = b'$.*
- *For all p, (t, Δ), and (t', Δ'), if $p \mapsto (t, \Delta)$ and $p \mapsto (t', \Delta')$, then $(t, \Delta) = (t', \Delta')$.*

Proof. Following the Barendregt's convention, we assume that all $\gamma \in \Gamma$ are fresh. As $\Gamma \vdash E$, translation $[\![\gamma]\!]^\Gamma$ is uniquely determined by γ, and that each p is fresh, the property holds. ☐

Lemma B.12 (Scope of E flows left)**.** *For all Γ, E, and $\gamma \mapsto e$, if $\Gamma \vdash E$ and $E = E_1, \gamma \mapsto e, E_2$, then E_1 contains bindings for all of the closed over variables of e.*

Proof. By induction on the operational semantic rules.

Rules S_VAR and S_TBETA do not alter E, so the binding cannot have been introduced using these rules. Rules S_APP, S_LET$_{\#a}$, S_TLAM, and S_TAPP do not alter E themselves, but instead take E' from their assumptions. Remaining are cases S_LET, S_LET$_{\#b}$, S_LAM, and S_LAM$_{\#}$, which each extend E by a binding.

For S_LET and S_LET$_{\#b}$, each rule extends E with a binding of γ to the right-hand side of the let binding. Therefore, if we match the situation with the proposition, we get $E_1, \gamma \mapsto e, \emptyset$, where $E = E_1$. As $\Gamma \vdash E_1$ for some Γ, and the right-hand sides are well-typed w.r.t. that same Γ, it follows that all closed over variables must be in E_1 as well.

Likewise, rules S_LAM and S_LAM$_{\#}$ each extend E to the right with a binding of the lambda's argument γ_1 to the expression bound to the variable applied to. Therefore, if we match the situation with the proposition, we get $E_1, \gamma \mapsto e, \emptyset$, where $E = E_1$ and $\gamma = \gamma_1$. Similarly, in both cases $\Gamma \vdash E_1$ for some Γ, and the new binding is required to be well-typed w.r.t. this same Γ. Therefore, it follows that all closed over variables must be in E_1 as well. ☐

Lemma B.13 (Environment translating binding consistency)**.** *If $\Gamma \vdash E$, $\gamma \mapsto e \in E$, and $[\![E]\!]^\Gamma = (\Delta, H)$, then, for some b, $[\![\gamma]\!]^\Gamma \mapsto b \in \Delta$.*

Proof. Straightforward induction on the translation rules for environments. ☐

Lemma B.14 (Variable lookup)**.** *For all Γ, E, and $\gamma \mapsto e$, if $\Gamma \vdash E$ and $\gamma \mapsto e \in E$, then for $[\![E]\!]^\Gamma = (\Delta, H)$, $[\![\gamma]\!]^\Gamma = x$, and $[\![e]\!]^\Gamma = t$, one of the following holds:*

- *Either e is of boxed kind, $x = y$, and there exits a p such that $\Delta[y] = p$ and $H[p] = (t, \Delta')$, where $\Delta' \vdash t$ WF.*
- *Or e is of unboxed kind, $x = z$, and $\Delta[z] = (t, \Delta')$, where $\Delta' \vdash t$ WF.*

Proof. If $\gamma \mapsto e \in E$, E is of form $E_1, \gamma \mapsto e, E_2$. As TR_BOXED and TR_UNBOXED only extend the intermediate result, $[\![E_1, \gamma \mapsto e]\!]^\Gamma \subseteq [\![E_1, \gamma \mapsto e, E_2]\!]^\Gamma$. If we show that our desired output is part of $[\![E_1, \gamma \mapsto e]\!]^\Gamma$, by lemma B.11 we know that looking up the binders in $[\![E_1, \gamma \mapsto e, E_2]\!]^\Gamma$ gives the desired result.

If e is of boxed kind, $x = y$, and TR_BOXED was used for $[\![E_1, \gamma \mapsto e]\!]^\Gamma$. We add $[\![\gamma]\!]^\Gamma \mapsto p$ and $p \mapsto ([\![e]\!], \Delta')$ to the intermediate environment and heap respectively. If e is of unboxed kind, $x = z$, and TR_UNBOXED was used for $[\![E_1, \gamma \mapsto e]\!]^\Gamma$. We add $[\![\gamma]\!]^\Gamma \mapsto ([\![e]\!], \Delta')$ to the intermediate environment.

In both cases, Δ' is the result of translating all to the left of $\gamma \mapsto e$, i.e. E_1, which by lemma B.12 contains bindings for all closed over variables of t. ☐

Lemma B.15 (E translation is well formed)**.** *For all* Γ*,* E*, if* $\Gamma \vdash E$ *and* $[\![E]\!]^\Gamma = (\Delta, H)$*, then* $H \vdash \Delta$ *WF.*

Proof. By induction on the translation rules for environments. For TR_EMPTY, the proposition trivially holds. For rule TR_BOXED, $E, \gamma \mapsto e$ is translated, where $[\![\gamma]\!]^\Gamma = y$ and $[\![e]\!]^\Gamma = t$. Δ is extended with a binding $y \mapsto p$, and H is extended with a binding $p \mapsto (t, \Delta)$. As by lemmas B.12 and B.13 all closed over variables of t are in Δ, we have that $\Delta \vdash t$ WF. As by induction we have that $H \vdash \Delta$ WF, the extended environment and heap are well-formed as well. Rule TR_UNBOXED is similar. Here, $E, \gamma \mapsto v$ is translated, where $[\![\gamma]\!]^\Gamma = z$ and $[\![v]\!]^\Gamma = w$. As by lemmas B.12 and B.13 all closed over variables of w are in Δ, we have that $\Delta \vdash t$ WF. As by induction we have that $H \vdash \Delta$ WF, the extended environment and heap are well-formed as well. ☐

