Doctoral Dissertation, 2019

Abstracting Control with Dependent Types

OCHANOMIZU UNIVERSITY
Advanced Sciences,
Graduate School of Humanities and Sciences

CONG Youyou

March, 2019

ABSTRACT

Dependent types are a powerful tool for ensuring safety. By interacting with terms, dependent types are able to precisely encode program specifications, guaranteeing the absence of runtime errors and other unexpected behaviors. Meanwhile, control operators have been extensively used to increase expressiveness. By talking about the surroundings of programs, control operators enable sophisticated manipulation of control flow, yielding a wide range of practical applications.

The two language ingredients are however known to pose various difficulties when mixed up together. Intuitively, the disharmony stems from their contrasting nature: dependent types are used for reasoning purposes and thus must be determined statically, whereas control operators are used to implement dynamic, non-local behaviors. To make their combination meaningful, previous work has imposed a purity restriction on type dependency, that is, types may depend only on effect-free terms.

In this thesis, we build a dependently typed, effectful language called Dellina. Dellina has support for essential features from the mainstream proof assistants, as well as the delimited control operators `shift` and `reset`. Similarly to the existing studies, we restrict types to depend only on pure terms, but additionally, we impose two constraints on the type of contexts surrounding effectful terms, in order to cope with the flexibility of the control operators. These restrictions make the resulting language type sound. We also define a selective CPS translation of the language, and prove that the translation preserves typing. Our key observation is that, in a dependently typed setting, selective translations not only yield efficient programs, but simplify the proof of the type preservation property.

Dellina is the first non-toy language where dependent types and control operators co-exist. To demonstrate its utility, we implement a type-safe evaluator that uses `shift`, `reset`, and dependent types all in a non-trivial manner. Our result further opens the door to integrating `shift` and `reset` into proof assistants. We discuss how we should extend the "proofs-as-programs" view to a language with delimited control, and what we can prove with the control operators.

Acknowledgements

I have been looking forward to writing the acknowledgement section, because there are so many people I would like to thank.

Kenichi Asai, my Ph.D. advisor, introduced me to the beauty of programming languages. He taught me DrRacket before I was exposed to “mainstream programming”, and control operators before I knew CPS. This made me think functionally and in direct style. After I joined his group, Kenichi gave me limitless freedom to pursue my interests, and has been a great listener of my ideas and problems. Through working with Kenichi, I learned not to be afraid of expressing my individuality. This had a crucial impact on my way of writing, talking, and even playing the piano—indeed, the most memorable “review” I got from Kenichi is the one on my performance of Chopin’s Barcarolle, which looked like a “weak reject”.

Daisuke Bekki, who supervised me during my Master’s studies, is the person who suggested this thesis topic. In 2013, Daisuke showed me how continuations and dependent types are useful in linguistics, leaving me with the question “how can we integrate these into a single programming language?” Four years later, I finally started to work on this challenge, and soon I realized that it was an extremely interesting topic to exlore. I also like Daisuke’s unique advices on research. My favorite one is “the best way to learn something new is to write a textbook on that topic”, from which I can see how he has been widening his expertise.

I was very fortunate to have multiple opportunities to visit universities outside Japan. My first internship was hosted by Chris Barker at New York University. Chris is amazingly good at explaining things for non-experts. His Lambda Seminar was one of the most enjoyable lectures I have ever participated in; I often recall his words and examples when I write papers or prepare talks.

My second destination was Northeastern University. The three months I spent there were critical to both this thesis and my future career. Matthias Felleisen, who hosted me at Northeastern, knows what is beneficial to students better than anyone else. He always asks his students to do more, which often goes beyond what they want or are able to do, but that is because he really cares about them.

A few months later, I started another internship at Chalmers University of Technology, hosted by Andreas Abel. I liked working in the department’s lunch room, waiting for the appearance of the great magician—Andreas knows everything about Agda; he always instantly identified what was wrong with my code and solved it in just a few minutes. These “kitchen magic shows” were one of the sweetest memories of my Ph.D. life.

Right before graduation, I visited Tiark Rompf at Purdue University. Tiark taught me the low-level behavior of programs, which I had never studied seriously before. Thanks to his advices and encouragement, I could co-author a paper with Tiark and his students, which was much more than what I had expected out of a one-month visit.

Besides the four hosts of my internships, William J. Bowman, who I met at Northeastern, made a noteworthy contribution to my Ph.D. I got to know William when I was ramdomly watching videos from ICFP 2015. At that time, he was just “a clever student studying thousands of miles away”. Several months later, William became my collaborator; more precisely, Matthias put me under supervision of William. This experience meant a lot to me. It both broadened and deepened my knowledge around continuations and type theory, giving me the confidence to dive into the research question posed by Daisuke years ago. But probably more importantly, I learned from William what it means to get a Ph.D. degree.

Most of my internships were funded by the Leading Graduate School Program of Ochanomizu University. Without those oversea experiences, I would not have been able to write this dissertation.

Back to the “local” community, there are two persons who significantly influenced me. Yukiyoshi Kameyama helped me notice my insufficient comprehension of various concepts. In fact, one of my must-do’s at conferences is to share my recent ideas with Yukiyoshi and ask for his feedback. Koji Mineshima is a living encyclopedia of linguistics and type theory. Every time I run up to Koji with a question, he instantly finds the right references for me, making me wonder how many papers and books he has studied so far.

And of course, my pleasant Ph.D. was due to the wonderful collegues at Asai and Bekki laboratories. I would especially like to thank Kanae Tsushima and Ribeka Tanaka, who advised me on my research and future career.

Lastly, I gratefully acknowledge many more people who shaped my work in various ways but are not mentioned in this list. Your support is part of the reason I am here today, and I hope to express my gratitude when I have a chance.

Typographical Conventions

In this thesis, we use colors and fonts to distinguish between different languages.

- non-bold, blue, sans-serif font: expressions in the Dellina- and Dellina languages, which have control operators
- **bold, red, serif font**: expressions in the target language of the CPS translation, which is free from control effects
- *non-bold*, *black*, *italic font*: expressions in languages other than the source or target of the CPS translation

We also use different brackets for source and target typing rules.

- (ROUND BRACKETS): typing rules for Dellina- and Dellina
- [SQUARE BRACKETS]: typing rules for the post-CPS language

Contents

1 Introduction **13**
1.1 Ensuring Safety: Dependent Types 14
1.2 Increasing Expressiveness: Delimited Control 20
1.3 Mixing Dependency and Control 27
1.4 Contributions and Outline 31

2 Shift, Reset, and Dependent Types **35**
2.1 Typing Programs with Shift and Reset 35
2.2 Simply Typed Shift and Reset 43
2.3 Three Restrictions on Type Dependency 46
2.4 Dependent Types and Effects 50

3 The Dellina- Language **54**
3.1 Syntax 55
3.2 Evaluation, Reduction, and Equivalence 57
3.3 Typing 66
3.4 Metatheory 75
3.5 Examples 101

4 CPS Translating Dellina- **104**
4.1 Challenges of CPS Translation 105
4.2 Past Solution: Answer-type Polymorphism + CPS Axioms 108
4.3 Our Solution: Selective Translation 112
4.4 Target Language 115
4.5 CPS Translation 124

4.6 Proof of Type Preservation 135
4.7 Related Work 148

5 The Dellina Language 153
5.1 Polymorphism and Type Operators 153
5.2 Prop, Set, and Universe Hierarchy 162
5.3 Inductive Datatypes 174
5.4 Local Definitions 190
5.5 Example 200

6 Call-by-name Dellina- 203
6.1 Syntax 204
6.2 Evaluation, Reduction, and Equivalence 205
6.3 Typing 213
6.4 CPS Translation 224

7 Logical Understanding of Dellina 232
7.1 Logical vs. Non-logical Objects 232
7.2 Intuitionistic vs. Classical Logic 234
7.3 Related Work 237

8 Conclusion and Perspectives 241
8.1 Multiple Effects 242
8.2 Control Effects at Higher Levels 245
8.3 Propositional Equality 246
8.4 Specialization by Unification 248

A Control Operators and Dependent Types in Natural Languages 250
A.1 Introduction to Natural Language Semantics 250
A.2 Solving NL Challenges using PL Techniques 253
A.3 Dellina for Natural Language Semantics 257

Chapter 1

Introduction

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

Lewis Carroll, *Alice in Wonderland*

Safety and *expressiveness* are key aspects of a programming language. Safety means the ability to ensure that a program works in the intended way. This includes the absence of runtime errors, as well as properties specific to individual programs, such as "when appending two lists of length n_1 and n_2, we obtain a list of length $n_1 + n_2$. Expressiveness, on the other hand, means the ability to express sophisticated behaviors. For instance, if we wish to rewrite the value of a variable, it is a good idea to work in a language that supports mutable references, rather than passing around an additional argument in every function.

This thesis aims at convenient implementation of safe programs. To achieve this goal, we build a language that provides *dependent types* and *delimited control operators*. The present chapter is devoted to give an overview of this work. We begin with an introduction to dependent types and control operators (Sections 1.1 and 1.2), focusing on how they are useful in everyday programming. We next discuss known challenges with combining them (Section 1.3), and the solutions reported by previous studies. With these in mind, we list specific contributions of our work (Section 1.4).

1.1 Ensuring Safety: Dependent Types

1.1.1 Simple Types

In programming languages, *types* are a notion that we use to describe properties of programs. For example, an integer 1 has type int, and a boolean true has type bool.

The existence of types gives rise to the notion of *well-typed programs.* For instance, $1 + 1$ is well-typed, whereas $1 + \mathsf{true}$ is not, since true is not a valid argument of the addition operator. In statically typed programming languages[1], such as ML and Haskell, well-typedness of programs is examined by a *type checker*, and only those programs that passed type checking may be executed. This means $1 + \mathsf{true}$ will never be executed in typed languages; instead, the type checker will report a *type error* before running it. Thus, types enable early detection of runtime errors, making programs more reliable.

Types also induce a beautiful connection between functional programming languages and logics, which is known as the *propositions-as-types principle* or *Curry-Howard isomorphism.* Consider the simply typed λ-calculus (STLC) [16], whose types and terms are generated by the following grammar:

$$\begin{array}{ll}
A ::= B & \text{base type} \\
\quad \mid A \to A & \text{function type} \\
e ::= x & \text{variable} \\
\quad \mid \lambda\, x.\, e & \text{abstraction} \\
\quad \mid e\ e & \text{application}
\end{array}$$

STLC is the core basis of functional programming languages, consisting of variables, functions, and applications. Decades ago, Curry [53] and Howard [95] discovered that the calculus corresponds to the implicational fragment of propositional logic, where types are propositions and terms are proofs of the proposition their type represents[2]. As a simple example, the function type $A \to A$ corresponds

[1] There are also dynamically typed languages, where type checking takes place "on the fly."

[2] The connection was in fact developed by many other researchers as well; see Section 30.5 of Harper [90] for a complete list of contributors.

to the law of identity, and the identity function $\lambda\, x.\, x$ of type $A \to A$ serves as a proof of this law. When viewing types and terms in this way, building a well-typed program can be considered as writing a mathematical proof, and checking well-typedness of a program is understood as verifying the correctness of a proof.

1.1.2 Enriching Type Systems

The type system of STLC is "simple" in that the type and term languages are mutually independent, *i.e.*, types do not refer to terms, and vice versa. To obtain a more expressive type system, one could add *polymorphism*, by allowing term-level functions to abstract over types:

$$
\begin{array}{ll}
A ::= \ldots & \\
\quad \mid \alpha & \text{type variable} \\
\quad \mid \forall\alpha.\, A & \text{polymorphic type} \\
\quad \mid A\ A & \text{type-to-type application} \\
e ::= \ldots & \\
\quad \mid \lambda\, \alpha.\, e & \text{term-level type abstraction} \\
\quad \mid e\ A & \text{term-to-type application}
\end{array}
$$

The resulting calculus is known as *System F* [84], and allows definition of the polymorphic identity function `pid`:

$$\texttt{pid} \ \overset{\text{def}}{\equiv}\ \lambda\, \alpha.\, \lambda\, x.\, x$$

The function has type $\forall\alpha.\, \alpha \to \alpha$, meaning that we can apply `pid` to a term e of any type, by instantiating α to the type of e. For instance, the following programs are both well-typed:

$$\texttt{pid}\ \mathsf{int}\ 1 \qquad\qquad \texttt{pid}\ \mathsf{bool}\ \mathsf{true}$$

We can also define polymorphic lists in the extended language. That is, we let the type constant list carry a parameter representing the type of its elements, so that [1; 2; 3] inhabits list int, and [true] inhabits list bool[3].

The extension by polymorphism has introduced dependency of terms on types, but the type language is still defined independently of the term language. From a logical perspective, the extended calculus corresponds to second-order propositional logic, where one can quantify over predicates.

System F is close to the underlying type system of the mainstream functional languages. However, sometimes it is not powerful enough to express the exact intension of the programmer. Consider the following head function, which returns the first element of a given list of integers:

```
(* head : list int -> int *)
let head lst = match lst with
  [] -> ?
| first :: rest -> first
```

When the given list is non-empty, the task is easy, but what if the list is empty? Since we can only take the head of a *non-empty* list, we wish to explicitly state that the head function can never be applied to an empty list. Unfortunately, in System F, we cannot rule out this unintended use along the same lines as we did for $1 +$ true. The reason is that the type list int is inhabited by lists containing *any* number of integers. This means an empty list is a valid argument to the head function, hence we must account for this case to make the pattern matching exhaustive.

Now the reader might ask: what kind of extension do we need to make the type system expressive enough to rule out taking the head of an empty list? The answer is: use *dependent types*! Dependent types are literally types that depend on terms; in other words, they represent program properties that are determined by terms. Supporting dependent types requires allowing application of types to terms, making the type and term languages mutually dependent:

[3]In ML and its dialects, list int is represented as int list. While the latter is more friendly when read as a natural language expression, the former makes it easier to view the type as an application.

$$
\begin{array}{lll}
A ::= \ldots & & \\
\quad \mid \lambda\, x.\, A & & \text{type-level term abstraction} \\
\quad \mid A\ e & & \text{type-term application}
\end{array}
$$

With this extended grammar, we can define a dependent variant of the list type L n. The type consists of a type constant L, and an index argument n, which is a natural number (of type $\mathbb{N}$). When composed together, L n represents a type inhabited by lists containing n natural numbers[4]. That is, if we inherit the standard list notation, [ ] has type L 0, and [1; 2; 3] has type L 3.

Using the depedent list type, we next re-define the type of the `head` function:

$$\texttt{head} : \Pi\, n : \mathbb{N}.\, \mathrm{L}\ (n+1) \to \mathbb{N}$$

The type tells us that `head` is a function from lists of length n+1 to natural numbers, *i.e.*, `head` only accepts a *non-empty* input. With this definition, application of `head` to [ ] is no longer well-typed, since [ ] has type L 0 and there is no natural number n satisfying $n+1 = 0$. In dependently typed languages supporting "smart" pattern matching [86, 42, 43], the user can skip the empty branch in the definition of `head`, without getting the "non-exhaustive pattern matching" warning.

Shifting the viewpoint to logics, dependency on terms allows types to express statements in *predicate* logic. For instance, function types $A \to B$ are refined to $\Pi\, x : A.\, B$, where type B may contain free occurrences of term-level variable x. From a logical point of view, the function type corresponds to universal quantification $\forall x \in A.\, B$. That is, when we have a function $\lambda\, x.\, b$ of type $\Pi\, x : A.\, B$, we can construct a proof of $B[a/x]$ via function application $(\lambda\, x.\, b)\ a$ for any $a : A$. Similarly, we have a dependent version of pair types $\Sigma\, x : A.\, B$, where x can occur free in type B. This type is interpreted as existential quantification $\exists x \in A.\, B$. That is, when we have a proof b of $B[a/x]$ for some $a : A$, we can view a as witnessing the existence of x satisfying B, and abstract this fact as an existential statement by constructing a pair $(a,\, b)$ of type $\Sigma\, x : A.\, B$.

[4]In a dependently typed language, natural numbers are often defined as separate data from integers.

1.1.3 Programming and Proving with Dependent Types

As we saw, dependent types greatly broaden the scope of expressible program specifications. This aspect is useful in two closely related but slightly different ways, as we describe below.

Building Correct-by-Construction Programs One typical use of dependent types is to build programs that are *correct by construction*. Let us consider the `append` function, which concatenate given lists. Intuitively, when a correctly defined `append` function is applied to two lists of length n_1 and n_2, we will obtain a list of length n_1+n_2. In a dendently typed language, we can express this correct behavior in the function definition using types. Specifically, we can declare `append` as having the following type:

$$\mathsf{append} : \Pi\, n_1\ n_2 : \mathbb{N}.\, \mathrm{L}\ n_1 \to \mathrm{L}\ n_2 \to \mathrm{L}\ (n_1 + n_2)$$

Observe how we use the indices of the input types to specify the output type. When the definition of `append` has passed type checking, we know that the `append` function must work correctly, that is, it produces a list of the right length.

Reasoning About Programs Another use of dependent types, which seems to be more popular, is to *reason about* programs that are possibly written in the non-dependent subset of the language. Consider the simply typed `append-smpl` function that concatenates two integer lists. The function is given the type $\mathsf{list\ int} \to \mathsf{list\ int} \to \mathsf{list\ int}$. Unlike the `append` function we discussed above, `append-smpl` does not say anything about the length of the input and output lists. However, in a dependently typed language, we can still show that the function works correctly, by proving its property separately from the function definition. That is, after defining `append-smpl`, we prove the correctness theorem `append-correct`, by writing a program of the following type (where `length` returns the length of a given list):

$$\begin{array}{l} \mathtt{append\text{-}correct} : \\ \quad \Pi\, l_1\ l_2 : \mathsf{int\ list}.\, \mathtt{length}\ (\mathtt{append\text{-}smpl}\ l_1\ l_2) = \mathtt{length}\ l_1 + \mathtt{length}\ l_2 \end{array}$$

The type says: for any integer lists l_1 and l_2, `append-smpl` produces a list whose length is the sum of the length of l_1 and l_2. The equality symbol $=$ is a type constructor, which has a single inhabitant `refl` $: a = a$ representing reflexivity. When we have found a term that has the above type (*i.e.*, if we could give a definition of `append-correct`), we know that `append-smpl` actually satisfies the expected property.

The `append-correct` example uses dependent types to represent a theorem that holds of the `append-smpl` function. In general, dependently typed languages serve as a *proof assistant*, aiding construction of 100% trustable proofs[5]. For this reason, dependent languages such as Coq [160], Agda [128], and Isabelle [127] are extensively used in the context of formal verification. Below are some of the notable contributions:

- CompCert [111]: a compiler for a large subset of the C language, accompanied by a Coq proof ensuring that the output program behaves the same as the input.
- CertiCoq [5]: a compiler of Coq that is mechanically verified by Coq itself, featuring efficiency, reliability, and compatibility with other languages.
- CertiKOS [89]: a framework for building OS kernels that are free from buffer overflow, null-pointer dereference, and other runtime errors.
- CertiCrypt [20]: a toolset for constructing machine-checked cryptographic proofs, where the interaction between an adversary and a cryptsystem is represented as probabilistic programs.
- DeepSpec [161]: a project aiming to build bug-free software that "actually does what it is supposed to do," by combining all of the ideas listed above.

1.1.4 Alternatives to Dependent Types

There are several alternative ways to make a type system richer. One approach is to simulate dependent types using *singleton types* [123]. Roughly speaking, singleton types are types indexed by a reflection of some term-level computation.

[5]Of course, we have to assume that the underlying logic of the language is consistent.

The simplest example is indexed integers $\breve{n}$: $\mathsf{sint}\ \hat{n}$, where $\hat{n}$ is the type-level counterpart of the term-level integer n.

A similar option is to adopt Generalized Algebraic Data Types (GADTs) [40, 173]. GADTs generalize ordinary ML datatypes in that they can be indexed by types of the meta-language. For instance, the indexed list type can be defined by first declaring natural numbers as types, and then declaring L as a type-returning function accepting a natural number.

The indexing mechanism of singleton types and GADTs enables implementation of non-toy programs with non-trivial invariants, such as safe database interface [71] and typeful normalization by evaluation [58]. Moreover, it has proven that languages with these types enjoy the *phase distinction* property [35, 123], that is, type checking and execution can be separated into two distinct phases. However, singletons and GADTs are obviously not as expressive as full dependent types, and working with them often makes program implementation less elegant.

A third variant of richer types are known as *refinement types* [142, 164]. Refinement types encode data properties by means of refinement predicates: for instance, the type of natural numbers can be represented as $\{v : \mathsf{int} \mid v \geq 0\}$. Note that the type mentions terms v and 0, just like dependent types. The main difference between refinement and dependent types is that the former restricts predicates to decidable ones. This restriction reduces the burden of manually providing proof terms, which is often required in full-spectrum dependent languages, but at the same time it limits the scope of expressible predicates.

In this thesis, we take the most "heavy-weight" approach, and focus our attention to dependent types. Although this decision complicates the language design, we believe that it is crucial for making our language truly practical.

1.2 Increasing Expressiveness: Delimited Control

Continuations represent the rest of the computation. Suppose we are evaluating $2 * 3$ in the program $1 + (2 * 3) - 4$. The current continuation is the computation "given the value of $2 * 3$, add 1 to it and subtract 4 from the result." The reader may think of this continuation as a computation with a hole (denoted [.]), namely $1 + [.] - 4$. It can also be understood as a function $\lambda\, x.\, 1 + x - 4$, which abstracts over the expression being executed (*i.e.*, the hole).

1.2.1 Handling Continuations in Continuation-Passing Style

While continuations are everywhere in our programs, they are usually not visible to us. If we wish to access and manipulate continuations, we have two options. One is to transform programs into *continuation-passing style (CPS)* [137]. CPS programs are different from ordinary, *direct-style (DS)* programs in that every function receives an additional continuation argument representing what to do after the function has returned a value. This allows us to simulate a wide range of side effects, such as exceptions, non-determinism, and mutable state. Here we demonstrate how to express exception raising in a CPS program. We start with the following direct-style function:

```
let rec times lst = match lst with
  [] -> 1
| h :: t -> if h = 0 then raise 0
            else h * (times t)
```

The `times` function computes the product of a given list of integers. When the list is empty, the function returns 1, and when it is not, the function checks whether the first element `h` is 0 or not. If this condition does not hold, we compute the answer in the normal way. Otherwise, we know that the ultimate answer must be 0, hence we want to ignore the rest of the computation (*i.e.*, computing the product of numbers we have traversed so far) and directly return 0 to the top-level. In the `times` function, we skip unnecessary computation by throwing an exception via the `raise` construct.

We next define `times-cps`, which is a CPS counterpart of the `times` function:

```
let rec times-cps lst k = match lst with
  [] -> k 1
| h :: t -> if h = 0 then 0
            else times-cps t (fun v -> k (h * v))
```

The `times-cps` function takes in a list `lst`, *and* a continuation `k`. When programming in CPS, we basically follow two principles: (i) to return a value v, we pass

v to the current continuation; and (ii) to compute a non-value e, we supply the computation with a continuation representing what to do when we have obtained the value of e. The empty clause is the return-case: we pass the value 1 to the continuation `k`. The `else`-clause of the cons branch is the compute-case, we supply the computation `times-cps t` with the continuation "given the product `v` of the rest of the list, return the value `h * v` to the current continuation `k`." The `then`-clause, however, does not fit into the CPS patterns: we only have a plain value 0, which is not applied or passed to a continuation. What is happening here is that we are *discarding* the continuation, which represents the redundant computation we do not want to execute. Since the continuation is now available as an explicit object `k`, we can drop it without using the language's exception facilities.

When running the `times-cps` function, we have to provide an initial continuation, which represents what we want to do with the value returned by the function. If we are just intereted in the value itself, we supply an empty continuation, *i.e.*, the identity function. Below we show two evaluation sequences of the `times-cps` function, corresponding to the normal and exception cases, respectively:

```
times-cps [1; 2; 3] (fun x -> x)
times-cps [2; 3] (fun v -> (fun x -> x) (1 * v))
times-cps [3] (fun v -> (fun v -> (fun x -> x) (1 * v)) (2 * v))
times-cps [] (fun v -> (fun v -> (fun v -> (fun x -> x) (1 * v)) (2 * v)) (3 * v))
(fun v -> (fun v -> (fun v -> (fun x -> x) (1 * v)) (2 * v)) (3 * v)) 1
(fun v -> (fun v -> (fun x -> x) (1 * v)) (2 * v)) 3
(fun v -> (fun x -> x) (1 * v)) 6
(fun x -> x) 6
6

times-cps [1; 0; 3] (fun x -> x)
times-cps [0; 3] (fun v -> (fun x -> x) (1 * v))
0
```

By writing programs in CPS, we obtain the ability of accessing continuations at any point of execution. However, naïve CPS translations are known to make programs slower, as they generate a number of non-source abstractions and applications. To avoid this problem, researchers have developed various *optimizing* CPS translations, which produce more compact and efficient programs [57, 126, 140, 61, 12].

1.2.2 Handling Undelimited Continuations in Direct Style

An alternative approach to handling continuations is to use *control operators*. Control operators turn continuations into user-accessible objects *as needed*, in other words, they provide access to continuations in direct-style programs. Among different variants of control operators, let us first look at Felleisen et al.'s $\mathcal{C}$ operator [75, 74]:

$$\begin{aligned}
&1 + (\mathcal{C}k.\, 2 + k\ 45) - 4 \\
&= (2 + k\ 45)[\lambda\, x : \mathtt{int}.\, \mathtt{abort}\ (1 + x - 4)/k] \\
&= 1 + 45 - 4 \\
&= 42
\end{aligned}$$

The $\mathcal{C}$ operator captures the whole continuation surrounding it, namely $1 + [.] - 4$, and turns it into an abortive function $\lambda\, x : \mathtt{int}.\, \mathtt{abort}\ (1 + x - 4)$. Then, it binds variable k to this continuation, and computes the body expression $2 + k\ 45$. When the value 45 is passed to k, the addition of 2 is discarded by the `abort` operator. Thus, the program reduces to 42.

Using $\mathcal{C}$, we can define a direct-style variant of the `times-cps` function, which supports efficient exception raising without using the `raise` construct:

```
let rec times-c lst = match lst with
  [] -> 1
| h :: t -> if h = 0 then C k 0
            else h * times-c t
```

Observe that the function takes one single argument; it does not require a continuation as an additional parameter. The continuation is made explicit only in the third line, where the $\mathcal{C}$ operator drops the rest of the work and returns 0 to the top-level. Other parts of the program are all written in direct style, without the clutters found in `times-cps`.

1.2.3 From Undelimited to Delimited Continuations

So far, we have been discussing *undelimited continuations*, *i.e.*, continuations that represent the *entire* rest of the computation. However, what we care in practice is often *part of* the computation to be done—just like we say things like "the rest of the day" instead of "the rest of my entire life." This gives rise to the notion of *delimited continuations*, *aka partial* or *composable* continuations [72, 55]. A delimited continuation is a continuation with a limited extent. Consider $\langle 1 + (2 * 3)\rangle - 4$, where $\langle\,\rangle$ represents the scope of the computation we are interested in. When evaluation of $2*3$ is happening, the current delimited continuation is $1+[.]$, or equivalently, the function $\lambda\, x.\, 1+x$. The subtraction of 4 is not included because we have delimited the scope of the relevant computation.

Delimited continuations are handled via a pair of a continuation reifier (like $\mathcal{C}$) and a continuation delimiter (like $\langle\,\rangle$). There are a variety of delimited control operators in the literature, differing in (i) whether the extent of a captured continuation is determined statically or dynamically; and (ii) whether the association between the reifier and the delimiter is established selectively or unselectively. In this thesis, we use Danvy and Filinski's `shift` and `reset` operators [56], which are classified as static and unselective. The behavior of these operators is simple: `shift` grabs a continuation delimited by the nearest enclosing `reset`. Consider the following program, where $\mathcal{S}$ and $\langle\,\rangle$ denote `shift` and `reset`, respectively:

$$
\begin{aligned}
&\langle 1 + \mathcal{S}k.\, 2 + k\ 45\rangle - 4 \\
&= \langle 2 + k\ 45[\lambda\, x.\, \langle 1 + x\rangle / k]\rangle - 4 \\
&= \langle 2 + 46\rangle - 4 \\
&= \langle 48\rangle - 4 \\
&= 48 - 4 \\
&= 44
\end{aligned}
$$

The `shift` operator captures the delimited continuation within the `reset` clause, namely $1+[.]$, and “shifts” the control to its body $k\ 45$, with k bound to the captured continuation $\lambda\, x : \mathsf{int}.\, \langle 1 + x \rangle$. When the expression inside `reset` has reduced to a value, we remove the surrounding `reset`, and perform the subtraction. Thus we obtain 44.

