Typed Closure Conversion with Sum Types for Analyzing Higher-Order Functions in Resource Aware ML

Lauren Sands

CMU-CS-25-134

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jan Hoffmann, Chair
Stephanie Balzer

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Higher-Order Functions, Static Analysis, Logical Relations, Closure Conversion, Resource Aware ML

Abstract

Resource Aware ML (RaML) is a tool that statically infers resource bounds for OCaml programs. However, RaML cannot derive bounds for functions whose cost depends on values that are not arguments. Partially-applied higher-order functions, which are common in OCaml code, often cannot be analyzed for this reason. This thesis introduces a closure conversion transformation that rewrites higher-order programs into ones analyzable by RaML. Unlike traditional closure conversion which wraps functions with environments represented using existential types, this algorithm uses sum types, which allows RaML to access internal types and infer bounds. The transformation is well-typed and preserves semantics and cost, ensuring that analysis of the transformed program remains sound with respect to the original code.

Acknowledgments

First, I have to thank my wonderful advisor, Jan Hoffmann. You introduced me to the world of PL in 15-312 and Compilers, and have continued to teach me valuable lessons—PL related and not—throughout the past two years. I could not have made it through this program without your support, encouragement, and guidance.

Thank you to David Kahn for guiding me when I was new to research. Harrison Grodin for fostering my interest in PL. Stephanie Balzer for your feedback and serving on my committee. Frank Pfenning, for your amazing HOT Compilation class and explanations of closure conversion. Amal Ahmed, for your help with our soundness proof. Matias Scharager, for your insights about logical relations. Mark Stehlik, for your willingness to listen no matter how busy you were, and your endless support when I wanted to give up. And thank you to all my amazing teachers throughout my life. There's too many to list, but some that deserve special mentions: David Kosbie, Daniel Anderson, Michael Caines, Scott Galson, Kailen Lee, Molly Spooner, Sara Folger, Jeff Edmonds, and Rachel Hill.

To my friends and family: Mom, Dad, Emily, Mana, Austin, Liv, Jason, and more that can't fit on the page, thank you for your never-ending love and support. Finally, I have to thank my grandma: I'm proud to be the second woman in the family with a Master of Science. When she sailed to the US in 1959, she joined a generation of women who paved the way for myself and countless others. She has been a role model to me for as long as I can remember.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overview of why Resource Aware ML cannot analyze many higher-order functions, and discusses multiple existing methods of translating higher-order programs to remove this problem. It summarizes the translation developed in this thesis, and provides an outline of what is covered in the remaining chapters.

1.1 Background

Resource Aware ML (RaML) is a tool that statically infers resource bounds for OCaml programs. For example, RaML could analyze the code in Listing 1 and derive the bound shown in Figure 1.1. Note that `Raml.tick n` denotes using `n` units of a resource.

```
let rec append (l1, l2) =
  match l1 with
  | []      -> l2
  | x::xs ->
    Raml.tick 1.0;
    x :: (append (xs, l2))
```

Listing 1: An append function

The current implementation of RaML cannot derive bounds when a function's cost depends on free variables. This issue commonly presents itself when RaML attempts to analyze partially-applied higher-order functions (HOFs).

```
Simplified bound:
 1*M
where
 M is the number of ::-nodes of the 1st component of the argument
```

Figure 1.1: RaML output for append function in Listing 1

For example, say `append l1 l2` appends `l1` to `l2` with a cost proportional to the length of `l1`. Then the function `f` in the code below will not be analyzable by RaML as it's cost depends on the length of `[1;2;3]`, which is not one of its arguments.

```
1 let f = append [1;2;3]
```

As higher-order functions are commonly used in OCaml programs, this thesis presents a code transformation which allows such programs to be analyzed by RaML.

1.2 Defunctionalization vs. Closure Conversion

Transforming higher-order programs has been studied in compiler design, as low-level compiler target languages often do not support higher-order functions. There are two primary methods: defunctionalization, and closure conversion.

Defunctionalization has been introduced by Reynolds [16] and later improved by Bell et al. to preserve typability [5]. Defunctionalization almost entirely eliminates functions, replacing them with data types and a single apply function. This approach is used in the MLton compiler for SML [8]. However, defunctionalization is not feasible for us because it changes the structure of the program so much that it would make it more difficult to analyze. RaML infers bounds per function, and it would complicate the implementation to have functions translated into data types.

The second option is closure conversion, which was applied to typed target languages by Minamide et al. [14]. Closure conversion eliminates partial applications by translating them into a pair of a code pointer and environment which maps variables to values. Free variables in the function body are transformed into environment lookups.

Traditionally, closure conversion represents the environment using an existential type [14]. However, this hides the internal type from the rest of the program which would pose a problem for RaML's analysis. RaML infers resource bounds by constructing linear constraints on potential assigned to base types. Without knowing the internal type, RaML would not be able to generate these constraints.

In this thesis, we present a modified closure conversion algorithm which uses sum types rather than existential types to construct programs amenable to analysis with RaML. The downside is that we have to do a global analysis before running closure conversion. However, the sum types give RaML access to the type information necessary to infer resource bounds.

1.3 Translation Overview

The source and target languages are the same. The language is similar to PCF, except it has lists rather than integers, and bounded recursion rather than a fixed point. These modifications were chosen for similarity to OCaml code, as well as expressibility of common examples using HOFs. The language is complex enough that the primary difficulties in proving soundness appear, but without too many additional features that would lengthen the proofs.

The dynamic semantics are small-step cost semantics which is necessary for the soundness proof. Importantly, the translation must not only preserve the functionality of programs, but also the cost. This ensures that RaML's analysis of the transformed programs will match the bounds on the original program.

Ticks in the code denote resource use. The translation preserves cost by preserving the number of ticks. Functions are translated into pairs where a new function is packaged with an environment which has the global sum type derived from an initial global analysis. Applications unwrap this pair and apply the new function to the argument and stored environment.

1.4 Outline

In this thesis, we define typed closure conversion with sum types, which is an effective tool for anaalyzing HOFs in RaML. We prove soundness of the translation using a logical relation which incorporates cost into the definition of equality so equal programs must evaluate to equal values with the same cost.

The remainder of the document is organized as follows:

- Chapter 2 provides background information about RaML and motivation for why the transformation is necessary. It shows a concrete example of a program that cannot currently be analyzed in RaML and how it can be translated into an analyzable program.
- Chapter 3 defines the source and target language. It presents static semantics and cost based dynamic semantics.
- Chapter 4 formalizes the translation. It describes how the global sum type for the environments is generated, then presents the translation of types, contexts, and expressions. It also proves that translated programs are well-typed.
- Chapter 5 presents the soundness proof. First, the definition of divergence is formalized.

Then, a logical relation is presented which says that two programs are related evaluate to equal values with the same cost. Finally, this logical relation is used to prove soundness of the translation.

- Chapter 6 compares our approach to related work in both closure conversion and soundness proof techniques.
- Chapter 7 summarizes the results and proposes future work.

Chapter 2

Motivation

This chapter provides an example of a program not currently analyzable by RaML and a translated version that can be analyzed. While the translation is not identical in structure to the formalization in Chapter 4, it is very close and provides intuition for our goal.

2.1 RaML

Automatic Amortized Resource Analysis (AARA) is a type-based technique to statically infer resource bounds. Base types are annotated with potential functions which indicate how much of a resource is available. Potential is used when a cost is incurred and gained when resources are returned. The potential assigned to the types can be imagined as vectors representing the linear, quadratic, cubic, etc. potential up to some maximum degree. In this way, AARA can represent non-linear bounds using a linear number of constraints.

RaML is an implementation of the AARA type system for OCaml programs. RaML determines potential annotations by solving a linear program (LP) on the set of constraints generated during type inference. An LP solver aims to minimize the initial potential subject to these constraints. Not all programs are analyzable, in which case, the LP will be infeasible. If the LP is feasible, RaML returns the corresponding resource bounds.

RaML has multiple modes to track resource use. In this paper, we use ticks. The expression `Raml.tick n` represents using `n` units of a resource, which therefore consumes `n` units of potential.

2.1.1 Higher-Order Functions in RaML

Currently, RaML cannot effectively generate resource bounds for programs involving partially-applied higher-order functions (HOFs). The reason for this is demonstrated by the implementation of the `append` function in Listing 2. In the sample code, we track the number of recursive

calls made to the `append` function, so we call `Raml.tick 1.0` immediately before the recursive call.

The second function, `useAppend`, demonstrates how `append` can be partially-applied, and the resulting function `f` can be used multiple times. Note that RaML does not limit the number of times a function is called. This increases the ease of writing RaML code, but also limits what can currently be analyzed.

```
1  let rec append l1 =
2    fun l2 ->  (* need 1 potential per element in l1 *)
3      match l1 with
4      | [] -> l2
5      | x::xs ->
6        Raml.tick 1.0;
7        x :: (append xs l2)
8  ;;
9
10 let useAppend l =
11   let f = append l in
12   (f [1], f [2])
13 ;;
```

Listing 2: Non-analyzable append function in RaML

In the code in Listing 2, the list `l1` passed into `append` must have one potential per element in the list. That is, one linear potential. However, consider what happens when we partially apply `append` on line 11. Since `f` can be called an arbitrary number of times, *each of those calls* requires one potential per element of `l`. Now `l` needs potential dependent on the number of times `f` is called. RaML cannot derive a bound in this case since it cannot know how many times `f` will be called.

Due to this issue, RaML has a rule that the cost of a function needs to be expressible as a function of the arguments. If this is not the case, the analysis fails. In the above case, the cost of `f` is dependent not on its argument, but on the length of `l`, which violates this rule. In this case, an error like the one in Figure 2.1 is raised.

```
A bound for useAppend could not be derived.
The linear program is infeasible.
```

Figure 2.1: Example RaML error for non-analyzable code

2.1.2 Transformed HOFs in RaML

In Listing 3, we present an intuitive translation of the code in Listing 2 that can be analyzed by RaML. This is not the exact translation we formalize later in this paper, but is helpful for gaining intuition for how that translation works and why it is analyzable in RaML.

```
let rec append l1 =
  (l1,
   fun (l1, l2) ->
     match l1 with
     | [] -> l2
     | x::xs ->
       Raml.tick 1.0;
       x :: (let xs, f = append xs in f (xs, l2))
  )
;;

let useAppend l =
  let l1, f = append l in
  (f (l1, [1]), f (l1, [2]))
;;
```

Listing 3: Intuitively translated append function analyzable in RaML

Recall the issue with the original append code: when applied to to a single argument, the function returned had a cost dependent on `l1`, which was not one of its arguments. We solve this by returning a pair rather than a function. This pair includes the list `l1` so we have access to it when analyzing the inner function. The new inner function takes a pair `(l1, l2)` rather than just `l2` so its cost now only depends on its arguments.

Whenever `append` is called, we replace it with a let expression. We call append with the first argument, unwrapping the pair returned to get the first argument back, as well as the new function that takes a pair rather than just the second argument. Then we call that new function with the pair containing the first and second argument together. Intuitively, we pass around the first argument so it's always an argument of any function whose cost can depend on it.

Now RaML can analyze the code, even when function `f` is called multiple times in `useAppend`. The output for the analysis of `useAppend` is shown in Figure 2.2. RaML correctly identifies that `useAppend` costs twice the number of elements in the list `l`.

The approach shown above, while more readable in simple examples, cannot generalize to every program. In the example, we only needed to package a single variable `l1` with the function during translating. However, the cost of the inner function could depend on multiple variables that are not arguments. In this case, we need to create an environment containing all those

```
Simplified bound:
  2*M
 where
  M is the number of ::-nodes of the argument
```

Figure 2.2: RaML output for translated useAppend

variables to package with the function.

In the following sections, we introduce our closure conversion algorithm using sum types for the environments. This will allow us to formalize a translation that can transform any code so that it is semantically equivalent, but doesn't fail for the above reason.

2.2 Closure Conversion

Closure conversion is a program transformation often used in compilers. Higher-order functions are packaged into a pair containing an environment and the code, where the environment maps free variables in the code to values. This is exactly what we wish to do to translate our programs so they can be analyzed by RaML, as demonstrated in the previous section. Importantly, Minamide et al. showed that closure conversion could be performed with a typed target language [14]. We need our source language and target language to be typed since AARA relies on the type system to generate resource bounds.