Lemma B.16 (Full translation is well formed)**.** *If* $[\![\langle \Gamma;\ E;\ e \rangle]\!]^S = \langle t;\ \Delta;\ S;\ H \rangle$ *and* $H \vdash S$ *WF, then* $\langle t;\ \Delta;\ S;\ H \rangle$ *WF.*

Proof. As our work item is t, we do not need to consider the cases of definition B.6 where $i = p$ and $i = (t', \Delta')$. As our stack is assumed to be well-formed, we only need to show that the closure (t, Δ) and binding storage (Δ, H) are well-formed. For the well-formedness of the closure we observe that $\Gamma \vdash E$, which means that E contains mappings for all closed over variables of e. By lemma B.13 it follows that $\Delta \vdash t$ WF. $H \vdash \Delta$ WF follows from lemma B.15, which means we have proven the proposition. ☐

Lemma B.17. *For all* E*, if* $\gamma \mapsto e \in E$ *and* e *is of unboxed kind, then* e *must be a value.*

Proof. By induction on the operational semantic rules.

Rules S_VAR and S_TBETA do not extend E, so the binding cannot have been introduced using these rules. Rules S_APP, S_LET$_{\#a}$, S_TLAM, and S_TAPP do not alter E themselves, but instead take E' from their assumptions. Remaining are the rules that extend E themselves.

S_LET extends E' with a binding of boxed kind, which means e is boxed. S_LET$_{\#b}$ extends E' with a binding of unboxed kind, which is a value. Finally, rules S_LAM and S_LAM$_{\#}$ extend E' with a pattern $\gamma \mapsto e$, where $e \in E$. As by induction we know it to hold for all elements of E, e must be a value, if it is of unboxed kind. ☐

Lemma B.18 (Compilation ignores type substitution)**.** *If* $[\![e]\!]^\Gamma = t$*, then* $[\![e[\tau/\alpha]]\!]^\Gamma = t$*.*

Proof. Straightforward induction on the rules for compiling terms. Here, only the rules C_TLAM and C_TAPP are relevant, as these are the only places where α can occur. As both rules erase the type variable in compilation, substitution does not affect the compilation. ☐

B.4 Simulation

Theorem 7.3 (Simulation)**.** *For all* $\langle \Gamma;\ E;\ e\rangle \longrightarrow \langle \Gamma';\ E';\ e'\rangle$ *and stacks* S_1 *and* S_1'*, let* $Q_1 = [\![\langle \Gamma;\ E;\ e\rangle]\!]^{S_1} = \langle t_1;\ \Delta_1;\ S_1;\ H_1\rangle$ *and* $Q_1' = [\![\langle \Gamma';\ E';\ e'\rangle]\!]^{S_1'} = \langle t_1';\ \Delta_1';\ S_1';\ H_1'\rangle$*.*

If $\Gamma \vdash e : \tau$*,* $\Gamma \vdash E$*,* $S_1 \sqsubseteq S_1'$*,* $H_1 \vdash S_1$ *WF, and* $H_1' \vdash S_1'$ *WF, there exists a* Q_2 *and a* Q_2' *such that* $Q_1 \xrightarrow{*} Q_2$*,* $Q_1' \xrightarrow{*} Q_2'$*,* $Q_2 \sqsubseteq Q_2'$*,* Q_2 *WF, and* Q_2' *WF.*

Proof. By induction on the typing derivation of e.

E_Var

$$\textsc{E\_Var}\ \frac{\gamma{:}\tau \in \Gamma}{\Gamma \vdash \gamma : \tau} \qquad \textsc{C\_Var}\ \frac{\kappa = \mathit{kindsOf}(\Gamma) \quad x = \mathit{lookup}(\kappa, \gamma)}{[\![\gamma]\!]^{\Gamma} = x} \qquad \textsc{S\_Var}\ \frac{\gamma \mapsto e \in E}{\langle \Gamma, E, \gamma\rangle \longrightarrow \langle \Gamma, E, e\rangle}$$

Here we case further on e being a value or not.

e is a value

$[\![\langle \Gamma;\ E;\ \gamma\rangle]\!]^{S}$: Lookup in $\mathcal{M}$ differs on whether a boxed or unboxed closure is looked up. However, as can be seen below, both cases result into the same state:

$$\begin{aligned}
\langle y;\ \Delta_1;\ S;\ H\rangle &\longrightarrow \langle y;\ \Delta_1[y] = p;\ S;\ H\rangle \\
&\longrightarrow \langle p;\ \Delta_1;\ S;\ H\rangle \\
&\longrightarrow \langle p;\ \Delta_1;\ S;\ H[p] = (w, \Delta_2)\rangle \\
&\longrightarrow \langle (w, \Delta_2);\ \Delta_1;\ S;\ H\rangle
\end{aligned}$$

For the boxed case, by lemma B.14 we know that the lookup $\Delta_1[y]$ is guaranteed to result in some p, that when looked up on the heap, i.e. $H[p]$, resolves into (w, Δ_2), where $[\![e]\!]^{\Gamma} = w$.

$$\begin{aligned}
\langle z;\ \Delta_1;\ S;\ H\rangle &\longrightarrow \langle z;\ \Delta_1[z] = (w, \Delta_2);\ S;\ H\rangle \\
&\longrightarrow \langle (w, \Delta_2);\ \Delta_1;\ S;\ H\rangle
\end{aligned}$$

For the unboxed case, by lemma B.14 we know that the lookup $\Delta_1[x]$ is guaranteed to result in (w, Δ_2), where $[\![e]\!]^{\Gamma} = w$.

For the well-formedness of both the boxed and unboxed case, we observe that our final Δ_1, S, and H are equal to the output of $[\![\langle \Gamma;\ E;\ \gamma\rangle]\!]^{S}$, which by lemma B.16 are known to be well-formed. Our work item has changed to (w, Δ_2), which in each case has been extracted from a well-formed structure. Therefore, $\Delta_2 \vdash w$ WF, and thus $\langle (w, \Delta_2);\ \Delta_1;\ S;\ H\rangle$ WF.