There are two differences between the above program and the one that uses the $\mathcal{C}$ operator. First, while $\mathcal{C}$ captures the whole continuation $1+[.]-4$ it is surrounded by, $\mathcal{S}$ captures part of the continuation that contains only the addition of 1. Second, whereas a $\mathcal{C}$-captured continuation is abortive, an $\mathcal{S}$-captured continuation returns normally and composes with the context $2 + [.]$ surrounding the application of k (hence delimited continuations are called composable). Indeed, although the word “delimited” may sound negatively, delimited continuations are more expressive than undelimited continuations [69, 140, 66], and in this sense, undelimitedness should be understood as a limitation.

1.2.4 Applications of Delimited Continuations

Delimited continuations have a diverse range of applications. Here we give three examples, each showing a different use of continuations.

Duplicating Continuations: Non-determinism The `times` example from Sections 1.2.1 and 1.2.2 simulates exception raising by calling the continuation *zero* times. If we call a continuation *multiple* times with different arguments, we can make programs behave non-deterministically. Let us look at the `either` function below, which is a simplified version of the `amb` operator in the Scheme language:

```
let either a b = shift k (k a; k b)

> reset (print_int (either 1 2))
12
- : unit = ()
```

Given two arguments `a` and `b`, the function first applies the captured continuation `k` to `a`, and then to `b`. If we call this function in a printing context, we will see two outputs `a` and `b` printed in this order. The non-deterministic behavior enables us to implement backtracking programs [56]: *e.g.*, we can find pythagorean triples [11], and solve the famous N-queens puzzle [117].

Suspending Continuations: Coroutines A captured continuation need not be called at once; we can temporarily save it somewhere for future use. The following function serves as the "yield" construct from languages with support for coroutines:

```
let yield v = shift k (v, k)
```

The `yield` function packs a given value `v` and the current continuation `k` in a pair, allowing us to do some other work first and get back to the original computation later. Notice that we are returning the continuation as part of the answer produced by `yield`. This shows that continuations captured by the `shift` opeartor has a *first-class* status (*i.e.*, they may escape).

Suspension and resumption of continuations help us describe Web interactions, where the server waits for the client's input, with which it continues the rest of the work [138]. This is the idea underlying the PLT Scheme Web Server [108], and as reported by Flatt et al. [78], delimited (instead of undelimited) continuations are particularily suited for concurrent handling of multiple requests.

Reordering Continuations: Program Transformation Uses of captured continuations in a non-tail position have the effect of reordering computation. That is, resumption of a continuation, which always happens at the last step when evaluating a pure term, can be followed by some additional computation specified by the programmer. This allows us to implement the `reverse` function in the following way:

```
let rec visit lst = match lst with
  [] -> []
| h :: t -> shift k (h :: k (visit t)) in
let rec reverse lst = reset (visit lst)
```

The reader may find this program a bit tricky, but the key observation is that we are swapping consing of the first element `h` and resumption of the captured continuation `k`. Under the call-by-value evaluation, we must reduce `visit t` to a value before applying the continuation, and if `t` was a non-empty list of the form `h' :: t'`, it may capture the `h`-consing context and store it to a continuation variable `k'`. Now, it is easy to see that the order of `h` and `h'` is reversed. Note that if we replace the body of `shift` by `k (h :: (visit t))`, `reverse` would be equivalent to the identity function. The trick used in `reverse` extends to *A-normalization* [77], a program optimization technique for specifying the order of evaluation. It has proven that `shift` and `reset` enable direct-style implementation of partial evaluation [54, 9] and dynamic code generation [101], where scope-safety is maintained by inserting `reset` in appropriate places.

1.3 Mixing Dependency and Control

We have seen that dependent types and control operators are useful in different ways: the former provide safety guarantees, whereas the latter allow efficient coding. Given this fact, one question naturally comes into our mind: why not using dependent types and control operators together? Couldn't we write safe programs in a convenient way? It turns out that integrating these elements into a single language is not an easy task. In this section, we review the challenges of designing a dependently typed language with control effects, as well as the solutions reported so far.

1.3.1 Σ-types and `call/cc` Lead to Inconsistency

In 2005, Herbelin [93] built a language consisting of strong Σ-types (dependent pairs with first and second projections), equality types, and `call/cc`, which is an undelimited control operator similar to Felleisen's $\mathcal{C}$. The resulting language turned out to be inconsistent, in that every proposition is provable. Specifically, Herbelin used the following existential proof term to derive inconsistency:

$$p \overset{\text{def}}{\equiv} \texttt{call/cc}\ k\ (0,\ \texttt{throw}\ k\ (1,\ \texttt{refl})) : \Sigma\, x : \mathbb{N}.\, x = 1$$

Here, $\texttt{throw}\ k\ e$ can roughly be understood as applying the (abortive) continuation k to the argument e. To see what causes inconsistency, let us first observe the reduction of $\mathsf{fst}\ p$:

$$\begin{aligned}\mathsf{fst}\ p &= \texttt{call/cc}\ k\ (\mathsf{fst}\ (0,\ \texttt{throw}\ k\ (\mathsf{fst}\ (1,\ \texttt{refl})))) \\ &= \texttt{call/cc}\ k\ 0 \\ &= 0\end{aligned}$$

The reduction sequence shows that `call/cc` makes two copies of the surrounding context $\mathsf{fst}\ [.]$, placing one of them around the body, and the other around the argument of `throw`. The computation reduces to 0, which stands for the witness of the proof p. This witness, however, is an incorrect one. Observe the type of p: it says there is some term that is equal to 1. The proof p has this type because it eventually drops the context $(0,\ [.])$ and returns $(1,\ \texttt{refl})$ to the continuation k (that is, it backtracks). For $(1,\ \texttt{refl})$ to be well-typed, `refl` must represent the equality between something and 1, hence the type. Now, since $\mathsf{snd}\ p$ is a proof witnessed by a specific term $\mathsf{fst}\ p$, this "something" must be instantiated to $\mathsf{fst}\ p$, and since we have $\mathsf{fst}\ p = 0$, we can deduce $0 = 1$! Thus, Herbelin concluded that undelimited control operators are incompatible with strong Σ-types and any dependently-eliminated inductive types.

A further study on the above example revealed that the inconsistency is caused by type dependency on an effectful proof. As we saw, p has two different witnesses depending on its surrounding context, due to its backtracking ability. The witnesses are unexpectedly joined up via an application of the second projection, yielding the absurd proposition $0 = 1$. Based on this observation, Herbelin [94] defined a syntactic category called the *negative-elimination free (NEF)* fragement, consisting of terms free from control effects. By restricting type dependency to NEF terms, Herbelin successfully built a classical calculus where one can constructively prove two weak instances of the axiom of choice.

1.3.2 CPS Translation Fails to Preseve Typing

Another issue with control effects and dependent types was noticed by Barthe and Uustalu [24]. Instead of control operators, they considered a CPS translation, which, as we saw, gives us the same power as control operators do. In examining different constructs from dependently typed languages, Barthe and Uustalu found that the standard call-by-name CPS translation[6], which corresponds to Kolmogorov's double-negtion translation [106], fails to be type-preserving in the presence of strong Σ types (dependent pairs with projections). To see why this is the case, let us look at the translation of second projection $\mathsf{snd}\ e$, which is one of the problematic cases. Below, $e^{\div}$ and A^{+} denote the CPS counterparts of term e and type A, and $\neg A$ is a shorthand for the type $A \to \bot$, representing an arbitrary continuation accepting an A-value:

$$(\mathsf{snd}\ e)^{\div} = \lambda\, k : \neg(B[\mathsf{fst}\ e/x])^{+}.\, e^{\div}\ (\lambda\, v : (\Sigma\, x : \neg\neg A^{+}.\, \neg\neg B^{+}).\, (\mathsf{snd}\ v)\ k)$$

The translation encodes the reduction of $\mathsf{snd}\ e$: we first evaluate e, and when it has reduced to a pair $v = (e_1, e_2)$, we return $\mathsf{snd}\ v = e_2$ as the result.

Now, suppose e has type $\Sigma\, x : A.\, B$. According to the typing rule of the second projection, $\mathsf{snd}\ e$ has type $B[\mathsf{fst}\ e/x]$. For the translation to preserve types, the whole translated term must have type $\neg\neg(B[\mathsf{fst}\ e/x])^{+}$. By the induction hypothesis, we have $v : \Sigma\, x : \neg\neg A^{+}.\, \neg\neg B^{+}$, which implies $\mathsf{snd}\ v : \neg\neg(B^{+}[\mathsf{fst}\ v/x])$. We have to show that application $(\mathsf{snd}\ v)\ k$ has type $\bot$. If the translation commutes with substitution, we could rewrite the domain $(B[\mathsf{fst}\ e/x])^{+}$ of k to $B^{+}[(\mathsf{fst}\ e)^{\div}/x]$, but clearly, k is not a valid argument to $\mathsf{snd}\ v$, since $(\mathsf{fst}\ e)^{\div}$ is distinct from $\mathsf{fst}\ v$. As Barthe and Uustalu point out, a similar type mismatch occurs when CPS translating dependent elimination of sum types $A + B$.

Since Barthe and Uustalu's negative result, CPS translation of dependently typed languages had been left open for years, until the recent breakthrough made by Bowman et al. [34]. They identify two sources of the type mismatch dis-

[6]Call-by-name is an evaluation strategy where we do not reduce arguments before function application: *e.g.*, when evaluating $(\lambda\, x.\, x)\ (1 + 2)$, we first obtain $1 + 2$ by substituting $1 + 2$ for x, and then obtain 3. Another popular strategy is call-by-value, where β-reduction happens after the arguments have been reduced to values: *i.e.*, we first reduce $(\lambda\, x.\, x)\ (1 + 2)$ to $(\lambda\, x.\, x)\ 3$, and then to 3.

cussed above. First, CPS translations change the interface to values. In the pre-CPS world, we obtain the value of a term by simply evaluating the term, but in the post-CPS world, every term has turned into a continuation-awaiting function, whose evaluation does not happen unless it is supplied a continuation. The second problem is in the design of the double-negation translation: when we supply a CPS function with a continuation, we would obtain a value of type $\bot$, which should not have any inhabitant. Since type checking a dependently typed term involves evaluation of terms appearing in types, disruption of the computation-to-value interface greatly affects typability of CPS programs. As a remedy, Bowman et al. give up the fixed answer type and use instead a polymorphic one. That is, they translate every term into a computation of type $\forall \alpha.\, (A \to \alpha) \to \alpha$. This translation gives us a type-safe interface to values through the identity continuation, together with a *free theorem* [165] that helps us reason about CPS terms.

1.3.3 CBV Application Breaks Subject Reduction

Miquey [121] found a third challenge in building a dependently typed variant of the $\lambda\mu\tilde{\mu}$-calculus [52], a calculus with facilities for expressing and manipulating continuations. Miquey showed that, the subject reduction property, which states that reduction preserves types, does not hold under the call-by-value evaluation strategy:

$$\langle \lambda\, x.\, b \| a \cdot e \rangle \rightsquigarrow \langle a \| \tilde{\mu} x.\, \langle b \| e \rangle \rangle$$

Programs (called *commands*) in the $\lambda\mu\tilde{\mu}$-calculus are represented as pairs $\langle p \| e \rangle$ of a term p and a continuation e. In the above reduction relation, the left-hand side represents a situation where we have a λ-abstraction $\lambda\, x.\, b$ being applied to the argument a in the context e. Since the calculus is call-by-value, the computation proceeds by reducing a to a value and evaluating b in the context e. This is represented by the command on the right-hand side, where we have a $\tilde{\mu}$ operator binding the value of a. The problem is that the post-reduction command is not well-typed, since the function's body b, whose type depends on x, is paired with the context e, whose hole type depends on a. In Miquey's words, this mismatch is caused by the desynchronization of the typing process with respect to the computation. Recall from Section 1.1 that a dependent function $f : \Pi\, x : A.\, B$ allows constructing a proof of $B[a/x]$ via application $f\ a$. In a call-by-value language, however, β-reduction happens when a has reduced to a value v, hence substitution yields a term of type $B[v/x]$. If the language has no control effects, we know $a = v$ hence $B[a/x]$ and $B[v/x]$ are equivalent, but if the language is effectful, as in $\lambda\mu\tilde{\mu}$, x is not necessarily replaced by the value of a. Indeed, the role of the $\tilde{\mu}$-abstraction is exactly to allow arbitrary values to be substituted for x, but this freedom affects typing in the presence of dependency.

This observation suggests two things. First, dependency of types must be restricted to terms that reduce to a unique value. Second, the binding information of $\tilde{\mu}$-bound variables must be made explicit in the typing rules. Miquey found that the set of terms satisfying the unique value requirement coincides with Herbelin's NEF fragment, and adopts Ariola et al.'s control delimiter $\hat{\mathsf{tp}}$ [8] to define NEF terms. He then introduced a *dependencies list*, which tells us what value is to be substituted for each $\tilde{\mu}$-bound variable. Thus, Miquey recovered the synchronization and obtains the subject reduction property.

1.4 Contributions and Outline

While previous work has shown us how to handle continuations with dependent types, we do not regard this as the end of the story. First, the control operators considered so far are not the most powerful variants, limiting the use of captured continuations. Second, the proposed languages only support a small set of expressions, preventing us from building interesting programs. Third, the CPS translations rely on impredicative polymorphism and parametricity, which are unavailable in certain dependent calculi.

This thesis aims to adress all of the three issues. To this end, we design a language that has (i) a flexible control facility via the `shift` and `reset` operators, (ii) a variety of constructs essential for dependent programming; and (iii) a translation semantics that does not require non-default features. Below is the outline of the thesis, showing what contribution each chapter makes:

- Chapter 2: As a first step, we identify the issues that arise when mixing dependent types and delimited control operators. We begin with a simple type system for `shift` and `reset`, highlighting the requirements that effectful terms impose on their surroundings. Then, we enrich the type system with dependent types, and see why naïve combination of delimited control and dependency fails to be meaningful. Our main contribution in this chapter is to establish three restrictions on type dependency. A key observation is that, in addition to the purity constraints proposed by past work on dependent types and undelimited control, we need to take care of dependencies associated with continuations. This is due to the flexibility of delimited control, and as far as we are aware, the additional constraints have not yet been discussed by others.

- Chapter 3: With these results in mind, we present Dellina-, a small, call-by-value language with dependent types and the `shift`/`reset` operators. The main idea is to impose the three restrictions identified in Chapter 2 by means of typing rules. Although the language lacks many features available in the full-spectrum dependently typed languages, the non-trivial interaction between dependency and control is pervasively present in the type system. After showing the specification, we prove various metatheoretic properties of Dellina-. Specifically, we show that Dellina- is *type sound*, in the sense that "well-typed programs do not go wrong." Lastly, we show three examples of Dellina- programs, which give the reader a rough idea of what we can do with dependent types and control operators.

- Chapter 4: To show that Dellina- is genuinely "good" as a language with control, we define a *selective* CPS translation, which serves as an elimination of the `shift` and `reset` operators. A selective translation is an optimized version of an ordinary CPS translation, in that it only converts effectful terms into CPS and keeps pure terms in direct style. While such a translation has an obvious advantage from a performance perspective, we find that it also greatly simplifies the type preservation argument of our translation. This is a significant result, since type preservation is hard to prove in the presence of dependent types, as we discussed earlier. The existence of a type-preserving CPS translation makes it possible to extend existing languages with `shift` and `reset` without all the effort to re-implement execution facilities.

- Chapter 5: At this moment, the reader should have internalized the general principles for dealing with dependency and control. Building on this intuition, we gradually extend Dellina- with advanced features, and thus obtain the full language Dellina. The features to be discussed includes polymorphism and type operators (Section 5.1), an infinite hierarchy of universes (Section 5.2), user-defined inductive types (Section 5.3), and local definitions (Section 5.4). We then present an example program of Dellina (Section 5.5), showing how the combination of dependent types and delimited control allows for convenient implementation of safe programs.

- Chapter 6: As control operators are usually discussed in a call-by-value setting, both Dellina- and Dellina have a call-by-value semantics. On the other hand, dependently typed languages are in many cases call-by-name, and their type system is also designed to go well with this semantics. To see how dependent types and delimited control interact in a call-by-name setting, we build Dellina^n, a call-by-name variant of Dellina-. Although we have not yet formally proved the metatheoretic properties of the language, we found that some elimination constructs can be given a more "generous" typing rule in the call-by-name language, but not all.

- Chapter 7: The "good" properties of Dellina encourage us to incorporate `shift` and `reset` into proof assistants. However, it is not clear what it means to use these operators in proofs, or what propositions we can prove using them. In answering the first question, we make an observation that impure types and terms carry the information of required contexts, which cannot be encoded in the standard logic. Based on this observation, we claim that, in Dellina, we may only view pure types as propositions, and pure terms as proofs. As for the second question, we conjecture that `shift` and `reset` only prove intuitionistic theorems. This is in contrast to undelimited control operators like `call/cc`, which are known to allow classical reasoning.

- Chapter 8: To our best knowledge, Dellina is the first language that allows implementing meaningful programs using control effects and type dependency. However, to use Dellina in real-world programming, we need to enrich the language with more powerful effect and typing facilities. In Chapter 8, we discuss some of the extentions we intend to investigate as future work, focusing on their practical impact and expected challenges.

This thesis is somewhat lengthy, as it deals with two topics that are studied by non-overlapping communities. We tried to make it as self-contained as possible, so that readers having either background are able to follow the technical content. We also provide a detailed discussion of related work in each chapter, which would serve as a good pointer to readers who wish to further explore a specific topic.

Chapter 2

Shift, Reset, and Dependent Types

2.1 Typing Programs with Shift and Reset

2.1.1 Typing Programs in a Pure Calculus

In the pure, simply typed λ-calculus, a *typing judgment* takes the following form:

$$\Gamma \vdash e : A$$

On the left-hand side of the turnsile, we have a typing environment Γ, which contains a sequence of variable-type pairs $x_i : A_i$. On the right-hand side, we have a term e (called *subject*), and a type A (called *predicate*). The judgment tells us that term e has type A in environment Γ.

Typing judgments are derived by a set of *typing rules*. For instance, the typing rule for application looks like:

$$\frac{\Gamma \vdash e_0 : A \to B \quad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0 \; e_1 : B}$$

Notice that the rule imposes two restrictions on the type of e_0 and e_1. First, e_0 must have an arrow type. Second, the type of e_1 must be equal to the domain of the type of e_0. Thus, to see whether an application is well-typed or not, it suffices to look at the type of the function and the argument.

2.1.2 Typing Terms with Undelimited Control

Now, let us look at how we type terms with undelimited control. Here we repeat the example from the introduction:

$$\begin{aligned}
&1 + (\mathcal{C}k.\, 2 + k\ 45) - 4 \\
&= (2 + k\ 45)[\lambda\, x.\, \mathtt{abort}\ (1 + x - 4)/k] \\
&= 1 + 45 - 4 \\
&= 42
\end{aligned}$$

As the first reduction step shows, the captured continuation k is used as a functional representation of the entire context surrounding the $\mathcal{C}$ construct. This function is however *abortive*, that is, all we can do with a continuation application is to return it as the final result. This means, the $\mathcal{C}$ construct evaluates in *any* context, as long as its hole type is the correct one. Therefore, when typing the $\mathcal{C}$ construct, there is no need to state what kind of context it requires; we can simply use the three-place judgment for pure terms:

$$\Gamma \vdash \mathcal{C}k.\, 2 + k\ 45 : \mathsf{int}$$

Since $\mathcal{C}$ requires no additional typing facility, incorporating this operator into a language requires no major change in the typing rules. The only task is to define a new rule for the $\mathcal{C}$ construct:

$$\frac{\Gamma,\, k : \neg A \vdash e : \bot}{\Gamma \vdash \mathcal{C}k.\, e : A}$$

Observe that the return type of the captured continuation k is $\bot$, *i.e.*, it can be any type. This corresponds to the fact that a $\mathcal{C}$ construct imposes no requirement on the surrounding context.

2.1.3 Typing Terms with Delimited Control, Informally

We saw that programs that use the $\mathcal{C}$ operator can be typed in the same way as programs that have no control effect. Then, what about `shift` and `reset`? It turns out that the standard typing no longer applies in the presence of these operators.

Consider the following program[1]:

(1) $1 :: \langle 2 + \mathcal{S}k.\,(k\ (k\ 3)) :: \mathsf{nil} \rangle$

The program (1) evaluates in the following way:

$$\begin{aligned}
&1 :: \langle 2 + \mathcal{S}k.\,(k\ (k\ 3)) :: \mathsf{nil} \rangle \\
&= 1 :: \langle ((k\ (k\ 3)) :: \mathsf{nil})[\lambda\, x.\, \langle 2 + x \rangle / k] \rangle \\
&= 1 :: \langle 7 :: \mathsf{nil} \rangle \\
&= 1 :: 7 :: \mathsf{nil}
\end{aligned}$$

The program illustrates two characteristics of delimited control operators. To make them easier to identify, let us separate the contexts within and outisde the `reset` clause:

$$\boxed{1 :: \boxed{\langle\ \boxed{2 + \boxed{\mathcal{S}k.\,(k\ (k\ 3)) :: \mathsf{nil}}}\ \rangle}}$$

Following the continuations terminology, we call the inner context $\boxed{2 + \square}$ a *delimited context*, and the outer context $\boxed{1 :: \square}$ a *meta context*.

Now, in the reduction sequence of program (1), we see that k is used as a functional representation of the delimited context. The delimitedness of the continuation is a natural consequence of adding a control delimiter, but there is a more important difference between undelimited and delimited continuations: delimited continuations return and compose like ordinary functions. Observe that in the body of `shift`, we use the result of the application $k\ 3$ for further computation, namely application of k. If k was a $\mathcal{C}$-captured continuation, the nested application $k\ (k\ 3)$ would reduce to 5 instead of 7. The non-abortive nature of delimited continuations means that a `shift` construct requires a specific kind of delimited context, *i.e.*, a delimited context whose return type agrees with the uses of k in the body.

We next find that the value of the `reset` clause, namely $7 :: \mathsf{nil}$, is supposed to compose with the meta context. The notion of meta contexts is again brought

[1]To make evaluation steps and typing derivations clearer, we use a more formal list representation ($1 :: 2 :: \mathsf{nil}$ instead of $[1;\ 2]$) throughout this section.

by the control delimiter, but of particular interest is the fact that the type of the value to be plugged into the meta context can differ from the return type of the delimited context. In our example, the `reset` clause initially surrounds an addition, hence the whole `reset` appears to have type int, but at the next reduction step, the addition is replaced by a list $(k\ (k\ 3)) :: \mathsf{nil}$, therefore the `reset` reduces to a value of type $\mathsf{list\ int}$. This means, a `shift` construct requires a specific kind of meta context, *i.e.*, a meta context whose hole type coincides with the value to be returned by the `reset` construct.

The above discussion suggests that the typing derivation of a `shift` construct must carry two additional types: (i) the expected return type of the delimited context; and (ii) the expected input type of the meta context. This is a central idea underlying various type systems for `shift` and `reset` [55, 10, 140, 27, 102]. For instance, if we adopt the notation of Rompf et al. [140], the `shift` construct in program (1) will have the following judgment:

$$\Gamma \vdash \mathcal{S}k.\, k\ (k\ 3) :: \mathsf{nil} : \mathsf{int}[\mathsf{int}, \mathsf{list\ int}]$$

The typing judgment has a pair of types $[\mathsf{int}, \mathsf{list\ int}]$, telling us that the `shift` operator requires an int-returning delimited context, and a $\mathsf{list\ int}$-accepting meta context. These allow us to determine the well-typedness of program (1), by observing that addition of 2 is an instance of the required delimited context, and consing of 1 is an instance of the required meta context.

2.1.4 Typing Terms with Delimited Control, Formally

In the previous subsection, we sketched a rough idea of what we need to type programs with `shift` and `reset`. Now, let us give a more formal description of how to design a type system for delimited control.

Pure and Impure Terms First, we define the notion of pure and impure terms. We say a term e is pure when its evaluation does not involve access to the context surrounding e. That is, values and `reset` constructs are unconditionally pure, and complex terms consisting of only pure terms are also pure. Here are some examples of pure terms:

$$1 \qquad 2+3 \qquad \lambda\, x.\, x \qquad \lambda\, x.\, \mathcal{S}k.\, 5 \qquad (\lambda\, x.\, x)\ y \qquad \langle 1 + \mathcal{S}k.\, 5\rangle$$

Conversely, we say a term e is impure when its evaluation involves access to the surrounding context. This means, a `shift` construct is trivially impure, and any term containing a `shift` in an executable position is also impure. Here are some examples:

$$\mathcal{S}k.\,5 \qquad 2 + \mathcal{S}k.\,5 \qquad (\lambda\, x.\, \mathcal{S}k.\,5)\ 1 \qquad (\lambda\, x.\, x)\ (\mathcal{S}k.\,5)$$

Typing Judgments with Effect Annotations From the definition of pure and impure terms, we can see that pure terms are those terms that evaluate to a value in an empty context, while impure terms are those terms that evaluate only in parcticular delimited and meta contexts. This means, the additional information of expexted contexts—namely the two types we put in the derivation of `shift` above—are only needed for impure terms. Therefore, in a language with `shift` and `reset`, a typing judgment has the following general form [140]:

$$\Gamma \vdash e : A\ \rho$$

Here, ρ is an optional component representing an effect annotation. When e is pure, the annotation is empty, hence the judgment takes the usual form:

$$\Gamma \vdash e : A$$

On the other hand, when e is impure, ρ is a pair of two types, hence the judgment looks like:

$$\Gamma \vdash e : A[\alpha, \beta]$$

Here, α is the return type of the delimited context required by e, and β is the hole type of the meta context required by e. In this thesis, we call α *initial answer type*, and β *final answer type.* The word "answer type" is commonly used in the continuations literature, and describes the return type of contexts. Readers familiar with CPS translations may recall that a program in CPS often has a type of the form $(A \to r) \to r$; in this type, the two occurrences of r represent the answer type of the program. What is unusual with the above judgment is that α and β are distinct in the general case, *i.e.*, e has two answer types. This is because a `shift` operator may cause *answer-type modification.* The phenomenon, often shortend for *ATM*, happens when elimination of a `shift` replacs the body of the surrounding `reset` with a term having a different type. In the case of program (1), the `shift` operator replaces an addition by a list, modifying the answer type from int to $\mathsf{list\ int}$. In terms of CPS, ATM corresponds to the situation where a CPS program has a type of the form $(A \to r_1) \to r_2$, where r_1 and r_2 are different types.

With these in mind, we look at the typing rules for `shift` and `reset`, which serve as the introduction and elimination rules of effect annotations[2]:

$$\frac{\Gamma,\, k : A \to \alpha \;\vdash\; e : \beta}{\Gamma \;\vdash\; \mathcal{S}k.\, e : A[\alpha, \beta]}\ (\textsc{Shift}) \qquad \frac{\Gamma \;\vdash\; e : B[B, A]}{\Gamma \;\vdash\; \langle e \rangle : A}\ (\textsc{Reset})$$

Let us first focus our attention to (Shift). We see that the initial answer type α in the conclusion comes from the return type of the captured continuation k. This can be understood as: if k is used as an α-returning function in the body e, then the whole `shift` construct must be surrounded by an α-returning delimited context. We next find that the final answer type β coincides with the type of the body e. This can be understood as: if e is a term that evaluates to a β-value, then the whole `shift` construct must be surrounded by a β-accepting meta context.

Now we observe (Reset). The premise says: when we evaluate the body e in an empty context (whose hole and return types are both B), we obtain a value of type A. Since we regard the value of e as the value of $\langle e \rangle$, we conclude $\langle e \rangle$ is a pure term of type A.