2.2.1 A Unified Environment Representation

When performing closure conversion, it is important that environments for two functions of the same type also have the same type. To see why, consider the following code, where `id` is the identity function and `append` is defined as in Listing 3.

```
1 let f = if ___ then id
2           else append l
```

If we used an intuitive environment representation such as the one in Listing 3, then the types of `id` and `append l` would be:

```
id : unit * ((unit * list) -> list)
append : list * ((list * list) -> list)
```

As we saw previously, `append l` needs a list stored in the environment. However, `id` doesn't need anything in its environment, which therefore has type `unit`. This raises a problem: after translation, `f` is not well-typed. Thus, we need a shared type that can be used to represent the environments for both of these functions. More generally, any two functions with the same type in the original code need to also have the same type (and thus the same environment type) in the translated code.

Traditionally, closure conversion algorithms handle this by using existential types to represent environments [14]. However, we do not want to use existential types because they hide the internal type—we only know what we can use the type for, not what it actually is. However, we need to know the structure of the values captured in the closure for analysis, as they may need to carry potential. Thus, we need a more concrete representation than existential types can give.

2.3 Closure Conversion with Sum Types

Rather than using existential types, we will represent our environments using a sum type. Sum types are used in the conversion performed by the MLton compiler [8]. They create a new sum type for every distinct set of lambdas that could appear at some program point. However, we will differ from this approach by creating a single sum type to represent all possible environments. While the sum type itself may be more complicated, it is a more intuitive representation and simplifies the translation of applications to use a single sum.

Each function in our translated code will have a corresponding case of the sum type whose type is a product of all the free variables in scope when the function is defined. This solves the problem with using existential types by giving RaML access to the actual values inside the environment which allows it to determine resource bounds. The downside to this approach is that we have to begin with a global analysis of the code. However, this is not unprecedented—the MLton compiler for SML also does global analysis before their closure conversion pass [8].

As before, functions are translated into pairs of an environment and new function. This new function takes the environment and argument. The primary difference with the example in Listing 3 is that we immediately match on the environment inside the function to extract the individual variables from within it. We know which case of the sum we should be in because we know which function we are translating, and each function has a unique corresponding case of the sum. In all other cases, we will raise an error: we should never reach this case.

The translation of the code from Listing 2 using sum types is shown in Listing 4. For readability, we did not translate the top-level functions. These do not have any free variables in scope when they are defined, so there is no need to wrap them with an environment as the environment would be empty. While our formal translation does not differentiate top-level functions, this is a simple optimization that would be implemented in practice. For readability, we also have a separate apply function whereas the translation would inline this in practice.

```
type t = Env of int list

let rec append l1 =
  (Env l1,
   fun (env, l2) ->
     match env with
     | Env l1 ->
       match l1 with
       | [] -> l2
       | x::xs ->
         Raml.tick 1.0;
         x :: (let env, f = append xs in f (env, l2))
    | _ -> raise (Failure "Should never reach this case")
  )
;;

let apply (f, arg) =
  let env, g = f in
  g (env, arg)
;;

let useAppend l =
  let f = append l in
  (apply (f, [1]), apply (f, [2]))
;;
```

Listing 4: Translated append function analyzable in RaML

Chapter 3

The Language: Bounded PCF

The source language is a variant of PCF that we call Bounded PCF . Rather than recursion using a fixed point, we have bounded recursive functions. Bounding the recusion is helpful for the soundness proof, and because of the compactness theorem, a recursive function with a large enough bound is equivalent to the unbounded version. This is further explained in Section 5.3.

The language doesn't include lambda functions since anything implemented with a lambda can also be implemented using our recursive functions. We also use lists rather than integers. Many of the examples of non-analyzable HOFs used lists, and these are just as expressive as natural numbers.

We also include product and sum types, which are required in the target language. Products are required since functions are translated into pairs. Sums are required since we use a sum type to represent the environments. While we could omit these from the source language, they are commonly used in simple examples, so we include them. This allows our source and target languages to be the same, which also simplifies our proofs. This language is just complex enough that all the main difficulties in the proofs become apparent, without adding too many extra cases.

3.1 Syntax

The full syntax for Bounded PCF is shown in Figure 3.1. We use labeled sums and products. $\langle\, e_1 \,,\, e_2 \,\rangle$ is shorthand for $\langle\, \ell \hookrightarrow e_1 \,,\, r \hookrightarrow e_2 \,\rangle$, and $\tau_1 * \tau_2$ is shorthand for $\langle\, \ell : \tau_1 \,,\, r : \tau_2 \,\rangle$.

Ticks, written in code as `Raml.tick n`, are denoted by

$$\mathbf{tick}(n)$$

Bounded recursive functions have the following syntax:

$$\mathbf{fun}^m \, \{\tau_1, \tau_2\} (f \,.\, x \,.\, e)$$

This represents a function with name f of type $\tau_1 \to \tau_2$ that can make a maximum of m recursive calls. Each recursive call decrements m, which is shown in the dynamic semantics in Figure 3.4.

Typ	τ	::=	$\mathbf{unit}$	
			$\tau\ \mathbf{list}$	
			$\langle\, \ell : \tau_\ell \,\rangle_{\ell \in L}$	
			$\tau \to \tau$	
			$[\, \ell : \tau_\ell \,]_{\ell \in L}$	
Exp	e	::=	x	
			$(\,)$	
			$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	
			$\mathbf{tick}(n)$	(consume resources)
			$[\,]\{\tau\}$	(list introduction)
			$e_1 :: e_2$	(list introduction)
			$\mathbf{match}\ e\ \{\, [\,]\{\tau\} \hookrightarrow e_1 \,;\, x :: xs \hookrightarrow e_2 \,\}$	(list elimination)
			$\langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L}$	(product introduction)
			$e.\ell$	(product elimination)
			$\mathbf{fun}^m\, \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)$	(bounded recursion introduction)
			$e_1(e_2)$	(bounded recursion elimination)
			$\mathbf{in}[\ell]\{\tau\}\,(e)$	(sum introduction)
			$\mathbf{case}\ e\ \{\, \ell(x_\ell) \hookrightarrow e_\ell \,\}_{\ell \in L}$	(sum elimination)

Figure 3.1: Syntax for Bounded PCF

3.2 Static Semantics

The type rules for Bounded PCF use the standard typing judgement

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

which states that e has type τ under context Γ. Note that ticks have type $\mathbf{unit}$:

$$\frac{}{\Gamma \vdash \mathbf{tick}(n) : \mathbf{unit}}\ \text{S-TICK}$$

Bounded recursive functions are typed as you would expect for general recursive functions:

$$\frac{\Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}^m\, \{\tau_1, \tau_2\}(f\,.\,x\,.\,e) : \tau_1 \to \tau_2}\ \text{S-FUN}$$

The full set of type rules are shown in Figure 3.2.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{S-VAR} \qquad \frac{}{\Gamma \vdash () : \mathbf{unit}}\ \text{S-UNIT}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \text{S-LET} \qquad \frac{}{\Gamma \vdash \mathbf{tick}(n) : \mathbf{unit}}\ \text{S-TICK}$$

$$\frac{}{\Gamma \vdash [\,]\{\tau\} : \tau\ \mathbf{list}}\ \text{S-NIL} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathbf{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathbf{list}}\ \text{S-CONS}$$

$$\frac{\Gamma \vdash e : \tau_1\ \mathbf{list} \qquad \Gamma \vdash e_1 : \tau_2 \qquad \Gamma, x : \tau_1, xs : \tau_1\ \mathbf{list} \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\} : \tau_2}\ \text{S-MATCH}$$

$$\frac{\Gamma \vdash e_\ell : \tau_\ell \quad \forall \ell \in L}{\Gamma \vdash \langle \ell \hookrightarrow e_\ell \rangle_{\ell \in L} : \langle \ell : \tau_\ell \rangle_{\ell \in L}}\ \text{S-PROD} \qquad \frac{\Gamma \vdash e : \langle \ell : \tau_\ell \rangle_{\ell \in L}}{\Gamma \vdash e.\ell : \tau_\ell}\ \text{S-PROJ}$$

$$\frac{\Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}^m\,\{\tau_1, \tau_2\}(f\,.\,x\,.\,e) : \tau_1 \to \tau_2}\ \text{S-FUN} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ \text{S-APP}$$

$$\frac{\Gamma \vdash e : \tau_\ell}{\Gamma \vdash \mathbf{in}[\ell]\{[\,\ell : \tau_\ell\,]_{\ell \in L}\}\,(e) : [\,\ell : \tau_\ell\,]_{\ell \in L}}\ \text{S-INJ}$$

$$\frac{\Gamma \vdash e : [\,\ell : \tau_\ell\,]_{\ell \in L} \qquad \Gamma, x_\ell : \tau_\ell \vdash e_\ell : \tau \quad \forall \ell \in L}{\Gamma \vdash \mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L} : \tau}\ \text{S-CASE}$$

Figure 3.2: Static semantics for Bounded PCF

3.3 Dynamic Semantics

In this section we define the values and small-step semantics for Bounded PCF . While RaML dynamics are traditionally written using big-step semantics, we need small-step for the soundness proof presented in Section 5.2.

3.3.1 Values

The judgement

$$e\ \mathbf{val}$$

states that e is a value in Bounded PCF . The values are shown in Figure 3.3. Note that lists, products, and sums are only values if all the expressions contained in them are also values.

$$\frac{}{(\,)\ \mathbf{val}}\ \text{V-UNIT} \qquad \frac{}{[\,]\{\tau\}\ \mathbf{val}}\ \text{V-NIL} \qquad \frac{v_1\ \mathbf{val} \qquad v_2\ \mathbf{val}}{v_1 :: v_2\ \mathbf{val}}\ \text{V-CONS}$$

$$\frac{v_\ell\ \mathbf{val} \quad \forall \ell \in L}{\langle\, \ell \hookrightarrow v_\ell \,\rangle_{\ell \in L}\ \mathbf{val}}\ \text{V-PROD} \qquad \frac{}{\mathbf{fun}^m\, \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)\ \mathbf{val}}\ \text{V-FUN} \qquad \frac{v\ \mathbf{val}}{\mathbf{in}[\ell]\{\tau\}\,(v)\ \mathbf{val}}\ \text{V-INJ}$$

Figure 3.3: Values in Bounded PCF

3.3.2 Small-Step Cost Semantics

The dynamic semantics are defined using the judgement

$$e \longmapsto_n e'$$

which states that e steps to e' with cost n. The non-standard feature is that this tracks cost. We use cost semantics because for our transformation to be sound, it not only must preserve the semantics of the program, but also the cost. Transforming a function so that it outputs the same value with a different cost would not be acceptable, as RaML's analysis would be incorrect.

The only expression that incurs cost when it steps is the tick, which immediately steps to a unit with the cost indicated:

$$\frac{}{\mathbf{tick}(n) \longmapsto_n (\,)}\ \text{D-TICK}$$

Since we use the ticks mode in RaML, ticks are the only way to consume resources. Thus, cost is preserved in all other rules, or 0 if the rule does not have a premise.

Note that we assume labeled products have an order. The dynamics evaluate the expressions from left to right. This is denoted in the rule by numbering the labels from 1 to k.

$$\frac{e_j\ \mathbf{val} \quad \forall j \in [1, i-1] \qquad e_i \longmapsto_n e_i'}{\langle\, \ell_1 \hookrightarrow e_1, \ldots, \ell_i \hookrightarrow e_i, \ldots, \ell_k \hookrightarrow e_k \,\rangle \longmapsto_n \langle\, \ell_1 \hookrightarrow e_1, \ldots, \ell_i \hookrightarrow e_i', \ldots, \ell_k \hookrightarrow e_k \,\rangle}\ \text{D-PROD}$$

The other interesting rule is function application to a value:

$$\frac{m > 0}{\mathbf{fun}^m\, \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)(v) \longmapsto_0 [v/x, \mathbf{fun}^{m-1}\, \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)/f]e}\ \text{D-APP3}$$

m denotes how many recursive calls can be made to the function. We substitute the entire function in for f in the body as usual, but decrement m by 1 to indicate that one fewer recursive call can now be made. Also, note that the premise requires $m > 0$. There are no rules for how to step the application if $m = 0$. This represents non-termination, which is defined in Section 3.3.4.