$[\![\langle \Gamma;\ E;\ e\rangle]\!]^{S}$:

$$\langle w;\ \Delta_1;\ S;\ H\rangle \quad \longrightarrow \quad \langle (w, \Delta_2);\ \Delta_1;\ S;\ H\rangle$$

For this case, we first observe that by lemma B.16, $\langle w;\ \Delta_1;\ S;\ H\rangle$ WF.

Since $[\![e]\!]^{\Gamma} = w$, w becomes our work item. As Γ, E, and S are all unchanged, we know w is placed in an context equal to the one of γ, i.e. $[\![\langle \Gamma;\ E;\ e\rangle]\!]^{S} = \langle w;\ \Delta_1;\ S;\ H\rangle$.

As $\Delta_1 \vdash t$ WF, we know that $\Delta_2 \subseteq \Delta_1$. As Δ_1 is our working environment, the lifting of w to (w, Δ_2) will always succeed. Therefore, we have arrived at a state that extends our previous state.

For the well-formedness of this case, we observe that our final Δ_1, S, and H are unchanged. As the lift operation takes exactly the closed over variables from Δ_1, it follows that $\Delta_2 \vdash w$ WF, which means $\langle (w, \Delta_2);\ \Delta_1;\ S;\ H\rangle$ WF.

e is a non-value

By S_Var, $\gamma \mapsto e \in E$. By lemma B.17, patterns where e is both a non-value and of unboxed kind cannot occur. Therefore, e can only represent a term of boxed kind.

$[\![\langle \Gamma;\ E;\ \gamma\rangle]\!]^S$:

$$\begin{array}{rcl} \langle y;\ \Delta_1;\ S;\ H\rangle & \longrightarrow & \langle y;\ \Delta_1[y] = p;\ S;\ H\rangle \\ & \longrightarrow & \langle p;\ \Delta_1;\ S;\ H\rangle \\ & \longrightarrow & \langle p;\ \Delta_1;\ S;\ H[p] = (t, \Delta_2)\rangle \\ & \longrightarrow & \langle t;\ \Delta_2;\ S;\ H\rangle \end{array}$$

Per lemma B.14 we know that the lookup $\Delta_1[y]$ is guaranteed to result in p, and the lookup $H[p]$ is guaranteed to resolve into (t, Δ_2), where $[\![e]\!]^\Gamma = w$, and $\Delta_2 \vdash t$ WF. Finally, we step into t under Δ_2.

For the well-formedness of this case, we observe that our final S and H are equal to the output of $[\![\langle \Gamma;\ E;\ \gamma\rangle]\!]^S$, which by lemma B.16 are known to be well-formed. As the closure (t, Δ_2) is extracted from the well-formed heap, it must be well-formed itself. Switching to this closure retains well-formedness, so our final state is well-formed.

$[\![\langle \Gamma;\ E;\ e\rangle]\!]^S$: As $[\![e]\!]^\Gamma = t$, and E is unchanged, $[\![\langle \Gamma;\ E;\ e\rangle]\!]^S = \langle t;\ \Delta_2';\ S;\ H\rangle$, which by lemma B.16 is well-formed. By lemma B.10, $\Delta_2 \subseteq \Delta_2'$, which means this state extends the previous state.

E_App

$$\text{E\_App}\ \frac{\Gamma \vdash e : \tau_1 \xrightarrow{A} \tau_2 \qquad \Gamma \vdash \gamma : \tau_1}{\Gamma \vdash e\ \gamma : \tau_2} \qquad \text{C\_App}\ \frac{[\![e]\!]^\Gamma = t \quad [\![\gamma]\!]^\Gamma = x}{[\![e\ \gamma]\!]^\Gamma = t\ x}$$

We have two cases, depending on how $e\ \gamma$ has stepped.

S_App

$[\![\langle \Gamma;\ E;\ e_1\ \gamma\rangle]\!]^S$:

$$\begin{array}{rcl} \langle t_1\ x;\ \Delta_1;\ S;\ H_1\rangle & \longrightarrow & \langle t_1\ x;\ \Delta_1[z] = b;\ S;\ H_1\rangle \\ & \longrightarrow & \langle t_1;\ \Delta_1;\ \mathrm{App}(b) \bullet S;\ H_1\rangle \\ & \longrightarrow^* & \langle t_2;\ \Delta_2;\ \mathrm{App}(b) \bullet S;\ H_2\rangle \end{array}$$

$[\![\langle\Gamma';\ E';\ e'_1\ \gamma\rangle]\!]^S$:

$$\begin{array}{rcl} \langle t'_1\ x;\ \Delta'_1;\ S;\ H'_1\rangle & \longrightarrow & \langle t'_1\ x;\ \Delta'_1[z]=b;\ S;\ H'_1\rangle \\ & \longrightarrow & \langle t'_1;\ \Delta'_1;\ \mathrm{App}(b)\bullet S;\ H_1\rangle \\ & \longrightarrow^* & \langle t'_2;\ \Delta'_2;\ \mathrm{App}(b)\bullet S;\ H'_2\rangle \end{array}$$

By lemmas A.3 and B.14, $\Delta_1[x] = \Delta'_1[x] = b$. For the application $e_1\ \gamma$, the resulting b is stored in the App continuation, and the left hand side t_1 is made the work item. For the application $e'_1\ \gamma$ the process is similar: the (same) result of the lookup b is stored in the App continuation, and the left hand side is made the work item, in this case t'_1.