Using (Shift) and (Reset), we can type program (1) in the following way:

$$\dfrac{\vdash\ 1 : \mathsf{int} \qquad \dfrac{\dfrac{\vdash\ 2 : \mathsf{int} \qquad \dfrac{\dfrac{k : \mathsf{int} \to \mathsf{int} \;\vdash\; k\ (k\ 3) : \mathsf{int} \qquad k : \mathsf{int} \to \mathsf{int} \;\vdash\; \mathsf{nil} : \mathsf{list\ int}}{k : \mathsf{int} \to \mathsf{int} \;\vdash\; k\ (k\ 3) :: \mathsf{nil} : \mathsf{list\ int}}}{\vdash\ \mathcal{S}k.\, k\ (k\ 3) :: \mathsf{nil} : \mathsf{int}[\mathsf{int}, \mathsf{list\ int}]}}{\vdash\ 2 + \mathcal{S}k.\, k\ (k\ 3) :: \mathsf{nil} : \mathsf{int}[\mathsf{int}, \mathsf{list\ int}]}}{\vdash\ \langle 2 + \mathcal{S}k.\, k\ (k\ 3) :: \mathsf{nil} \rangle : \mathsf{list\ int}}}{\vdash\ 1 :: \langle 2 + \mathcal{S}k.\, k\ (k\ 3) :: \mathsf{nil} \rangle : \mathsf{list\ int}}$$

Observe that the `shift` clause has an effect annotation $[\mathsf{int}, \mathsf{list\ int}]$, saying that it

[2] These rules are slightly informal: the body e in (Shift) can be impure, and dually, the body e in (Reset) can be pure. The precise rules will be presented in Section 3.3.

requires an `int`-returning delimited context, and a `list int`-returning meta context. This information is propagated to the derivation of the addition, and since its type and initial answer type agree, we can apply the (Reset) rule, and conclude that the `reset` has type `list int`. Thus, we know that the consing operation is type-safe.

Composing Impure Terms In program (1), there is only one `shift` operator, and its control effect simply propagates to the whole term inside the surrounding `reset`. When a program has multiple `shift` operators, we have to reason about their composition more carefully, keeping in mind in what order we evaluate each `shift`. Consider the following program:

$$(2)\quad \langle (\mathcal{S}k_1.\,(k_1\ 1) :: [\texttt{"4"}]) + (\mathcal{S}k_2.\,\texttt{string-of-int}\ (k_2\ 2))\rangle$$

$$\begin{aligned}
&\langle (\mathcal{S}k_1.\,(k_1\ 1) :: [\texttt{"4"}]) + (\mathcal{S}k_2.\,\texttt{string-of-int}\ (k_2\ 2))\rangle\\
&= \langle (k_1\ 1) :: [\texttt{"4"}][\lambda\,x : \mathsf{int}.\,\langle x + (\mathcal{S}k_2.\,\texttt{string-of-int}\ (k_2\ 2))\rangle/k_1]\rangle\\
&= \langle\langle 1 + (\mathcal{S}k_2.\,\texttt{string-of-int}\ (k_2\ 2))\rangle :: [\texttt{"4"}]\rangle\\
&= \langle\langle \texttt{string-of-int}\ (k_2\ 2)[\lambda\,x.\,\langle 1 + x\rangle/k_2]\rangle :: [\texttt{"4"}]\rangle\\
&= \langle\langle \texttt{string-of-int}\ 3\rangle :: [\texttt{"4"}]\rangle\\
&= [\texttt{"3"};\ \texttt{"4"}]
\end{aligned}$$

Program (2) has two `shift` operators. Among which, we first eliminate the one on the left. What we should pay attention to is the initial answer type of this `shift` construct: as we can see from $(k_1\ 1) :: [\texttt{"4"}]$, we are using k_1 as a `string`-returning function. By careflly observing the reduction sequence, we find that the string is formed by `string-of-int` $(k_2\ 2)$, which is a computation that happens when we eliminate the second `shift` construct. Thus, when we have two successive occurrences of `shift`, it looks like the initial answer type of the first `shift` must coincide with the final answer type of the second one.

This finding is indeed true, and can be justified by the reduction rule of the `shift` operator. Let us make two observations in the above reduction sequence. First, the body of a `shift`-captured continuation is surrounded by a `reset`. Second, elimination of the second `shift` happens when we call the continuation k_1 captured by the first `shift`. What this means is that the return type of k_1 (*i.e.*, the initial answer type of the first `shift`) is determined by what the second `shift` eventually returns (*i.e.*, the final answer type of the second `shift`).

The answer types of constituent terms also determine the answer types of the whole term. In the reduction sequence of program (2), we see that the addition operation happens when we call the continuation k_2 captured by the second `shift`. This seems to suggest that the initial answer type of the whole term is equal to the initial answer type of the last-eliminated `shift` operator. The reduction sequence further shows that elimination of the first `shift` replaces the addition by a string list, and this overall structure is fixed in later reduction steps. This seems to imply that the final answer type of the whole term is equal to the final answer type of the first-evaluated `shift`.

These observations again hold in the general case, and can be explained in terms of the `reset` surrounding the body of `shift`-captured continuations. When we have successive occurrences of `shift`, as in $\langle op\ \mathcal{S}k_1.\, e_1\ \mathcal{S}k_2.\, e_2\ ...\ \mathcal{S}k_n.\, e_n\rangle$ (where *op* is an n-ary operator), evaluation of later `shift`'s happens when the continuation captured by a preceding `shift` is applied to an argument. That is, the second `shift` is evaluated when k_1 is called, and the third one is evaluated when k_2 is called, and so on. As a captured continuation has a `reset` in its body, every k_i must share the same overall structure, and differ only in the position of the hole. That is, if k_1 takes the form $\langle op\ [.]\ \mathcal{S}k_2.\, e_2\ ...\ \mathcal{S}k_n.\, e_n\rangle$, then k_2 takes the form $\langle op\ a_1\ [.]\ ...\ \mathcal{S}k_n.\, e_n\rangle$, and k_n takes the form $\langle op\ a_1\ a_2\ ...\ [.]\rangle$. Now, it is easy to see that the computation of *op* really takes place when k_n is applied to some argument a_n. The result of $k_n\ a_n$ must compose with the rest of the computation in e_n, hence return type of *op*, which is going to be the initial answer type of the whole term inside `reset`, must agree with the use of k_n. On the other hand, when the first `shift` is evaluated, the application of *op* is replaced by e_1. As we saw, evaluation of succeeding `shift`'s happens when k_1 is applied, but these `shift`'s cannot affect e_1 because their effects are encapsulated in the `reset` surrounding the body of k_1. Therefore, the final answer type of the whole *op* construct is determined by that of e_1.

To check the composability of (possibly) impure terms, Rompf et al. [140] use an effect composition operator in the typing rule. For instance, the rule of addition looks like:

$$\frac{\Gamma \vdash e_1 : \mathsf{int}\ \rho \quad \Gamma \vdash e_2 : \mathsf{int}\ \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)}{\Gamma \vdash e_1 + e_2 : \mathsf{int}\ \tau}\ (\textsc{Add})$$

where $\mathrm{comp}(\rho, \sigma)$ is defined as:

$$\frac{}{\mathrm{comp}(\epsilon, \rho) = \rho} \qquad \frac{}{\mathrm{comp}(\rho, \epsilon) = \rho} \qquad \frac{\mathrm{comp}(\overline{\rho}) = [\alpha, \beta]}{\mathrm{comp}([\beta, \gamma], \overline{\rho}) = [\alpha, \gamma]}$$

Observe how the last rule generalizes our three findings: (i) two adjacent effects has a common type β (which serves as either an initial or the final answer type); (ii) the overall initial answer type is that of the last effectful computation (α); and (iii)the overall final answer type is that of the first effectful computation (γ).

2.2 Simply Typed Shift and Reset

Type Systems for Shift and Reset The `shift` and `reset` operators have a solid type theoretical foundation. The original type system, given by Danvy and Filinski [55], uses a five-place judgment of the following form:

$$\Gamma;\, \alpha \vdash e : A;\, \beta$$

The judgment communicates the same information as $\Gamma \vdash e : A[\alpha, \beta]$, but the two answer types are *not* optional, and every term—including pure values—must use this non-standard judgment. The initial and final answer types for pure terms can be arbitrary, since pure terms evaluates in any context, but they must be equal, because pure terms never cause answer-type modification.

The Danvy-Filinski type system was later extended with different forms of purity distinction. One of the variants was given by Asai and Kameyama [10], who incorporate the ordinary three-place judgment $\Gamma \vdash_p e : A$ into the original type system. The judgment is used for syntactically pure terms, which include variables, functions, and `reset` constructs. Asai and Kameyama further refine the type of continuations so that we may treat them as pure functions, *i.e.*, functions having a pure body. This refinement allows us to use a continuation in different context, making more programs typable[3].

[3]Asai and Kameyama give the following example:

```
let rec visit lst = match lst with
  [] -> shift k []
```

The type system of Rompf et al. [140], which we used in the previous section, can be viewed as extending Asai and Kameyama's system with a finer purity distinction. Recall that we had a composition operation, which we use to decide whether the effects of subterms meet the chaining rule. Since each effect annotation can be either empty or non-empty, the composition operater absorbs all possible combinations of the subterms' effects. In Asai and Kameyama's type system, on the other hand, all subterms are derived using the impure judgment. For instance, in the typing rule of addition, the two arguments must carry a non-empty effect annotation. This design choice does not limit typability of terms in a simply typed setting, since the type system has the following rule for casting a pure term into an impure one:

$$\frac{\Gamma \vdash_p e : A}{\Gamma;\, \alpha \vdash e : A;\, \alpha}$$

In a dependently typed setting, however, application of the casting rule may turn a well-typed term into an ill-typed one, since allowing impure terms in types leads to undesired consequences. What this means is that, when dealing with control effects and dependent types at the same time, we need to be precise about which terms *actually* involve control effects. For this reason, we use Rompf et al.'s judgment throughout this thesis.

ATM and Parameterized Monads The typing mechanism of `shift` and `reset` can also be explained by means of *monads* [166]. First, it is widely recognized that there is a strong connection between CPS and the continuations monad. If we look at the MonadCont class of Haskell, we will find the following constructor/destructor declaration:

```
cont :: ((a -> r) -> r) -> Cont r a
```

```
  | a :: rest -> a :: shift k (k [] :: reset (k (visit rest)))
  let prefix lst = reset (visit lst)
```

The `prefix` function receives a list and returns a list consisting of its prefixes: *e.g.*, `prefix [1; 2; 3]` = `[[1]; [1; 2]; [1; 2; 3]]`. Among the two occurrences of `k`, the first one is used as a function whose body expects a `list (list int)`-returning context (since the body of `shift` returns a list of integer lists), whereas the second one is used as a function whose body expects a `list int`-returning context (since subsequent `shift`'s require a `list int`-context). This use of `k` is not possible in Danvy and Filinski's type system, where we must fix the type and effect of `k` when extending the typing environment.

```
runCont :: Cont r a -> (a -> r) -> r
```

The type of `cont` says, in the continuations monad, computations are functions awaiting a continuation. Dually, the type of `runCont` says, we can run a computation by supplying a continuation. The continuations monad is however insufficient to express `shift` and `reset`, because the answer type `r` is fixed. To support modification of answer types, we replace the two occurrences of `r` with different types `r1` and `r2`, obtaining the *parameterized monad* [13]:

```
pcont :: ((a -> r1) -> r2) -> PCont r2 r1 a
runPCont :: PCont r2 r1 a -> (a -> r1) -> r2
```

An advantage of studying delimited control in terms of the parameterized monad is that we can understand the chaining of answer types via the `bind` operation, which we use to compose two monadic computations. Specifically, the `bind` operator for the parameterized monad has the following type:

```
bind :: (PCont r3 r2 A) -> (A -> PCont r2 r1 B) -> PCont r3 r1 B
```

The duplicated occurrences of `r2` and the parameters of the conclusion are exactly what we observed in program (2).

Shift and Reset without ATM The type systems we have discussed so far, as well as the parameterized monad, are all adapted to account for ATM, but there is also a variant of `shift` and `reset` that does not have this ability. The variant is obviously weaker than the full `shift` and `reset`, as it limits the kind of computation we may have in the body of `shift`. Nevertheless, it has proven that the ATM-free `shift` and `reset` can express any monadic effects—including exceptions, non-determinism, and state—in direct style[4] [76]. The weaker `shift` and `reset` have also been studied from a logical perspective [96]. Interestingly, the fixed answer type makes it easier to extend the proofs-as-programs notion to delimited control, as we will see in Chapter 7.

[4]In this sense, `shift` and `reset` are the mother of all monads, as summarized by Koppel et al. [107].

2.3 Three Restrictions on Type Dependency

Having discussed `shift` and `reset` in a simply typed setting, let us integrate them into a dependently typed world. As we saw in Section 1.3, handling control in the presence of dependent types is a delicate issue. To solve the challenges, previous studies restrict types to depend only on pure terms [94, 121]. The restriction guarantees that, whenever a term appears in a type, that term must reduce to a unique value, regardless of its surroundings. This helps us maintain fundamental properties such as logical consistency and subject reduction.

In this section, we establish three restrictions for soundly extending a dependently typed language with the `shift` and `reset` operators. We observe that dependency on impure terms makes the whole type unuseful, and furthermore, the flexibility of `shift` and `reset` necessitates additional care on the dependency of continuations, as well as the dependency on continuations. In the subsequent sections, we study a seriese of examples illustrating invalid dependencies that may arise when unrestrictedly using the `shift` and `reset` operators.

2.3.1 Types Dependent on Impure Terms

Let us begin by discussing what kind of terms can appear in types and what cannot. Compare the following programs:

(3) $(\lambda\, x : \mathrm{L}\ (1 + 1).\, x)\ [0;\ 1]$

(4) $(\lambda\, x : \mathrm{L}\ \langle 1 + \mathcal{S}k.\, 2\rangle.\, x)\ [0;\ 1]$

(5) $(\lambda\, x : \mathrm{L}\ (1 + \mathcal{S}k.\, 2).\, x)\ [0;\ 1]$

In dependently typed languages, types may contain term-level computations. Therefore, when deciding equality between two types, we first normalize terms inside types, and then compare the results. Observe program (3): the identity function requires a term of type $\mathrm{L}\ (1+1)$, whereas the actual argument $[0;\ 1]$ has a syntactically different type $\mathrm{L}\ 2$. However, these types can be considered equivalent, since $1 + 1$ normalizes to 2. Thus, we conclude that the application is type-safe.

Program (4) differs from program (3) in that the index of the domain type uses control operators. But still, we can type check the application in the standard manner, because the index is a pure term as a whole, which we can normalize to a value. In this case, $\langle 1 + \mathcal{S}k.\, 2\rangle$ normalizes to 2, hence the list [0; 1] is a valid argument to the identity function.

Now we look at program (5). Compared with the previous one, this program lacks a `reset` clause surrounding the list index. The difference may appear subtle, but once we try type checking the application, we immediately find that things do not go as before. With the absence of `reset`, the index $1 + \mathcal{S}k.\, 2$ is judged impure. To see if the application is well-typed, we are tempted to somehow eliminate the `shift` operator, but we cannot do so because `shift` captures a *delimited* continuation—it requires a `reset`. What this implies is that the normalize-and-compare method does not apply to types dependent on impure terms.

Aside from type checking, having non-normalizable types does not seem to be beneficial to the user either. The reason is that such types do not tell us what property their inhabitants have. Consider program (5) again. In the body of the identity function, we know that x has type $\mathrm{L}\ (1 + \mathcal{S}k.\, 2)$, but we still do not know how many elements x contains. If the reader finds the example not clear enough, think of L (`get` a), where the index accesses the value of some globally defined mutable reference a. At type checking time, we cannot know what the type means as a whole, since the value of a depends on the runtime environment. Furthermore, if we had two lists of this type, it would not necessarily be the case that they must have the same length, because impure terms are not *pervasive* [70] and hence their copies should all be considered different.

The above discussion tells us that, if we wish to use types to reason about programs, then we must restrict types to depend only on pure terms, which we can locally normalize.

2.3.2 Continuations Having A Dependent Type

We next turn our viewpoint to dependencies associated with continuations. Recall that, in a dependently typed language, functions have a type of the form $\Pi\, x : A.\, B$, where x may occur free in the codomain B. Since continuations are functions, we will naturally wonder whether they can have a dependent type as well. Let us consider the programs below, assuming `mk-lst` : $\Pi\, x : \mathbb{N}.\, \mathrm{L}\ x$ and `append` : $\Pi\, m : \mathbb{N}.\, \Pi\, n : \mathbb{N}.\, \mathrm{L}\ m \to \mathrm{L}\ n \to \mathrm{L}\ (m + n)$:

(6) $\langle$`mk-lst` $\mathcal{S}k.\, k\ 1\rangle$

(7) $\langle$`mk-lst` $(\mathcal{S}k.$ `append` $1\ 2\ (k\ 1)\ (k\ 2))\rangle$

(8) $\langle$`mk-lst` $\mathcal{S}k.\, k\rangle$

(9) $\langle$`mk-lst` $\mathcal{S}k.\, k\rangle\ 1$

In all the four programs, the captured continuation k is the application of `mk-lst`, hence it has a dependent function type $\Pi\, x : \mathbb{N}.\, \mathrm{L}\ x$. To type the `shift` constructs in these programs, we have to slightly modify the (Shift) typing rule from Section 2.1, by replacing the assumption $k : A \to \alpha$ with $k : \Pi\, x : A.\, \alpha$. Then, the `shift` constructs in the above programs will have a derivation of the following form:

$$\frac{k : \Pi\, x : \mathbb{N}.\, \mathrm{L}\ x \vdash\ e : \beta}{\vdash\ \mathcal{S}k.\, e : \mathbb{N}[\mathrm{L}\ x, \beta]}$$

Recall that the initial answer type of a `shift` construct coincides with the return type of the captured continuation. In the above examples, the continuation returns a value of type $\mathrm{L}\ x$. Clearly, this type does not serve as a valid initial answer type of the `shift` construct, because it has a free occurrence of variable x.

The reader might think that we could close off this variable by substituting the actual argument we pass to the continuation. In the case of program (6), the remedy seems to work: when we type check the `shift` construct, we can see that k is called exactly once with argument 1, hence we could “instantiate” $\mathrm{L}\ x$ to a closed type $\mathrm{L}\ 1$. However, this does not work when we call the captured continuation multiple times (as in program (7)), or zero times (as in program (8)). In the former case, we would have more than one candidate value for closing off the free variable, while in the latter case, we have no candidate value at all. Another case where type instantiation fails is illustrated by program (9): the captured continuation receives an argument outside the control construct, which is not visible when type checking the `shift` construct. To find this argument, we have to analyze the whole program during type checking, violating the principle of static typing—that is, types are determined using local information.

Summing up these observations, we conclude that continuations must be non-dependent functions. Put differently, we must use the `shift` operator only in a non-dependent context.

Remark In languages with undelimited control, dependency on holes can never arise, because continuations have a type of the form $\neg A$.

2.3.2.1 Types Dependent on Continuations

What we discussed in the previous section is dependency *of* continuations. Now, let us look at dependency *on* continuations. Consider the following program:

(10) $\langle 1 + \mathcal{S}k.\, \texttt{mk-lst}\ (k\ 2)\rangle$

The typing rule from Section 2.1 gives us the following derivation for the `shift` construct:

$$\frac{\bullet,\ k : \mathbb{N} \to \mathbb{N} \vdash \texttt{mk-lst}\ (k\ 2) : \mathrm{L}\ (k\ 2)}{\bullet \vdash \mathcal{S}k.\, \texttt{mk-lst}\ (k\ 2) : \mathbb{N}[\mathbb{N}, \mathrm{L}\ (k\ 2)]}\ (\textsc{Shift})$$

Recall that the final answer type of a `shift` construct is determined by what its body evaluates to. In program (10), the body evaluates to a value of type L $(k\ 2)$. Similarly to the situation with dependent continuations, this type does not serve as a valid final answer type of the `shift` construct, because it contains a free continuation variable k.

Just like we tried to obtain a closed initial answer type by substituting the actual argument, we might attempt to obtain a closed final answer type by substituting the actual continuation. In the case of program (10), it is not hard to see that k will be the function $\lambda\, x : \mathbb{N}.\, \langle 1 + x\rangle$, hence we could instantiate L $(k\ 2)$ to a closed type L $((\lambda\, x : \mathbb{N}.\, \langle 1 + x\rangle)\ 2)$. However, in the general case, figuring out what continuation is to be captured requires evaluating the whole program, which is not allowed at type checking time if the language is statically typed. Moreover, there is no guarantee that we have a candidate continuation to substitute for every continuation variable, because well-typedness of terms does not require `shift` operators to have a matching `reset`.

Wrapping up the discussion, we conclude that continuations may appear only in types that are not used as the final answer type of a `shift` construct. In other words, a `shift` construct may give us different values depending on the context, but all the values must have a common type.

Remark In languages with undelimited control, dependency on continuations can never arise, because the body of undelimited control constructs either has an empty type $\perp$, or has the same type as the hole of the continuation.

2.4 Dependent Types and Effects

The combination of dependent types and (control and non-control) effects has been extensively studied over the last twenty years. With a few exceptions, most studies share the same design principle: types may depend only on pure terms. In this section, we review different approaches to restricting type dependency.

No Restriction The earliest attempt at building dependent and effectful calculi goes back to 1997. Barthe et al. [21] invent Classical Pure Type System (CPTS), which extends ordinary Pure Type System (PTS) [16] with a $\mathcal{C}$-like control operator. CPTS allows arbitrary use of the control operator, and enjoys a number of desirable properties, including subject reduction, consistency, decidability of type checking, and type preservation of the CPS translation[5]. This may sound surprising, but a large part of the result relies on the specification of their calculus: the only computation in CPTS is application, which is evaluated in a call-by-name manner. This keeps us away from various issues reported by subsequent studies [24, 93, 121].

Almost at the same time, Augustsson [14] designed the Cayenne language, which can be understood as a subset of Haskell extended with dependent types. Cayenne supports general recursion, and allows the user to freely use this construct. Consequently, Cayenne's type checking is undecidable, but experiments show that the type checker "works remarkably well" in practice.

Separate Index Language Dependent ML (DML) [174, 172] is a variant of ML [120] with support for a restricted form of dependent types. Types in DML are allowed to depend on terms generated by an index language, which is defined

[5]Strictly speaking, the translation is only given for the *domain-free* variant of CPTS, where λ-abstractions do not have a type annotation. Giving up annotations makes type checking undecidable, but simplifies the type preservation argument of the CPS translation. See Section 3.2 for details.

separately from runtime programs and has no effects. This approach makes it possible to accommodate dependent types without complicating type checking too much: for instance, the resulting language does not require the user to provide type annotations everywhere in her program.

Syntactic Value Restriction Instead of having a distinguished index language, some languages impose a syntactic value restriction on type dependency. AURA [98] is a domain-specific language for access control, which combines read-like operations and value-dependent types. An old version of the $F^\star$ language [155], which was designed for secure distributed programming, soundly mixes dependency and different kinds of computational effects by imposing a similar value restriction.

Restricting dependency to values, however, leads to slowdown and inconvenience in languages featuring *computational irrelevance*. Readers with experience in dependently typed programming would have seen functions requiring arguments that are only used for type checking. In some languages, such arguments can be marked irrelevant, and get erased after the type checking phase. This is a practically useful feature, since it helps us reduce computation steps at runtime, and often unnecessitates verbose type casting in the implementation phase. However, when limiting type dependency to values, many computationally useless arguments can no longer be marked as irrelevant. We invite the interested reader to visit Section 3.8 of Sjöberg [150] for a detailed discussion.

Semantic Value Restriction Lepigre [110] extends the syntactic value restriction to a semantic one. He builds a dependently typed version of the $\lambda\mu$-calculus [131] with ML-like constructors and records, where types may depend on terms that are observationally equal to values. The relaxation admits strictly more terms in types, but makes it harder to decide whether dependency on a certain term is allowed or not. Indeed, the observational equivalence is undecidable, as pointed out by Miquey [121].

Negative-Elimination-Free Condition A more lightweight, but reasonably permissive, approach is Herbelin's negative-elimination-free (NEF) condition [94], which we saw in Section 1.3. Herbelin classifies a proof p as NEF when evaluation of p causes no effect. That is, an NEF proof is either a value, or a computation that does not trigger control effects involved in its subterm. The latter rules out application, which eliminates a function type. Since function types are classified as "negative" in logics, we call the condition "NEF."

Miquey [121] incorporates the NEF restriction into his dependent $\lambda\mu\tilde{\mu}$-calculus, by introducing delimited continuations. This solves the issue with subject reduction for application, as we saw in Section 1.3. In a more recent study, Miquey extends the calculus with *co-delimited continuations* [122], to make the standard reduction of dependent pairs type-safe. The notion of co-delimited continuations had never appeared before, but considering the fact that Π and Σ types are the dual of each other, it is quite natural that their elimination forms require dual notions to be type-preserving.

Side-effect-free Fragment A third approach to relaxing the value restriction is to build some form of type-and-effect system, and allow types to depend on effect-free terms. This idea is used in the dependent calculi proposed by Ou et al. [129], Casinghino et al. [38], and Gordon [87], which feature non-termination. In particular, Casinghino et al. distinguish between logical and programmatic fragments of their language, and allow non-termination only in the latter fragement. By restricting proofs and dependency to the logical fragment, they obtain a language that is both powerful as a programming language and consistent as a proof assistant.

The same idea extends to languages with different kinds of effects. Sheldon and Gifford [149] observe that first-class modules give rise to type dependency on module inhabitants. To guarantee static type checking, they propose *static dependent types*, which are essentially types dependent on terms having an empty effect annotation.

Call-By-Push-Value So far, we have looked at languages where type dependency is restricted to a certain fragment defined by the language's designer. There are also frameworks that have an intrinsic distinction between pure and effectful terms. The Call-By-Push-Value (CBPV) calculus of Levy [113] is an instance of such frameworks. In CBPV, terms are divided into two categories: *value terms* (which are pure) and *computation terms* (which are impure). Using this distinction, Ahman [1, 2] and Vákár [163] independently develop a dependent version of CBPV, which only allows types to depend on value terms. The restricted dependency helped Ahman and Vákár define a category-theoretical semantics of their calculus, but it turns out that this single strategy is not sufficient to work with dependently typed CBPV syntax.

CBPV has a syntactic construct called *sequential composition*, which takes the form e_1 to $x : A$ in e_2. This construct is similar to a call-by-value `let` expression: we evaluate the computation e_1, bind x to the resulting value, and then evaluate the computation e_2. The challenge stems from the fact that, in a dependently typed setting, the type of e_2 may contain occurreces of x, which would remain free in the conclusion of the typing rule below:

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma, x : A \vdash e_2 : B(x)}{\Gamma \vdash e_1 \;\mathsf{to}\; x : A \;\mathsf{in}\; e_2 : B(x)}$$

We know that x is to be replaced by the value of e_1, but this value is unknown at type-checking time. Note that we cannot simply substitute e_1 for x, because it is an effectful computation.

Ahman solves this issue by introducing *computational* Σ *types*. The idea is to close off the free variable by existentially quantifying over possible values, which amounts to saying "the result type depends on some x, though I don't know what exactly x will be." Vákár, on the other hand, gives a different solution to this problem. He turns effectful e_1 into a pure value by thunkifying it (*i.e.*, by delaying the evaluation of e_1 via $\lambda\,().\,e_1$), and lets the result type depend on the thunk.

The free variable in the conclusion reminds us of the open initial and final answer types we observed earlier in this chapter. In particular, Ahman's idea gives us some hope to obtain a closed initial answer type in programs (7) and (9), where we have a candidate value for x (the argument to the continuation). The naïve value instantiation fails because the value is not unique or currently invisible, but it seems plausible to abstract the value using an existential quantifier. However, Ahman's approach implicitly assumes that there is always a candidate value for the free variable to be closed off. This assumption is inappropriate in program (8), where no value is substituted for x in the course of evaluation. While Ahman does not aim to support control operators, our observation suggests that the computational Σ types would not work for first-class continuations.