The other rules are standard with costs added. The full dynamic semantics are in Figure 3.4.

$$\frac{e_1 \longmapsto_n e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \longmapsto_n \textbf{let } x = e_1' \textbf{ in } e_2} \text{ D-Let1} \qquad \frac{v \textbf{ val}}{\textbf{let } x = v \textbf{ in } e_2 \longmapsto_0 [v/x]e_2} \text{ D-Let2}$$

$$\frac{}{\textbf{tick}(n) \longmapsto_n ()} \text{ D-Tick} \qquad \frac{e_1 \longmapsto_n e_1'}{e_1 :: e_2 \longmapsto_n e_1' :: e_2} \text{ D-Cons1} \qquad \frac{v_1 \textbf{ val} \qquad e_2 \longmapsto_n e_2'}{v_1 :: e_2 \longmapsto_n v_1 :: e_2'} \text{ D-Cons2}$$

$$\frac{e \longmapsto_n e'}{\textbf{match } e \{ [\,]\{\tau\} \hookrightarrow e_1 ; x :: xs \hookrightarrow e_2 \} \longmapsto_n \textbf{match } e' \{ [\,]\{\tau\} \hookrightarrow e_1 ; x :: xs \hookrightarrow e_2 \}} \text{ D-Match}$$

$$\frac{}{\textbf{match } [\,]\{\tau\} \{ [\,]\{\tau\} \hookrightarrow e_1 ; x :: xs \hookrightarrow e_2 \} \longmapsto_0 e_1} \text{ D-MatchNil}$$

$$\frac{v_1 \textbf{ val} \qquad v_2 \textbf{ val}}{\textbf{match } v_1 :: v_2 \{ [\,]\{\tau\} \hookrightarrow e_1 ; x :: xs \hookrightarrow e_2 \} \longmapsto_0 [v_1/x, v_2/xs]e_2} \text{ D-MatchCons}$$

$$\frac{e_j \textbf{ val} \quad \forall j \in [1, i-1] \qquad e_i \longmapsto_n e_i'}{\langle \ell_1 \hookrightarrow e_1, \ldots, \ell_i \hookrightarrow e_i, \ldots, \ell_k \hookrightarrow e_k \rangle \longmapsto_n \langle \ell_1 \hookrightarrow e_1, \ldots, \ell_i \hookrightarrow e_i', \ldots, \ell_k \hookrightarrow e_k \rangle} \text{ D-Prod}$$

$$\frac{e \longmapsto_n e'}{e.\ell \longmapsto_n e'.\ell} \text{ D-Proj1} \qquad \frac{}{(\langle \ell \hookrightarrow e_\ell \rangle_{\ell \in L}).k \longmapsto_0 e_k} \text{ D-Proj2}$$

$$\frac{e_1 \longmapsto_n e_1'}{e_1(e_2) \longmapsto_n e_1'(e_2)} \text{ D-App1} \qquad \frac{v \textbf{ val} \qquad e_2 \longmapsto_n e_2'}{v(e_2) \longmapsto_n v(e_2')} \text{ D-App2}$$

$$\frac{m > 0 \qquad v \textbf{ val}}{\textbf{fun}^m \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)(v) \longmapsto_0 [v/x, \textbf{fun}^{m-1} \{\tau_1, \tau_2\}(f\,.\,x\,.\,e)/f]e} \text{ D-App3}$$

$$\frac{e \longmapsto_n e'}{\textbf{in}[\ell]\{\tau\}(e) \longmapsto_n \textbf{in}[\ell]\{\tau\}(e')} \text{ D-Inj}$$

$$\frac{e \longmapsto_n e'}{\textbf{case } e \{ \ell(x_\ell) \hookrightarrow e_\ell \}_{\ell \in L} \longmapsto_n \textbf{case } e' \{ \ell(x_\ell) \hookrightarrow e_\ell \}_{\ell \in L}} \text{ D-Case1}$$

$$\frac{v \textbf{ val}}{\textbf{case } (\textbf{in}[k]\{\tau\}(v)) \{ \ell(x_\ell) \hookrightarrow e_\ell \}_{\ell \in L} \longmapsto_0 [v/x_k]e_k} \text{ D-Case2}$$

Figure 3.4: Dynamic semantics for Bounded PCF

3.3.3 Multi-Step Definition

The notation

$$e \longmapsto_n^* e'$$

means that e multi-steps to e', and is formally defined by:

$$e \longmapsto_0^* e$$

$$\text{If } e \longmapsto_{n_1} e', \text{ and } e' \longmapsto_{n_2}^* e'', \text{ then } e \longmapsto_{n_1+n_2}^* e''$$

We will use this notation in the soundness proof in Section 5.2.

3.3.4 Divergence

Since we can have non-terminating programs, we must define what it means for a program to diverge. The judgement

$$e \longmapsto_\infty^* \bot$$

states that e does not terminate. Intuitively, this means the program can no longer step to anything and also has not reached a value.

The only expression that immediately diverges is an application of $\mathbf{fun}^0\,\{\tau_1, \tau_2\}(f\,.\,x\,.\,e)$: the function abstraction with 0 unrollings allowed. This is the only expression that cannot step to anything based on the dynamics. Naturally, anything that steps to a diverging program also diverges.

$$\frac{}{(\mathbf{fun}^0\,\{\tau_1, \tau_2\}(f\,.\,x\,.\,e_1))(e_2) \longmapsto_\infty^* \bot}\ \textsc{Div-Fun} \qquad \frac{e \longmapsto_n^* e' \qquad e' \longmapsto_\infty^* \bot}{e \longmapsto_\infty^* \bot}\ \textsc{Div-Step}$$

Figure 3.5: Divergence base case and step

We also need to capture that an expression with a diverging subexpression diverges. This is done with the full set of divergence rules are shown in Figure 3.6.

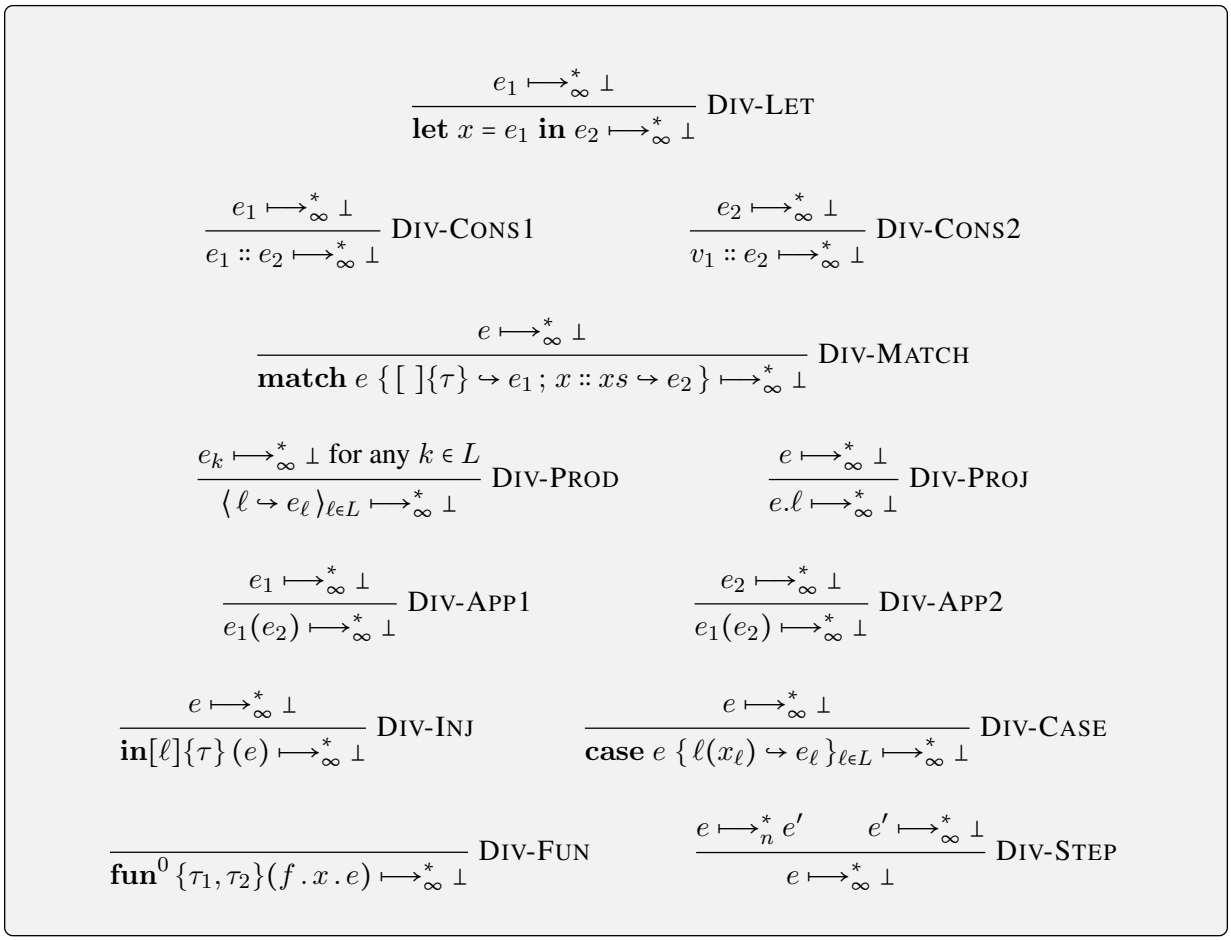

Figure 3.6: Divergence of programs

3.4 Type Soundness

Lemma 1 (Progress). If $\Gamma \vdash e : \tau$ then either e **val**, $e \longmapsto_n e'$, or $e \longmapsto^*_\infty \bot$.

This can be proven by induction on the type judgement. For each rule that is not a base case, we case on the result given by the inductive hypothesis. As usual, we consider if the subexpressions are values or step to another expression. Additionally, we consider the case where a subexpression diverges, in which case the entire expression will also diverge by the rules in Figure 3.6. Thus, in each case, we can either claim the overall expression is a value, step it, or say that it diverges.

Lemma 2 (Preservation). If $\Gamma \vdash e : \tau$ and $e \longmapsto_n e'$ then $\Gamma \vdash e' : \tau$.

This can be proven by induction on the step judgement. For each rule that is not a base case, the inductive hypothesis tells us that a subexpression steps to something of the same type. From there, we can reconstruct the type of the overall stepped expression, which will be preserved.

Chapter 4

The Translation

The translation is defined using the judgement:

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

which states that an expression e of type τ under context Γ translates to the expression e'.

In Section 4.3, we will show that this translation preserves types. That is, under the translated context Γ', the expression e' has the translated type τ'.

4.1 Global Sum Type

A sum type T is used to store function environments in the translated programs. To generate this type, we begin by enumerating all functions in the program from 1 to n, where n is the number of function definitions. Going forward, function names will be subscripted with their index for clarity.

To determine the variables in scope for a particular function definition, we typecheck the program. When the typechecker reaches a function abstraction, the variables in the context are the variables in scope that must be stored in the environment. For function i, say $\Gamma = \{ z_1 : \sigma_1, \ldots, z_{n_i} : \sigma_{n_i} \}$. Then label i of the sum type T should map to a product type with an element for each of these variables. Each variable should have its translated type (denoted σ'_j), where the translation of types follows the rules in Section 4.2.1. In this case:

$$\langle j : \sigma'_j \rangle_{j \in [n_i]}$$

Note that order matters. If z_1 is the first element in the context when we generate type T, it must also be the first element when we create the function's environment in the translation pass. For this reason, we consider the contexts to be ordered, and always add new elements to the end of the context.

Putting this all together, we generate the type:

$$T = [\, i : \langle\, j : \sigma'_j \,\rangle_{j \in [n_i]} \,]_{i \in [n]}$$

We assume this type is global and always accessible during translation.

4.2 Translation Definition

This section defines how types, contexts, and expressions are translated. Note that we need to consider both open and closed expressions when defining the translation.