As $\Gamma \vdash e_1 : \tau_1$, $\Gamma \vdash E$, $\langle\Gamma, E, e_1\rangle \longrightarrow \langle\Gamma', E', e'_1\rangle$, $[\![\langle\Gamma;\ E;\ e_1\rangle]\!]^{\mathrm{App}(b)\bullet S} = \langle t_1;\ \Delta_1;\ \mathrm{App}(b)\bullet S;\ H_1\rangle$, and $[\![\langle\Gamma';\ E';\ e'_1\rangle]\!]^{\mathrm{App}(b)\bullet S} = \langle t'_1;\ \Delta'_1;\ \mathrm{App}(b)\bullet S;\ H'_1\rangle$, we can apply the induction hypothesis to both states to arrive in states satisfying the proposition.

S_Lam

We case further on the kinds of γ_1 and γ_2. As γ_1 and γ_2 must be of the same type (and thus kind) for an application to be well-typed, we only need to consider the case where both are of boxed kind, and the case where both are of unboxed kind.

γ_1 and γ_2 of boxed kind From the information that γ_2 is boxed and lemma B.14 we get that $[\![\gamma_2]\!]^\Gamma = y_2$, and $\Delta_1[y_2] = p$. From this, we get the following reduction steps:

$[\![\langle\Gamma;\ E;\ (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2\rangle]\!]^S$:

$$\begin{array}{rcl} \langle(\lambda y_1.t_1)\ y_2;\ \Delta_1;\ S;\ H\rangle & \longrightarrow & \langle(\lambda y_1.t_1)\ y_2;\ \Delta_1[y_2]=p;\ S;\ H\rangle \\ & \longrightarrow & \langle\lambda y_1.t_1;\ \Delta_1;\ \mathrm{App}(p)\bullet S;\ H\rangle \\ & \longrightarrow & \langle(\lambda y_1.t_1, \Delta_2);\ \Delta_1;\ \mathrm{App}(p)\bullet S;\ H\rangle \\ & \longrightarrow & \langle t_1;\ y_1 \mapsto p\bullet\Delta_2;\ S;\ H\rangle \\ & & \text{where } \Delta_2 = fv(t_1),\ \Delta_2 \subseteq \Delta_1 \end{array}$$

For the well-formedness of this case, we observe that our final S and H are equal to the output of $[\![\langle\Gamma;\ E;\ (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2\rangle]\!]^S$, which by lemma B.16 are known to be well-formed. As lifting $\lambda y_1.t_1$ to $(\lambda y_1.t_1, \Delta_2)$ stores all closed over variables of the term in Δ_2, we have that $\Delta_2 \vdash \lambda y_1.t_1\ \mathsf{WF}$. The inner term of this lambda, t_1, contains one additional closed over variable, namely a binding for y_1. As we extend Δ_2 by exactly this binding, $y_1 \mapsto p\bullet\Delta_2 \vdash t_1\ \mathsf{WF}$. Therefore, the final state is well-formed as well.

$[\![\langle\Gamma, \gamma_1{:}\tau;\ E, \gamma_1 \mapsto e_2;\ e_1\rangle]\!]^S$: As $[\![E]\!]^\Gamma = (\Delta_1, H)$ and γ_1 represents a term of boxed kind, we know that $[\![E, \gamma_1 \mapsto e_2]\!]^{\Gamma,\gamma_1:\tau} = (y_1 \mapsto p'\bullet\Delta_1, p' \mapsto (t_1, \Delta_1)\bullet H)$. This gives us that $[\![\langle\Gamma, \gamma_1{:}\tau;\ E, \gamma_1 \mapsto e_2;\ e_1\rangle]\!]^S = \langle t_1;\ y_1 \mapsto p'\bullet\Delta_1;\ S;\ p' \mapsto (t_1, \Delta_1)\bullet H\rangle$.

As $\Delta_2 \subseteq \Delta_1$, $(\Delta_2, H) \sqsubseteq (\Delta_1, H)$. Furthermore, note that $(y_1 \mapsto p\bullet\Delta_2)[y_1] = p$, $H[p] = (t_1, \Delta_3)$, for some Δ_3, $(y_1 \mapsto p'\bullet\Delta_1)[y_1] = p'$, and $(p' \mapsto (t_1, \Delta_1)\bullet H)[p'] = (t_1, \Delta_1)$. By lemma B.10, $\Delta_3 \subseteq \Delta_1$. Therefore, $(y_1 \mapsto p\bullet\Delta_2, H) \sqsubseteq (y_1 \mapsto p'\bullet\Delta_1, H)$. Furthermore, by lemma B.16, the state is well-formed.

γ_1 and γ_2 of unboxed kind

$[\![\langle \Gamma;\ E;\ (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2\rangle]\!]^S$: From the information that γ_2 is unboxed and lemma B.14 we get that $[\![\gamma_2]\!]^\Gamma = z_2$, $\Delta_1[z_2] = (t_2, \Delta_3)$. From this, we get the following reduction steps:

$$\begin{array}{rcl}
\langle (\lambda z_1.t_1)\ z_2;\ \Delta_1;\ S;\ H\rangle & \longrightarrow & \langle (\lambda z_1.t_1)\ z_2;\ \Delta_1[z_2] = (t_2, \Delta_3);\ S;\ H\rangle \\
 & \longrightarrow & \langle \lambda z_1.t_1;\ \Delta_1;\ \mathrm{App}(t_2, \Delta_3) \bullet S;\ H\rangle \\
 & \longrightarrow & \langle (\lambda z_1.t_1, \Delta_2);\ \Delta_1;\ \mathrm{App}(t_2, \Delta_3) \bullet S;\ H\rangle \\
 & \longrightarrow & \langle t_1;\ z_1 \mapsto (t_2, \Delta_3) \bullet \Delta_2;\ S;\ H\rangle \\
 & & \text{where } \Delta_2 = fv(t_1),\ \Delta_2 \subseteq \Delta_1
\end{array}$$

For the well-formedness of this case, we observe that our final S and H are equal to the output of $[\![\langle \Gamma;\ E;\ (\lambda\gamma_1{:}\tau.e_1)\ \gamma_2\rangle]\!]^S$, which by lemma B.16 are known to be well-formed. As lifting $\lambda y_1.t_1$ to $(\lambda y_1.t_1, \Delta_2)$ stores all closed over variables of the term in Δ_2, we have $\Delta_2 \vdash \lambda y_1.t_1\ \mathsf{WF}$. The inner term of this lambda, t_1, contains one additional closed over variable, namely a binding for y_1. As we extend Δ_2 by exactly this binding, $z_1 \mapsto (t_2, \Delta_3) \bullet \Delta_2 \vdash t_1\ \mathsf{WF}$. Therefore, $\langle t_1;\ z_1 \mapsto (t_2, \Delta_3) \bullet \Delta_2;\ S;\ H\rangle\ \mathsf{WF}$.