Chapter 3

The Dellina- Language

In this chapter, we present Dellina-[1], a dependently typed language with delimited control via the `shift` and `reset` operators. As we saw in the previous chapter, unconstrained dependency in the presence of control effects brings unuseful types and open answer types. To avoid these, we have established three restrictions on type dependency: (i) types should not depend on impure terms; (ii) continuations should not be dependent functions; and (iii) final answer types should not refer to continuations. In Dellina-, we enforce these restrictions by means of typing rules. To focus our attention to this "real issue", we equip Dellina- with a minimal set of features available in the mainstream dependently typed languages, and defer the discussion of the full language, Dellina, to Chapter 5.

In what follows, we present the specification of Dellina- in three steps: syntax (Section 3.1), reduction and equivalence (Section 3.2), and typing (Section 3.3). Then, we prove a series of metatheoretic properties (Section 3.4). Our ultimate goal is to show that the type system is *sound* [170], that is, once a program is judged well-typed, it never "goes wrong" at runtime. Dellina- is a small language, but it already allows implementing simple programs where dependency and effects co-exist. We show two such examples (Section 3.5), which build a dependent list via non-determinism and mutable state.

[1]The name is inspired by Gallina, the specification language of the Coq proof assistant. The intention is that our language is a variant of Gallina with DELimited control.

$$\begin{array}{lll l}
\text{Environments} & \Gamma & ::= & \bullet \mid \Gamma, \mathsf{x} : \mathsf{A} \\
\text{Kinds} & \kappa & ::= & * \mid \Box \\
\text{Types} & \mathsf{A}, \alpha & ::= & \mathsf{Unit} \mid \mathbb{N} \mid \mathsf{L}\ \mathsf{e} \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho \\
\text{Effects} & \rho & ::= & \epsilon \mid [\alpha, \beta] \\
\text{Values} & \mathsf{v} & ::= & \mathsf{x} \mid \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e} \mid \mathsf{rec}\ \mathsf{f}_{\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho}\, \mathsf{x}.\, \mathsf{e} \\
 & & \mid & () \mid \mathsf{z} \mid \mathsf{suc}\ \mathsf{v} \mid \mathsf{nil} \mid\ ::\ \mathsf{v}\ \mathsf{v}\ \mathsf{v} \\
\text{Terms} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{e}\ \mathsf{e} \mid \mathsf{suc}\ \mathsf{e} \mid\ ::\ \mathsf{e}\ \mathsf{e}\ \mathsf{e} \\
 & & \mid & \mathsf{pm}\ \mathsf{x}\ \mathsf{as}\ \mathbb{N}\ \mathsf{in}\ \mathsf{P}\ \mathsf{ret}\ \mathsf{e}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e} \\
 & & \mid & \mathsf{pm}\ \mathsf{x}\ \mathsf{as}\ \mathsf{L}\ \mathsf{a}\ \mathsf{in}\ \mathsf{P}\ \mathsf{ret}\ \mathsf{e}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e} \mid\ ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e} \\
 & & \mid & \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} \mid \langle \mathsf{e} \rangle
\end{array}$$

Figure 3.1: Dellina- Syntax

3.1 Syntax

We define the syntax of Dellina- in Figure 3.1. Throughtout this thesis, we use a blue, sans-serif font to typeset Dellina- (and Dellina) expressions. The first category, Γ, denotes typing environments. We build an environment by extending an empty environment $\bullet$ with variable declarations of the form $\mathsf{x} : \mathsf{A}$. The snoc-list-style presentation of environments is popular in dependently typed languages, because each type A may refer to the previously introduced variables. We will explain how this reference is made available when we present the environment well-formedness rules in Section 3.3.

Next, we have kinds κ. Since Dellina- does not allow formation of type-level functions, we only have a base kind $*$, representing the type of types, and a higher kind $\Box$, representing the type of $*$. Note that these kinds never show up in the user program; they are only used to define the formation rules of types and kinds.

Types A, α consist of primitive inductive datatypes and function types. Inductive types include the unit type Unit, the natural number type $\mathbb{N}$, and the length-indexed list type $\mathsf{L}\ \mathsf{e}$ from Section 1.1. Remember that $\mathsf{L}\ \mathsf{e}$ is inhabited by lists containing e natural numbers. Function types take the form $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$. Here, A is the domain, B is the co-domain, and ρ is an effect annotation. The annotation tells us what kind of control effect the function's body has. When the body is pure, the annotation is empty (ϵ when written explicitly), and the function type looks like $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$. Otherwise, ρ is a pair $[\alpha, \beta]$ of initial and final answer types, and the whole function type takes the form $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}[\alpha, \beta]$. As a notational convention, we usually use greek letters to evoke answer types. Note that effect annotations are present only in the co-domain part of function types. This is because Dellina- is a call-by-value language: functions can only ever receive a value, which is free from effects[2]. Note also that we allow the variable x to occur free in the co-domain B, as well as the answer types α and β. When we want to explicitly state the absence of such dependency (*e.g.*, in the (E-NDApp) typing rule in Figure 3.9), we use the arrow type $\mathsf{A} \rightarrow \mathsf{B}\ \rho$ instead[3].

Lastly, we have terms e, consisting of values v and computations. A value is either a variable x, an abstraction $\lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e}$, a recursive function $\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x}:\mathsf{A}.\, \mathsf{B}\ \rho}\, \mathsf{x}.\, \mathsf{e}$, or an inductive datum of the value form (*i.e.*, constructor application where all arguments are values). We use Church-style abstractions, where the bound variable has an explicit type annotation. This is necessary for deciding typability and purity of terms [64, 29], as we will detail in the next section. Among inductive data, $()$ is the unit value, z is the zero constructor, $\mathsf{suc}\ \mathsf{v}$ is the successor of v, nil is an empty list, and $::\ \mathsf{v}_0\ \mathsf{v}_1\ \mathsf{v}_2$ means consing the value v_1 to the list v_2, whose length is v_0. For simplicity, we require constructors to be fully applied. Computations include application, inductive data of the computation form, pattern matching for natural numbers and lists, and the `shift`/`reset` constructs. Dellina- supports *dependent pattern matching* [49], a variant of pattern matching constructs where the result type may depend on the term being analyzed. The dependency necessitates additional annotations for typing purposes: for instance, when we pattern match on a natural number e, we use the following syntax:

$$\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e}_2$$

Here, the type $\mathbb{N}$ explicitly tells us that we are inspecting a natural number, and P denotes the return type of the whole construct. What is different from non-dependent pattern matching is that P may contain free occurrences of the variable x, which abstracts over all possible scrutinees (*i.e.*, natural numbers). The corresponding construct for lists can be understood in a similar way. The difference is

[2]As we will see in Chapter 6, the call-by-name variant of Dellina- has function types whose domain carries an effect annotation.

[3]When $\rho = [\alpha, \beta]$, $\mathsf{A} \rightarrow \mathsf{B}\ \rho$ is equivalent to $\mathsf{A}/\alpha \rightarrow \mathsf{B}/\beta$ of Danvy and Filinski [55] and $\mathsf{A}\ {}_{\alpha}\!\!\rightarrow_{\beta}\ \mathsf{B}$ of Biernacka and Biernacki [27].

that P may depend on two variables: `a`, which abstracts over length indices, and `x`, which abstracts over lists.

As a notational convention, we use the metavariable `t` to mean an expression of any category. That is, `t` can be a kind, a type, or a term.

3.2 Evaluation, Reduction, and Equivalence

We next define rules for running Dellina- expressions. Following previous studies [151, 38, 37, 150], we define two kinds of reduction: one for runtime evaluation, and the other for type checking. Having separate reductions has various benefits when working with dependent types; we will motivate this at the beginning of Section 3.2.2.

3.2.1 Runtime Evaluation

In Figure 3.2, we present evaluation contexts, which guide us searching for redexes (reducible expressions) in a deterministic manner. The reader will find that there are two kinds of contexts, E and F. These contexts denote general and *pure* contexts, repectively. Observe that the definition of F lacks the pattern $\langle\mathsf{F}\rangle$. This ensures that pure contexts never contain a `reset` surrounding a hole, which, as we will see shortly, helps us define the elimination rule of the `shift` operator. All other patterns are shared by both kinds, imposing a call-by-value, left-to-right evaluation strategy. For instance, when we have an application $\mathsf{e}_0\ \mathsf{e}_1$, we first evaluate the function e_0, which corresponds to the context $\mathsf{E}\ \mathsf{e}$, then evaluate the argument e_1, which corresponds to $\mathsf{v}\ \mathsf{E}$.

The plug function tells us how to build a term by plugging a term into the hole of a pure evaluation context[4]. By observing the definition, the reader will find that evaluation contexts are represented "inside-out", that is, the top-level shape of an evaluation context determines the shape of the term to be plugged into, not that of the resulting term. As a notational convenience, we often write $\mathsf{F}[\mathsf{e}]$ to mean the result of plug $\mathsf{F}\ \mathsf{e}$.

[4]The definition extends to E by adding the clause plug $\langle\mathsf{E}\rangle\ \mathsf{e} =$ plug $\mathsf{E}\ \langle\mathsf{e}\rangle$, but we do not need this variant in the reduction rules defined below.

$\boxed{\text{Evaluation Contexts } \mathsf{E}, \mathsf{F}}$

$$
\begin{array}{lcl}
\mathsf{E} & ::= & [\,] \mid \mathsf{E}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{E} \\
 & \mid & \mathsf{suc}\ \mathsf{E} \mid\ ::\ \mathsf{E}\ \mathsf{e}\ \mathsf{e} \mid\ ::\ \mathsf{v}\ \mathsf{E}\ \mathsf{e} \mid\ ::\ \mathsf{v}\ \mathsf{v}\ \mathsf{E} \\
 & \mid & \mathsf{pm}\ \mathsf{E}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \rightarrow \mathsf{e} \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e} \\
 & \mid & \mathsf{pm}\ \mathsf{E}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \rightarrow \mathsf{e} \mid\ ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e} \\
 & \mid & \langle \mathsf{E} \rangle \\
\mathsf{F} & ::= & [\,] \mid \mathsf{F}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{F} \\
 & \mid & \mathsf{suc}\ \mathsf{F} \mid\ ::\ \mathsf{F}\ \mathsf{e}\ \mathsf{e} \mid\ ::\ \mathsf{v}\ \mathsf{F}\ \mathsf{e} \mid\ ::\ \mathsf{v}\ \mathsf{v}\ \mathsf{F} \\
 & \mid & \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \rightarrow \mathsf{e} \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e} \\
 & \mid & \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \rightarrow \mathsf{e} \mid\ ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e}
\end{array}
$$

$\boxed{\text{Plugging Function } \mathsf{plug}\ F\ \mathsf{e} = \mathsf{e}'}$

$$
\begin{array}{rcl}
\mathsf{plug}\ [\,]\ \mathsf{e} & = & \mathsf{e} \\
\mathsf{plug}\ (\mathsf{F}\ \mathsf{e}_1)\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (\mathsf{e}\ \mathsf{e}_1) \\
\mathsf{plug}\ (\mathsf{v}_0\ \mathsf{F})\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (\mathsf{e}_0\ \mathsf{e}) \\
\mathsf{plug}\ (\mathsf{suc}\ \mathsf{F})\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (\mathsf{suc}\ \mathsf{e}) \\
\mathsf{plug}\ (::\ \mathsf{F}\ \mathsf{e}_1\ \mathsf{e}_2)\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (::\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2) \\
\mathsf{plug}\ (::\ \mathsf{e}_0\ \mathsf{F}\ \mathsf{e}_2)\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (::\ \mathsf{e}_0\ \mathsf{e}\ \mathsf{e}_2) \\
\mathsf{plug}\ (::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{F})\ \mathsf{e} & = & \mathsf{plug}\ \mathsf{F}\ (::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}) \\
\begin{array}{r}\mathsf{plug}\ (\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\ \mathsf{z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e}_2)\ \mathsf{e}\end{array} & = & \begin{array}{r}\mathsf{plug}\ \mathsf{F}\ (\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\ \mathsf{z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e}_2)\end{array} \\
\begin{array}{r}\mathsf{plug}\ (\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\ \mathsf{nil} \rightarrow \mathsf{e}_1 \mid\ ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e}_2)\ \mathsf{e}\end{array} & = & \begin{array}{r}\mathsf{plug}\ \mathsf{F}\ (\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\ \mathsf{nil} \rightarrow \mathsf{e}_1 \mid\ ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e}_2)\end{array}
\end{array}
$$

Figure 3.2: Dellina- Evaluation Contexts

$\boxed{\text{Reduction Rules } \mathsf{e} \rhd \mathsf{e}'}$

$$
\begin{array}{rcl}
(\lambda\, x : A.\, e)\ v & \rhd_\beta & e[v/x] \\
(\mathrm{rec}\ f_{\Pi\, x : A.\, B\ \rho}\, x.\, e)\ v & \rhd_\mu & e[\mathrm{rec}\ f_{\Pi\, x : A.\, B\ \rho}\, x.\, e/f, v/x] \\
\begin{array}{r}\mathrm{pm}\ z\ \mathrm{as}\ x\ \mathrm{in}\ \mathbb{N}\ \mathrm{ret}\ P\ \mathrm{with} \\ z \to e_1 \mid \mathrm{suc}\ n \to e_2\end{array} & \rhd_\iota & e_1 \\
\begin{array}{r}\mathrm{pm}\ \mathrm{suc}\ v\ \mathrm{as}\ x\ \mathrm{in}\ \mathbb{N}\ \mathrm{ret}\ P\ \mathrm{with} \\ z \to e_1 \mid \mathrm{suc}\ n \to e_2\end{array} & \rhd_\iota & e_2[v/n] \\
\begin{array}{r}\mathrm{pm}\ \mathrm{nil}\ \mathrm{as}\ x\ \mathrm{in}\ L\ a\ \mathrm{ret}\ P\ \mathrm{with} \\ \mathrm{nil} \to e_1 \mid ::\ m\ h\ t \to e_2\end{array} & \rhd_\iota & e_1 \\
\begin{array}{r}\mathrm{pm}\ ::\ v_0\ v_1\ v_2\ \mathrm{as}\ x\ \mathrm{in}\ L\ a\ \mathrm{ret}\ P\ \mathrm{with} \\ \mathrm{nil} \to e_1 \mid ::\ m\ h\ t \to e_2\end{array} & \rhd_\iota & e_2[v_0/m, v_1/h, v_2/t] \\
\langle F[\mathcal{S}k : A \to \alpha.\, e]\rangle & \rhd_{\mathcal{S}} & \langle e[\lambda\, x : A.\, \langle F[x]\rangle/k]\rangle \\
\langle v\rangle & \rhd_{\mathcal{R}} & v
\end{array}
$$

$\boxed{\text{Single-step Evaluation}}$

$$\frac{e\ \rhd\ e'}{E[e]\ \rhd\ E[e']}\ \text{(R-Eval)}$$

$\boxed{\text{Multi-step Evaluation } \mathsf{e}\ \rhd^\star\ \mathsf{e}'}$

$$\frac{}{e\ \rhd^\star\ e}\ \text{(RS-Refl)} \qquad \frac{e_0\ \rhd\ e_1 \qquad e_1\ \rhd^\star\ e_2}{e_0\ \rhd^\star\ e_2}\ \text{(RS-Trans)}$$

Figure 3.3: Dellina- Runtime Reduction Rules

We next define reduction rules in Figure 3.3. These rules take the form $\mathsf{e} \triangleright \mathsf{e}'$, where e is a redex and e' is a reduct. The first rule is the familar β-reduction, which happens when a function is applied to a value. We use the notation $\mathsf{e}[\mathsf{v}/\mathsf{x}]$ to mean capture-avoiding substitution of v for free occurrences of x in e. The μ-reduction is similar to β-reduction, and is used for application of a recursive function; notice that the rule involves an extra substitution for the name f of the function. We have four rules for pattern matching, called ι-rules, which choose an appropriate branch and perform substitution for pattern variables. Since Dellina- is call-by-value, ι-rules only apply when all constructor arguments have reduced to values. The last two rules account for the control constructs. We eliminate a `shift` operator when it is surrounded by an F-context and a `reset` clause. In the post-reduction expression, k is replaced by the function $\lambda\,\mathsf{x} : \mathsf{A}.\,\langle \mathsf{F}[\mathsf{x}] \rangle$ representing the context that was surrounding the `shift` construct. The last rule lets us drop a `reset` surrounding a value. This reflects the fact that a value-surrounding `reset` is redundant, since values cannot have any control effects.

Rule (R-Eval) defines single-step evaluation: if e reduces to e' via one of the rules listed in Figure 3.3, then $\mathsf{E}[\mathsf{e}]$ reduces to $\mathsf{E}[\mathsf{e}']$. Single-step evaluation is extended to multi-step evaluation $\mathsf{e} \triangleright^{\star} \mathsf{e}'$ by the reflexivity rule (RS-Refl), and the transitivity rule (RS-Trans).

3.2.2 Parallel Reduction

As we saw in Section 2.3, type checking a dependently typed program involves reduction of terms that appear in types. This makes the notion of type equivalence dependent on reduction. What this implies is that the choice of reduction strategy affects typability of programs: the more terms a strategy rewrites, the more types it equates. Consider for instance the following terms:

$$\lambda\,\mathsf{x} : \mathbb{N}.\,1 + 1 \qquad \lambda\,\mathsf{x} : \mathbb{N}.\,2$$

Under the call-by-value strategy, we do not reduce under binders, because we do not have an evaluation context of the form $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{E}$. Therefore, the former term does not reduce to the latter. This means, if we defined equivalence in terms of runtime evaluation, we would not be able to equate types dependent on these terms.

Call-by-value reduction also poses several challenges in the proof of the preservation theorem. We defer the details until Section 3.4.5, but the intuition is that reduction of terms requires corresponding reduction of their type, and the type-level reduction is not always valid under the call-by-value strategy.

For these reasons, we define a more generous notion of reduction, called *parallel reduction* [158], and discuss equivalence of expressions in terms of this reduction.

Figures 3.4 – 3.6 show the definition of parallel reduction $\mathsf{t} \rhd_p \mathsf{t}'$. As the name suggests, parallel reduction reduces all subterms—including those under binders, and those serving as type annotations—in parallel. Reduction of annotations means that parallel reduction is defined not only on terms, but on types as well. However, what happens at the level of types is actually term-level reduction, because Dellina- has no construct that gives rise to type-level computation. That is, the reduction on types simply recurses on the structure of types, and when it encounters $\mathsf{L}\ \mathsf{e}$, it applies term-level reduction to the length index e. Note that reduction on a non-empty effect annotation $[\alpha, \beta]$ is defined as $[\alpha', \beta']$ when $\alpha \rhd_p \alpha'$ and $\beta \rhd_p \beta'$.

We will hereafter write $\mathsf{t} \rhd_p^\star \mathsf{t}'$ to mean the reflexive and transitive closure of the $\rhd_p$-relation, as defined by rules (PS-Refl) and (PS-Trans).

3.2.3 Equivalence

Using prarllel reduction, we define the notion of equivalence (Figure 3.7). We say two expressions $\mathsf{t}_1, \mathsf{t}_2$ are equivalent when they parallel-reduce to a common expression t. Since we are using parallel reduction, we can equate the terms $\lambda \mathsf{x} : \mathbb{N}.\, 1 + 1$ and $\lambda \mathsf{x} : \mathbb{N}.\, 2$, as well as the types $\mathsf{A}(\lambda \mathsf{x} : \mathbb{N}.\, 1 + 1)$ and $\mathsf{A}(\lambda \mathsf{x} : \mathbb{N}.\, 2)$. The derived type equivalence is used in the type conversion rule, as we will see in the next section.

An important design decision of our equivalence is that it is *untyped*, *i.e.*, we do not require the left- and right-hand sides of $\equiv$ to have the same type, or even be well-typed. This is in contrast to the above-mentioned previous studies, which require t_1 and t_2 to have *some* (possibly different) type. The reason we use an untyped equivalence is that the typing preconditions on the two expressions would introduce an unfortunate circularity into the type preservation argument of the CPS translation. This problem has been noticed by others [23, 34], and will be

$$\frac{}{\mathsf{t} \rhd_p \mathsf{t}}\ (\text{P-Refl})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{L\ e} \rhd_p \mathsf{L\ e}'}\ (\text{P-List})$$

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \qquad \mathsf{B} \rhd_p \mathsf{B}' \qquad \rho \rhd_p \rho'}{\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho \rhd_p \Pi\,\mathsf{x} : \mathsf{A}'.\,\mathsf{B}'\ \rho'}\ (\text{P-Pi})$$

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \qquad \mathsf{e} \rhd_p \mathsf{e}'}{\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e} \rhd_p \lambda\,\mathsf{x} : \mathsf{A}'.\,\mathsf{e}'}\ (\text{P-Abs})$$

$$\frac{\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho \rhd_p \Pi\,\mathsf{x} : \mathsf{A}'.\,\mathsf{B}'\ \rho' \qquad \mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{rec}\ \mathsf{f}_{\Pi\,\mathsf{x}:\mathsf{A}.\,\mathsf{B}\ \rho}\,\mathsf{x}.\,\mathsf{e} \rhd_p \mathsf{rec}\ \mathsf{f}_{\Pi\,\mathsf{x}:\mathsf{A}'.\,\mathsf{B}'\ \rho'}\,\mathsf{x}.\,\mathsf{e}'}\ (\text{P-Rec})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \qquad \mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{e}_0\ \mathsf{e}_1 \rhd_p \mathsf{e}_0'\ \mathsf{e}_1'}\ (\text{P-App})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \qquad \mathsf{v}_1 \rhd_p \mathsf{v}_1'}{(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_0)\ \mathsf{v}_1 \rhd_p \mathsf{e}_0'[\mathsf{v}_1'/\mathsf{x}]}\ (\text{P-AppBeta})$$

$$\frac{\begin{array}{c}\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho \rhd_p \Pi\,\mathsf{x} : \mathsf{A}'.\,\mathsf{B}'\ \rho' \\ \mathsf{e}_0 \rhd_p \mathsf{e}_0' \qquad \mathsf{v}_1 \rhd_p \mathsf{v}_1'\end{array}}{(\mathsf{rec}\ \mathsf{f}_{\Pi\,\mathsf{x}:\mathsf{A}.\,\mathsf{B}\ \rho}\,\mathsf{x}.\,\mathsf{e}_0)\ \mathsf{v}_1 \rhd_p \mathsf{e}_0'[\mathsf{rec}\ \mathsf{f}_{\Pi\,\mathsf{x}:\mathsf{A}'.\,\mathsf{B}'\ \rho'}\,\mathsf{x}.\,\mathsf{e}_0'/\mathsf{f}, \mathsf{v}_1'/\mathsf{x}]}\ (\text{P-AppMu})$$

Figure 3.4: Dellina- Parallel Reduction (Types and λ-terms)

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{suc}\ \mathsf{e} \rhd_p \mathsf{suc}\ \mathsf{e}'}\ (\text{P-Suc})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 \rhd_p\ ::\ \mathsf{e}_0'\ \mathsf{e}_1'\ \mathsf{e}_2'}\ (\text{P-Cons})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm}\ \mathsf{e}'\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}'\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1' \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2'\end{array}}\ (\text{P-MatchN})$$

$$\frac{\mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{pm}\ \mathsf{z}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 \rhd_p \mathsf{e}_1'}\ (\text{P-MatchZero})$$

$$\frac{\mathsf{v} \rhd_p \mathsf{v}' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\mathsf{pm}\ \mathsf{suc}\ \mathsf{v}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 \rhd_p \mathsf{e}_2'[\mathsf{v}'/\mathsf{n}]}\ (\text{P-MatchSuc})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm}\ \mathsf{e}'\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}'\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1' \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2'\end{array}}\ (\text{P-MatchL})$$

$$\frac{\mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{pm}\ \mathsf{nil}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 \rhd_p \mathsf{e}_1'}\ (\text{P-MatchNil})$$

$$\frac{\mathsf{v}_0 \rhd_p \mathsf{v}_0' \quad \mathsf{v}_1 \rhd_p \mathsf{v}_1' \quad \mathsf{v}_2 \rhd_p \mathsf{v}_2' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm}\ ::\ \mathsf{v}_0\ \mathsf{v}_1\ \mathsf{v}_2\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 \rhd_p \\ \mathsf{e}_2'[\mathsf{v}_0'/\mathsf{m}, \mathsf{v}_1'/\mathsf{h}, \mathsf{v}_2'/\mathsf{t}]\end{array}}\ (\text{P-MatchCons})$$

Figure 3.5: Dellina- Parallel Reduction (Inductive Data)

$$\frac{\mathsf{A} \to \alpha \ \triangleright_p \ \mathsf{A}' \to \alpha' \qquad \mathsf{e} \triangleright_p \mathsf{e}'}{\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} \triangleright_p \mathcal{S}\mathsf{k} : \mathsf{A}' \to \alpha'.\, \mathsf{e}'} \ (\text{P-Shift})$$

$$\frac{\mathsf{e} \triangleright_p \mathsf{e}'}{\langle \mathsf{e} \rangle \triangleright_p \langle \mathsf{e}' \rangle} \ (\text{P-Reset})$$

$$\frac{\mathsf{A} \triangleright_p \mathsf{A}' \qquad \mathsf{F}[\mathsf{x}] \triangleright_p \mathsf{F}'[\mathsf{x}] \qquad \mathsf{e} \triangleright_p \mathsf{e}'}{\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \triangleright_p \langle \mathsf{e}'[\lambda\, \mathsf{x} : \mathsf{A}'.\, \mathsf{F}'[\mathsf{x}]/\mathsf{k}] \rangle} \ (\text{P-ResetS})$$

$$\frac{\mathsf{v} \triangleright_p \mathsf{v}'}{\langle \mathsf{v} \rangle \triangleright_p \mathsf{v}'} \ (\text{P-ResetV})$$

$$\frac{}{\mathsf{t} \triangleright_p^\star \mathsf{t}} \ (\text{PS-Refl}) \qquad \frac{\mathsf{t}_0 \triangleright_p \mathsf{t}_1 \qquad \mathsf{t}_1 \triangleright_p^\star \mathsf{t}_2}{\mathsf{t}_0 \triangleright_p^\star \mathsf{t}_2} \ (\text{PS-Trans})$$

Figure 3.6: Dellina- Parallel Reduction (Control Operators, Reflexibity, Transitivity)

$$\frac{\mathsf{t}_0 \triangleright_p^\star \mathsf{t} \qquad \mathsf{t}_1 \triangleright_p^\star \mathsf{t}}{\mathsf{t}_1 \equiv \mathsf{t}_2} \ (\equiv)$$

Figure 3.7: Dellina- Equivalence

elaborated in Section 4.5.

Remark In dependently typed languages, reduction is often defined on an implicitly typed version of the language, where all type annotations have been erased. This is because annotations do not affect the runtime behavior of programs; they are present only for the type-checking purposes. Equivalence of the explicitly typed language, then, is defined via an erasure function $|\,\mathsf{t}\,|$: two expressions $\mathsf{t}_1, \mathsf{t}_2$ are equivalent if their annotation-free counterparts $|\,\mathsf{t}_1\,|, |\,\mathsf{t}_2\,|$ reduce to a common expression t.

We have designed Dellina- as an explicitly typed language, and defined reduction directly on this language. The reason mainly comes from the presence of control effects and our fine-grained purity distinction. As reported in the literature [12], the absence of type annotations may allow terms to have multiple derivations concluding with different types. Consider the following function[5]:

$$\lambda\,\mathsf{f}.\,\lambda\,\mathsf{g}. :: 1\ 2\ \langle \mathsf{f}\ 3 + \mathsf{g}\ 4 \rangle$$

Since addition reduces to a natural number, we know that either f, or g, or both, change the answer type in the course of evaluation. However, we do not know exactly which. As a consequence, the two types below can both be assigned to the above function:

$$(\mathbb{N} \rightarrow \mathbb{N}[\mathbb{N}, \mathsf{L}\ 1]) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathsf{L}\ 2 \quad \text{and} \quad (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}[\mathbb{N}, \mathsf{L}\ 1]) \rightarrow \mathsf{L}\ 2$$

This poses a problem to the type-preservation argument of the CPS translation. Since our translation converts function application differently depending on the function's purity, the above function is mapped to two distinct CPS images. As a consequence, we lose the guarantee that the translation preserves equivalence, at least in the usual sense. If we define a more advanced notion of equivalence (via logical relations [135]), it might be possible to relate CPS images of different derivations [29], but here we avoid the additional complication by simply adding

[5]This is a modified version of the example from Asai and Uehara [12], who develop an effect-annotating algorithm for a simply and implicitly typed language with `shift` and `reset`. Their algorithm is designed to yield as many pure terms as possible, but this example shows that there is no best annotation in general.

type annotations.