4.2.1 Translation of Types

Using the definition of type T from Section 4.1, the translation of types is defined in Figure 4.1. The interesting case is the function case. A function with type $\tau_1 \to \tau_2$ is translated into a pair of type $T * (T * \tau'_1 \to \tau'_2)$. This pair contains the environment and the new function. The new function takes a pair containing the environment of type T, and an argument of the translated argument type τ_1. It returns a value of the translated return type τ_2.

$$\frac{}{\mathbf{unit} \rightsquigarrow \mathbf{unit}}\ \text{T-UNIT} \qquad \frac{\tau \rightsquigarrow \tau'}{\tau\ \mathbf{list} \rightsquigarrow \tau'\ \mathbf{list}}\ \text{T-LIST} \qquad \frac{\tau_\ell \rightsquigarrow \tau'_\ell \quad \forall \ell \in L}{\langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow \langle\, \ell : \tau'_\ell \,\rangle_{\ell \in L}}\ \text{T-PROD}$$

$$\frac{\tau_1 \rightsquigarrow \tau'_1 \qquad \tau_2 \rightsquigarrow \tau'_2}{\tau_1 \to \tau_2 \rightsquigarrow T * ((T * \tau'_1) \to \tau'_2)}\ \text{T-FUN} \qquad \frac{\tau_\ell \rightsquigarrow \tau'_\ell \quad \forall \ell \in L}{[\, \ell : \tau_\ell \,]_{\ell \in L} \rightsquigarrow [\, \ell : \tau'_\ell \,]_{\ell \in L}}\ \text{T-SUM}$$

Figure 4.1: Translation of types

4.2.2 Translation of Contexts

Contexts are translated pointwise:

$$\Gamma \rightsquigarrow \Gamma' \text{ if and only if } \Gamma' = \{x : \tau' \mid x : \tau \in \Gamma \wedge \tau \rightsquigarrow \tau'\}$$

4.2.3 Translation of Expressions

We begin with the translation of functions. As in Section 4.1, let

$$T = [\, i : \langle\, j : \sigma'_j \,\rangle_{j \in [n_i]} \,]_{i \in [n]}$$

where n is the number of functions in the program, and n_i is the number of local variables when function i is defined. The full rule is shown in Figure 4.2.

$$\frac{\begin{array}{c}\Gamma = \{\, z_1 : \sigma_1,\, \ldots,\, z_{n_i} : \sigma_{n_i} \,\} \\ \Gamma, x : \tau_1, f_i : \tau_1 \to \tau_2 \vdash e : \tau_2 \rightsquigarrow e' \qquad f,\, y,\, y_1,\, y_2,\, z \;\text{ fresh}\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{fun}^m\,\{\tau_1, \tau_2\}(f_i\,.\,x\,.\,e) : \tau_1 \to \tau_2 \rightsquigarrow \langle\, \mathbf{in}[i]\{T\}\,(\langle\, j \hookrightarrow z_j \,\rangle_{j \in [n_i]})\,, \\ \qquad\qquad \mathbf{fun}^m\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,. \\ \qquad\quad \mathbf{let}\ f_i = \langle\, \mathbf{in}[i]\{T\}\,(\langle\, j \hookrightarrow z_j \,\rangle_{j \in [n_i]})\,, f \,\rangle\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ y_1 = y.l\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ y_2 = y.r\ \mathbf{in} \\ \qquad\quad \mathbf{case}\ y_1\ \{ \\ \qquad\qquad \mid i(z) \hookrightarrow \\ \qquad\qquad\quad \mathbf{let}\ z_1 = z.1\ \mathbf{in} \\ \qquad\qquad\quad \ldots \\ \qquad\qquad\quad \mathbf{let}\ z_{n_i} = z.n_i\ \mathbf{in}\ [y_2/x]e' \\ \qquad\qquad \mid \_ \hookrightarrow (\mathbf{fun}^0\,\{\mathbf{unit}, \tau_2'\}(f\,.\,x\,.\,e))()\,\})\,\rangle\end{array}}\ \textsc{E-Fun}$$

Figure 4.2: Translation of functions

A function translates to a pair that wraps the environment with a new function. The environment has type T and consists of everything in the context at this point translated. By the construction of the sum type T, we know that function f_i corresponds to label i in the sum. Thus, with context $\Gamma = \{\, z_1 : \sigma_1,\, \ldots,\, z_{n_i} : \sigma_{n_i} \,\}$, the environment is:

$$\mathbf{in}[i]\{T\}\,(\langle\, j \hookrightarrow z_j' \,\rangle_{j \in [n_i]})$$

where we assume $\Gamma \vdash z_j : z_j' \rightsquigarrow \sigma_j$ for all $j \in [n_i]$.

The new function takes both an environment and a translated argument as a pair. The second and third lines of the function unwrap the pair into variables y_1 for the environment and y_2 for the argument. Line one of the function sets f_i to the pair containing the environment and the function f. This ensures the translated program typechecks. f is a function with type $T * \tau_1' \to \tau_2'$, but recursive references to f_i in the body e' expect type $T * (T * \tau_1' \to \tau_2')$, which matches the translation of the overall function.

Lines six through eight of the function extract the variables from the environment and perform the necessary substitutions. We know y_1 should have type T and be an injection with label i since we are translating function i. If this is not the case, we error by evaluating to an immediately diverging computation. In the label i case, given that the variables in the context were $z_1, \ldots, z_{n_i}$,

we extract the values from the environment into these same variables in the same order. Thus, the usages of these variables within e' will use the correct values. Finally, we substitute the argument variable y_1 in for the original argument x in the translated body.

We now describe the translation of applications. Translated functions include their environment and a new function that takes an environment and argument. Thus, we translate applications by extracting the environment from the translated function, and passing this as well as the translated argument into the new function. The rule is shown in Figure 4.3.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2' \qquad x_1, x_2 \text{ fresh}}{\Gamma \vdash e_1(e_2) : \tau_2 \rightsquigarrow \textbf{let } x_1 = e_1'.l \textbf{ in let } x_2 = e_1'.r \textbf{ in } x_2(\langle x_1 , e_2' \rangle)} \text{ E-App}$$

Figure 4.3: Translation of applications

The remaining expressions do not change structure upon translation. The full set of translation rules is shown in figure Figure 4.4.

$$\frac{}{\Gamma \vdash x : \tau \rightsquigarrow x} \text{ E-Var} \qquad \frac{}{\Gamma \vdash ( ) : \textbf{unit} \rightsquigarrow ( )} \text{ E-Unit}$$

$$\frac{\tau \rightsquigarrow \tau'}{\Gamma \vdash [\,]\{\tau\} : \tau \textbf{ list} \rightsquigarrow [\,]\{\tau'\}} \text{ E-Nil} \qquad \frac{\Gamma \vdash e_1 : \tau \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau \textbf{ list} \rightsquigarrow e_2'}{\Gamma \vdash e_1 :: e_2 : \tau \textbf{ list} \rightsquigarrow e_1' :: e_2'} \text{ E-Cons}$$

$$\frac{\begin{array}{c}\tau \rightsquigarrow \tau' \\ \Gamma \vdash e : \tau_1 \textbf{ list} \rightsquigarrow e' \qquad \Gamma \vdash e_1 : \tau_2 \rightsquigarrow e_1' \qquad \Gamma, x : \tau_1, xs : \tau_1 \textbf{ list} \vdash e_2 : \tau_2 \rightsquigarrow e_2'\end{array}}{\Gamma \vdash \textbf{match } e \,\{ [\,]\{\tau_1\} \hookrightarrow e_1 \,;\, x :: xs \hookrightarrow e_2 \} : \tau_2 \rightsquigarrow \textbf{match } e' \,\{ [\,]\{\tau_1'\} \hookrightarrow e_1' \,;\, x :: xs \hookrightarrow e_2' \}} \text{ E-Match}$$

$$\frac{\Gamma \vdash e_\ell : \tau_\ell \rightsquigarrow e_\ell' \quad \forall \ell \in L}{\Gamma \vdash \langle \ell \hookrightarrow e_\ell \rangle_{\ell \in L} : \langle \ell : \tau_\ell \rangle_{\ell \in L} \rightsquigarrow \langle \ell \hookrightarrow e_\ell' \rangle_{\ell \in L}} \text{ E-Prod} \qquad \frac{\Gamma \vdash e : \langle \ell : \tau_\ell \rangle_{\ell \in L} \rightsquigarrow e'}{\Gamma \vdash e.k : \tau_k \rightsquigarrow e'.k} \text{ E-Proj}$$

$$\frac{\Gamma \vdash e : \tau_k \rightsquigarrow e' \qquad \tau_\ell \rightsquigarrow \tau_\ell' \quad \forall \ell \in L}{\Gamma \vdash \textbf{in}[k]\{[\,\ell : \tau_\ell\,]_{\ell \in L}\}\,(e) : \tau_k \rightsquigarrow \textbf{in}[k]\{[\,\ell : \tau_\ell'\,]_{\ell \in L}\}\,(e')} \text{ E-Inj}$$

$$\frac{\Gamma \vdash e : [\,\ell : \tau_\ell\,]_{\ell \in L} \rightsquigarrow e' \qquad \Gamma, x_\ell : \tau_\ell \vdash e_\ell : \tau \rightsquigarrow e_\ell' \quad \forall i \in L}{\Gamma \vdash \textbf{case } e \,\{\, \ell(x_\ell) \hookrightarrow e_\ell \,\}_{\ell \in L} : \tau \rightsquigarrow \textbf{case } e' \,\{\, \ell(x_\ell) \hookrightarrow e_\ell' \,\}_{\ell \in L}} \text{ E-Case}$$

Figure 4.4: Translation rules

4.3 Transformed Program is Well-Typed

Theorem 1 . If $\Gamma \vdash e : \tau \rightsquigarrow e'$, then $\Gamma' \vdash e' : \tau'$, where $\Gamma \rightsquigarrow \Gamma'$ and $\tau \rightsquigarrow \tau'$.

Proof. We proceed by induction on the judgement $\Gamma \vdash e : \tau \rightsquigarrow e'$.

Case E-Fun: $e = \mathbf{fun}^m \{\tau_1, \tau_2\}(f_i \,.\, x \,.\, e_b)$.

Name	Expression
e_{env}	$\mathbf{in}[i]\{T\}\,(\langle\, j \hookrightarrow z'_j \,\rangle_{j \in [n_i]})$
f^*	$\mathbf{fun}^m \{T * \tau'_1, \tau'_2\}(f \,.\, y \,.\, e_1)$
e_1	$\mathbf{let}\ f_i = \langle\, e_{env} \,,\, f \,\rangle\ \mathbf{in}$ $\mathbf{let}\ y_1 = y.l\ \mathbf{in}$ $\mathbf{let}\ y_2 = y.r\ \mathbf{in}\ e_2$
e_2	$\mathbf{case}\ y_1\ \{\, i(z) \hookrightarrow e_3 \mid \_ \hookrightarrow (\mathbf{fun}^0\,\{\mathbf{unit}, \tau'_2\}(f \,.\, x \,.\, e))() \,\}$
e_3	$\mathbf{let}\ z_1 = z.1\ \mathbf{in}\ \ldots \mathbf{let}\ z_n = z.n_i\ \mathbf{in}\ e_4$
e_4	$[y_2/x]e'_b$

Table 4.1: Expression names for E-Fun case in Theorem 1.

By T-Fun, $\tau_1 \to \tau_2 \rightsquigarrow T * ((T * \tau'_1) \to \tau'_2)$. We know $e' = \langle\, e_{env} \,,\, f^* \,\rangle$, where e_{env} and f^* are given in Table 4.1. We want to show that $\Gamma' \vdash \langle\, e_{env} \,,\, f^* \,\rangle : T * ((T * \tau'_1) \to \tau'_2)$. By S-Prod, this entails proving both of the following:

$$\Gamma \vdash e_{env} : T \tag{4.1}$$

$$\Gamma \vdash f^* : (T * \tau'_1) \to \tau'_2 \tag{4.2}$$

We begin by showing 4.1. By construction of T in Section 4.1, we know that when we reach the abstraction for function i, $\Gamma = \{\, z_1 : \sigma_1, \ldots, z_{n_i} : \sigma_{n_i} \,\}$. By the inductive hypothesis, $\Gamma' \vdash z'_j : \sigma'_j$ for all $j \in [n_i]$. Thus, by S-Prod,

$$\Gamma \vdash \langle\, j \hookrightarrow z'_j \,\rangle_{j \in [n_i]} : \langle\, j : \sigma'_j \,\rangle_{j \in [n_i]}$$

Then by S-Inj,

$$\Gamma \vdash e_{env} : T$$

Now we show 4.2. By S-Fun, we need to show:

$$\Gamma', y : (T * \tau'_1),\ f : ((T * \tau'_1) \to \tau'_2) \vdash e_1 : \tau'_2$$

By S-Let and S-Proj, we need to show:

$$\Gamma', y : (T * \tau_1'), f : ((T * \tau_1') \to \tau_2'), f_i : T * ((T * \tau_1') \to \tau_2'), y_1 : T, y_2 : \tau_1' \vdash e_2 : \tau_2'$$

Since y and f are fresh, we can remove them from the context since we know that they are not used anywhere in e_2. Thus, the above statement is equivalent to:

$$\Gamma', f_i : T * ((T * \tau_1') \to \tau_2'), y_1 : T, y_2 : \tau_1' \vdash e_2 : \tau_2'$$