$[\![\langle \Gamma, \gamma_1{:}\tau;\ \ E, \gamma_1 \mapsto e_2;\ \ e_1\rangle]\!]^S$: As $[\![E]\!]^\Gamma = (\Delta_1, H)$ and γ_1 represents a term of unboxed kind, we know that $[\![E, \gamma_1 \mapsto e_2]\!]^{\Gamma, \gamma_1:\tau} = (z_1 \mapsto (t_2, \Delta_1) \bullet \Delta_1, H)$. Therefore, it follows that $[\![\langle \Gamma, \gamma_1{:}\tau;\ (E, \gamma_1 \mapsto e_2);\ e_1\rangle]\!]^S = \langle t_1;\ y_1 \mapsto (t_2, \Delta_1) \bullet \Delta_1;\ S;\ H\rangle$.

As $\Delta_2 \subseteq \Delta_1$, $(\Delta_2, H) \sqsubseteq (\Delta_1, H)$, and thus $(\Delta_2, H) \sqsubseteq (y_1 \mapsto (t_2, \Delta_1) \bullet \Delta_1, H)$, which means the state extends the previous state. Furthermore, by lemma B.16, the state is well-formed.

E_App$_\#$

Similar to the case of E_App.

E_TLam

$$\text{E\_TLam}\ \frac{\Gamma \bullet \alpha{:}\kappa \vdash e : \tau \qquad \Gamma \vdash_\kappa \kappa\,\text{kind}}{\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\,\alpha{:}\kappa.\ \tau} \qquad \text{C\_TLam}\ \frac{[\![e]\!]^\Gamma = t}{[\![\Lambda\alpha{:}\kappa.e]\!]^\Gamma = t}$$

$$\text{S\_TLam}\ \frac{\langle \Gamma, E, e\rangle \longrightarrow \langle \Gamma', E', e'\rangle}{\langle \Gamma, E, \Lambda\alpha{:}\kappa.e\rangle \longrightarrow \langle \Gamma', E', \Lambda\alpha{:}\kappa.e'\rangle}$$

In this case, the proposition holds trivially, as by C_TLam $[\![\Lambda\alpha{:}\kappa.e]\!]^\Gamma = [\![e]\!]^\Gamma = t$, and $[\![\Lambda\alpha{:}\kappa.e']\!]^{\Gamma'} = [\![e']\!]^{\Gamma'} = t'$. Therefore, $[\![\langle \Gamma;\ E;\ \Lambda\alpha{:}\kappa.e\rangle]\!]^S = [\![\langle \Gamma;\ E;\ e\rangle]\!]^S$ and $[\![\langle \Gamma';\ E';\ \Lambda\alpha{:}\kappa.e'\rangle]\!]^S = [\![\langle \Gamma';\ E';\ e'\rangle]\!]^S$, which means we can use the induction hypothesis directly.

E_TApp

$$\text{E\_TApp}\ \frac{\Gamma \vdash e : \forall\,\alpha{:}\kappa.\ \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/\alpha]} \qquad \text{C\_TApp}\ \frac{[\![e]\!]^\Gamma = t}{[\![e\ \tau]\!]^\Gamma = t}$$

We have two cases, depending on how $e\ \tau$ has stepped.

S_TApp

$$\textsc{S\_Tapp}\ \frac{\langle \Gamma, E, e\rangle \longrightarrow \langle \Gamma', E', e'\rangle}{\langle \Gamma, E, e\ \tau\rangle \longrightarrow \langle \Gamma', E', e'\ \tau\rangle}$$

In this case, the proposition holds trivially. By C_TApp, $[\![e\ \tau]\!]^{\Gamma} = [\![e]\!]^{\Gamma} = t$, and $[\![e'\ \tau]\!]^{\Gamma'} = [\![e']\!]^{\Gamma'} = t'$. Therefore, $[\![\langle \Gamma;\ E;\ e\ \tau\rangle]\!]^{S} = [\![\langle \Gamma;\ E;\ e\rangle]\!]^{S}$ and $[\![\langle \Gamma;\ E';\ e'\ \tau\rangle]\!]^{S} = [\![\langle \Gamma';\ E';\ e'\rangle]\!]^{S}$, which means we can use the induction hypothesis directly.

S_TBeta

$$\textsc{S\_TBeta}\ \frac{\begin{array}{c}\Gamma_1 = \Gamma \bullet \alpha{:}\kappa \bullet \Gamma' \\ \Gamma_2 = \Gamma \bullet \Gamma'\end{array}}{\begin{array}{l}\langle \Gamma_1, E, (\Lambda\alpha{:}\kappa.v)\ \tau\rangle \\ \quad\longrightarrow \langle \Gamma_2[\tau/\alpha], E[\tau/\alpha], v[\tau/\alpha]\rangle\end{array}}$$

In this case, the proposition holds trivially. By C_TApp, C_TLam, and lemma B.18, we have that $[\![(\Lambda\alpha{:}\kappa.v)\ \tau]\!]^{\Gamma} = [\![\Lambda\alpha{:}\kappa.v]\!]^{\Gamma} = [\![v]\!]^{\Gamma} = [\![v[\tau/\alpha]]\!]^{\Gamma}$. Therefore, $[\![\langle \Gamma;\ E;\ (\Lambda\alpha{:}\kappa.v)\ \tau\rangle]\!]^{S} = [\![\langle \Gamma;\ E;\ \Lambda\alpha{:}\kappa.v\rangle]\!]^{S}$. As by lemma B.15 our singular state is well-formed, the property holds.