3.3 Typing

In this section, we define typing rules of Dellina-. The rules play a key role in our language, as they enforce the three restrictions on type dependency. Remember that our motto is:

1. Types do not depend on impure terms;
2. Continuations are non-depnedent functions; and
3. Final answer types do not refer to continuations.

As we will see below, we rule out dependency on impure terms by defining separate rules for the dependent and non-dependent forms of elimination constructs (*i.e.*, application and pattern matching). The other two dependencies are excluded by two premises we put in the rule for the `shift` construct.

In Figure 3.8, we define well-formed typing environments, kinds, and types. The first two rules are completely standard. (G-Empty) tells us that an empty environment $\bullet$ is unconditionally well-formed. (G-Ext) states that an extended context $\Gamma, x : A$ is well-formed if Γ is well-formed, and the type A is well-formed in Γ. As the second premise suggests, determining well-formedness of types requires an environment, since types may contain terms. Hence, when we extend an environment with a type A, we must make sure that A only refers to variables that are currently available. For instance, when $\Gamma = \bullet, a : \mathbb{N}$, we can extend Γ with $x : \mathsf{L}\ a$, but not $x : \mathsf{L}\ b$.

The next three rules, (K-Star), (T-Unit), and (T-Nat), share a common pattern: they all require well-formedness of the typing environment Γ in the conclusion. As these rules all deal with a constant, we do not actually use the environment to determine the well-formedness of the subject, but we still need the premise to guarantee that every derivation uses a well-formed environment.

Rule (T-List) tells us how to form a list type $\mathsf{L}\ e$. We can see that the rule requires e to be a *pure* term of type $\mathbb{N}$. The purity restriction comes from our principle that types may depend only on pure terms.

$\boxed{\text{Well-formed Environments } \vdash \Gamma}$

$$\frac{}{\vdash \bullet}\ (\text{G-Empty}) \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \mathsf{A} : *}{\vdash \Gamma, \mathsf{x} : \mathsf{A}}\ (\text{G-Ext})$$

$\boxed{\text{Well-formed Kinds } \Gamma \vdash \kappa : \Box}$

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \Box}\ (\text{K-Star})$$

$\boxed{\text{Well-formed Types } \Gamma \vdash \mathsf{A} : *}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Unit} : *}\ (\text{T-Unit}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : *}\ (\text{T-Nat}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathbb{N}}{\Gamma \vdash \mathsf{L}\ \mathsf{e} : *}\ (\text{T-List})$$

$$\frac{\Gamma \vdash \mathsf{A} : * \quad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : * \quad \Gamma, \mathsf{x} : \mathsf{A} \vdash \rho}{\Gamma \vdash \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho : *}\ (\text{T-Pi})$$

Figure 3.8: Dellina- Well-formed Environments, Kinds and Types

$\boxed{\text{Well-typed Terms } \Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho}$

$$\frac{\vdash \Gamma \quad \mathsf{x} : \mathsf{A} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{A}} \text{ (E-Var)}$$

$$\frac{\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho} \text{ (E-Abs)}$$

$$\frac{\begin{array}{c}\Gamma, \mathsf{f} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho \\ \Gamma \vdash \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho : * \quad \text{guard}(\mathsf{f}, \mathsf{x}, \mathsf{e}, \{\,\})\end{array}}{\Gamma \vdash \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho}\, \mathsf{x}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho} \text{ (E-Rec)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_0 : (\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \tau)\ \rho \quad \Gamma \vdash \mathsf{e}_1 : \mathsf{A} \\ \Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : * \quad \nu = \text{comp}(\rho, \tau[\mathsf{e}_1/\mathsf{x}]) \quad \Gamma \vdash \nu\end{array}}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}]\ \nu} \text{ (E-DApp)}$$

$$\frac{\Gamma \vdash \mathsf{e}_0 : (\mathsf{A} \to \mathsf{B}\ \tau)\ \rho \quad \Gamma \vdash \mathsf{e}_1 : \mathsf{A}\ \sigma \quad \nu = \text{comp}(\rho, \sigma, \tau)}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}\ \nu} \text{ (E-NDApp)}$$

Figure 3.9: Dellina- Typing Rules (Lambda Terms)

The last rule (T-Pi) accounts for function types. A function type $\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$ is well-formed if its subcomponents A, B, and ρ are well-formed. The notion of well-formed effect annotations is simple: $\Gamma \vdash \rho$ trivially holds when $\rho = \epsilon$, and requires $\Gamma \vdash \alpha : *$ and $\Gamma \vdash \beta : *$ when $\rho = [\alpha, \beta]$. An important observation is that well-formedness of B and ρ is checked in the extended environment $\Gamma, \mathsf{x} : \mathsf{A}$, judstifying reference to x from these components.

Now we look at typing rules for terms. Let us sort the rules into several groups, and elaborate them one by one.

Values We begin with the rules for values, which we show (together with application rules) in Figure 3.9. A variable x has type A if there is a declaration $\mathsf{x} : \mathsf{A}$ in the environment Γ. An abstraction is given two different types depending on the purity of its body e. If e is a pure term of type B (*i.e.*, if $\rho = \epsilon$), the whole abstraction is given a pure function type $\Pi\, x : A.\, B$. If e is an impure term that modifies the answer type from α to β (*i.e.*, if $\rho = [\alpha, \beta]$), we give the abstraction an impure function type $\Pi\, x : A.\, B[\alpha, \beta]$.

The rule for recursive functions is different from abstractions in three ways. First, the rule allows references to f from the body e by making the type information of f available during type checking of e. Second, the rule requires the type $\Pi\, x : A.\, B\ \rho$ of the function to be well-formed under Γ, in order to exclude dependency of B on f (which would be allowed by the first premise). Thirdly, the rule checks if f is *guarded* in the body e, via the premise guard(f, x, e, V). The guard condition ensures termination of recursive functions by requiring every recursive call to be made on a structurally smaller argument [83]. Among the four components, f is the name of the recursive function, x is its parameter, e is a subexpression of f's body, and V is a set of variables f is allowed to be applied to. This variable set is initially an empty set, and is extended while we recurse on the body e. Formally, the guard condition is defined as follows:

- When f does not appear in e, guard(f, x, e, V) holds without any condition.
- When $e = L\ e'$, guard(f, x, e', V).
- When $e = \Pi\, x' : A.\, B\ \rho$, guard$(f, x, A, V)$, guard$(f, x, B, V)$, and guard$(f, x, \rho, V)$.
- When $e = \lambda\, x' : A.\, e'$, guard$(f, x, A, V)$ and guard(f, x, e', V).
- When $e = \mathsf{rec}\ f'_{\Pi\, x : A'.\, B'\ \rho'}\ x'.\, e'$, guard$(f, x, A', V)$, guard$(f, x, B', V)$, guard$(f, x, \rho', V)$, and guard$(f, x, e', V)$.
- When $e = e_0\ e_1$,
  - If $e_0 = f$, $e_1 \in V$.
  - Otherwise, guard(f, x, e_0, V) and guard(f, x, e_1, V).
- When $e = \mathsf{suc}\ e'$, guard(f, x, e', V).
- When $e = ::\ e_0\ e_1\ e_2$, guard(f, x, e_0, V), guard(f, x, e_1, V), and guard(f, x, e_2, V).
- When $e = \mathsf{pm}\ y\ \mathsf{as}\ x'\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ P\ \mathsf{with}\ z \to e_1 \,|\, \mathsf{suc}\ n \to e_2$,

  - If $\mathsf{y} = \mathsf{x}$, guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_1, \mathsf{V}$) and guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_2, \mathsf{V} \cup \{\mathsf{n}\}$).
  - Otherwise, guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_1, \mathsf{V}$) and guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_0, \mathsf{V}$).

- When $\mathsf{e} = \mathsf{pm\ y\ as\ x'\ in\ L\ a\ ret\ P\ with\ nil} \rightarrow \mathsf{e}_1 \mid :: \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2$,
  - If $\mathsf{y} = \mathsf{x}$, guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_1, \mathsf{V}$) and guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_2, \mathsf{V} \cup \{\mathsf{t}\}$).
  - Otherwise, guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_1, \mathsf{V}$) and guard($\mathsf{f}, \mathsf{x}, \mathsf{e}_2, \mathsf{V}$).

- If $\mathsf{e} = \mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e}'$ or $\langle \mathsf{e}' \rangle$, guard($\mathsf{f}, \mathsf{x}, \mathsf{e}', \mathsf{V}$).

Observe that, in the case where e is a pattern matching scrutinizing x, we use an extended variable set $\mathsf{V} \cup \{\mathsf{n}\}$ or $\mathsf{V} \cup \{\mathsf{t}\}$ for guardedness check on the second branch e_2. The added variables are called *recursive arguments* of suc and $::$, as they inhabit the same datatype as the whole pattern. Now, if we look at the application case $\mathsf{f}\ \mathsf{e}_1$, we can see that the guard condition holds if e_1 is a member of V, *i.e.*, if it is a recursive argument of the datum we are scrutinizing. Since a recursive argument is smaller than the scrutinee by one constructor, any recursive function that passes guardedness checking must terminate.

Application We next look at rules for application, which is defined at the bottom of Figure 3.9. The first one, (E-DApp), derives a *dependent* application. We see that the function e_0 has a dependent Π type, which means the result type of this application involves substitution of the argument e_1 for the free occurrences of x in B. Since we do not allow dependency on impure terms, we require that the argument e_1 is a pure term, that is, its derivation ends with a judgment that has no effect annotation.

The second rule, (E-NDApp), derives a *non-dependent* application. Here, we see that the argument e_1 is potentially an impure term, which means it is unsafe to substitute this term into the result type. Therefore, we require that the function e_0 has a non-dependent $\rightarrow$ type, that is, the co-domain does not refer to the argument.

By separating dependent and non-dependent rules in this way, we obtain the guarantee that, if an application is judged well-typed, then it can never be the case that the result type depends on an impure argument. However, these rules are not sufficient to ensure the result type and effect annotation are well-formed; we need

$$\frac{\vdash \Gamma}{\Gamma \vdash () : \mathsf{Unit}}\ (\text{E-Unit})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ (\text{E-Zero}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho}{\Gamma \vdash \mathsf{suc}\ \mathsf{e} : \mathbb{N}\ \rho}\ (\text{E-Suc})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nil} : \mathsf{L}\ \mathsf{z}}\ (\text{E-Nil}) \frac{\Gamma \vdash \mathsf{e}_0 : \mathbb{N} \quad \Gamma \vdash \mathsf{e}_1 : \mathbb{N}\ \rho \quad \Gamma \vdash \mathsf{e}_2 : (\mathsf{L}\ \mathsf{e}_0)\ \sigma \\ \tau = \mathrm{comp}(\rho, \sigma)}{\Gamma \vdash ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 : (\mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0))\ \tau}\ (\text{E-Cons})$$

Figure 3.10: Dellina- Typing Rules (Inductive Data)

explicit proofs $\Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : *$ and $\Gamma \vdash \tau[\mathsf{e}_1/\mathsf{x}]$ in the dependent rule (E-DApp)[6]. The need for these premises comes from the call-by-value nature of Dellina-: when e_1 is a non-value, some reduction that takes place in B can be blocked by the substitution $\mathsf{B}[\mathsf{e}_1/\mathsf{x}]$. We will revisit this issue in Section 3.4.3.

Turning our viewpoint to the effect annotations, we see that the chaining of answer types goes the same way as we described in Section 2.1. In (E-DApp), the argument e_1 is a pure term, hence the composition operator only takes two arguments: ρ, which represents the effect of the function e_0, and $\tau[\mathsf{e}_1/\mathsf{x}]$, which represents the effect of its body. We pass the two effect arguments in this order because evaluation of the function happens before that of the body. In (E-NDApp), the effect σ of the argument comes in between the other two effects. So, if all these effects are non-empty, we will have:

$$\rho = [\gamma, \delta], \quad \sigma = [\beta, \gamma], \quad \tau = [\alpha, \beta], \quad \text{and} \quad \nu = [\alpha, \delta]$$

for some α, β, γ, and δ.

Inductive Data Having seen how to type application, it would be easy to see how to type inductive data. In Figure 3.10, we present rules for unit, natural numbers, and indexed lists. Rules (E-Unit) and (E-Zero) are similar to those

[6]Substitution on effect annotations distributes to their constituent types, if any. That is, $\tau[\mathsf{e}_1/\mathsf{x}] = \epsilon$ when $\tau = \epsilon$ and $[\alpha[\mathsf{e}_1/\mathsf{x}], \beta[\mathsf{e}_1/\mathsf{x}]]$ when $\tau = [\alpha, \beta]$.

for constant kinds and types: they have a premise stating that the environment Γ is well-formed. (E-Suc) is also simple, but it would be worth noting that the effect of $\mathsf{suc}\ \mathsf{e}$ is determined by the effect of e, since evaluation of $\mathsf{suc}\ \mathsf{e}$ requires evaluation of e. (E-Nil) and (E-Cons) tell us how list indices are determined: an empty list is indexed by z, and consing an element to an e_0-indexed list yields a $\mathsf{suc}\ \mathsf{e}_0$-indexed list. Notice that the rule does not allow impure e_0, because it would make the result type $\mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0)$ ill-formed[7]. The other two arguments, e_1 and e_2, may be effectful, as long as their effects ρ and σ compose.

Pattern Matching What we saw in Figure 3.10 are the introduction rules of datatypes. The elimination rules, namely rules for pattern matching, are shown in Figure 3.11. As mentioned earlier, Dellina- supports dependent pattern matching, where the return type P may depend on the scrutinee. Thus, we have two rules for each pattern matching construct, similarly to application. Let us look at (E-DMatchN), where we scrutinize a pure term e of type $\mathbb{N}$. The first branch, e_1, is what we compute when e has evaluated to zero. We find that the rule requires e_1 to have type $\mathsf{P}[\mathsf{z}/\mathsf{x}]$ (we are ignoring the effect annotation for readability), *i.e.*, a variant of the return type P where x is replaced by the current pattern z. The second branch, e_2, is what we compute when e has evaluated to the successor of some natural number n. We see that the rule requires e_2 to have type $\mathsf{P}[\mathsf{suc}\ \mathsf{n}/\mathsf{x}]$ under the assumption $\mathsf{n} : \mathbb{N}$, that is, the variable x is instantiated to an arbitrary non-zero number. If both branches have the correct type, we conclude that the whole pattern matching construct has type $\mathsf{P}[\mathsf{e}/\mathsf{x}]$, which depends on the actual scrutinee e.

The three different instantiations of variable x allow us to view the typing rule as an induction principle of natural numbers: if $\mathsf{P}(\mathsf{z})$ holds, and $\mathsf{P}(\mathsf{n}+1)$ holds for an arbitrary n, then $\mathsf{P}(\mathsf{e})$ holds for any natural number e. Indeed, we use symbol P to communicate with the reader that it corresponds to a predicate. The dependency is however not allowed in (E-NDMatchN), where e is an impure term. The reason is, of course, that substitution of an impure e into P results in

[7]We can see that the argument e_0 is present only for the type checking purpuse. Indeed, if the list type had no length index, the argument would be redundant. This means e_0 is not supposed to be used in the computation, *i.e.*, it is *computationally irrelevant*. Therefore, the purity restriction on e_0 does not limit computational expresiveness of the language.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N} \quad \Gamma, \mathsf{x} : \mathbb{N} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{x}] \quad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ n}/\mathsf{x}]\ \rho[\mathsf{suc\ n}/\mathsf{x}] \\ \Gamma \vdash \mathsf{P}[\mathsf{e}/\mathsf{x}] : * \quad \Gamma \vdash \rho[\mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in}\ \mathbb{N}\ \mathsf{ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \,|\, \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}[\mathsf{e}/\mathsf{x}]\ \rho[\mathsf{e}/\mathsf{x}]} \text{ (E-DMatchN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho \quad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \quad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ \_\ in}\ \mathbb{N}\ \mathsf{ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \,|\, \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}\ \tau} \text{ (E-NDMatchN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{L\ n} \quad \Gamma, \mathsf{a} : \mathbb{N}, \mathsf{x} : \mathsf{L\ a} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}] \\ \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L\ m} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ m}/\mathsf{a}, ::\ \mathsf{m\ h\ t}/\mathsf{x}]\ \rho[\mathsf{suc\ m}/\mathsf{a}, ::\ \mathsf{m\ h\ t}/\mathsf{x}] \\ \Gamma \vdash \mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}] : * \quad \Gamma \vdash \rho[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \rightarrow \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2 : \mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]\ \rho[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]} \text{ (E-DMatchL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : (\mathsf{L\ n})\ \rho \quad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \quad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L\ m} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ \_\ in\ L\ \_\ ret\ P\ with\ nil} \rightarrow \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2 : \mathsf{P}\ \tau} \text{ (E-NDMatchL)}$$

Figure 3.11: Dellina- Typing Rules (Pattern Matching)

$$\frac{\begin{array}{c}\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \beta \ \text{ or } \ \Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta] \\ \Gamma \vdash \beta : *\end{array}}{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} : \mathsf{A}[\alpha, \beta]} \ (\text{E-Shift})$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \ \text{ or } \ \Gamma \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \mathsf{A}]}{\Gamma \vdash \langle \mathsf{e} \rangle : \mathsf{A}} \ (\text{E-Reset})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \quad \Gamma \vdash \mathsf{B} : * \quad \Gamma \vdash \sigma \\ \mathsf{A} \equiv \mathsf{B} \quad \rho \equiv \sigma\end{array}}{\Gamma \vdash \mathsf{e} : \mathsf{B}\ \sigma} \ (\text{E-Conv})$$

Figure 3.12: Dellina- Typing Rules (Control Operators and Conversion)

an ill-formed type.

Pattern matching on lists has two similar rules, but there is one difference: in the dependent case, (E-DMatchL), the result type P depends also on the length of the scrutinee, which is represented by variable a. The variable is replaced by z when type checking the first branch e_1, and by $\mathsf{suc\ m}$ when type checking the second branch e_2. Note that there is no need to check the purity of the length n of the scrutinee e, since well-typedness of e implies well-formedness of its type $\mathsf{L\ n}$ (by the regularity property proved in Section 3.4.2), and well-formedness of $\mathsf{L\ n}$ implies purity of n (by (T-List)).

Control Operators and Conversion Lastly, we present rules for control constructs and type conversion in Figure 3.12. Among the three rules, (E-Shift) imposes the two restrictions on the continuation-related dependency, which we have not yet addressed so far. The rule tells us that, a `shift` operator may either have a pure body of type β, or have an impure body that requires an empty context and returns an answer of type β. In either case, we type check the body e with the assumption that the continuation k has a non-dependent arrow type $\mathsf{A} \to \alpha$. This ensures that the initial answer type α of the `shift` construct is closed under the environment Γ. The rule further has a well-formed precondition $\Gamma \vdash \beta : *$ on β. This guarantees that the final answer type β of the `shift` construct is closed under Γ. The next rule, (E-Reset), is identical to the rule from simply typed calculi.

Similarly to (E-Shift), we accept both a pure body and an impure body, and in the pure case, the type of the whole `reset` construct is simply determined by the type of its body.

The conversion rule plays a critical role in a dependently typed language, as it lets us convert between two syntactically different but semantically equivalent types. Specifically, we can change the type of a term from A to B as long as $\mathsf{A} \equiv \mathsf{B}$. For instance, when we have a function f expecting an argument of type $\mathsf{L}\ \mathsf{z}$, and a term e of type $\mathsf{L}\ ((\lambda \mathsf{x} : \mathbb{N}. \mathsf{x})\ \mathsf{z})$, we can make the application $\mathsf{f}\ \mathsf{e}$ well-typed by casting the type of e to $\mathsf{L}\ \mathsf{z}$, since $\mathsf{L}\ ((\lambda \mathsf{x} : \mathbb{N}. \mathsf{x})\ \mathsf{z}) \ \triangleright_p^\star\ \mathsf{L}\ \mathsf{z}$. Note that the rule accounts for conversion of effect annotations as well: we say $[\alpha, \beta] \equiv [\alpha', \beta']$ holds when $\alpha \equiv \alpha'$ and $\beta \equiv \beta'$.

3.4 Metatheory

In this section, we prove metatheoretic properties of Dellina-. Our goal is to show that the type system of Dellina- is *sound*. Following Wright and Felleisen [170], we prove type soundness by showing *preservation* and *progress*. Compared to the simply typed, effect-free setting (*c.f.*, [134]), we have to deal with subtleties introduced by dependent types, and prove additional statements about impure terms. Fortunately, our "pure terms only" restriction on type dependency allows us to combine the proof techniques from other call-by-value dependent languages [151, 38, 37, 150], and simply typed languages with control effects [10].

3.4.1 Confluence

Our first goal is to show the *confluence* property of parallel reduction. Confluence states that if an expression e reduces to e_1 and e_2, then these reducts must reduce to a common expression e'. This property is necessary for showing transitivity of equivalence, which we rely on in the preservation proof. We proceed by first proving basic facts about substitution and reduction, and then proving the main theorem.

3.4.1.1 Substitution and Parallel Reduction

Lemma 3.4.1 (Substitution of Computations and Parallel Reduction)**.** *If* $\mathsf{e} \rhd_p \mathsf{e}'$, *then* $\mathsf{t}[\mathsf{e}/\mathsf{x}] \rhd_p \mathsf{t}[\mathsf{e}'/\mathsf{x}]$.

Proof. The proof is by induction on the structure of t.

Case 1: $\mathsf{t} = \mathsf{x}$
This case is trivial, since $\mathsf{x}[\mathsf{e}/\mathsf{x}] = \mathsf{e}$ and $\mathsf{e} \rhd_p \mathsf{e}'$.

Case 2: $\mathsf{t} = \mathsf{y}$ where $\mathsf{y} \neq \mathsf{x}$
This case is also trivial, since $\mathsf{y}[\mathsf{e}/\mathsf{x}] = \mathsf{y}[\mathsf{e}'/\mathsf{x}] = \mathsf{y}$.

Case 3: $\mathsf{t} = \lambda\, \mathsf{x}' : \mathsf{A}.\, \mathsf{e}_0$
By the induction hypothesis, we have $\mathsf{e}_0[\mathsf{e}/\mathsf{x}] \rhd \mathsf{e}_0[\mathsf{e}'/\mathsf{x}]$. The goal follows by (P-Abs). All other cases are similar.

□

Lemma 3.4.2 (Substitution of Values and Parallel Reduction)**.** *If* $\mathsf{t} \rhd_p \mathsf{t}'$, *then* $\mathsf{t}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{t}'[\mathsf{v}/\mathsf{x}]$.

Proof. The proof is by induction on the derivation of $\mathsf{t} \rhd_p \mathsf{t}'$.

Case 1: (P-AppBeta)
Our goal is to show

$$((\lambda\, \mathsf{x}' : \mathsf{A}.\, \mathsf{e}_0)\ \mathsf{v}_1)[\mathsf{v}/\mathsf{x}] \rhd_p (\mathsf{e}_0'[\mathsf{v}_1'/\mathsf{x}'])[\mathsf{v}/\mathsf{x}]$$

By the induction hypothesis, we have

$$\mathsf{e}_0[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{e}_0'[\mathsf{v}/\mathsf{x}] \;\text{ and }\; \mathsf{v}_1[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{v}_1'[\mathsf{v}/\mathsf{x}]$$

Using (P-AppBeta), we obtain

$$((\lambda\, \mathsf{x}' : \mathsf{A}.\, \mathsf{e}_0)\ \mathsf{v}_1)[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{e}_0'[\mathsf{v}/\mathsf{x}][\mathsf{v}_1'[\mathsf{v}/\mathsf{x}]/\mathsf{x}']$$

The goal follows by the definition of substitution.

□

Lemma 3.4.3 (Substitution and Parallel Reduction)**.** *If* $\mathsf{t} \rhd_p \mathsf{t}'$ *and* $\mathsf{v} \rhd_p \mathsf{v}'$, *then* $\mathsf{t}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{t}'[\mathsf{v}'/\mathsf{x}]$.

Proof. This is a direct consequence of Lemma 3.4.1 and Lemma 3.4.2: we can either first reduce $\mathsf{t}[\mathsf{v}/\mathsf{x}]$ to $\mathsf{t}[\mathsf{v}'/\mathsf{x}]$, and then reduce the result to $\mathsf{t}'[\mathsf{v}'/\mathsf{x}]$, or the other way around. □

3.4.1.2 Confluence of Reduction

Lemma 3.4.4 (One-step Diamond Property of $\rhd_p$)**.** *If* $\mathsf{t} \rhd_p \mathsf{t}_1$ *and* $\mathsf{t} \rhd_p \mathsf{t}_2$*, then there exists some* t' *such that* $\mathsf{t}_1 \rhd_p \mathsf{t}'$ *and* $\mathsf{t}_2 \rhd_p \mathsf{t}'$*.*

Proof. The proof is by induction on the structure of t. Following Sjöberg et al. [151], we consider all possible combinations of the two reductions $\mathsf{t} \ \rhd^\star \ \mathsf{t}_1$ and $\mathsf{t} \ \rhd^\star \ \mathsf{t}_2$.

Case 1: $\mathsf{t} = \mathsf{e}_0\ \mathsf{e}_1$

Sub-Case 1: One reduction is (P-Refl)
This case is trivial.

Sub-Case 2: Both reductions are (P-App)
We have

$$\mathsf{e}_0\ \mathsf{e}_1 \ \rhd_p\ \mathsf{e}_{00}\ \mathsf{e}_{10}$$

where $\mathsf{e}_0 \ \rhd_p\ \mathsf{e}_{00}$ and $\mathsf{e}_1 \ \rhd_p\ \mathsf{e}_{10}$, and

$$\mathsf{e}_0\ \mathsf{e}_1 \ \rhd_p\ \mathsf{e}_{01}\ \mathsf{e}_{11}$$

where $\mathsf{e}_0 \rhd_p \mathsf{e}_{01}$ and $\mathsf{e}_1 \rhd_p \mathsf{e}_{11}$. By the induction hypothesis on e_0, there exists some e_0' such that $\mathsf{e}_{00} \ \rhd_p\ \mathsf{e}_0'$ and $\mathsf{e}_{01} \ \rhd_p\ \mathsf{e}_0'$. Similarly, by the induction hypothesis on e_1, there is an e_1' such that $\mathsf{e}_{10} \rhd_p \mathsf{e}_1'$ and $\mathsf{e}_{11} \rhd_p \mathsf{e}_1'$. Now, we can derive $\mathsf{e}_{00}\ \mathsf{e}_{10} \rhd_p \mathsf{e}_0'\ \mathsf{e}_1'$ and $\mathsf{e}_{01}\ \mathsf{e}_{11} \ \rhd_p\ \mathsf{e}_0'\ \mathsf{e}_1'$ using (P-App), which is exactly what we want.

Sub-Case 3: One reduction is (P-AppBeta)
In this case, t must have the form $(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_0)\ \mathsf{v}_1$.

Sub-Sub-Case 1: The other reduction is (P-App)
We have

$$(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_0)\ \mathsf{v}_1 \ \rhd_p\ \mathsf{e}_{00}[\mathsf{v}_{10}/\mathsf{x}]$$

and

$$(\lambda\, x : A.\, e_0)\ v_1 \ \rhd_p \ (\lambda\, x : A_1.\, e_{01})\ v_{11}$$

By the induction hypothesis, there is an e'_0 such that $e_{00} \ \rhd_p \ e'_0$ and $e_{01} \ \rhd_p \ e'_0$. We also have a similar v'_1. Now, the substitution lemma (Lemma 3.4.3) gives us $e_0[v_1/x] \ \rhd_p \ e'_0[v'_1/x]$, and (P-AppBeta) gives us $(\lambda\, x : A_1.\, e_{01})\ v_{11} \ \rhd_p \ e'_0[v'_1/x]$, which imply the goal.