By S-App, we know that the catch-all case in e_2 has type τ_2'. Thus, to show that e_2 has type τ_2', we need to show:

$$\Gamma', f_i : T * ((T * \tau_1') \to \tau_2'), y_2 : \tau_1', z : \langle\, j : \sigma_j' \,\rangle_{j \in [n_i]} \vdash e_3 : \tau_2'$$

Note that again, we could remove y_1 since it is fresh and does not appear in e_3. By S-Let, we want to show:

$$\Gamma', f_i : T * ((T * \tau_1') \to \tau_2'), y_2 : \tau_1', z_1 : \sigma_1', \ldots, z_{n_i} : \sigma_{n_i}' \vdash e_4 : \tau_2'$$

Since contexts are converted pointwise, we know that $\Gamma' = \{\, z_1 : \sigma_1', \ldots, z_{n_i} : \sigma_{n_i}' \,\}$. Thus, adding the substitutions $z_1 : \sigma_1', \ldots, z_{n_i} : \sigma_{n_i}'$ is redundant, so we can remove them. Additionally, since y_2 is substituted for x in e_b', we want to show:

$$\Gamma', f_i : T * ((T * \tau_1') \to \tau_2'), x : \tau_1' \vdash e_b' : \tau_2'$$

This is exactly what the inductive hypothesis states. Therefore, we have shown that

$$\Gamma \vdash f^* : (T * \tau_1') \to \tau_2'$$

Case $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2' \qquad x_1, x_2 \text{ fresh}}{\Gamma \vdash e_1(e_2) : \tau_2 \rightsquigarrow \textbf{let } x_1 = e_1'.l \textbf{ in let } x_2 = e_1'.r \textbf{ in } x_2(\langle\, x_1 \,,\, e_2' \,\rangle)}$ E-App

We want to show that $\Gamma' \vdash \textbf{let } x_1 = e_1'.l \textbf{ in let } x_2 = e_1'.r \textbf{ in } x_2(\langle\, x_1 \,,\, e_2' \,\rangle) : \tau_2'$. By the inductive hypothesis,

$$\Gamma' \vdash e_1' : T * ((T * \tau_1') \to \tau_2')$$

Then by S-Let and S-Proj, we need to show:

$$\Gamma', x_1 : T, x_2 : (T * \tau_1') \to \tau_2' \vdash x_2(\langle\, x_1 \,,\, e_2' \,\rangle) : \tau_2'$$

By the inductive hypothesis, $\Gamma' \vdash e_2' : \tau_1'$. Thus, by S-Prod,

$$\Gamma', x_1 : T, x_2 : (T * \tau_1') \to \tau_2' \vdash \langle\, x_1 \,,\, e_2' \,\rangle : T * \tau_1'$$

Then by S-App,

$$\Gamma', x_1 : T, x_2 : (T * \tau_1') \to \tau_2' \vdash x_2(\langle\, x_1 \,,\, e_2' \,\rangle) : \tau_2'$$

Case $\dfrac{}{\Gamma \vdash x : \tau \rightsquigarrow x}$ E-VAR

Since the context converts pointwise, we know

$$\Gamma' \vdash x : \tau'$$

Case $\dfrac{}{\Gamma \vdash ( ) : \mathbf{unit} \rightsquigarrow ( )}$ E-UNIT

By T-Unit, $\mathbf{unit} \rightsquigarrow \mathbf{unit}$, so we want to show that the translated expression has type $\mathbf{unit}$. By S-Unit,

$$\Gamma' \vdash ( ) : \mathbf{unit}$$

Case $\dfrac{\tau \rightsquigarrow \tau'}{\Gamma \vdash [\,]\{\tau\} : \tau\ \mathbf{list} \rightsquigarrow [\,]\{\tau'\}}$ E-NIL

By T-List, $\tau\ \mathbf{list} \rightsquigarrow \tau'\ \mathbf{list}$, so we want to show that the translated expression has type $\tau'\ \mathbf{list}$. By S-Nil,

$$\Gamma' \vdash [\,]\{\tau'\} : \tau'\ \mathbf{list}$$

Case $\dfrac{\Gamma \vdash e_1 : \tau \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau\ \mathbf{list} \rightsquigarrow e_2'}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathbf{list} \rightsquigarrow e_1' :: e_2'}$ E-CONS

By T-List, $\tau\ \mathbf{list} \rightsquigarrow \tau'\ \mathbf{list}$, so we want to show that the translated expression has type $\tau'\ \mathbf{list}$. By the inductive hypothesis, $\Gamma' \vdash e_1' : \tau'$ and $\Gamma' \vdash e_2' : \tau'\ \mathbf{list}$. Then by S-Cons,

$$\Gamma' \vdash e_1' :: e_2' : \tau'\ \mathbf{list}$$

Case $\dfrac{\tau \rightsquigarrow \tau' \qquad \Gamma \vdash e : \tau_1\ \mathbf{list} \rightsquigarrow e' \qquad \Gamma \vdash e_1 : \tau_2 \rightsquigarrow e_1' \qquad \Gamma, x : \tau_1, xs : \tau_1\ \mathbf{list} \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash \mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\} : \tau_2 \rightsquigarrow \mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}}$ E-MATCH

By the inductive hypothesis, $\Gamma' \vdash e' : \tau_1'$, $\Gamma' \vdash e_1' : \tau_2'$, and $\Gamma', x : \tau_1', xs : \tau_1'\ \mathbf{list} \vdash e_2' : \tau_2'$. Then by S-Match,

$$\Gamma' \vdash \mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\} : \tau_2'$$

Case $\dfrac{\Gamma \vdash e_\ell : \tau_\ell \rightsquigarrow e_\ell' \quad \forall \ell \in L}{\Gamma \vdash \langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L} : \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow \langle\, \ell \hookrightarrow e_\ell' \,\rangle_{\ell \in L}}$ E-PROD

By T-Prod, $\langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow \langle\, \ell : \tau_\ell' \,\rangle_{\ell \in L}$, so we want to show that the translated expression has type $\langle\, \ell : \tau_\ell' \,\rangle_{\ell \in L}$. By the inductive hypothesis, $\Gamma' \vdash e_\ell' : \tau_\ell'$ for all $\ell \in L$. Then by S-Prod,

$$\Gamma' \vdash \langle\, \ell \hookrightarrow e_\ell' \,\rangle_{\ell \in L} : \langle\, \ell : \tau_\ell' \,\rangle_{\ell \in L}$$

Case $\dfrac{\Gamma \vdash e : \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow e'}{\Gamma \vdash e.k : \tau_k \rightsquigarrow e'.k}$ E-PROJ

By the inductive hypothesis, $\Gamma' \vdash e' : \langle\, \ell : \tau'_\ell \,\rangle_{\ell \in L}$. Then by S-Proj,

$$\Gamma' \vdash e'.k : \tau'_k$$

Case $\dfrac{\Gamma \vdash e : \tau_k \rightsquigarrow e' \qquad \tau_\ell \rightsquigarrow \tau'_\ell \quad \forall \ell \in L}{\Gamma \vdash \mathbf{in}[k]\{[\, \ell : \tau_\ell \,]_{\ell \in L}\}\,(e) : \tau_k \rightsquigarrow \mathbf{in}[k]\{[\, \ell : \tau'_\ell \,]_{\ell \in L}\}\,(e')}$ E-INJ

By the inductive hypothesis, $\Gamma' \vdash e' : \tau'_k$. Then by S-Inj,

$$\Gamma' \vdash \mathbf{in}[k]\{[\, \ell : \tau'_\ell \,]_{\ell \in L}\}\,(e') : \tau'_k$$

Case $\dfrac{\Gamma \vdash e : [\, \ell : \tau_\ell \,]_{\ell \in L} \rightsquigarrow e' \qquad \Gamma, x_\ell : \tau_\ell \vdash e_\ell : \tau \rightsquigarrow e'_\ell \quad \forall i \in L}{\Gamma \vdash \mathbf{case}\ e\ \{\, \ell(x_\ell) \hookrightarrow e_\ell \,\}_{\ell \in L} : \tau \rightsquigarrow \mathbf{case}\ e'\ \{\, \ell(x_\ell) \hookrightarrow e'_\ell \,\}_{\ell \in L}}$ E-CASE

By the inductive hypothesis, $\Gamma' \vdash e' : [\, \ell : \tau'_\ell \,]_{\ell \in L}$ and $\Gamma', x_\ell : \tau'_\ell \vdash e'_\ell : \tau'$ for all $\ell \in L$. Then by S-Case,

$$\Gamma' \vdash \mathbf{case}\ e'\ \{\, \ell(x_\ell) \hookrightarrow e'_\ell \,\}_{\ell \in L} : \tau'$$

□

Chapter 5

Soundness

In this chapter, we show that our translation does not change the semantics of the program. We define a logical relation to define equality of two programs. Then, we prove soundness by showing that the original and translated programs are related.

5.1 Program Equality

To prove soundness, we need to formalize what it means for two programs to be equal. We do this with a logical relation, and specify rules for equality of values, closed expressions, and open expressions.

5.1.1 Equality of Values

Equality of values is defined in Figure 5.1. The interesting rule is for the function case. Intuitively, two functions should be equal if they evaluate to equivalent expressions when applied to equivalent values. This idea is captured in the rule, but the translated function is applied to the pair including both the environment and the value.

5.1.2 Equality of Closed Expressions

We define equality of closed expressions in Figure 5.2. Any two diverging programs are equivalent. For terminating programs, they must evaluate to equivalent values *with the same cost*. This is necessary to ensure the translation preserves cost. Note that since the transformation does not add or remove any ticks, it should not affect the total cost. We prove this formally in Theorem 2.

Note that EQ-Step only states that two expressions e_1 and e_2 are equivalent if they step to equivalent *values*. We show that e_1 and e_2 are also equivalent if they step to equivalent *expressions*.

$$\frac{}{() \approx () : \mathbf{unit}}\ \text{EQ-UNIT} \qquad \frac{\tau \rightsquigarrow \tau'}{[\,]\{\tau\} \approx [\,]\{\tau'\} : \tau\ \mathbf{list}}\ \text{EQ-NIL}$$

$$\frac{v_1 \approx v_1' : \tau \qquad v_2 \approx v_2' : \tau\ \mathbf{list}}{v_1 :: v_2 \approx v_1' :: v_2' : \tau\ \mathbf{list}}\ \text{EQ-CONS}$$

$$\frac{v_\ell \approx v_\ell' : \tau_\ell \quad \forall i \in [k]}{\langle\, \ell \hookrightarrow v_\ell \,\rangle_{\ell \in L} \approx \langle\, \ell \hookrightarrow v_\ell \,\rangle_{\ell \in L} : \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L}}\ \text{EQ-PROD}$$

$$\frac{\begin{array}{c} f^* = \mathbf{fun}^m\,\{\tau_1, \tau_2\}(f\,.\,x\,.\,e) \qquad f^{*'} = \mathbf{fun}^m\,\{T * \tau_1', \tau_2'\}(f'\,.\,x\,.\,e') \\ f^*(v) \approx f^{*'}(\langle\, e_{env}\,,\, v' \,\rangle) : \tau_2 \quad \forall v \approx v' : \tau_1 \end{array}}{f^* \approx \langle\, e_{env}\,,\, f^{*'} \,\rangle : \tau_1 \to \tau_2}\ \text{EQ-FUN}$$

$$\frac{\tau = [\,\ell : \tau_\ell\,]_{\ell \in L} \qquad v \approx v' : \tau_\ell}{\mathbf{in}[\ell]\{\tau\}\,(v) \approx \mathbf{in}[\ell]\{\tau\}\,(v') : \tau}\ \text{EQ-INJ}$$

Figure 5.1: Equality of values

$$\frac{\tau \rightsquigarrow \tau' \qquad \Gamma \vdash v_1 : \tau \qquad \Gamma \vdash v_2 : \tau' \qquad e_1 \longmapsto_n^* v_1 \qquad e_2 \longmapsto_n^* v_2 \qquad v_1 \approx v_2 : \tau}{e_1 \approx e_2 : \tau}\ \text{EQ-STEP}$$

$$\frac{e_1 \longmapsto_\infty^* \bot \qquad e_2 \longmapsto_\infty^* \bot}{e_1 \approx e_2 : \tau}\ \text{EQ-DIVERGE}$$

Figure 5.2: Equality of closed expressions

Lemma 3 (Head Expansion). Let $e_1' \approx e_2' : \tau$. If $e_1 \longmapsto_n^* e_1'$ and $e_2 \longmapsto_n^* e_2'$, then $e_1 \approx e_2 : \tau$.

Proof. Since $e_1' \approx e_2' : \tau$, we know there are two cases:

Case Both e_1' and e_2' diverge.