E_Let

$$\textsc{E\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash \tau_1 : \mathit{TYPE\ P\ A} \\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \textsc{C\_Let}\ \frac{\begin{array}{ll}\kappa = \mathit{kindsOf}(\Gamma) & [\![e_1]\!]^{\Gamma} = t_1 \\ x = \mathit{fresh}(\kappa) & [\![e_2]\!]^{\Gamma\bullet\gamma:\tau} = t_2\end{array}}{\begin{array}{l}[\![\mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2]\!]^{\Gamma} \\ \quad = \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\end{array}}$$

$$\textsc{S\_Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma' = \Gamma \bullet \gamma{:}\tau \\ E' = E, \gamma \mapsto e_1\end{array}}{\langle \Gamma, E, \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2\rangle \longrightarrow \langle \Gamma', E', e_2\rangle}$$

$[\![\langle \Gamma;\ E;\ (\mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2)\rangle]\!]^{S}$:

$$\langle \mathbf{let}\ y = t_1\ \mathbf{in}\ t_2;\ \Delta;\ S;\ H\rangle \quad\longrightarrow\quad \langle t_2;\ y \mapsto p \bullet \Delta;\ S;\ p \mapsto (t_1, \Delta) \bullet H\rangle$$

By lemma B.16, $\langle \mathbf{let}\ y = t_1\ \mathbf{in}\ t_2;\ \Delta;\ S;\ H\rangle$ WF. Therefore, $\Delta \vdash \mathbf{let}\ y = t_1\ \mathbf{in}\ t_2$ WF, which means that $fv(\mathbf{let}\ y = t_1\ \mathbf{in}\ t_2) \subseteq \Delta$. As $[\![\mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2]\!]^{\Gamma} = \mathbf{let}\ y = t_1\ \mathbf{in}\ t_2$ and $\Gamma \vdash \mathbf{let}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau$, by rule E_Let, $\Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2$. Therefore, t_2 ranges over exactly one additional closed over variable, namely the binding $y = t_1$. As this exact binding is supplied by extending Δ and H, we have that $y \mapsto p \bullet \Delta \vdash t_2$ WF and $p \mapsto (t_1, \Delta) \bullet H \vdash y \mapsto p \bullet \Delta$ WF, and thus $\langle t_2;\ y \mapsto p \bullet \Delta;\ S;\ p \mapsto (t_1, \Delta) \bullet H\rangle$ WF.

$[\![\langle \Gamma, \gamma{:}\tau;\ (E, \gamma \mapsto e_1);\ e_2\rangle]\!]^{S}$: As $[\![E]\!]^{\Gamma} = (\Delta, H)$ and γ represents a term of boxed kind, we know that $[\![E, \gamma \mapsto e_2]\!]^{\Gamma,\gamma:\tau} = (y \mapsto p' \bullet \Delta, p' \mapsto (t_1, \Delta) \bullet H)$. Therefore, we get that $[\![\langle \Gamma, \gamma{:}\tau;\ (E, \gamma \mapsto e_1);\ e_2\rangle]\!]^{S} = \langle t_2;\ y \mapsto p' \bullet \Delta;\ S;\ p' \mapsto (t_1, \Delta) \bullet H\rangle$, which means this state extends the previous state. Furthermore, by lemma B.16, the state is well-formed.

$\textbf{E\_Let}_{\#}$

$$\textsc{E\_Let}_{\#}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1\\ \Gamma \vdash \tau_1 : \mathit{TYPE}\ U\ A\\ \Gamma \bullet \gamma{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \textsc{C\_Let}_{\#}\ \frac{\begin{array}{cc}\kappa = \mathit{kindsOf}(\Gamma) & [\![e_1]\!]^{\Gamma} = t_1\\ x = \mathit{fresh}(\kappa) & [\![e_2]\!]^{\Gamma\bullet\gamma:\tau} = t_2\end{array}}{\begin{array}{c}[\![\mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2]\!]^{\Gamma}\\ = \mathbf{let}_{\#}\ x = t_1\ \mathbf{in}\ t_2\end{array}}$$

We have two cases, depending on how $\mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2$ has stepped.

$\textbf{S\_Let}_{\#a}$

$$\textsc{S\_Let}_{\#a}\ \frac{\langle \Gamma, E, e_1\rangle \longrightarrow \langle \Gamma', E', e_1'\rangle}{\begin{array}{c}\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2\rangle\\ \longrightarrow \langle \Gamma', E', \mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2\rangle\end{array}}$$

$[\![\langle\Gamma;\ E;\ (\mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2)\rangle]\!]^S$:

$$\begin{array}{rcl}\langle \mathbf{let}_{\#}\ z = t_1\ \mathbf{in}\ t_2;\ \Delta_1;\ S;\ H_1\rangle & \longrightarrow & \langle t_1;\ \Delta_1;\ \mathrm{Let}(z, t_2, \Delta_1) \bullet S;\ H_1\rangle\\ & \longrightarrow^* & \langle t_3;\ \Delta_2;\ \mathrm{Let}(z, t_2, \Delta_1) \bullet S;\ H_2\rangle\end{array}$$

$[\![\langle\Gamma';\ E';\ (\mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2)\rangle]\!]^S$:

$$\begin{array}{rcl}\langle \mathbf{let}_{\#}\ z = t_1'\ \mathbf{in}\ t_2;\ \Delta_1';\ S;\ H_1'\rangle & \longrightarrow & \langle t_1';\ \Delta_1';\ \mathrm{Let}(z, t_2, \Delta_1') \bullet S;\ H_1'\rangle\\ & \longrightarrow^* & \langle t_3;\ \Delta_2';\ \mathrm{Let}(z, t_2, \Delta_1') \bullet S;\ H_3'\rangle\end{array}$$