Sub-Sub-Case 2: The other reduction is (P-AppBeta)
We have

$$(\lambda\, x : A_0.\, e_0)\ v_1 \ \rhd_p \ e_{00}[v_{10}/x]$$

and

$$(\lambda\, x : A_1.\, e_0)\ v_1 \ \rhd_p \ e_{01}[v_{11}/x]$$

By the induction hypothesis, there is an e'_0 such that $e_{00} \ \rhd_p \ e'_0$ and $e_{01} \ \rhd_p \ e'_0$. We also have a similar v'_1. By Lemma 3.4.3, we know that $e_{00}[v_{10}/x] \ \rhd_p \ e'_0[v'_1/x]$ and $e_{01}[v_{11}/x] \ \rhd_p \ e'_0[v'_1/x]$. Therefore the statement holds.

Sub-Case 4: One reduction is (P-AppMu)
This case is analogous to the β-reduction case.

Case 2: $e = \langle e \rangle$

Sub-Case 1: One reduction is (P-Refl)
This case is trivial.

Sub-Case 2: Both reductions are (P-Reset)
We have

$$\langle e \rangle \ \rhd_p \ \langle e_0 \rangle \quad \text{and} \quad \langle e \rangle \ \rhd_p \ \langle e_1 \rangle$$

By the induction hypothesis, there is an e' such that $e_0 \ \rhd_p \ e'$ and $e_1 \ \rhd_p \ e'$. The goal follows by (P-Reset).

Sub-Case 3: One reduction is (P-ResetS)
In this case, t must have the form $\langle F[\mathcal{S}k.\, e] \rangle$.

Sub-Sub-Case 1: The other reduction is (P-Reset)
We have

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \;\rhd_p\; \langle \mathsf{e}_0[\lambda\, \mathsf{x} : \mathsf{A}_0.\, \langle \mathsf{F}_0[\mathsf{x}] \rangle / \mathsf{k}] \rangle$$

and

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \;\rhd_p\; \langle \mathsf{F}_1[\mathcal{S}\mathsf{k} : \mathsf{A}_1 \to \alpha_1.\, \mathsf{e}_1] \rangle$$

By the induction hypothesis, there is an A' such that $\mathsf{A}_0 \;\rhd_p\; \mathsf{A}'$ and $\mathsf{A}_1 \;\rhd_p\; \mathsf{A}'$. We have similar $\mathsf{F}'[\mathsf{x}]$ and e'. The substitution lemma (Lemma 3.4.3) gives us $\langle \mathsf{e}_0[\lambda\, \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}_0[\mathsf{x}] \rangle / \mathsf{k}] \rangle \;\rhd_p\; \langle \mathsf{e}'[\lambda\, \mathsf{x} : \mathsf{A}'.\, \langle \mathsf{F}'[\mathsf{x}] \rangle / \mathsf{k}] \rangle$ (note that $\mathsf{F}[\mathsf{e}] = \mathsf{F}[\mathsf{x}][\mathsf{e}/\mathsf{x}]$). Using (P-ResetS), we can also derive $\langle \mathsf{F}_1[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}_1] \rangle \;\rhd_p\; \langle \mathsf{e}'[\lambda\, \mathsf{x} : \mathsf{A}'.\, \langle \mathsf{F}'[\mathsf{x}] \rangle / \mathsf{k}] \rangle$, which completes the proof.

Sub-Sub-Case 2: The other reduction is (P-ResetS)
We have

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \;\rhd_p\; \langle \mathsf{e}_0[\lambda\, \mathsf{x} : \mathsf{A}_0.\, \langle \mathsf{F}_0[\mathsf{x}] \rangle / \mathsf{k}] \rangle$$

and

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \;\rhd_p\; \langle \mathsf{e}_1[\lambda\, \mathsf{x} : \mathsf{A}_1.\, \langle \mathsf{F}_1[\mathsf{x}] \rangle / \mathsf{k}] \rangle$$

By the induction hypothesis, there is an A' such that $\mathsf{A}_0 \rhd_p \mathsf{A}'$ and $\mathsf{A}_1 \rhd_p \mathsf{A}'$. We have similar F' and e'. By Lemma 3.4.3, we know that $\langle \mathsf{e}_0[\lambda\, \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}_0[\mathsf{x}] \rangle / \mathsf{k}] \rangle \;\rhd_p\; \langle \mathsf{e}'[\lambda\, \mathsf{x} : \mathsf{A}'.\, \langle \mathsf{F}'[\mathsf{x}] \rangle / \mathsf{k}] \rangle$ and $\langle \mathsf{e}_1[\lambda\, \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}_1[\mathsf{x}] \rangle / \mathsf{k}] \rangle \;\rhd_p\; \langle \mathsf{e}'[\lambda\, \mathsf{x} : \mathsf{A}'.\, \langle \mathsf{F}'[\mathsf{x}] \rangle / \mathsf{k}] \rangle$. Therefore the statement holds.

Sub-Case 4: One reduction is (P-ResetV)
In this case, t must have the form $\langle \mathsf{v} \rangle$.

Sub-Sub-Case 1: The other reduction is (P-Reset)
We have

$$\langle \mathsf{v} \rangle \;\rhd_p\; \mathsf{v}_0 \quad\text{and}\quad \langle \mathsf{v} \rangle \;\rhd_p\; \langle \mathsf{v}_1 \rangle$$

By the induction hypothesis, there is a v' such that $\mathsf{v}_0 \;\rhd_p\; \mathsf{v}'$ and $\mathsf{v}_1 \;\rhd_p\; \mathsf{v}'$. The goal follows by (P-ResetV).

Sub-Sub-Case 2: The other reduction is (P-ResetV)
We have

$$\langle \mathsf{v} \rangle \ \triangleright_p \ \mathsf{v}_0 \quad \text{and} \quad \langle \mathsf{v} \rangle \ \triangleright_p \ \mathsf{v}_1$$

By the induction hypothesis, there is a v' such that $\mathsf{v}_0 \triangleright_p \mathsf{v}'$ and $\mathsf{v}_1 \triangleright_p \mathsf{v}'$. Therefore the statement holds.

□

Theorem 3.4.1 (Confluence of $\triangleright_p^\star$)**.** *If* $\mathsf{t} \triangleright_p^\star \mathsf{t}_1$ *and* $\mathsf{t} \triangleright_p^\star \mathsf{t}_2$*, then there exists some* t' *such that* $\mathsf{t}_1 \ \triangleright_p^\star \ \mathsf{t}'$ *and* $\mathsf{t}_2 \ \triangleright_p^\star \ \mathsf{t}'$*.*

Proof. The proof is by induction on the length of the reduction sequence. The base case is trivial. The inductive case follows by the induction hypothesis and Lemma 3.4.4. □

Corollary 3.4.1 (Confluence of $\triangleright^\star$)**.** *The reduction relation* $\triangleright^\star$ *is confluent.*

Proof. This is a direct consequence of Theorem 3.4.1, since $\triangleright^\star$ is a subrelation of $\triangleright_p$. □

3.4.1.3 Properties of Equivalence

Using the confluence theorem, we can easily show that our equivalence is transitive:

Lemma 3.4.5 (Transitivity of Equivalence)**.** *If* $\mathsf{t}_0 \equiv \mathsf{t}_1$ *and* $\mathsf{t}_1 \equiv \mathsf{t}_2$*, then* $\mathsf{t}_0 \equiv \mathsf{t}_2$*.*

Proof. If we have $\mathsf{t}_0 \equiv \mathsf{t}_1$, then we know that there is some t such that $\mathsf{t}_0 \ \triangleright_p^\star \ \mathsf{t}$ and $\mathsf{t}_1 \ \triangleright_p^\star \ \mathsf{t}$. Similarly, if we have $\mathsf{t}_1 \equiv \mathsf{t}_2$, then there is some t' such that $\mathsf{t}_1 \ \triangleright_p^\star \ \mathsf{t}'$ and $\mathsf{t}_2 \ \triangleright_p^\star \ \mathsf{t}'$. By Theorem 3.4.1, the two reductions $\mathsf{t}_1 \ \triangleright_p^\star \ \mathsf{t}$ and $\mathsf{t}_1 \ \triangleright_p^\star \ \mathsf{t}'$ must be confluent, that is, there is some t'' such that $\mathsf{t} \ \triangleright_p^\star \ \mathsf{t}''$ and $\mathsf{t}' \ \triangleright_p^\star \ \mathsf{t}''$. This implies $\mathsf{t}_0 \ \triangleright_p^\star \ \mathsf{t}''$ and $\mathsf{t}_2 \ \triangleright_p^\star \ \mathsf{t}''$, allowing us to derive $\mathsf{t}_0 \equiv \mathsf{t}_2$ via ($\equiv$). □

We can also prove the following injectivity lemma, which we use in the preservation proof:

Lemma 3.4.6 (Injectivity of Equivalence)**.**

1. *If* $\mathsf{L}\ \mathsf{e}_1 \equiv \mathsf{L}\ \mathsf{e}_2$*, then* $\mathsf{e}_1 \equiv \mathsf{e}_2$*.*
2. *If* $\Pi\, \mathsf{x} : \mathsf{A}_1 .\, \mathsf{B}_1\ \rho_1 \equiv \Pi\, \mathsf{x} : \mathsf{A}_2 .\, \mathsf{B}_2\ \rho_2$*, then* $\mathsf{A}_1 \equiv \mathsf{A}_2$*,* $\mathsf{B}_1 \equiv \mathsf{B}_2$*, and* $\rho_1 \equiv \rho_2$*.*

Proof. These facts easily follow by the definition of parallel reduction. □

Lemma 3.4.7 (Substitution and Equivalence)**.** *If* $\mathsf{t} \equiv \mathsf{t}'$*, then* $\mathsf{t}[\mathsf{v}/\mathsf{x}] \equiv \mathsf{t}'[\mathsf{v}/\mathsf{x}]$.

Proof. This is a direct consequence of Lemma 3.4.2. □

3.4.2 Regularity

We next prove regularity: if an expression is well-typed, then its type and effect annotation are also well-formed. Below, we use the judgment $\Gamma \vdash \mathsf{t} : \mathsf{T}\ \rho$ to mean t is a valid expression of any category. In our current setting, this means:

- If $\mathsf{T} = \square$ and $\rho = \epsilon$, then t is a kind
- If $\mathsf{T} = *$ and $\rho = \epsilon$, then t is a type
- Otherwise, t is a term

Lemma 3.4.8 (Environment Regularity)**.** *If* $\Gamma \vdash \mathsf{t} : \mathsf{T}\ \rho$*, then* $\vdash \Gamma$.

Proof. The proof is by induction on the derivation of t.

Case 1: (K-Star), (T-Unit), (T-Nat), (E-Var), (E-Unit), (E-Zero), (E-Nil)
These cases are trivial since the rules require a well-formed context.

Case 2: (E-Abs)
Suppose we have $\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e} : \mathsf{T}$. By the induction hypothesis on the derivation of e, we know that the extended environment $\vdash \Gamma, \mathsf{x} : \mathsf{A}$ is well-formed. This environment must be derived by (G-Ext), which requires well-formedness of Γ. Therefore the statement holds.

Case 3: All other cases
The goal easily follows by the induction hypothesis.

□

Lemma 3.4.9 (Environment Inversion)**.** *If* $\vdash \Gamma, \mathsf{x} : \mathsf{A}, \Gamma'$*, then* $\Gamma \vdash \mathsf{A} : *$.

Proof. The proof is by induction on the derivation of $\Gamma, \mathsf{x} : \mathsf{A}, \Gamma'$.

Case 1: (G-Empty)
This case is impossible.

Case 2: (G-Ext)

Sub-Case 1: $\Gamma' = \bullet$
The goal immediately follows by the premise $\Gamma \vdash A : *$.

Sub-Case 2: $\Gamma' = \Gamma'', x' : A'$
The goal follows by the induction hypothesis on Γ, Γ'.

□

Lemma 3.4.10 (Subderivation of Types)**.** *If* $\Gamma, x : A, \Gamma' \vdash t : T\ \rho$*, then there is a subderivation of* $\Gamma \vdash A : *$.

Proof. This is a consequence of Lemma 3.4.8 and Lemma 3.4.9. □

Lemma 3.4.11 (Weakening)**.** *Suppose* $\Gamma \vdash A : *$ *and* $x \notin \Gamma, \Gamma'$*. Then, the following hold.*

1. *If* $\vdash \Gamma, \Gamma'$*, then* $\vdash \Gamma, x : A, \Gamma'$.
2. *If* $\Gamma, \Gamma' \vdash t : T\ \rho$*, then* $\Gamma, x : A, \Gamma' \vdash t : T\ \rho$.

Proof. The proof is by mutual induction on the derivation of the environment and the expression.

Case 1: (G-Empty)
This case is impossible.

Case 2: (G-Ext)

Sub-Case 1: $\Gamma' = \bullet$
The goal immadiately follows by the well-formedness of Γ and A.

Sub-Case 2: $\Gamma' = \Gamma'', x' : A'$
Our goal is to show

$$\vdash \Gamma, x : A, \Gamma'', x' : A'$$

By the induction hypothesis, we have

$$\vdash \Gamma, x : A, \Gamma'' \quad \text{and} \quad \Gamma, x : A, \Gamma'' \vdash A' : *$$

The goal follows by (G-Ext).

Case 3: (K-Star), (T-Unit), (T-Nat), (E-Var), (E-Unit), (E-Zero), (E-Nil)
These cases can be proven easily using the induction hypothesis on the typing environment.

Case 4: All other cases
The goal follows by the induction hypothesis on the subderivations.

□

Lemma 3.4.12 (Environment Conversion)**.**

1. *If* $\vdash \Gamma, x : A, \Gamma'$*, and* $\Gamma \vdash B : *$ *with* $A \equiv B$*, then* $\vdash \Gamma, x : B, \Gamma'$*.*
2. *If* $\Gamma, x : A, \Gamma' \vdash t : T\ \rho$*, and* $\Gamma \vdash B : *$ *with* $A \equiv B$*, then* $\Gamma, x : B, \Gamma' \vdash t : T\ \rho$*.*

Proof. The proof is by mutual induction on the derivation of the environment and the expression.

Case 1: (G-Empty)
This case is impossible.

Case 2: G-Ext

Sub-Case 1: $\Gamma' = \bullet$
Our goal is to show $\vdash \Gamma, x : B$, which immediately follows by $\Gamma \vdash B : *$.

Sub-Case 2: $\Gamma' = \Gamma'', x' : A'$
Our goal is to show

$$\vdash \Gamma, x : B, \Gamma'', x' : A'$$

By the induction hypothesis, we have

$$\vdash \Gamma, x : B, \Gamma'' \text{ and } \Gamma, x : B, \Gamma'' \vdash A' : *$$

which imply the goal.

Case 3: (K-Star), (T-Unit), (T-Nat), (E-Unit), (E-Zero), (E-Nil)
These cases follow by the induction hypothesis on the typing environment.

Case 4: (E-Var)

Sub-Case 1: $t = x$
Our goal is to show

$$\Gamma, x : B, \Gamma' \vdash x : A$$

By the induction hypothesis, we have

$$\vdash \Gamma, x : B, \Gamma'$$

The fact that $\Gamma, x : A, \Gamma' \vdash x : A$ implies we have a subderivation of

$$\Gamma \vdash A : *$$

By weakening (Lemma 3.4.11), we obtain

$$\Gamma, x : B, \Gamma \vdash A : *$$

Now we can derive the goal using the equivalence between A and B via (E-Conv).

Sub-Case 2: $t = y$
This case follows by the induction hypothesis on the environment.

Case 5: All other cases
The goal follows by the induction hypothesis on the subderivations.

□

Lemma 3.4.13 (Regularity). *If $\Gamma \vdash e : A\ \rho$, then $\Gamma \vdash A : *$ and $\Gamma \vdash \rho$.*

Proof. The proof is by induction on the derivation of e. We show some representative cases:

Case 1: (E-Var)

Our goal is to show $\Gamma \vdash \mathsf{A} : *$. By the premise of the typing rule, we have $\vdash \Gamma$. The goal easily follows by environment inversion (Lemma 3.4.9) and weakening (Lemma 3.4.11).

Case 2: (E-Abs)

Our goal is to show $\Gamma \vdash \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho : *$. By the induction hypothesis on e, we have $\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : *$ and $\Gamma, \mathsf{x} : \mathsf{A} \vdash \rho$. By environment regularity (Lemma 3.4.8) and inversion (Lemma 3.4.9), we obtain $\Gamma \vdash \mathsf{A} : *$. The goal follows by (T-Pi).

Case 3: (E-DApp)

The goal immediately follows by the premises $\Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : *$ and $\Gamma \vdash \rho[\mathsf{e}_1/\mathsf{x}]$ of the typing rule.

Case 4: (E-NDApp)

The goal immediately follows by the premises of the typing rule.

Case 5: (E-Shift)

Our goal is to show $\Gamma \vdash \mathsf{A} : *$, $\Gamma \vdash \alpha : *$, and $\Gamma \vdash \beta : *$. The well-formedness of β follows by the premise of the typing rule, so it suffices to check the other two types. By applying environment inversion (Lemma 3.4.9) to the derivation of e, we obtain $\Gamma \vdash \mathsf{A} \to \alpha : *$. This type can only be derived by (T-Pi), which requires $\Gamma \vdash \mathsf{A} : *$ and $\Gamma \vdash \alpha : *$. Therefore the statement holds.

Case 6: (E-Reset)

Our goal is to show $\Gamma \vdash \mathsf{A} : *$. This immediately follows by the induction hypothesis on e.

Case 7: (E-Conv)

Our goal is to show $\Gamma \vdash \mathsf{B} : *$ and $\Gamma \vdash \rho'$, which are explicitly given as the premises of the typing rule.

□

3.4.3 Substitution

As in simply typed calculi, the preservation theorem requires a substitution lemma, which states that closing off a free variable by replacing it with a value of the correct type preserves typability. Specifically, we use the substitution lemma in the cases where the subject takes step via a β-like rule, which involves substitution. Below is the formal statement of the lemma we are going to prove:

Lemma 3.4.14 (Substitution)**.** *Suppose* $\Gamma \vdash \mathsf{v} : \mathsf{A}$*. Then, the following hold.*

1. *If* $\vdash \Gamma, \mathsf{x} : \mathsf{A}, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[\mathsf{v}/\mathsf{x}]$.

2. *If* $\Gamma, \mathsf{x} : \mathsf{A}, \Gamma' \vdash \mathsf{t} : \mathsf{T}\ \rho$*, then* $\Gamma, \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{t}[\mathsf{v}/\mathsf{x}] : \mathsf{T}[\mathsf{v}/\mathsf{x}]\ \rho[\mathsf{v}/\mathsf{x}]$.

The statement is different from its simply typed counterpart in that substitution happens not only at the level of terms, but at the level of types as well. Thus, we have $[\mathsf{v}/\mathsf{x}]$ in all the four components of a typing judgment. Note that we only need to consider the case where we substitute a value v, because no reduction rule of Dellina- performs substitution of a computation.

Proof. The proof is by mutual induction on the derivation of the environment and the expression.

Case 1: (G-Empty)
This case is impossible.

Case 2: (G-Ext)

Sub-Case 1: $\Gamma' = \bullet$
Our goal is to show $\vdash \Gamma$. This follows immediately by the premise of the formation rule.

Sub-Case 2: $\Gamma' = \Gamma'', \mathsf{x}' : \mathsf{A}'$
Our goal is to show $\vdash \Gamma, \Gamma''[\mathsf{v}/\mathsf{x}], \mathsf{x}' : \mathsf{A}'[\mathsf{v}/\mathsf{x}]$. By the induction hypothesis on $\Gamma, \mathsf{x} : \mathsf{A}, \Gamma''$, we have $\vdash \Gamma, \Gamma''[\mathsf{v}/\mathsf{x}]$. By the induction hypothesis on A', we also have $\Gamma, \Gamma''[\mathsf{v}/\mathsf{x}] \vdash \mathsf{A}'[\mathsf{v}/\mathsf{x}] : *$. These imply the goal.

Case 3: (T-Unit), (T-Nat), (E-Unit), (E-Zero), (E-Nil)

These cases can be shown by appealing to environment regularity (Lemma 3.4.8) and the induction hypothesis on the typing environment.

Case 4: (E-Var)

Sub-Case 1: $\mathsf{t} = \mathsf{x}$

Our goal is to show

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{v} : \mathsf{A}$$

which follows by the weakning lemma 3.4.11.

Sub-Case 2: $\mathsf{t} = \mathsf{y}$ where $\mathsf{y} \neq \mathsf{x}$

Our goal is to show

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{y} : \mathsf{B}[\mathsf{v}/\mathsf{x}]$$

By the induction hypothesis on $\Gamma,\ \mathsf{x} : \mathsf{A},\ \Gamma'$, we have

$$\vdash \Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}]$$

The goal follows by the fact that $\mathsf{B}[\mathsf{v}/\mathsf{x}] \in \Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}]$.

Case 5: (E-DApp)

Our goal is to show

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{e}_0\ \mathsf{e}_1[\mathsf{v}/\mathsf{x}] : (\mathsf{B}[\mathsf{e}_1/\mathsf{x}'])[\mathsf{v}/\mathsf{x}]\ \nu[\mathsf{v}/\mathsf{x}]$$

By the induction hypothesis, we have

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{e}_0[\mathsf{v}/\mathsf{x}] : (\Pi\,\mathsf{x}' : \mathsf{A}'.\,\mathsf{B}\ \tau)[\mathsf{v}/\mathsf{x}]\ \rho[\mathsf{v}/\mathsf{x}]\ ,\quad \Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{e}_1[\mathsf{v}/\mathsf{x}] : \mathsf{A}'[\mathsf{v}/\mathsf{x}]$$

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash (\mathsf{B}[\mathsf{e}_1/\mathsf{x}'])[\mathsf{v}/\mathsf{x}] : *\ ,\quad \text{and}\quad \Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \nu[\mathsf{v}/\mathsf{x}]$$

By (E-DApp), we obtain

$$\Gamma,\ \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{e}_0[\mathsf{v}/\mathsf{x}]\ \mathsf{e}_1[\mathsf{v}/\mathsf{x}] : (\mathsf{B}[\mathsf{v}/\mathsf{x}])[\mathsf{e}_1[\mathsf{v}/\mathsf{x}]/\mathsf{x}']\ \nu[\mathsf{v}/\mathsf{x}]$$

The goal follows by the definition of substitution.

Case 6: (E-Conv)
Our goal is to show

$$\Gamma, \Gamma'[v/x] \vdash e[v/x] : B[v/x]\ \rho'[v/x]$$

By the induction hypothesis, we have

$$\Gamma, \Gamma'[v/x] \vdash e[v/x] : A'[v/x]\ \rho[v/x]$$

To make this case go through, we have to show

$$A' \equiv B \Rightarrow A'[v/x] \equiv B[v/x]$$

and

$$\Gamma, x : A, \Gamma' \vdash B : * \Rightarrow \Gamma, \Gamma'[v/x] \vdash B[v/x] : *$$

By ($\equiv$), we know that there is a type C such that $A' \triangleright_p^\star C$ and $B \triangleright_p^\star C$. By Lemma 3.4.2, we also know that $A'[v/x] \triangleright_p^\star C[v/x]$, and similarly for B. The well-formedness of $B[v/x]$ also follows by the induction hypothesis on B. After proving similar statements for the effect annotation, we obtain the desired result.

□

Remark Since Dellina- allows dependency on non-value terms as long as they are pure, the reader might think we could prove a stronger substitution lemma, which looks like:

If $\Gamma \vdash e : A$ and $\Gamma, x : A, \Gamma' \vdash t : T\ \rho$, then $\Gamma, \Gamma'[e/x] \vdash t[e/x] : T[e/x]\ \rho[e/x]$.

If we had this lemma, then we would not need the well-formedness certificates of the result type in the typing rules for dependent constructs. Unfortunately, it is *not* possible to prove this lemma. Specifically, we would get stuck in the type conversion case, which requires us to show:

$$A' \equiv B \Rightarrow A'[e/x] \equiv B[e/x]$$

This statement does not hold, because $A' \triangleright_p C$ does not imply $A'[e/x] \triangleright_p C[e/x]$ for a non-value e under our call-by-value strategy. To give an example, we have

$$L\ ((\lambda y : \mathbb{N}.\, y)\ x) \triangleright_p L\ x$$

but not

$$\mathsf{L}\ ((\lambda\,\mathsf{y} : \mathbb{N}.\,\mathsf{y})\ (\mathsf{loop}\ ()))\ \rhd_p\ \mathsf{L}\ (\mathsf{loop}\ ())$$

if loop is a non-terminating function. Since we require recursive functions to be guarded, we should not be able to build a looping term, but as we have not yet established normalization of pure terms, we must take this possibility into account.

3.4.4 Inversion

The last thing we need for proving preservation is a set of inversion lemmas. These lemmas literally allow us to invert a typing relation and extract some useful facts about subterms. In the preservation proof, we use inversion when *e.g.* checking well-typedness of a β-reduct $(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e})\ \mathsf{v}$, where we need $\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}$ to show $\mathsf{e}[\mathsf{v}/\mathsf{x}] : \mathsf{B}[\mathsf{v}/\mathsf{x}]$. This cannot be obtained directly when the last rule used for the derivation of the abstraction was (Conv), but we still know that e satisfies some "equivalent" properties—this is what inversion lemmas tell us.

Lemma 3.4.15 (Inversion for Function Types)**.** *If* $\Gamma \vdash \Pi\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho : *$, *then* $\Gamma \vdash \mathsf{A} : *$, $\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : *$, *and* $\Gamma, \mathsf{x} : \mathsf{A} \vdash \rho$.

Proof. This is obvious from (T-Pi). □

Lemma 3.4.16 (Inversion for λ)**.** *If* $\Gamma \vdash \lambda\mathsf{x} : \mathsf{A}.\,\mathsf{e} : \mathsf{C}$, *then* $\mathsf{C} \equiv \Pi\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho$ *for some* B *and* ρ, *such that* $\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}$.

Proof. The proof is by induction on the derivation of $\lambda\mathsf{x} : \mathsf{A}.\,\mathsf{e}$.

Case 1: (E-Abs)
This case is trivial.

Case 2: (Conv)
Suppose we have the following derivation:

$$\frac{\Gamma \vdash \lambda\mathsf{x} : \mathsf{A}.\,\mathsf{e} : \mathsf{C}_0 \qquad \Gamma \vdash \mathsf{C}_1 : * \qquad \mathsf{C}_0 \equiv \mathsf{C}_1}{\Gamma \vdash \lambda\mathsf{x} : \mathsf{A}.\,\mathsf{e} : \mathsf{C}_1}\ (\textsc{Conv})$$

By the induction hypothesis, we have

$$\mathsf{C}_0 \equiv \Pi\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho \quad \text{and} \quad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho$$

The goal follows by transitivity of equivalence (Lemma 3.4.5).

□

Lemma 3.4.17 (Inversion for Recursive Functions). *If* $\Gamma \vdash \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho} \mathsf{x}.\, \mathsf{e} : \mathsf{C}$, *then* $\mathsf{C} \equiv \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$, *and* $\Gamma, \mathsf{f} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho$.

Proof. The proof is analogous to the previous lemma. □

Lemma 3.4.18 (Inversion for Shift). *If* $\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A}_0 \to \alpha_0.\, \mathsf{e} : \mathsf{C}[\alpha', \beta']$, *then the following hold.*

- $\mathsf{A}_0 \to \alpha_0 \equiv \mathsf{A} \to \alpha$ *for some* A *and* α
- $\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \beta$ *or* $\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta]$ *for some* B *and* β
- $\Gamma \vdash \beta : *$
- $\mathsf{C} \equiv \mathsf{A}$, $\alpha' \equiv \alpha$, *and* $\beta' \equiv \beta$.

Proof. The proof is by induction on the derivation of $\mathcal{S}\mathsf{k} : \mathsf{A}_0 \to \alpha_0.\, \mathsf{e}$.

Case 1: E-Shift
This case is trivial.