Then by Div-Step, both e_1 and e_2 diverge. Thus, by EQ-Diverge, $e_1 \approx e_2 : \tau$.

Case Both e_1' and e_2' terminate.

Then $e_1' \longmapsto_{n'}^* v_1$ and $e_2 \longmapsto_{n'}^* v_2$ where $v_1 \approx v_2 : \tau$. Thus,

$$e_1 \longmapsto_{n+n'}^* v_1$$

$$e_2 \longmapsto^*_{n+n'} v_2$$

Therefore, by EQ-Step, $e_1 \approx e_2 : \tau$.

□

5.1.3 Equality of Open Expressions

Equivalence of open expressions is defined in Figure 5.3 using closing substitutions. Equivalence of closing substitutions γ and γ' is defined as follows:

$$\Gamma \vdash \gamma \approx \gamma' \text{ if and only if } \forall x : \tau \in \Gamma,\ \gamma(x) \approx \gamma'(x) : \tau$$

That is, both closing substitutions must have bindings of the correct type for each variable in Γ, and the corresponding values must be equivalent.

$$\frac{\Gamma \vdash \gamma \approx \gamma' \qquad \gamma(e) \approx \gamma'(e') : \tau}{\Gamma \vdash e \sim e' : \tau}\ \text{EQ-EXP}$$

Figure 5.3: Equality of open expressions

5.2 Soundness Proof

Theorem 2 . Let $\Gamma \vdash \gamma \approx \gamma'$. If $\Gamma \vdash e : \tau \rightsquigarrow e'$ then $\gamma(e) \approx \gamma'(e') : \tau$.

Proof. Let $\Gamma \vdash \gamma \approx \gamma'$. We proceed by induction over the translation judgement.

Case E-Fun: $e = \mathbf{fun}^m \{\tau_1, \tau_2\}(f_i \,.\, x \,.\, e_b)$.

We know $\gamma'(e') = \langle\, e_{env}\,,\ f^* \,\rangle$, where e_{env} and f^* are given in Table 5.1. We want to show that $\gamma(e) \approx \gamma'(e') : \tau_1 \to \tau_2$.

From EQ-Fun, we need to show that for all $v \approx v' : \tau_1$, $\Gamma \vdash \gamma\,(\mathbf{fun}^m\,\{\tau_1, \tau_2\}(f_i \,.\, x \,.\, e)(v)) \approx f^*(\langle\, e_{env}\,,\ v' \,\rangle) : \tau_2$. By Lemma 3, it is sufficient to show that both sides step to equivalent expressions with the same cost. We will show this is true for all m and proceed by induction on m.

When $m = 0$, both sides diverge, and by EQ-Diverge, $\gamma\left(\left(\mathbf{fun}^0\,\{\tau_1, \tau_2\}(f_i \,.\, x \,.\, e)\right)(v)\right) \approx \left(\mathbf{fun}^0\,\{T * \tau_1', \tau_2'\}(f \,.\, y \,.\, e_1)\right)(\langle\, e_{env}\,,\ v' \,\rangle) : \tau_2$.

Name	Expression
e_{env}	$\mathbf{in}[i]\{T\}\,(\langle\, j \hookrightarrow \gamma'(z_j)\,\rangle_{j\in[n_i]})$
f^*	$\mathbf{fun}^m\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)$
e_1	$\mathbf{let}\ f_i = \langle\, e_{env}\,,\, f\,\rangle\ \mathbf{in}\ e_2$
e_2	$\mathbf{let}\ y_1 = y.l\ \mathbf{in}\ e_3$
e_3	$\mathbf{let}\ y_2 = y.r\ \mathbf{in}\ e_4$
e_4	$\mathbf{case}\ y_1\ \{\, i(z) \hookrightarrow e_5 \mid \_ \hookrightarrow (\mathbf{fun}^0\,\{\mathbf{unit}, \tau_2'\}(f\,.\,x\,.\,e))()\,\}$
e_5	$\mathbf{let}\ z_1 = z.1\ \mathbf{in}\ \ldots \mathbf{let}\ z_n = z.n_i\ \mathbf{in}\ e_6$
e_6	$[y_2/x]\gamma'(e_b')$

Table 5.1: Expression names for E-Fun case in Theorem 2.

Now we consider the case where $m > 0$. We step both applications as follows:

$$\gamma\,((\mathbf{fun}^m\,\{\tau_1,\tau_2\}(f_i\,.\,x\,.\,e_b))\,(v)) \longmapsto_0 [v/x\,,\ \mathbf{fun}^{m-1}\,\{\tau_1,\tau_2\}(f_i\,.\,x\,.\,e)/f_i](\gamma(e_b))$$

$$\begin{aligned}
f^*(\langle\, e_{env}\,,\, v'\,\rangle) &\longmapsto_0 [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f]e_1 \\
&\longmapsto_0 [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f\,, \\
&\qquad \langle\, e_{env}\,,\, \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i]e_2 \\
&\longmapsto_0 [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f\,, \\
&\qquad \langle\, e_{env}\,,\, \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i\,,\ e_{env}/y_1]e_3 \\
&\longmapsto_0 [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f\,, \\
&\qquad \langle\, e_{env}\,,\, \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i\,,\ e_{env}/y_1\,, \\
&\qquad v'/y_2]e_4 \\
&\longmapsto_0 [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f\,, \\
&\qquad \langle\, e_{env}\,,\, \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i\,,\ e_{env}/y_1\,, \\
&\qquad v'/y_2\,,\ \langle\, j \hookrightarrow \gamma'(z_j)\,\rangle_{j\in[n_i]}/z]e_5 \\
&\longmapsto_0^* [\langle\, e_{env}\,,\, v'\,\rangle/y\,,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)/f\,, \\
&\qquad \langle\, e_{env}\,,\, \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i\,,\ e_{env}/y_1\,, \\
&\qquad v'/y_2\,,\ \langle\, j \hookrightarrow \gamma'(z_j)\,\rangle_{j\in[n_i]}/z\,, \\
&\qquad \gamma'(z_1)/z_1\,,\ldots\,,\ \gamma'(z_{n_i})/z_{n_i}\,,\ y_2/x](\gamma'(e_b'))
\end{aligned}$$

Let δ'' be the final substitution above. Define:

$$\delta = \gamma \cup [v/x\,,\ \mathbf{fun}^{m-1}\,\{\tau_1,\tau_2\}(f_i\,.\,x\,.\,e)/f_i]$$

$$\delta' = \gamma' \cup \delta''$$

Since f, y, y_1, and z are all fresh, they do not appear in Γ. Thus, removing them from δ'' does not affect the equivalence of the substitutions under Γ, so we have

$$\Gamma, x : \tau_1,\ f_i : \tau_1 \to \tau_2 \vdash \\ \quad \delta'' \approx [v'/x\ ,\ \langle\, e_{env}\ ,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i\ ,\ \gamma'(z_1)/z_1\ ,\ \ldots\ ,\ \gamma'(z_n)/z_n]$$

Note that δ' already includes the substitutions from γ', so the substitutions $\gamma'(z_1)/z_1\ ,\ \ldots\ ,\ \gamma'(z_{n_i})/z_{n_i}$ are redundant. Thus,

$$\Gamma,\ x : \tau_1,\ f_i : \tau_1 \to \tau_2 \vdash \delta' \approx \gamma' \cup [v'/x\ ,\ \langle\, e_{env}\ ,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i]$$

Since $\Gamma \vdash v \approx v' : \tau_1$ and $\Gamma \vdash \gamma \approx \gamma'$, we know:

$$\Gamma,\ x : \tau_1 \vdash \gamma \cup [v/x] \approx \gamma' \cup [v'/x]$$

By the inductive hypothesis for the inner induction,

$$\mathbf{fun}^{m-1}\,\{\tau_1, \tau_2\}(f_i\,.\,x\,.\,e) \approx \langle\, e_{env}\ ,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle$$

Thus,

$$\Gamma,\ x : \tau_1,\ f_i : \tau_1 \to \tau_2 \vdash \gamma \cup [v/x,\ \mathbf{fun}^{m-1}\,\{\tau_1, \tau_2\}(f_i\,.\,x\,.\,e)/f_i] \\ \approx \gamma' \cup [v'/x,\ \langle\, e_{env}\ ,\ \mathbf{fun}^{m-1}\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1)\,\rangle/f_i]$$

Therefore, $\Gamma,\ x : \tau_1,\ f_i : \tau_1 \to \tau_2 \vdash \delta \approx \delta'$. By the inductive hypothesis, $\Gamma,\ x : \tau_1 \vdash \delta(e_b) \approx \delta'(e_b') : \tau_2$. Then by Lemma 3,

$$\gamma\left((\mathbf{fun}^m\,\{\tau_1, \tau_2\}(f_i\,.\,x\,.\,e))\,(v)\right) \approx (\mathbf{fun}^m\,\{T * \tau_1', \tau_2'\}(f\,.\,y\,.\,e_1))\,(\langle\, e_{env}\ ,\ v'\,\rangle) : \tau_2$$

Case $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2' \qquad x_1,\ x_2 \text{ fresh}}{\Gamma \vdash e_1(e_2) : \tau_2 \rightsquigarrow \mathbf{let}\ x_1 = e_1'.l\ \mathbf{in\ let}\ x_2 = e_1'.r\ \mathbf{in}\ x_2(\langle\, x_1\ ,\ e_2'\,\rangle)}$ E-App

By the inductive hypothesis, $\gamma(e_1) \approx \gamma'(e_1') : \tau_1 \to \tau_2$, so by the rules for equality of closed expressions, there are two cases: either both sides diverge, or both terminate after evaluating to values with equal cost.

Case Both $\gamma(e_1)$ and $\gamma'(e_1')$ diverge.

By Div-App1, $\gamma(e_1(e_2)) \longmapsto_\infty^* \bot$, and $\gamma'(e_1'(e_2')) \longmapsto_\infty^* \bot$. Then by EQ-Diverge,

$$\gamma(e_1(e_2)) \approx \gamma'(e_1'(e_2')) : \tau_1 \to \tau_2$$

Case Both $\gamma(e_1)$ and $\gamma'(e_1')$ terminate.

Since both terminate, $\gamma(e_1) \longmapsto^*_{n_1} v_1$ and $\gamma(e_1') \longmapsto^*_{n_1} v_1'$, where $v_1 \approx v_1' : \tau_1 \to \tau_2$. By canonical forms, $v_1 = \mathbf{fun}^m \{\tau_1, \tau_2\}(f \,.\, x \,.\, e)$, and $v_1' = \langle\, v_{env} \,,\, f^* \,\rangle$, where $f^* = \mathbf{fun}^m \{T * \tau_1', \tau_2'\}(f' \,.\, x \,.\, e)$.

By the inductive hypothesis, $\Gamma \vdash \gamma(e_2) \approx \gamma'(e_2') : \tau_1$. By the rules for equality of closed expressions, there are two cases: either both sides diverge, or both terminate after evaluating to values with the same cost.

Case Both $\gamma(e_2)$ and $\gamma'(e_2')$ diverge.

By Div-App2, $\gamma(e_1(e_2)) \longmapsto^*_\infty \bot$ and $\gamma'(e_1'(e_2')) \longmapsto^*_\infty \bot$. Then by EQ-Diverge,

$$\gamma(e_1(e_2)) \approx \gamma'(e_1'(e_2')) : \tau_1 \to \tau_2$$

Case Both $\gamma(e_2)$ and $\gamma'(e_2')$ terminate.

Since both terminate, $\gamma(e_2) \longmapsto^*_{n_2} v_2$ and $\gamma'(e_2') \longmapsto^*_{n_2} v_2'$, where $v_2 \approx v_2' : \tau_1$. We want to show that $\gamma(e_1(e_2)) \approx \gamma(e_1'(e_2')) : \tau_2$.