In both $[\![\langle\Gamma;\ E;\ (\mathbf{let}_{\#}\ \gamma = e_1\ \mathbf{in}\ e_2)\rangle]\!]^S$ and $[\![\langle\Gamma';\ E';\ (\mathbf{let}_{\#}\ \gamma = e_1'\ \mathbf{in}\ e_2)\rangle]\!]^S$, the first step saves the let continuation on the stack. This continuation differs slightly: while the variable and term stored are equal, the environments differ (Δ_1 versus Δ_1'). However, by lemma A.3, $E \subseteq E'$, which means $\Delta_1 \subseteq \Delta_1'$. Therefore, $\mathrm{Let}(z, t_2, \Delta_1) \bullet S \sqsubseteq \mathrm{Let}(z, t_2, \Delta_1') \bullet S$, which means we can apply the induction hypothesis and arrive at states satisfying the proposition.

$\textbf{S\_Let}_{\#b}$

$$\textsc{S\_Let}_{\#b}\ \frac{\begin{array}{c}\Gamma \vdash v : \tau\\ \Gamma' = \Gamma \bullet \gamma{:}\tau\\ E' = E, \gamma \mapsto v\end{array}}{\langle \Gamma, E, \mathbf{let}_{\#}\ \gamma = v\ \mathbf{in}\ e_2\rangle \longrightarrow \langle \Gamma', E', e_2\rangle}$$

$[\![\langle\Gamma;\ E;\ (\mathbf{let}_{\#}\ \gamma = v\ \mathbf{in}\ e_2)\rangle]\!]^S$:

$$\begin{array}{rcl}\langle \mathbf{let}_{\#}\ z = w\ \mathbf{in}\ t_2;\ \Delta_1;\ S;\ H\rangle & \longrightarrow & \langle w;\ \Delta_1;\ \mathrm{Let}(z, t_2, \Delta_1) \bullet S;\ H\rangle\\ & \longrightarrow & \langle (w, \Delta_2);\ \Delta_1;\ \mathrm{Let}(z, t_2, \Delta_1) \bullet S;\ H\rangle\\ & \longrightarrow & \langle t_2;\ z \mapsto (w, \Delta_2) \bullet \Delta_1;\ S;\ H\rangle\\ & & \text{where } \Delta_2 = fv(w),\ \Delta_2 \subseteq \Delta_1\end{array}$$

For the well-formedness of this case, we observe that our final S and H are equal to the output of $[\![\langle\Gamma;\ E;\ (\mathbf{let}_{\#}\ \gamma = v\ \mathbf{in}\ e_2)\rangle]\!]^S$, which by lemma B.16 are known to be well-formed. Δ_2 is a subset of the well-formed Δ_1, and contains all closed over variables of t_2. Therefore, $\Delta_2 \vdash t_2\ \mathsf{WF}$ and $H \vdash \Delta_2\ \mathsf{WF}$, which means the state is well-formed.

$[\![\langle\Gamma, \gamma{:}\tau;\ (E, \gamma \mapsto v);\ e_2\rangle]\!]^S$: As $[\![E]\!]^\Gamma = (\Delta, H)$ and γ represents a term of unboxed kind, we know that $[\![E, \gamma \mapsto v]\!]^{\Gamma,\gamma:\tau} = (z \mapsto (w, \Delta_1) \bullet \Delta_1, H)$. This gives us that $[\![\langle\Gamma, \gamma{:}\tau;\ (E, \gamma \mapsto v);\ e_2\rangle]\!]^S = \langle t_2;\ z \mapsto (w, \Delta_1) \bullet \Delta_1;\ S;\ H\rangle$. As $\Delta_2 \subseteq \Delta_1$, the state extends the previous state. Furthermore, by lemma B.16, the state is well-formed.

E_Lam, E_Lam$_\#$, E_IntLit, E_Forget, E_Forget$_\#$

Cases impossible, as these do not step. $\square$

Appendix C

Eventual correctness

C.1 Observational equivalence

For our eventual correctness we do not use a decode step, but instead define an observational equivalence relation $\cong$ that relates observationally equivalent $\mathcal{L}$ and $\mathcal{M}$ values. For both $\mathcal{L}$ and $\mathcal{M}$, the only values that can be observed are integers i.

We do not need a full definition of observational equivalence [9] for our proof. Instead, we leave its definition abstract, and assume the following (in our opinion reasonable) property:

Assumption 7.1 (Compiled integers are observationally equivalent)**.** *For any $\mathcal{L}$ state $\langle \Gamma;\ E;\ i_{\mathcal{L}} \rangle$, if $\Gamma \vdash v : \tau$ and $\Gamma \vdash E$, then $[\![\langle \Gamma;\ E;\ v \rangle]\!]^{\emptyset} = \langle i_{\mathcal{M}};\ \Delta;\ \emptyset;\ H \rangle$, and $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.*

C.2 Lemmas

Lemma 7.4 (Equivalent states step to equivalent states)**.** *Let $Q_1 = \langle t_1;\ \Delta_1;\ S_1;\ H_1 \rangle$, $Q_1' = \langle t_1';\ \Delta_1';\ S_1';\ H_1' \rangle$, $Q_2 = \langle t_2;\ \Delta_2;\ S_2;\ H_2 \rangle$, and $Q_2' = \langle t_2';\ \Delta_2';\ S_2';\ H_2' \rangle$.*

If $Q_1 \sqsubseteq Q_1'$, Q_1 WF, Q_1' WF, and $Q_1' \xrightarrow{*} Q_2'$, then there exists some Q_2 such that $Q_1 \xrightarrow{*} Q_2$ and $Q_2 \sqsubseteq Q_2'$.