Case 2: Conv
Suppose we have the following derivation:

$$\frac{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A}_0 \to \alpha_0.\, \mathsf{e} : \mathsf{C}_0[\alpha_1, \beta_1] \qquad \mathsf{C}_0 \equiv \mathsf{C}_1 \qquad [\alpha_1, \beta_1] \equiv [\alpha_2, \beta_2]}{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A}_0 \to \alpha_0.\, \mathsf{e} : \mathsf{C}_1[\alpha_2, \beta_2]} \ (\textsc{Conv})$$

By the induction hypothesis, we have

$$\mathsf{A}_0 \to \alpha_0 \equiv \mathsf{A} \to \alpha,$$

$$\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \beta \quad \text{or} \quad \Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta],$$

$$\mathsf{C}_0 \equiv \mathsf{A}, \alpha_1 \equiv \alpha, \quad \text{and} \quad \beta_1 \equiv \beta.$$

The goal follows by transitivity of equivalence (Lemma 3.4.5).

□

3.4.5 Preservation

Using the lemmas we have proved so far, we show preservation: evaluation of a well-typed, non-value term preserves its type and purity. Instead of directly proving this theorem, we first show preservation under parallel reduction, and then derive preservation under call-by-value evaluation as a corollary.

Theorem 3.4.2 (Preservation under Parallel Reduction)**.** *If* $\Gamma \vdash \mathsf{t} : \mathsf{T}\ \rho$ *and* $\mathsf{t} \rhd_p \mathsf{t}'$*, then* $\Gamma \vdash \mathsf{t}' : \mathsf{T}\ \rho$*.*

Proof. The proof is by induction on the derivation of t.

Case 1: (K-Star), (T-Unit), (T-Nat), (E-Unit), (E-Zero), (E-Nil)
These cases are impossible because the subject cannot take step.

Case 2: (E-Abs)
An abstraction $\lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e}$ may reduce to $\lambda \mathsf{x} : \mathsf{A}'.\, \mathsf{e}'$ via (P-Abs). Our goal is to show

$$\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}'.\, \mathsf{e}' : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$$

By the induction hypothesis, we have

$$\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e}' : \mathsf{B}\ \rho$$

By Lemma 3.4.10, we know that there is a subderivation of

$$\Gamma \vdash \mathsf{A} : *$$

The induction hypothesis on this derivation gives us

$$\Gamma \vdash \mathsf{A}' : *$$

and since $\mathsf{A} \equiv \mathsf{A}'$, we obtain

$$\Gamma, \mathsf{x} : \mathsf{A}' \vdash \mathsf{e}' : \mathsf{B}\ \rho$$

by Lemma 3.4.12. Now, by (E-Abs), we can derive

$$\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}'.\, \mathsf{e}' : \Pi \mathsf{x} : \mathsf{A}'.\, \mathsf{B}\ \rho$$

What remains to be done is to cast the type to the original one, which requires us to show the well-formedness of $\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$. As we saw, the domain A is well-formed.

By regularity (Lemma 3.4.13) on e, we also know that B and ρ are well-formed. Thus we obtain the goal.

Case 3: (E-DApp)

Sub-Case 1: $\mathsf{e}_0\ \mathsf{e}_1 \ \triangleright_p\ \mathsf{e}_0'\ \mathsf{e}_1'$ by (P-App)
Our goal is to show:

$$\Gamma \vdash \mathsf{e}_0'\ \mathsf{e}_1' : \mathsf{B}[\mathsf{e}_1/\mathsf{x}]\ \tau$$

By the induction hypothesis on e_0 and e_1, we have

$$\Gamma \vdash \mathsf{e}_0' : (\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \sigma)\ \rho \quad \text{and} \quad \Gamma \vdash \mathsf{e}_1' : \mathsf{A}$$

Since $\mathsf{e}_1 \ \triangleright_p\ \mathsf{e}_1'$, Lemma 3.4.1 gives us $\mathsf{B}[\mathsf{e}_1/\mathsf{x}] \ \triangleright_p\ \mathsf{B}[\mathsf{e}_1'/\mathsf{x}]$ and $\sigma[\mathsf{e}_1/\mathsf{x}] \ \triangleright_p\ \sigma[\mathsf{e}_1'/\mathsf{x}]$. This allows us to use the induction hypothesis on $\mathsf{B}[\mathsf{e}_1/\mathsf{x}]$ and $\sigma[\mathsf{e}_1/\mathsf{x}]$, that is, $\Gamma \vdash \mathsf{B}[\mathsf{e}_1'/\mathsf{x}] : *$ and $\Gamma \vdash \sigma[\mathsf{e}_1'/\mathsf{x}]$. Now by (E-DApp), we can derive $\Gamma \vdash \mathsf{e}_0'\ \mathsf{e}_1' : \mathsf{B}[\mathsf{e}_1'/\mathsf{x}]\ \tau'$, where $\tau' = \text{comp}(\rho, \sigma[\mathsf{e}_1'/\mathsf{x}])$. As we have $\mathsf{B}[\mathsf{e}_1/\mathsf{x}] \equiv \mathsf{B}[\mathsf{e}_1'/\mathsf{x}]$ and $\sigma[\mathsf{e}_1/\mathsf{x}] \equiv \sigma[\mathsf{e}_1'/\mathsf{x}]$, plus well-formedness of the left-hand sides of these equivalences, we obtain $\Gamma \vdash \mathsf{e}_0'\ \mathsf{e}_1' : \mathsf{B}[\mathsf{e}_1/\mathsf{x}]\ \tau$ by (E-Conv), as desired.

Sub-Case 2: $(\lambda\, \mathsf{x} : \mathsf{A}_1.\, \mathsf{e}_0)\ \mathsf{v}_1 \ \triangleright_p\ \mathsf{e}_0'[\mathsf{v}_1'/\mathsf{x}]$ by (P-AppBeta)
Our goal is to show

$$\Gamma \vdash \mathsf{e}_0'[\mathsf{v}_1'/\mathsf{x}] : \mathsf{B}[\mathsf{v}_1/\mathsf{x}]\ \sigma[\mathsf{v}_1/\mathsf{x}]$$

Note that both the function and argument must be a pure term, hence the overall effect is $\sigma[\mathsf{v}_1/\mathsf{x}]$, which comes from the function's body. By the induction hypothesis on $\lambda\, \mathsf{x} : \mathsf{A}_1.\, \mathsf{e}_0$, v_1, and $\mathsf{B}[\mathsf{v}_1/\mathsf{x}]$, we have

$$\Gamma \vdash \lambda\, \mathsf{x} : \mathsf{A}_1'.\, \mathsf{e}_0' : \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \sigma\ , \quad \Gamma \vdash \mathsf{v}_1' : \mathsf{A}\ ,$$

$$\Gamma \vdash \mathsf{B}[\mathsf{v}_1'/\mathsf{x}] : *\ , \quad \text{and} \quad \Gamma \vdash \sigma[\mathsf{v}_1'/\mathsf{x}]$$

We also have the following facts:

1. $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \sigma \equiv \Pi\, \mathsf{x} : \mathsf{A}_1'.\, \mathsf{B}_1\ \sigma_1$ (by inversion for λ (Lemma 3.4.16))

2. $\Gamma, \mathsf{x} : \mathsf{A}_1' \vdash \mathsf{e}_0' : \mathsf{B}_1\ \sigma_1$ (by inversion for λ)

3. $\mathsf{A} \equiv \mathsf{A}'_1$ and $\mathsf{B} \equiv \mathsf{B}_1$ (by item 1 and injectivity of Π (Lemma 3.4.6))

4. There is a subderivation of $\Gamma \vdash \mathsf{A}'_1 : *$ (by item 2 and Lemma 3.4.10)

Using items 3, 4 and (E-Conv), we can derive $\Gamma \vdash \mathsf{v}'_1 : \mathsf{A}'_1$. Then, using item 2 and the substitution lemma (Lemma 5.4.4), we obtain

$$\Gamma \vdash \mathsf{e}'_0[\mathsf{v}'_1/\mathsf{x}] : \mathsf{B}_1[\mathsf{v}'_1/\mathsf{x}]\ \sigma_1[\mathsf{v}'_1/\mathsf{x}]$$

Item 2 and Lemma 3.4.7 further give us $\mathsf{B}[\mathsf{v}'_1/\mathsf{x}] \equiv \mathsf{B}_1[\mathsf{v}'_1/\mathsf{x}]$ and $\sigma[\mathsf{v}'_1/\mathsf{x}] \equiv \sigma_1[\mathsf{v}'_1/\mathsf{x}]$. These imply

$$\Gamma \vdash \mathsf{e}'_0[\mathsf{v}'_1/\mathsf{x}] : \mathsf{B}[\mathsf{v}'_1/\mathsf{x}]\ \sigma[\mathsf{v}'_1/\mathsf{x}]$$

Now, Lemma 3.4.2 tells us that $\mathsf{B}[\mathsf{v}_1/\mathsf{x}] \ \rhd_p\ \mathsf{B}[\mathsf{v}'_1/\mathsf{x}]$ and $\sigma[\mathsf{v}_1/\mathsf{x}] \ \rhd_p\ \sigma[\mathsf{v}'_1/\mathsf{x}]$. Using these facts, and the well-formedness premises of the result type and effect annotation, and (E-Conv), we can derive $\Gamma \vdash \mathsf{e}'_0[\mathsf{v}'_1/\mathsf{x}] : \mathsf{B}[\mathsf{v}_1/\mathsf{x}]\ \sigma[\mathsf{v}_1/\mathsf{x}]$ as desired.

Case 4: (E-Shift)
Our goal is to show

$$\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A}' \to \alpha'.\, \mathsf{e}' : \mathsf{A}[\alpha, \beta]$$

By the induction hypothesis on e, we have

$$\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e}' : \mathsf{B}[\mathsf{B}, \beta]$$

By Lemma 3.4.10, we know that there is a subderivation of

$$\Gamma \vdash \mathsf{A} \to \alpha : *$$

The induction hypothesis on this derivation gives us

$$\Gamma \vdash \mathsf{A}' \to \alpha' : *$$

and since $\mathsf{A} \to \alpha \equiv \mathsf{A}' \to \alpha'$, we obtain

$$\Gamma, \mathsf{k} : \mathsf{A}' \to \alpha' \vdash \mathsf{e}' : \mathsf{B}[\mathsf{B}, \beta]$$

by Lemma 3.4.12. The goal now follows by (E-Shift).

Case 5: (E-Reset)

$$\frac{\alpha \rhd_p \alpha' \quad \mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1'}{(\mathcal{S}\mathsf{k} : \mathsf{A}' \to \alpha.\,\mathsf{e})\ \mathsf{e}_1 \rhd_p \mathcal{S}\mathsf{k}' : \mathsf{B}[\mathsf{e}_1/\mathsf{x}] \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho.\,\langle \mathsf{k}'\ (\mathsf{v}\ \mathsf{e}_1)\rangle/\mathsf{k}]}\ \text{(P-SApp1)}$$

$$\frac{\alpha \rhd_p \alpha' \quad \mathsf{v}_0 \rhd_p \mathsf{v}_0' \quad \mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{v}_0\ (\mathcal{S}\mathsf{k} : \mathsf{A}' \to \alpha.\,\mathsf{e}) \rhd_p \mathcal{S}\mathsf{k}' : \mathsf{B} \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathsf{A}.\,\langle \mathsf{k}'\ (\mathsf{v}_0\ \mathsf{v})\rangle/\mathsf{k}]}\ \text{(P-SApp2)}$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \alpha \rhd_p \alpha'}{\mathsf{suc}\ (\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e}) \rhd_p \mathcal{S}\mathsf{k}' : \mathbb{N} \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathbb{N}.\,\langle \mathsf{k}'\ (\mathsf{suc}\ \mathsf{v})\rangle/\mathsf{k}]}\ \text{(P-SSuc)}$$

$$\frac{\alpha \rhd_p \alpha' \mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{::\ \mathsf{e}_0\ (\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e})\ \mathsf{e}_2 \rhd_p \mathcal{S}\mathsf{k}' : \mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0') \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathsf{L}\ \mathsf{e}_0'.\,\langle \mathsf{k}'\ (::\ \mathsf{e}_0'\ \mathsf{v}\ \mathsf{e}_2')\rangle/\mathsf{k}]}\ \text{(P-SCons1)}$$

$$\frac{\alpha \rhd_p \alpha' \quad \mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e} \rhd_p \mathsf{e}'}{::\ \mathsf{e}_0\ \mathsf{e}_1\ (\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e}) \rhd_p \mathcal{S}\mathsf{k}' : \mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0') \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathsf{L}\ \mathsf{e}_0'.\,\langle \mathsf{k}'\ (::\ \mathsf{e}_0'\ \mathsf{e}_1'\ \mathsf{v})\rangle/\mathsf{k}]}\ \text{(P-SCons2)}$$

$$\frac{\alpha \rhd_p \alpha' \quad \mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm}\ (\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e})\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 \rhd_p \\ \mathcal{S}\mathsf{k}' : \mathsf{P} \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathbb{N}.\,\langle \mathsf{k}'\ (\mathsf{pm}\ \mathsf{v}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2)\rangle/\mathsf{k}]\end{array}}\ \text{(P-SMatchN)}$$

$$\frac{\alpha \rhd_p \alpha' \quad \mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm}\ (\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e})\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 \rhd_p \\ \mathcal{S}\mathsf{k}' : \mathsf{P} \to \alpha'.\,\mathsf{e}[\lambda\,\mathsf{v} : \mathsf{L}\ \mathsf{n}.\,\langle \mathsf{k}'\ (\mathsf{pm}\ \mathsf{v}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2)\rangle/\mathsf{k}]\end{array}}\ \text{(P-SMatchL)}$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\langle \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\,\mathsf{e}\rangle \rhd_p \langle \mathsf{e}'[\lambda\,\mathsf{v} : \mathsf{A}.\,\mathsf{v}/\mathsf{k}]\rangle}\ \text{(P-SEmpty)}$$

Figure 3.13: Small Reductions

Sub-Case 1: (P-Reset)
Our goal is to show

$$\Gamma \vdash \langle e' \rangle : A$$

This can be easily obtained by the induction hypothesis on e.

Sub-Case 2: (P-ResetS)
Following Asai and Kameyama [10], we prove this case by decomposing the reduction into small reductions, which capture one context frame at a time. These reductions are defined in Figure 3.13; observe that when there is no more context between `shift` and `reset`, we substitute the identity function for the continuation variable.

We show one instance of the first reduction, and one instance of the last reduction.

Sub-Sub-Case 1: (P-SApp1)
Suppose we have

$$\frac{\Gamma \vdash \mathcal{S}k : A_0 \to \alpha_0.\, e : \Pi x : A.\, B[\alpha, \beta] \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash (\mathcal{S}k : A_0 \to \alpha_0.\, e)\ e_1 : B[e_1/x][\alpha, \beta]}\ \text{(E-DApp)}$$

where e is a pure term. Our goal is to show

$$\Gamma \vdash \mathcal{S}k' : B[e_1'/x] \to \alpha_0'.\, e'[\lambda\, v_0 : \Pi x : A.\, B.\, \langle k'\ (v_0\ e_1')\rangle/k] : B[e_1/x][\alpha, \beta]$$

By the induction hypothesis, we have

$$\Gamma \vdash \mathcal{S}k : A_0' \to \alpha_0'.\, e' : \Pi x : A.\, B[\alpha, \beta] \ \text{ and } \ \Gamma \vdash e_1' : A$$

Using inversion for `shift` (Lemma 3.4.18), we obtain the following derivations:

$$\Gamma, k : A_1 \to \alpha_1 \vdash e' : \beta_1 \ \text{ and } \ \Gamma \vdash \beta_1 : *$$

and the following equivalence relations:

$$\Pi x : A.\, B \equiv A_1\ ,\quad A_0' \to \alpha_0' \equiv A_1 \to \alpha_1\ ,\quad \alpha \equiv \alpha_1\ ,\quad \text{and} \quad \beta \equiv \beta_1$$

For our purpose, we must show

$$\Gamma, k' : B[e_1'/x] \to \alpha_0' \vdash e'[\lambda\, v_0 : \Pi x : A.\, B.\, \langle k'\ (v_0\ e_1')\rangle/k] : \beta$$

By (E-DApp), we have $\mathsf{v_0\ e'_1} : \mathsf{B[e'_1/x]}$. By (E-NDApp), we obtain $\mathsf{k'\ (v_0\ e'_1)} : \alpha'_0$. Since we have $\mathsf{A'_0} \rightarrow \alpha'_0 \equiv \mathsf{A_1} \rightarrow \alpha_1$ and $\alpha \equiv \alpha_1$, we know that $\alpha \equiv \alpha'_0$ (by transitivity and injectivity), allowing us to derive $\mathsf{k'\ (v_0\ e'_1)} : \alpha$ via (E-Conv) (note that well-formedness of α follows by regularity). Now by (E-Reset), we obtain $\langle \mathsf{k'\ (v_0\ e'_1)} \rangle : \alpha$. Using (E-Abs), we further obtain $\lambda\, \mathsf{v_0} : \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}.\, \langle \mathsf{k'\ (v_0\ e'_1)} \rangle : \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightarrow \alpha$. To safely substitute this function for k, we have to cast its type to $\mathsf{A_1} \rightarrow \alpha_1$. As we saw, inversion gives us $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \equiv \mathsf{A_1}$ and $\alpha \equiv \alpha_1$, which imply $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightarrow \alpha \equiv \mathsf{A_1} \rightarrow \alpha_1$. By Lemma 3.4.10, we also know $\Gamma \vdash \mathsf{A_1} \rightarrow \alpha_1 : *$. These allow us to cast the type of the function we are substituting for k, obtaining

$$\Gamma,\ \mathsf{k'} : \mathsf{B[e'_1/x]} \rightarrow \alpha'_0 \vdash \mathsf{e'}[\lambda\, \mathsf{v_0} : (\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}).\, \langle \mathsf{k'\ (v_0\ e'_1)} \rangle / \mathsf{k}] : \beta_1$$

via the substitution lemma (Lemma 5.4.4). Since $\beta \equiv \beta_1$ and $\Gamma \vdash \beta : *$, (E-Conv) gives us

$$\Gamma,\ \mathsf{k'} : \mathsf{B[e'_1/x]} \rightarrow \alpha'_0 \vdash \mathsf{e'}[\lambda\, \mathsf{v_0} : (\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}).\, \langle \mathsf{k'\ (v_0\ e'_1)} \rangle / \mathsf{k}] : \beta$$

and (E-Shift) lets us conclude

$$\Gamma \vdash \mathcal{S}\mathsf{k'} : \mathsf{B[e'_1/x]} \rightarrow \alpha'_0.\, \mathsf{e'}[\lambda\, \mathsf{v_0} : (\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}).\, \langle \mathsf{k'\ (v_0\ e'_1)} \rangle / \mathsf{k}] : \mathsf{B[e_1/x]}[\alpha'_0, \beta]$$

Now the goal follows by (E-Conv), using $\alpha \equiv \alpha'_0$ and $\Gamma \vdash \alpha : *$.

Sub-Sub-Case 2: (P-SEmpty)
Suppose we have

$$\frac{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A_0} \rightarrow \alpha_0.\, \mathsf{e} : \mathsf{A}[\alpha, \beta]}{\Gamma \vdash \langle \mathcal{S}\mathsf{k} : \mathsf{A_0} \rightarrow \alpha_0.\, \mathsf{e} \rangle : \beta}\ \text{(E-Reset)}$$

where e is an impure term. Our goal is to show

$$\Gamma \vdash \langle \mathsf{e'}[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}] \rangle : \beta$$

By the induction hypothesis, we have

$$\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A'_0} \rightarrow \alpha'_0.\, \mathsf{e'} : \mathsf{A}[\alpha, \beta]$$

Using inversion for `shift` (Lemma 3.4.18), we obtain the following derivations:

$$\Gamma,\ \mathsf{k} : \mathsf{A_1} \rightarrow \alpha_1 \vdash \mathsf{e'} : \mathsf{B}[\mathsf{B}, \beta_1] \quad \text{and} \quad \Gamma \vdash \beta_1 : *$$

and the following equivalence relations:

$$\mathsf{A} \equiv \mathsf{A}_1\ ,\quad \mathsf{A}'_0 \to \alpha'_0 \equiv \mathsf{A}_1 \to \alpha_1\ ,\quad \alpha \equiv \alpha_1\ ,\quad \text{and}\quad \beta \equiv \beta_1$$

For our purpose, we must show

$$\Gamma \vdash \mathsf{e}'[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}] : \mathsf{B}[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}][\mathsf{B}[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}], \beta]$$

By (E-Abs), we have $\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v} : \mathsf{A} \to \mathsf{A}$. To safely substitute this function for k, we must cast its type to $\mathsf{A}_1 \to \alpha_1$. Since we know $\mathsf{A} \equiv \mathsf{A}_1$ by inversion, it suffices to show $\mathsf{A} \equiv \alpha_1$. Now, recall that there is no further context between `shift` and `reset`. This means $\mathsf{A} \equiv \alpha$ holds. Since we have $\alpha \equiv \alpha_1$ by inversion, we can conclude $\mathsf{A} \equiv \alpha_1$ by transitivity. Using well-formedness of $\mathsf{A}_1 \to \alpha_1$ obtained by inversion and Lemma 3.4.10, we can derive $\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v} : \mathsf{A}_1 \to \alpha_1$, and then

$$\Gamma \vdash \mathsf{e}'[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}] : \mathsf{B}[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}][\mathsf{B}[\lambda\, \mathsf{v} : \mathsf{A}.\, \mathsf{v}/\mathsf{k}], \beta_1]$$

The goal follows by $\beta \equiv \beta_1$ and well-formedness of β.

Sub-Case 3: (P-ResetV)
Our goal is to show

$$\Gamma \vdash \mathsf{v}' : \mathsf{A}$$

This can be obtained easily by the induction hypothesis on v.

□

Corollary 3.4.2 (Preservation of Runtime Evaluation)**.** *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho$ *and* $\mathsf{e} \triangleright \mathsf{e}'$, *then* $\Gamma \vdash \mathsf{e}' : \mathsf{A}\ \rho$.

Proof. This is a direct consequence of Theorem 3.4.2; remember that runtime evaluation is a subrelation of parallel reduction. □

3.4.6 Progress

We now prove the progress theorem: a well-typed, closed, pure term does not get stuck. This theorem requires a *canonical forms lemma*, which states that we can determine the shape of closed values by looking at their type. For instance, if we have a closed value of type $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$, then we know that it must be an abstraction or a recursive function. While the lemma may appear trivial, it requires special care due to the presence of the conversion rule. Suppose we have casted the type of a value v from A to B using the equivalence $\mathsf{A} \equiv \mathsf{B}$. If A and B were different kinds of types—*e.g.*, $\mathsf{A} = \mathbb{N}$ and $\mathsf{B} = \Pi\, \mathsf{x} : \mathbb{N}.\, \mathbb{N}$—then what holds of $\Gamma \vdash \mathsf{v} : \mathsf{A}$ would no longer be true for $\Gamma \vdash \mathsf{v} : \mathsf{B}$. Therefore, before proving progress, we show soundness of our equivalence, using the notion of *head constructors* defined below.

Definition 3.4.1 (Head Constructors)**.** *The head constructor of a type* A*, written* $\mathsf{head}(\mathsf{A})$*, is defined as follows:*

$$\begin{aligned}
\mathsf{head}(\mathsf{Unit}) &= \mathsf{Unit} \\
\mathsf{head}(\mathbb{N}) &= \mathbb{N} \\
\mathsf{head}(\mathsf{L}\ \mathsf{e}) &= \mathsf{L} \\
\mathsf{head}(\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho) &= \Pi
\end{aligned}$$

Lemma 3.4.19 (Soundness of Equivalence)**.** *If* $\mathsf{A} \equiv \mathsf{B}$*, then* A *and* B *have the same head constructor.*

Proof. This can be easily shown by observing that reduction preserves head constructors. □

Lemma 3.4.20 (Canonical Forms)**.** *If* $\bullet \vdash \mathsf{v} : \mathsf{A}$*, then the following hold:*

1. *If* $\mathsf{A} \equiv \Pi\, \mathsf{x} : \mathsf{A}_1.\, \mathsf{B}\ \rho$*, then* v *is either* $\lambda\, \mathsf{x} : \mathsf{A}'.\, \mathsf{e}$ *or* $\mathsf{rec}\ \mathsf{f}_{\Pi\, \mathsf{x} : \mathsf{A}'.\, \mathsf{B}'\ \rho'}\ \mathsf{x}.\, \mathsf{e}$.
2. *If* $\mathsf{A} \equiv \mathsf{Unit}$*, then* v *is* $()$.
3. *If* $\mathsf{A} \equiv \mathbb{N}$*, then* v *is either* z *or* $\mathsf{suc}\ \mathsf{v}'$.
4. *If* $\mathsf{A} \equiv \mathsf{L}\ \mathsf{e}$*, then* v *is either* nil *or* $::\ \mathsf{v}_0\ \mathsf{v}_1\ \mathsf{v}_2$.

Proof. The proof is by induction on the derivation of v.

Case 1: (E-Var)
This case is impossible because well-typedness of variables requires a non-empty context.

Case 2: (E-DApp), (E-DLet), (E-DMatchN), (E-DMatchL), (E-Reset)
These cases are also impossible because their conclusion has a non-value subject.

Case 3: (E-Abs), (E-Rec)

These rules conclude with a function type $\Pi x : A.\, B\ \rho$. The subject is a λ or a recursive function, as required by item 1.

Case 4: (E-Unit)

This rule concludes with the unit type Unit. The subject is a unit value, as required by item 2.

Case 5: (E-Zero), (E-Suc)

These rules conclude with the natural number type $\mathbb{N}$. The subject is either z or $\mathsf{suc}\ \mathsf{v}$, as required by item 3.

Case 6: (E-Nil), (E-Cons)

These rules conclude with a list type $\mathsf{L}\ \mathsf{e}$. The subject is either nil of $::\ \mathsf{v}_0\ \mathsf{v}_1\ \mathsf{v}_2$, as required by item 4.

Case 7: (E-Conv)

We have

$$\frac{\Gamma \vdash \mathsf{v} : A \qquad \Gamma \vdash B : * \qquad A \equiv B}{\Gamma \vdash \mathsf{v} : B}\ \text{(E-Conv)}$$

The induction hypothesis tells us that v has the specific form required by A. To say the same thing for type B, we need the fact that A and B have the same shape. This follows by the soundness of equivalence (Lemma 3.4.19).

□

Theorem 3.4.3 (Progress)**.**

1. *If $\bullet \vdash \mathsf{e} : A$, then either e is a value, or there is an e' such that $\mathsf{e} \rhd \mathsf{e}'$.*

2. *If $\bullet \vdash \mathsf{e} : A[\alpha, \beta]$, then either e is a stuck term of the form $F[\mathcal{S}\mathsf{k} : A' \to \alpha'.\, \mathsf{e}']$, or there is an e' such that $\mathsf{e} \rhd \mathsf{e}'$.*

Proof. The proof is by induction on the derivation of e. The progress property in the usual sense holds only for pure terms, because impure terms are not executable in general (*e.g.*, a `shift` clause is a stuck term).

Case 1: (E-Var)

This case is impossible since no variable can be well-typed in an empty context.

Case 2: (E-Abs), (E-Rec), (E-Unit), (E-Zero), (E-Nil)

These cases conclude with a value subject, which trivially satisfies the statement.

Case 3: (E-App)

Suppose both e_0 and e_1 are pure. By the induction hypothesis, we know e_0 is either a value or there is an e_0' such that $e_0 \triangleright e_0'$, and similarly for e_1. If e_0 is a value, we know from Lemma 3.4.20 that it has the form $\lambda x : A'. e_0'$. If e_1 is also a value, the application is a β-redex, hence $(\lambda x : A'. e_0')\ e_1 \triangleright e_0'[e_1/x]$. If e_1 is a non-value, $e_0\ e_1 \triangleright e_0\ e_1'$. If e_0 is a non-value, $e_0\ e_1 \triangleright e_0'\ e_1$.

Next, suppose e_1 is impure. By the induction hypothesis, we know e_1 is either a stuck term of the form $F[\mathcal{S}k : A' \to \alpha'. e]$, or there is a e_1' such that $e_1 \triangleright e_1'$. If e_0 is a value, the the whole application is either a stuck term $e_0\ F[\mathcal{S}k : A' \to \alpha'. e]$ or it reduces to $e_0\ e_1'$. If e_0 is a non-value, $e_0\ e_1 \triangleright e_0'\ e_1$.