Recall that $v_1' = \langle\, v_{env} \,,\, f^* \,\rangle$. We step both sides as follows:

$$\begin{aligned} \gamma(e_1(e_2)) &\longmapsto_{n_1} \gamma(v_1(e_2)) \\ &\longmapsto_{n_2} \gamma(v_1(v_2)) \end{aligned}$$

$$\begin{aligned} \gamma'(e_1'(e_2')) &\longmapsto_{n_1} \gamma'\,(\mathbf{let}\ x_1 = v_1'.l\ \mathbf{in}\ \mathbf{let}\ x_2 = v_1'.r\ \mathbf{in}\ x_2(\langle\, x_1 \,,\, e_2' \,\rangle)) \\ &\longmapsto_{n_2} \gamma'\,(\mathbf{let}\ x_1 = v_1'.l\ \mathbf{in}\ \mathbf{let}\ x_2 = v_1'.r\ \mathbf{in}\ x_2(\langle\, x_1 \,,\, v_2' \,\rangle)) \\ &\longmapsto_0 \quad \gamma'\,(\mathbf{let}\ x_2 = e_1'.r\ \mathbf{in}\ x_2(\langle\, v_{env} \,,\, e_2' \,\rangle)) \\ &\longmapsto_0 \quad \gamma'\,(f^*(\langle\, v_{env} \,,\, e_2' \,\rangle)) \end{aligned}$$

Note that x_1 and x_2 are fresh, so they do not appear in e_2 or e_2'. Thus, after substituting, there is no need to continue writing the substitution in the expressions above.

Since $v_1 \approx \langle\, v_{env} \,,\, f^* \,\rangle : \tau_1 \to \tau_2$, by inversion on EQ-Fun,

$$\gamma(v_1(v)) \approx \gamma'(f^*(\langle\, e_{env} \,,\, v' \,\rangle)) : \tau_1 \to \tau_2 \quad \forall v \approx v' : \tau_1$$

Combining $v_2 \approx v_2'$ with the above, we have:

$$\gamma(v_1(v_2)) \approx \gamma'(f^*(\langle\, e_{env} \,,\, v_2' \,\rangle)) : \tau_1 \to \tau_2$$

Therefore, by Lemma 3,

$$\gamma(e_1(e_2)) \approx \gamma'(e_1'(e_2')) : \tau_1 \to \tau_2$$

Case $\dfrac{}{\Gamma \vdash x : \tau \rightsquigarrow x}$ E-VAR

Since $\Gamma \vdash \gamma \approx \gamma'$ and $\Gamma \vdash x : \tau$, we know $\gamma(x) \approx \gamma'(x) : \tau$.

Case $\dfrac{}{\Gamma \vdash (\,) : \mathbf{unit} \rightsquigarrow (\,)}$ E-UNIT

By EQ-Unit, $(\,) \approx (\,) : \mathbf{unit}$.

Case $\dfrac{\tau \rightsquigarrow \tau'}{\Gamma \vdash [\,]\{\tau\} : \tau\ \mathbf{list} \rightsquigarrow [\,]\{\tau'\}}$ E-NIL

By EQ-Nil, $[\,]\{\tau\} \approx [\,]\{\tau\} : \tau\ \mathbf{list}$.

Case $\dfrac{\Gamma \vdash e_1 : \tau \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau\ \mathbf{list} \rightsquigarrow e_2'}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathbf{list} \rightsquigarrow e_1' :: e_2'}$ E-CONS

By the inductive hypothesis, $\gamma(e_1) \approx \gamma'(e_1') : \tau$ and $\gamma(e_2) \approx \gamma'(e_2') : \tau\ \mathbf{list}$. By the rules for equality of closed expressions, there are two cases:

Case Both $\gamma(e_1)$ and $\gamma'(e_1')$ diverge, and/or both $\gamma(e_2)$ and $\gamma'(e_2')$ diverge.

By either Div-Cons1 or Div-Cons2, $\gamma(e_1 :: e_2) \longmapsto_\infty^* \bot$ and $\gamma'(e_1' :: e_2') \longmapsto_\infty^* \bot$. Then by EQ-Diverge,

$$\gamma(e_1 :: e_2) \approx \gamma'(e_1' :: e_2') : \tau\ \mathbf{list}$$

Case $\gamma(e_1)$, $\gamma'(e_1')$, $\gamma(e_2)$, and $\gamma'(e_2')$ all terminate.

Since they all terminate, $\gamma(e_1) \longmapsto_{n_1}^* v_1$, $\gamma(e_2) \longmapsto_{n_2}^* v_2$, $\gamma'(e_1') \longmapsto_{n_1}^* v_1'$, and $\gamma'(e_2') \longmapsto_{n_2}^* v_2'$, where $v_1 \approx v_1' : \tau$ and $v_2 \approx v_2' : \tau\ \mathbf{list}$.

By EQ-Cons, $v_1 :: v_2 \approx v_1' :: v_2' : \tau\ \mathbf{list}$. Therefore, by EQ-Step,

$$\gamma(e_1 :: e_2) \approx \gamma'(e_1' :: e_2') : \tau\ \mathbf{list}$$

Case $\dfrac{\tau \rightsquigarrow \tau' \qquad \Gamma \vdash e : \tau_1\ \mathbf{list} \rightsquigarrow e' \qquad \Gamma \vdash e_1 : \tau_2 \rightsquigarrow e_1' \qquad \Gamma, x : \tau_1, xs : \tau_1\ \mathbf{list} \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash \mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\} : \tau_2 \rightsquigarrow \mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}}$ E-MATCH

By the inductive hypothesis, $\gamma(e) \approx \gamma'(e') : \tau_1\ \mathbf{list}$. By the rules for equality of closed expressions, equality of lists, and canonical forms, there are three cases: either both sides diverge, both terminate and step to the Nil case, or both terminate and step to the Cons case.

Case Both $\gamma(e)$ and $\gamma'(e')$ diverge.

By Div-Match,

$$\gamma(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \longmapsto_\infty^* \bot$$

$$\gamma'(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) \longmapsto_\infty^* \bot$$

Then by EQ-Diverge,

$$\gamma\,(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \\ \approx \gamma'\,(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) : \tau_2$$

Case Both $\gamma(e)$ and $\gamma'(e')$ terminate and step to the Nil case.

Then $\gamma(e) \longmapsto_n^* [\,]\{\tau_1\}$ and $\gamma(e') \longmapsto_n^* [\,]\{\tau_1'\}$. By D-MatchNil,

$$\gamma\,(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \longmapsto_n^* \gamma(e_1)$$

$$\gamma'\,(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) \longmapsto_n^* \gamma'(e_1')$$

By the inductive hypothesis, $\gamma(e_1) \approx \gamma(e_1') : \tau_1\ \mathbf{list}$. Therefore, by Lemma 3,

$$\gamma\,(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \\ \approx \gamma'\,(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) : \tau_2$$

Case $\gamma(e)$ and $\gamma'(e')$ terminate and step to the Cons case.

Then $\gamma(e) \longmapsto_n^* v_3 :: v_4$ and $\gamma(e') \longmapsto_n^* v_3' :: v_4'$. By inversion on EQ-Cons, $v_3 \approx v_3' : \tau_1$ and $v_4 \approx v_4' : \tau_1\ \mathbf{list}$. By D-MatchCons,

$$\gamma\,(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \longmapsto_n^* [v_3/x,\, v_4/xs]\gamma(e_2)$$

$$\gamma'\,(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) \longmapsto_n^* [v_3'/x,\, v_4'/xs]\gamma'(e_2')$$

Define

$$\delta = \gamma \cup [v_3/x,\, v_4/xs]$$

$$\delta' = \gamma' \cup [v_3'/x,\, v_4'/xs]$$

Note that $\Gamma, x : \tau_1, xs : \tau_1\ \mathbf{list} \vdash \delta \approx \delta'$ since $\Gamma \vdash \gamma \approx \gamma'$, $v_3 \approx v_3' : \tau_1$, and $v_4 \approx v_4' : \tau_1\ \mathbf{list}$. By the inductive hypothesis, $\delta(e_2) \approx \delta'(e_2') : \tau_2\ \mathbf{list}$. Therefore, by Lemma 3,

$$\gamma\,(\mathbf{match}\ e\ \{\,[\,]\{\tau_1\} \hookrightarrow e_1\,;\, x :: xs \hookrightarrow e_2\,\}) \\ \approx \gamma'\,(\mathbf{match}\ e'\ \{\,[\,]\{\tau_1'\} \hookrightarrow e_1'\,;\, x :: xs \hookrightarrow e_2'\,\}) : \tau_2$$

Case $\dfrac{\Gamma \vdash e_\ell : \tau_\ell \rightsquigarrow e_\ell' \quad \forall \ell \in L}{\Gamma \vdash \langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L} : \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow \langle\, \ell \hookrightarrow e_\ell' \,\rangle_{\ell \in L}}$ E-Prod

Let $\tau = \langle\, \ell : \tau_\ell \,\rangle_{\ell \in k}$. By the inductive hypothesis, $\gamma(e_\ell) \approx \gamma'(e_\ell') : \tau_\ell$ for all $\ell \in L$. By the rules for equality of closed expressions, there are two cases:

Case There exists k such that both $\gamma(e_k)$ and $\gamma'(e'_k)$ diverge.

By Div-Prod, $\gamma(\langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L}) \longmapsto^*_\infty \bot$ and $\gamma'(\langle\, \ell \hookrightarrow e'_\ell \,\rangle_{\ell \in L}) \longmapsto^*_\infty \bot$. Then by EQ-Diverge,

$$\gamma(\langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L}) \approx \gamma'(\langle\, \ell \hookrightarrow e'_\ell \,\rangle_{\ell \in L}) : \tau$$

Case For all $\ell \in L$, both $\gamma(e_\ell)$ and $\gamma'(e'_\ell)$ terminate.

Since both terminate, $\gamma(e_\ell) \longmapsto^*_{n_\ell} v_\ell$ and $\gamma'(e'_\ell) \longmapsto^*_{n_\ell} v'_\ell$, where $v_\ell \approx v'_\ell : \tau_\ell$ for all $\ell \in L$. By EQ-Prod, $\langle\, \ell \hookrightarrow v_\ell \,\rangle_{\ell \in L} \approx \langle\, \ell \hookrightarrow v'_\ell \,\rangle_{\ell \in L} : \tau$. Therefore, by EQ-Step,

$$\gamma(\langle\, \ell \hookrightarrow e_\ell \,\rangle_{\ell \in L}) \approx \gamma'(\langle\, \ell \hookrightarrow e'_\ell \,\rangle_{\ell \in L}) : \tau$$

Case $\dfrac{\Gamma \vdash e : \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L} \rightsquigarrow e'}{\Gamma \vdash e.k : \tau_k \rightsquigarrow e'.k}$ E-PROJ

Let $\tau = \langle\, \ell : \tau_\ell \,\rangle_{\ell \in L}$. By the inductive hypothesis, $\gamma(e) \approx \gamma'(e') : \tau$. By the rules for equality of closed expressions, there are two cases: either both sides diverge, or both terminate after evaluating to values.

Case Both $\gamma(e)$ and $\gamma'(e')$ diverge.

By Div-Proj, $\gamma(e.k) \longmapsto^*_\infty \bot$ and $\gamma'(e'.k) \longmapsto^*_\infty \bot$. Then by EQ-Diverge,

$$\gamma(e.k) \approx \gamma'(e'.k) : \tau_k$$

Case Both $\gamma(e)$ and $\gamma'(e')$ terminate.

Then $\gamma(e) \longmapsto^*_n v$ and $\gamma'(e') \longmapsto^*_n v'$, where $v \approx v' : \tau$. By canonical forms, $v = \langle\, \ell \hookrightarrow v_\ell \,\rangle_{\ell \in L}$ and $v' = \langle\, \ell \hookrightarrow v'_\ell \,\rangle_{\ell \in L}$, where $v_\ell \approx v'_\ell : \tau_\ell$ for all $\ell \in L$.

By D-Proj, $\gamma(v.k) \longmapsto_0 v_k$ and $\gamma'(v'.k) \longmapsto_0 v'_k$, where $v_k \approx v'_k : \tau_k$. Thus, $\gamma(e.k) \longmapsto^*_n v_k$ and $\gamma'(e'.k) \longmapsto^*_n v'_k$, where $v_k \approx v'_k : \tau_k$. By EQ-Step,

$$\gamma(e.k) \approx \gamma'(e'.k) : \tau_k$$

Case $\dfrac{\Gamma \vdash e : \tau_k \rightsquigarrow e' \qquad \tau_\ell \rightsquigarrow \tau'_\ell \quad \forall \ell \in L}{\Gamma \vdash \mathbf{in}[k]\{[\, \ell : \tau_\ell \,]_{\ell \in L}\}\,(e) : \tau_k \rightsquigarrow \mathbf{in}[k]\{[\, \ell : \tau'_\ell \,]_{\ell \in L}\}\,(e')}$ E-INJ

Let $\tau = [\, \ell : \tau_\ell \,]_{\ell \in L}$ and let $\tau \rightsquigarrow \tau'$. By the inductive hypothesis, $\gamma(e) \approx \gamma'(e') : \tau_k$. By the rules for equality of closed expressions, there are two cases: either both sides diverge, or both terminate after evaluating to values.