Proof. Straightforward induction from the definition of state well-formedness (definition B.6) and state extension (definition B.9). ☐

C.3 Eventual correctness

Theorem C.1 (Open eventual correctness)**.** *For all $\langle \Gamma;\ E;\ e \rangle$, $\langle \Gamma';\ E';\ i_{\mathcal{L}} \rangle$, and S, if $\langle \Gamma;\ E;\ e \rangle \xrightarrow{*} \langle \Gamma';\ E';\ v \rangle$, $[\![\langle \Gamma;\ E;\ i_{\mathcal{L}} \rangle]\!]^{S} = \langle t;\ \Delta;\ S;\ H \rangle$, and $H \vdash S$ WF, then there exists a $\langle i_{\mathcal{M}};\ \Delta';\ \emptyset;\ H' \rangle$ such that $\langle t;\ \Delta;\ S;\ H \rangle \xrightarrow{*} \langle i_{\mathcal{M}};\ \Delta';\ S';\ H' \rangle$ and $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.*

Proof. By induction on the length of the derivation $\langle \Gamma;\ E;\ e \rangle \xrightarrow{*} \langle \Gamma';\ E';\ i_{\mathcal{L}} \rangle$.

Case $\langle \Gamma;\ E;\ e\rangle = \langle \Gamma';\ E';\ i_{\mathcal{L}}\rangle$

As $\langle \Gamma;\ E;\ e\rangle = \langle \Gamma';\ E';\ i_{\mathcal{L}}\rangle$, $[\![\langle \Gamma;\ E;\ e\rangle]\!]^{\emptyset} = [\![\langle \Gamma';\ E';\ i_{\mathcal{L}}\rangle]\!]^{\emptyset} = \langle i_{\mathcal{M}};\ \Delta;\ S;\ H\rangle$. By assumption 7.1, $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.

Case $\langle \Gamma_1, E_1, e_1\rangle \longrightarrow (\langle \Gamma_1';\ E_1';\ e_1'\rangle \xrightarrow{*} \langle \Gamma_2;\ E_2;\ i_{\mathcal{L}}\rangle)$

Let $Q_1 = [\![\langle \Gamma_1;\ E;\ e_1\rangle]\!]^{S} = \langle t_1;\ \Delta_1;\ S;\ H_1\rangle$ and $Q_1' = [\![\langle \Gamma_1';\ E';\ e_1'\rangle]\!]^{S} = \langle t_1';\ \Delta_1';\ S;\ H_1'\rangle$.

By theorem 7.3, there exist an $Q_2 = \langle t_2;\ \Delta_2;\ S_2;\ H_2\rangle$ and $Q_2' = \langle t_2';\ \Delta_2';\ S_2';\ H_2'\rangle$ such that $Q_1 \longrightarrow^{*} Q_2$, $Q_1' \longrightarrow^{*} Q_2'$, $Q_2 \sqsubseteq Q_2'$, Q_2 WF, and Q_2' WF.

Furthermore, by induction we know that there exists a state $Q_i' = \langle i_{\mathcal{M}};\ \Delta_3';\ \emptyset;\ H_3'\rangle$ such that $Q_1' \xrightarrow{*} Q_i'$ and $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.

As $Q_1' \longrightarrow^{*} Q_2'$, $Q_1' \xrightarrow{*} Q_i'$ and $\xrightarrow{*}$ for $\mathcal{M}$ is deterministic, it follows that either $Q_1' \xrightarrow{*} (Q_2' \overset{*}{\to} Q_i')$ or $Q_1' \xrightarrow{*} (Q_i' \overset{*}{\to} Q_2')$. We case on these possibilities.

Case $Q_1' \xrightarrow{*} (Q_2' \overset{*}{\to} Q_i')$

As $Q_1 \sqsubseteq Q_1'$ and $Q_1' \xrightarrow{*} Q_i'$ by lemma 7.4 there exists some $Q_i = \langle i_{\mathcal{M}};\ \Delta_3;\ \emptyset;\ H_3\rangle$ such that $Q_1 \xrightarrow{*} Q_i$, which means we are done.

Case $Q_1' \xrightarrow{*} (Q_w \overset{*}{\to} Q_2')$

Q_i' cannot step further, as its work item is $i_{\mathcal{M}}$, and its stack is empty. Therefore, $Q_i' = Q_2'$. As $Q_1 \xrightarrow{*} Q_2$ and $Q_2 \sqsubseteq Q_2'$, we have that Q_2 is of form $\langle i_{\mathcal{M}};\ \Delta_2;\ S_2;\ H_2\rangle$, which means we are done. □

Theorem 7.2 (Eventual correctness). *If* $\langle \emptyset;\ \emptyset;\ e\rangle \xrightarrow{*} \langle \Gamma;\ E;\ i_{\mathcal{L}}\rangle$ *and* $[\![\langle \emptyset;\ \emptyset;\ e\rangle]\!]^{\emptyset} = \langle t;\ \emptyset;\ \emptyset;\ \emptyset\rangle$, *then there exists a* $\langle i_{\mathcal{M}};\ \Delta;\ \emptyset;\ H\rangle$ *such that* $\langle t;\ \emptyset;\ \emptyset;\ \emptyset\rangle \xrightarrow{*} \langle i_{\mathcal{M}};\ \Delta;\ S;\ H\rangle$ *and* $i_{\mathcal{L}} \cong i_{\mathcal{M}}$.

Proof. Corollary from theorem C.1 and the fact that the empty stack $\emptyset$ is well-formed w.r.t. any heap. That is, for any H, $H \vdash \emptyset$ WF. □

Appendix D

Further attachments

D.1 C code

```c
int add3(int a, int b, int c) {
  int x = a + (b + c);
  return(x);
}
```

```
add3(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     DWORD PTR [rbp-28], edx
        mov     edx, DWORD PTR [rbp-24]
        mov     eax, DWORD PTR [rbp-28]
        add     edx, eax
        mov     eax, DWORD PTR [rbp-20]
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```