Case Both $\gamma(e)$ and $\gamma'(e')$ diverge.

By Div-Inj, $\gamma(\mathbf{in}[k]\{\tau\}\,(e)) \longmapsto^*_\infty \bot$ and $\gamma'(\mathbf{in}[k]\{\tau\}\,(e')) \longmapsto^*_\infty \bot$. Then by EQ-Diverge,

$$\gamma(\mathbf{in}[k]\{\tau\}\,(e)) \approx\ \gamma'(\mathbf{in}[k]\{\tau\}\,(e')) : \tau$$

Case Both $\gamma(e)$ and $\gamma'(e')$ terminate.

Then $\gamma(e) \longmapsto_n^* v$ and $\gamma'(e') \longmapsto_n^* v'$, where $v \approx v' : \tau_k$. By EQ-Inj, $\mathbf{in}[k]\{\tau\}\,(v) \approx \mathbf{in}[k]\{\tau'\}\,(v') : \tau$. Therefore, by EQ-Step,

$$\gamma(\mathbf{in}[k]\{\tau\}\,(e)) \approx\ \gamma'(\mathbf{in}[k]\{\tau\}\,(e')) : \tau$$

Case $\dfrac{\Gamma \vdash e : [\,\ell : \tau_\ell\,]_{\ell \in L} \rightsquigarrow e' \qquad \Gamma, x_\ell : \tau_\ell \vdash e_\ell : \tau \rightsquigarrow e'_\ell \quad \forall i \in L}{\Gamma \vdash \mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L} : \tau \rightsquigarrow \mathbf{case}\ e'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}}$ E-CASE

Let $\tau = [\,\ell : \tau_\ell\,]_{\ell \in L}$ and let $\Gamma \vdash \tau \rightsquigarrow \tau'$. By the inductive hypothesis, $\gamma(e) \approx \gamma(e') : \tau$. By the rules for equality of closed expressions, there are two cases: either both sides diverge, or both terminate after evaluating to values.

Case Both $\gamma(e)$ and $\gamma'(e')$ diverge.

By Div-Case,

$$\gamma(\mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L}) \longmapsto_\infty^* \bot$$
$$\gamma'(\mathbf{case}\ e'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}) \longmapsto_\infty^* \bot$$

Then by EQ-Diverge,

$$\gamma\,(\mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L}) \approx \gamma'\,(\mathbf{case}\ e'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}) : \tau$$

Case Both $\gamma(e)$ and $\gamma'(e')$ terminate.

Then $\gamma(e) \longmapsto_n^* v$ and $\gamma(e') \longmapsto_n^* v'$, where $v \approx v' : \tau$. By canonical forms, $v = \mathbf{in}[l_i]\{\tau\}\,(v)$ and $v' = \mathbf{in}[l_i]\{\tau\}\,(v')$.

By D-Case2,

$$\gamma\,(\mathbf{case}\ v\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L}) \longmapsto_0 [v/x_k]e_k$$
$$\gamma'\,(\mathbf{case}\ v'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}) \longmapsto_0 [v'/x_k]e'_k$$

Therefore,

$$\gamma\,(\mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L}) \longmapsto_n^* [v/x_k]e_k$$
$$\gamma'\,(\mathbf{case}\ e'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}) \longmapsto_n^* [v'/x_k]e'_k$$

Define

$$\delta = \gamma \cup [v/x_k]$$
$$\delta' = \gamma' \cup [v'/x_k]$$

Note that $\Gamma, x : \tau \vdash \delta \approx \delta'$ since $\Gamma \vdash \gamma \approx \gamma'$, and $v \approx v' : \tau$. By the inductive hypothesis, $\delta(e_k) \approx \delta(e'_k) : \tau_2$. Therefore, by Lemma 3,

$$\gamma\,(\mathbf{case}\ e\ \{\,\ell(x_\ell) \hookrightarrow e_\ell\,\}_{\ell \in L}) \approx \gamma'\,(\mathbf{case}\ e'\ \{\,\ell(x_\ell) \hookrightarrow e'_\ell\,\}_{\ell \in L}) : \tau$$

□

5.3 General Recursive Functions

Thus far, we have only considered bounded recursive functions, which denote the number of recursive calls they are allowed to make. However, our results also extend to general recursive functions by compactness.

Theorem 3 (Compactness). Only finitely many unwindings of a fixed point expression are needed in a complete evaluation of a a well-typed, closed expression.

Compactness is a known result for PCF [11]. Although we are using a modified version of PCF, the theorem still holds.

Intuitively, for any terminating program, there is some maximum number of recursive calls n that any function makes. In this case, the code written with general recursive functions is equivalent to bounded recursion with the maximum number of recursive calls set to n. We can formalize this idea as follows.

5.3.1 Translating Unbounded to Bounded PCF

Let Unbounded PCF (UPCF) be identical to Bounded PCF (BPCF) except it has general recursive functions rather than bounded recursion. Say we have an expression e in UPCF. We know by compactness that we can translate e into an equivalent expression $\bar{e}$ in BPCF by adding a large enough m to each function. Then the reverse should also be true—if we remove these numbers, we should get back an expression in UPCF. This is captured in Lemma 4.

Lemma 4 . Given $e : \tau$ in BPCF such that $e \longmapsto_n^* v$ where v is not $\bot$, let e' be the UPCF expression derived from removing the function bounds. Then $e' \longmapsto_n v$.

Proof. $\bar{e}'$ cannot need more recursive calls to terminate than $\bar{e}$. If this was the case, then the soundness proof in Section 5.2 would not hold because an initially terminating expression might no longer terminate. Given this, removing the bounds should give a valid UPCF expression. □

Let e be a UPCF expression and $\bar{e}$ be an equivalent BPCF expression which we know can be generated by compactness. Applying the translation defined in Chapter 4 to $\bar{e}$ gives $\bar{e}'$, which we know is equivalent to $\bar{e}$ by the soundness proof. Thus, it is also equivalent to e. By Lemma 4, removing the bounds on the functions gives an equivalent UPCF expression e', which also must be equivalent to e. Thus, our translation can be expanded using compactness and Lemma 4 to also work for general recursive functions in UPCF.

Chapter 6

Related Work

Existing work that transforms higher-order into first order programs falls into two categories: defunctionalization, and closure conversion. In this section, we survey relevant work in both of these categories, highlighting how our work differs from or builds on previous approaches. We also discuss alternate techniques for proving soundness using logical relations.

6.1 Defunctionalization

The original method of defunctionalization introduced by Reynolds essentially eliminates functions, replacing them with data types and a single top-level apply function [16]. Defunctionalization is implemented in the MLton compiler for SML, which improves upon previous work by using a flow analysis to determine sets of lambdas that can occur at the same program point. This allows them to use multiple different sum types for the environments rather than one large sum type, which also reduces the complexity of each corresponding apply function [8]. More recent work has also developed specializing defunctionalization which allows a single HOF to be transformed into multiple first-order functions, specialized for different call sites [7].

While these approaches are useful for compiling to low level languages, they fundamentally change the structure of the code. This is not necessary for our purposes and would make analysis using RaML more difficult. Thus, we do not opt for defunctionalization. However, similarly to these approaches, we use sum types to represent our environments. Like Reynolds's original approach, we use a single sum type for all our environments.

6.2 Closure Conversion

Closure conversion can be implemented as a type-preserving translation, where functions are transformed into pairs of a code pointer and environment [14]. Minamide et al. used a two phase approach to conversion, first translating to an intermediate language with abstract closures, and

then implementing closures using primitives in a second phase. Because we do not need our transformation to be sound for multiple representations of closures, we have only one phase, immediately representing closures using products and sums.

Traditional closure conversion algorithms use existential types for closure representation [2, 10, 14]. This supports separate compilation, but would not be feasible for our purposes because RaML needs access to the types inside the closure to generate constraints on the potential. Since data structures carry potential, without this knowledge, RaML would not be able to infer bounds. This is why we use sum types, combining that idea from defunctionalization with closure conversion.

6.3 Logical Relations

Similar to prior work, our soundness proof uses a logical relation to define program equality. However, unlike most prior work, we do not use a step-indexed logical relation. Step-indexing was introduced by Appel and McAllester. in [4]. Ahmed applied this to System F with mutable referencesin [1], and recursive and existential types in [3]. Biorthogonal logical relations have been used to prove program equality [13, 15] and step-indexing was applied in [6] to show correctness of a simple compiler. Kripke logical relations also had step-indexing applied in [9, 12]. Step indexed logical relations are indexed by the number of steps available for evaluation, which is beneficial when analyzing recursive functions and types. However, we elected to use bounded recursion and compactness instead, which we felt made the soundness proof simpler while still allowing us to prove equivalence.

Chapter 7

Conclusion

7.1 Summary

Currently, RaML fails to infer bounds for partially-applied higher-order functions whose cost depends on free variables rather than explicit arguments. To resolve this, we presented a typed closure conversion translation that packages a function with an environment. Unlike traditional closure conversion algorithms, ours uses a sum type for the environment rather than an existential type. This gives RaML access to the structure of the values in the environment so it can derive resource bounds.

The translation is designed so the number of tick operations is unchanged, meaning cost is preserved. We proved:

- Type preservation: The translated program is well-typed.
- Soundness: The original and translated programs evaluate to extensionally equal results with identical cost. We proved this using a logical relation that encodes resource use in the equivalence relation.

Together, these results show that our translation safely widens RaML's reach to a class of higher-order OCaml programs that were previously not analyzable.

7.2 Future Work

Now that we have proven soundness of the transformation, the next step is to implement it in RaML. Ideally, this will be implemented in the new version of RaML, RaML 2, which also includes other improvements to the original implementation. Once implemented, tests should be performed to identify the compile-time overhead.

Future improvements could also target preventing the global analysis required to generate the sum type for environments. Also, since the sum type could become quite large, we could investi-

gate leveraging flow-directed analyses to create local sum types, similar to the MLton compiler.

Bibliography

[1] Amal Ahmed. *Semantics of Types for Mutable State*. Phd thesis, Princeton University, November 2004. https://www.ccs.neu.edu/home/amal/ahmedthesis.pdf. 6.3

[2] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 157–168. ACM, 2008. doi: 10.1145/1411204.1411227. URL https://doi.org/10.1145/1411204.1411227. 6.2

[3] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006. doi: 10.1007/11693024_6. URL https://doi.org/10.1007/11693024_6. 6.3

[4] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi: 10.1145/504709.504712. URL https://doi.org/10.1145/504709.504712. 6.3

[5] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 25–37. ACM, 1997. doi: 10.1145/258948.258953. URL https://doi.org/10.1145/258948.258953. 1.2

[6] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 97–108. ACM, 2009. doi: 10.1145/1596550.1596567. URL https://doi.org/10.1145/1596550.1596567. 6.3

[7] William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano. Better defunctionalization through lambda set specialization. *Proc. ACM Program. Lang.*, 7 (PLDI):977–1000, 2023. doi: 10.1145/3591260. URL https://doi.org/10.1145/3591260. 6.1

[8] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion

for typed languages. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2000. doi: 10.1007/3-540-46425-5_4. URL https://doi.org/10.1007/3-540-46425-5_4. 1.2, 2.3, 6.1

[9] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012. doi: 10.1017/S095679681200024X. URL https://doi.org/10.1017/S095679681200024X. 6.3

[10] Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in haskell. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 83–92. ACM, 2007. doi: 10.1145/1291201.1291212. URL https://doi.org/10.1145/1291201.1291212. 6.2

[11] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016. ISBN 9781107150300. URL https://www.cs.cmu.edu/%7Erwh/pfpl/index.html. 5.3

[12] Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ML and assembly. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 133–146. ACM, 2011. doi: 10.1145/1926385.1926402. URL https://doi.org/10.1145/1926385.1926402. 6.3

[13] Paul-André Melliès and Jerome Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 82–91. IEEE Computer Society, 2005. doi: 10.1109/LICS.2005.42. URL https://doi.org/10.1109/LICS.2005.42. 6.3

[14] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 271–283. ACM Press, 1996. doi: 10.1145/237721.237791. URL https://doi.org/10.1145/237721.237791. 1.2, 2.2, 2.2.1, 6.2

[15] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.*, 10(3):321–359, 2000. URL http://journals.cambridge.org/action/displayAbstract?aid=44651. 6.3

[16] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. doi: 10.1145/800194.805852. URL https://doi.org/10.1145/800194.805852. 1.2, 6.1