Lastly, suppose e_0 is impure. By the induction hypothesis, we know e_0 is either a stuck term of the form $F[\mathcal{S}k : A' \to \alpha'. e]$, or there is a e_0' such that $e_0 \triangleright e_0'$. In the former case, the whole application is a stuck term $F[\mathcal{S}k : A' \to \alpha'. e]\ e_1$. In the latter case, $e_0\ e_1 \triangleright e_0'\ e_1$.

Case 4: (E-Shift)

This case is trivial, because a `shift` construct is itself a stuck term.

Case 5: (E-Reset)

Suppose e is pure. By the induction hypothesis, we know either e is a value, or there exists some e' such that $e \triangleright e'$. If e is a value, $\langle e \rangle \triangleright e$. Otherwise, $\langle e \rangle \triangleright \langle e' \rangle$.

Next, suppose e is impure. By the induction hypothesis, we know e is either a stack term $F[\mathcal{S}k : A' \to \alpha'. e']$, or there exists some e' such that $e \triangleright e'$. If e is a stuck term, $\langle F[\mathcal{S}k : A' \to \alpha'. e'] \rangle \triangleright \langle e'[\lambda x : A. \langle F[x] \rangle / k] \rangle$. Otherwise, $\langle e \rangle \triangleright \langle e' \rangle$.

□

Remark The preservation and progress theorems imply something more than "well-typed programs do not *go wrong*"; they tell us that well-typed, closed programs evaluate to a value of the same type, if they terminate. This property is known as *strong type soundness* in the literature [170].

3.5 Examples

In the previous sections, we studied the design and properties of Dellina-. Now it's time to see what kind of programs we can build using control operators and dependent types. In this section, we show two example programs of Dellina-: one with non-deterministic behaviors, and the other with mutable state.

3.5.1 Non-deterministic Choice

Let us first look at how to program with dependent types and non-deterministic choice. Our goal is to implement the following behavior:

$$\mathsf{run\text{-}choose}\ (\lambda\,()\,:\,\mathsf{Unit}.\,10 + (\mathsf{choose}\ 1\ 2)) \triangleright^{\star} [11;\ 12]$$

The intention is that, when we call the choose function with arguments a and b in the context E, we want back a two-element list $[\mathsf{E}[\mathsf{a}]; \mathsf{E}[\mathsf{b}]]$. It would be easy for the reader to identify the role of choose and $\mathsf{run\text{-}choose}$: they are basically `shift` and `reset` operators!

$$\begin{aligned}
\mathsf{choose} &\overset{\text{def}}{\equiv} \lambda\,\mathsf{a} : \mathbb{N}.\,\lambda\,\mathsf{b} : \mathbb{N}.\,\mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{N}.\,[\mathsf{k}\ \mathsf{a}; \mathsf{k}\ \mathsf{b}] \\
\mathsf{run\text{-}choose} &\overset{\text{def}}{\equiv} \lambda\,\mathsf{f} : \mathsf{Unit} \to \mathbb{N}[\mathbb{N}, \mathsf{L}\ 2].\,\langle \mathsf{f}\ ()\rangle
\end{aligned}$$

The choose function brings non-determinism by calling the captured continuation twice with different arguments. Since our list type is dependent, the function additionally guarantees that the resulting list has exactly two elements. This invariant is trivial, but it is something we cannot explicitly state in a simply typed language.

3.5.2 Mutable State

Our next example is yet another list-building function. The goal is to make the following program work:

$$\mathsf{run\text{-}state}\ (\lambda\,()\,:\,\mathsf{Unit}.\,\mathsf{mk\text{-}lst}\ 3) \triangleright^{\star} [1;\ 2;\ 3]$$

To make the program interesting, we generate each element using a mutable state, which we simulate using `shift` and `reset`[8] [11]:

$$\begin{aligned}
\mathsf{inc} &\overset{\text{def}}{\equiv} \mathcal{S}\mathsf{k} : \mathsf{Unit} \to \mathbb{N} \to \mathsf{L}\ 3.\ \lambda\,\mathsf{s} : \mathbb{N}.\ \mathsf{k}\ ()\ (\mathsf{suc}\ \mathsf{s}) \\
\mathsf{get} &\overset{\text{def}}{\equiv} \mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{N} \to \mathsf{L}\ 3.\ \lambda\,\mathsf{s} : \mathbb{N}.\ \mathsf{k}\ \mathsf{s}\ \mathsf{s} \\
\mathsf{e}_1;\ \mathsf{e}_2 &\overset{\text{def}}{\equiv} (\lambda\,() : \mathsf{Unit}.\ \mathsf{e}_2)\ \mathsf{e}_1
\end{aligned}$$

The inc and get are structurally similar: they both capture the current continuation k, receive the value of the state s, and apply k to two arguments. Among the two arguments, first one is what to be returned to the current context, and the second one is the state used by the subsequent computation. In the case of inc, we want to increment the state but do not expect a result, therefore we have an application k () (suc s). In the case of get, we want the value of the current state but do not need to change it, hence we have an application k s s. With these functions, we define mk-lst using an auxiliary function mk-lst′:

$$\begin{aligned}
\mathsf{mk\text{-}lst} \overset{\text{def}}{\equiv}\ & \mathsf{rec}\ \mathsf{f}_{\Pi\,\mathsf{m}:\mathbb{N}.\,\mathsf{L}\ \mathsf{m}[\mathbb{N}\to\mathsf{L}\ 3,\mathbb{N}\to\mathsf{L}\ 3]}\ \mathsf{m}.\ \mathsf{pm}\ \mathsf{m}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{L}\ \mathsf{x}\ \mathsf{with} \\
& \quad \mathsf{z} \to \mathcal{S}\mathsf{k} : \mathsf{L}\ \mathsf{z} \to \mathbb{N} \to \mathsf{L}\ 3.\ \mathsf{k}\ \mathsf{nil} \mid \mathsf{suc}\ \mathsf{n} \to\ ::\ \mathsf{n}\ (\mathsf{inc}\ ();\ \mathsf{get}\ ())\ (\mathsf{mk\text{-}lst}\ \mathsf{n})
\end{aligned}$$

The mk-lst function pattern matches on the first argument, and returns an empty list if it is z. The use of `shift` here is necessary for making the two branches have the same control effect (which is required by (E-DMatchN)); computationally, the entire branch is equivalent to nil [100]. If the argument is non-zero, we build a list whose first element is an increment of the current state, and whose tail is the result of a recursive call.

Similarly to the choose function, inc and get require a run-state function that resets the context:

$$\mathsf{run\text{-}state} \overset{\text{def}}{\equiv} \lambda\,\mathsf{f} : () \to \mathsf{L}\ 3[\mathbb{N} \to \mathsf{L}\ 3, \mathbb{N} \to \mathsf{L}\ 3].\ \langle(\lambda\,\mathsf{r} : \mathsf{L}\ 3.\ \lambda\,\mathsf{s} : \mathbb{N}.\ \mathsf{r})\ (\mathsf{f}\ ())\rangle\ \mathsf{z}$$

The function plays two roles. First, it initializes the value of the state to z. Second, it builds a delimited context of the form "given a state, return a value." This

[8] Note that the first two functions are specialized for uses in an L 3-returning context as in the goal program. A more general definition is possible once we have extended our language with polymorphism.

function, together with inc and get, allows us to manipulate the state without explicitly referring to s passed around behind the scene.

Shifting the viewpoint to types, we again find an invariant that holds for the input and output of mk-lst: the output list has exactly m elements when the input is m.

Chapter 4

CPS Translating Dellina-

When studying control operators, one of the must-have discussions is how to describe their semantics using pure λ-terms. This question reduces to whether there exists a CPS translation that eliminates the control operators. If the source language is typed, we further require that CPS translation is type-preserving, *i.e.*, a well-typed program that uses control operators is converted to a well-typed program that does not use them.

The `shift` and `reset` operators are notable for their succinct CPS semantics, and the translation is known to preserve typing in simply and polymorphically typed settings [10, 28]. This is a pleasant property from a practical point of view, since it allows supporting `shift` and `reset` in an existing language without rebuilding runtime facilities.

In this chapter, we give a CPS translation of Dellina-. This is a non-trivial task, because CPS translations are known to misbehave in the presence of dependent types. Recall from Section 1.3 that the double negation translation fails to preserve typing when the language has strong Σ types and dependent case analysis. Recent studies managed to recover type preservation by assuming impredicativity and parametricity in the target language [121, 34, 45], but the solution is not satisfactory because the two assumed features are not available by default in dependently typed languages [128, 30].

We solve this challenge by adopting a *selective* CPS translation [126]. A selective translation can be understood as an *on-demand* CPS translation: it translates impure terms (which manipulate continuations) into CPS, and keeps pure terms (which do not touch continuations) in direct style. Selective CPS translations have been studied as an efficient approach to simulating control operators, since they yield more compact programs compared to unselective translations. We, on the other hand, discovered another wonderful advantage of using a selective translation: it gives us the type presevation property *for free*. The intuition is that, since a selective translation does not change the shape of pure terms, and since pure terms are the only things we admit in types, any type equality that holds in Dellina- stays as is in the target.

In the rest of this chapter, we first revisit the challenges with unselective CPS translations of dependently typed languages (Section 4.1), and elaborate the parametricity-based solution of Bowman et al. [34] (Section 4.2). Then we show how selectiveness avoids the known issues without relying on parametricity or impredicativity (Section 4.3). The full translation of Dellina- is given in Section 4.5, and proved type-preserving in Section 4.6.

4.1 Challenges of CPS Translation

Before going into technical details, let us introduce some CPS terminology. When discussing CPS translations, we distinguish between *computations* and *values* in the post-translation world. Computations are terms in CPS, whose evaluation is suspended until they are given a continuation. Values, on the other hand, are direct-style terms that run on their own. Following the literature [3, 34], we use the $^{\div}$-superscript to denote *computation translation*, which produces CPS computations, and $^{+}$-superscript to denote *value translation*, which produces direct-style values. These translations differ in their ability to change the overall structure of the given expression. Suppose we have a function $\lambda\, x.\, x : \mathbb{N} \to \mathbb{N}$. When applied to the function, the $^{\div}$-translation produces a suspended computation $\lambda\, k.\, k\ (\lambda\, x.\, x)^{+}$, where $(\lambda\, x.\, x)^{+} = \lambda\, x.\, \lambda\, k'.\, k'\ x$. Notice that while $^{\div}$ introduces a new λ wrapping around the whole term, $^{+}$ keeps the structure of the original function. When applied to the type $\mathbb{N} \to \mathbb{N}$, the $^{\div}$-translation generates a doubly negated type $\neg\neg(\mathbb{N} \to \mathbb{N})^{+}$, where $(\mathbb{N} \to \mathbb{N})^{+} = \mathbb{N} \to \neg\neg\mathbb{N}$. Observe that while $^{\div}$ introduced two new arrows at the top-level, $^{+}$ preserves the head constructor of the original type.

Now, recall the negative result reported by Barthe and Uustalu [24]: the double-negation-based, call-by-name CPS translation do not scale to Σ and sum types. The conclusion is derived from the observation that second projection and dependent case analysis are mapped to an ill-typed term. For instance, the CPS image of the former looks like:

$$(\mathsf{snd}\ e)^{\div} = \lambda\, k : \neg(B[\mathsf{fst}\ e/x])^{+}.\, e^{\div}\ (\lambda\, v : \Sigma\, x : \neg\neg A^{+}.\, \neg\neg B^{+}.\, (\mathsf{snd}\ v)\ k)$$

As we saw in Section 1.3, the problem is in the application $(\mathsf{snd}\ v)\ k$: while $\mathsf{snd}\ v : \neg\neg B^{+}[\mathsf{fst}\ v/x]$, we have $k : \neg(B[\mathsf{fst}\ e/x])^{+} = \neg B^{+}[(\mathsf{fst}\ e)^{\div}/x]$. For this application to be well-typed, we need $\mathsf{fst}\ v = (\mathsf{fst}\ e)^{\div}$, which is not immediately true.

It turns out that call-by-value translations are even more "broken", in that the type mismatch is already present in the CPS image of function application [34]. Consider the following translation of application $e_0\ e_1$:

$$(e_0\ e_1)^{\div} = \lambda\, k : \neg(B[e_1/x])^{+}.\, e_0^{\div}\ (\lambda\, v_0 : \Pi\, x : A^{+}.\, \neg\neg B^{+}.\, e_1^{\div}\ (\lambda\, v_1 : A^{+}.\, v_0\ v_1\ k))$$

We are presuming that $e_0 : \Pi\, x : A.\, B$ and $e_1 : A$. What is wrong with the translation is the application $v_0\ v_1\ k$: while $v_0\ v_1$ requires a $B^{+}[v_1/x]$-accepting continuation, k is a $(B[e_1/x])^{+}$-accepting continuation. In the call-by-name translation of second projection, we assumed commutativity and rewrote $(B[\mathsf{fst}\ e/x])^{+}$ to $B^{+}[(\mathsf{fst}\ e)^{\div}/x]$, but here we cannot convert $(B[e_1/x])^{+}$ into $B^{+}[e_1^{\div}/x]$, since $e_1^{\div}$ represents a *computation* (of type $\neg\neg A^{+}$), while x must be substituted by a *value* (of type A^{+}).

Interestingly, if we compare call-by-name projection $\mathsf{snd}\ e$ and call-by-value application $e_0\ e_1$, we find that their typing and reduction share a common pattern: both terms have a type dependent on their subterm, and the subterm must be evaluated during evaluation of the whole term.

Let us look at this shared pattern in more detail. $\mathsf{snd}\ e$ is an elimination form of dependent pairs of type $\Sigma\, x : A.\, B$. The reduction of this construct goes as follows: we first evaluate e to a pair (e_1, e_2), and then convert $\mathsf{snd}\ (e_1, e_2)$ into e_2 (note that in a call-by-name setting, the reduction rule applies even when e_1

and e_2 are non-value). Typingwise, $\textsf{snd}\ e$ is given the type $B[\textsf{fst}\ e/x]$. We see that the type depends on the subterm e, which we evaluate when evaluating $\textsf{snd}\ e$. The post-reduction term e_2 canonically has type $B[e_1/x]$, but we know this type is equivalent to the original type $B[\textsf{fst}\ e/x]$ because $\textsf{fst}\ e$ evaluates to e_1.

On the other hand, $e_0\ e_1$ is an elimination form of dependent functions of type $\Pi\, x : A.\, B$. The reduction of this construct goes in the following way: we first evaluate e_0 to a function $\lambda\, x : A.\, e_0'$ (which is not important here), and evaluate e_1 to a value v_1, then convert $(\lambda\, x : A.\, e_0')v_1$ into $e_0'[v_1/x]$. Typingwise, $e_0\ e_1$ is given the type $B[e_1/x]$. We see that the type depends on the subterm e_1, which we evaluate when evaluating $e_0\ e_1$. The post-reduction term $e_0'[v_1/x]$ canonically has type $B[v_1/x]$, but we know this type is equivalent to the original type $B[e_1/x]$ because e_1 evaluates to v_1.

Now we look at their translations. In the call-by-name CPS image of $\textsf{snd}\ e$, evaluation of e is represented by the application $e^{\div}\ (\lambda\, v : (\Sigma\, x : \neg\neg A^{+}.\, \neg\neg B^{+}).\, (\textsf{snd}\ v)\ k)$. Here, the problematic application is $(\textsf{snd}\ v)\ k$, where we are applying a v-dependent computation to an e-dependent continuation.

Similarly, in the call-by-value CPS image of $e_0\ e_1$, evaluation of e_1 is represented by the application $e_1^{\div}\ (\lambda\, v_1 : A^{+}.\, v_0\ v_1\ k)$. Here, the problematic application is $v_0\ v_1\ k$, where we are applying a v_1-dependent computation to an e_1-dependent continuation.

Now the question is: what are v and v_1? It turns out that if the source terms e and e_1 are pure, these variables are to be replaced by a *unique* value, which, in essence, corresponds to the value of e and e_1 [34]. The key to recovering typability, then, is to find a way to communicate this fact to the type system.

As Bowman et al. [34] suggests, our task can be decomposed into two subtasks: (i) representing the unique value in terms of e and e_1; and (ii) expressing its uniqueness. Let us focus our attention on call-by-value application and see what the puzzle is. If we try to tackle the first task, we will find that the fixed answer type $\bot$ gets in the way. In the source language, we obatin the value of e_1 by evaluating it in an empty context. Therefore, in the target of the translation, we want to obtain the unique value by running $e_1^{\div}$ with an empty continuation $\lambda\, v : A^{+}.\, v$. Clearly, this results in a type error, because $e_1^{\div}$ has type $\neg\neg A^{+}$, and hence cannot be applied to the identity function.

In fact, this type error stems from a deeper mismatch in the nature of the continuation $e_1^{\div}$ expects and that of the continuation we supply. In a double-negation-based translation, computations demand a continuation that never returns, but what we are trying to do with the identity continuation is to obtain a value, which means we are implicitly expecting the continuation to return something. This suggests that the use of the $\perp$ type is one source of the challenges with CPS translating dependent types.

The second task, expressing uniqueness of the value, is also non-trivial. Suppose we have managed to obtain the unique value to be substituted for v_1. Since the variable is bound by a λ, we have to type check the body $v_0\ v_1\ k$—which contains the problematic application—only with the assumption that $v_1 : A^+$. This means, in the body of λv_1, v_1 represents an *arbitrary* value inhabiting A^+, instead of some specific value. That is, if we type check continuations as we do for the standard λ-abstraction, we cannot use the unique value even if we have one.

The incapability of expressing uniqueness, however, is an unsurprising consequence, since the double-negation CPS translation allows implementing control operators (such as `call/cc`), which can return an arbitrary value to the context surrounding them. Indeed, if the source term is impure, it is invalid to assume that its continuation receives a unique value. What this means is that, when translating pure terms, we have to use a translation that cannot express control effects in the first place, and then, enrich the type system in such a way that we can soundly impose the unique-value requirement on the continuation of pure terms.

4.2 Past Solution: Answer-type Polymorphism + CPS Axioms

As we saw, the failure of type preservation is due to the lack of a type-safe representation of CPS values, and the incapability of expressing the uniqueness of the value represented by λ-bound variables. How can we overcome these difficulties?

Bowman et al. [34] give the following solution. First, they adopt a specific variant of CPS translation that uses a *polymorphic answer type*. That is, instead of translating a source term $e : A$ to a computation $\lambda\, k : \neg A^+ .\, e'$ of type $\neg\neg A^+$, they translate e to $\lambda\, \alpha : *.\, \lambda\, k : A^+ \to \alpha .\, e'$ of type $\Pi\, \alpha : *.\, (A^+ \to \alpha) \to \alpha$.

Now, $e^{\div}$ expects a continuation that returns as ordinary functions do, and since the answer type can be *any* type, we are always able to run $e^{\div}$ with an empty continuation. More specifically, we first instantiate the answer type of $e^{\div}$ to A^+, and then pass $\lambda\, v : A^+.\, v$ of type $A^+ \to A^+$ (we will hereafter abbreviate the identity function on A^+ as id_{A^+}). Thus, we obtain a type-safe interface from CPS computations to values.

The second idea of Bowman et al. is to equip the target language of the translation with a new equivalence rule, and a new typing rule. These rules are used to reason about CPS computations with a polymorphic type, and are defined roughly as follows:

$$\frac{}{e_1\ A\ (\lambda\, v : B.\, e_2) \equiv (\lambda\, v : B.\, e_2)\ (e_1\ B\ \mathsf{id}_B)}\ [\equiv\text{-}\textsc{Cont}]$$

$$\frac{\Gamma \vdash e_1 : \Pi\, \alpha : *.\, (B \to \alpha) \to \alpha \qquad \Gamma \vdash A : * \qquad \Gamma,\, v = e_1\ B\ \mathsf{id}_B : B \vdash e_2 : A}{\Gamma \vdash e_1\ A\ (\lambda\, v : B.\, e_2) : A}\ [\text{T-}\textsc{Cont}]$$

The equivalence rule, [≡-Cont], gives us an equivalence relation that holds of polymorphic CPS computations. The rule reads: running a CPS computation e_1 with a given continuation $\lambda\, v : B.\, e_2$ is equivalent to first running e_1 with the identity continuation, and then passing the result to the actual continuation. This is a *free theorem* [165] obtained from answer-type polymorphism, and essentially, it tells us that a CPS translated pure term uses its continuation in a trivial way: it calls the continuation with the value it evaluates to. And this value is the result of running the computation with the identity continuation. Since the continuation is called exactly once with this value, no other value can be substituted for the variable v, *i.e.*, v denotes a *unique* value. The equivalence is also called *naturality* [162] and *continuation shuffling* [3, 34] in the literature.

The typing rule, [T-Cont], makes the uniqueness information available when typing an application of a CPS computation to its continuation. The rule says: when type checking the body e_2 of the continuation $\lambda\, v : B.\, e_2$ passed to a CPS computation e_1, we can assume that the variable v represents a unique value $e_1\ B\ \mathsf{id}_B$. The assumption can be used via the following reduction rule:

$$\Gamma \vdash x \quad \rhd_\delta \quad e \text{ if } x = e : A \in \Gamma$$

The rule, often called δ-reduction, allows replacing a variable with its definition [143]. Notice that the reduction rule is indexed by a typing environment Γ, which serves as a store of definitions. Extension by definitions happens in languages featuring *dependent* `let`, which has a typing rule like the following one:

$$\frac{\Gamma,\, x = e_1 : A \vdash e_2 : B}{\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 : B[e_1/x]}\ (\textsc{Let})$$

With this in mind, we can view [T-Cont] as turning the continuation $\lambda\, v : B.\, e_2$ into a `let` expression $\mathsf{let}\ v = e_1\ B\ \mathsf{id}_B : B\ \mathsf{in}\ e_2$. That is, it makes a λ, which binds a variable representing an arbitrary value, behave like a `let`, which binds a variable denoting a unique value.

The sharp-eyed reader might have noticed that, while the equivalence [$\equiv$-Cont] holds only when e_1 has a polymorphic type, the rule has no typing precondition (and it cannot have one, because Bowman et al., like us, use an untyped equivalence), which makes this equivalence a bit suspicious. To prohibit uses of ill-typed equivalence, Bowman et al. introduce a special syntax $e_1\ @\ A\ e_2$ for application of CPS computations to their continuation. This can be understood as telling the type system that we have turned on CPS reasoning. Thus, the actual equivalence and typing rules look like:

$$\frac{}{e_1\ @\ A\ (\lambda\, v : B.\, e_2) \equiv (\lambda\, v : B.\, e_2)\ (e_1\ B\ \mathsf{id}_B)}\ [\equiv\text{-}\textsc{Cont}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \Pi\, \alpha : *.\, (B \to \alpha) \to \alpha \qquad \Gamma \vdash A : * \\ \Gamma,\, v = e_1\ B\ \mathsf{id}_B : B \vdash e_2 : A\end{array}}{\Gamma \vdash e_1\ @\ A\ (\lambda\, v : B.\, e_2) : A}\ [\text{T-}\textsc{Cont}]$$

Now, we obtain the following guarantee: when we use [$\equiv$-Cont], e_1 cannot have a wrong type. This is because (i) equivalence can only ever be used via the conversion rule, which checks the well-formedness of the types on the left- and right-hand sides of $\equiv$; and (ii) the only introduction rule of @ is [T-Cont], which requires e_1 to have a polymorphic type.

Using a polymorphic answer type, a special application form, and two additional rules, Bowman et al. show that it is possible to give a type-preserving translation of call-by-name second projection, as well as call-by-value application. Let us see how the proof goes. We first look at the call-by-name translation of $\mathsf{snd}\ e$, which, when using a polymorphic answer type, takes the following form (for brevity, we write $A^{\div}$ to mean $\Pi\,\alpha : *.\,(A^{+} \to \alpha) \to \alpha$):

$$\lambda\,\alpha : *.\,\lambda\,k : (B[\mathsf{fst}\ e/x])^{+} \to \alpha.\,e^{\div}\ @\ \alpha\ (\lambda\,v : \Sigma\,x : A^{\div}.\,B^{\div}.\,(\mathsf{snd}\ v)\ \alpha\ k)$$

By the induction hypothesis, we have

$$e^{\div} : \Pi\,\alpha : *.\,(\Sigma\,x : A^{\div}.\,B^{\div} \to \alpha) \to \alpha$$

Since the application of $e^{\div}$ uses @, we type this application using [T-CONT]. The rule makes the definition $v = e^{\div}\ S\ \mathsf{id}_S$ available for typing checking of $(\mathsf{snd}\ v)\ \alpha\ k$, where S stands for $\Sigma\,x : A^{\div}.\,B^{\div}$. This allows us to cast the type of $\mathsf{snd}\ v$ from $B^{\div}[\mathsf{fst}\ v/x]$ to $B^{\div}[\mathsf{fst}\ (e^{\div}\ S\ \mathsf{id}_S)/x]$ via the δ-rule. The domain of k, on the other hand, can be rewritten to $B^{+}[(\mathsf{fst}\ e)^{\div}/x]$, since the translation commutes with substitution (this can be proved easily). What remains to be shown is the equivalence $(\mathsf{fst}\ e)^{\div} \equiv \mathsf{fst}\ (e^{\div}\ S\ \mathsf{id}_S)$. By the definition of the translation, we have:

$$(\mathsf{fst}\ e)^{\div} = \lambda\,\alpha : *.\,\lambda\,k : A^{+} \to \alpha.\,e^{\div}\ @\ \alpha\ (\lambda\,v : \Sigma\,x : A^{\div}.\,B^{\div}.\,(\mathsf{fst}\ v)\ \alpha\ k)$$

Using [≡-CONT], we can rewrite the application of $e^{\div}$ in the following way:

$$\begin{aligned}
& e^{\div}\ @\ \alpha\ (\lambda\,v : \Sigma\,x : A^{\div}.\,B^{\div}.\,(\mathsf{fst}\ v)\ \alpha\ k) \\
& \equiv (\lambda\,v : \Sigma\,x : A^{\div}.\,B^{\div}.\,(\mathsf{fst}\ v)\ \alpha\ k)\ (e^{\div}\ S\ \mathsf{id}_S) \qquad \text{by [≡-CONT]} \\
& \rhd_\beta\ \mathsf{fst}\ (e^{\div}\ S\ \mathsf{id}_S)\ \alpha\ k
\end{aligned}$$

Now, by η-equivalence, we obtain $(\mathsf{fst}\ e)^{\div} \equiv \mathsf{fst}\ (e^{\div}\ S\ \mathsf{id}_S)$ as desired.

We next sketch the case where we translate a call-by-value application:

$$\begin{aligned}
& \lambda\,\alpha : *.\,\lambda\,k : (B[e_1/x])^{+} \to \alpha. \\
& \quad e_0^{\div}\ \alpha\ (\lambda\,v_0 : \Pi\,x : A^{+}.\,\Pi\,\alpha : *.\,(B^{+} \to \alpha) \to \alpha.\,e_1^{\div}\ @\ \alpha\ (\lambda\,v_1 : A^{+}.\,v_0\ v_1\ k))
\end{aligned}$$

By the induction hypothesis, we have

$$e_0 : \Pi\, \alpha : *. (\Pi\, x : A^+. (\Pi\, \alpha' : *. (B^+ \to \alpha') \to \alpha') \to \alpha) \to \alpha$$

and

$$e_1 : \Pi\, \alpha : *. (A^+ \to \alpha) \to \alpha$$

Similarly to the snd case, we type the application of $e_1^{\div}$ using [T-Cont], which makes the definition $v_1 = e_1^{\div}\ A^+\ \mathsf{id}_{A+}$ available when type checking $v_0\ v_1\ k$. This allows us to convert the type of $v_0\ v_1$ from $B^+[v_1/x]$ to $B^+[e_1^{\div}\ A^+\ \mathsf{id}_{A+}/x]$. Since we are doing a call-by-value translation, commutativity tells us that the domain of k, $(B[e_1/x])^+$, is equivalent to $B^+[e_1^{\div}\ A^+\ \mathsf{id}_{A+}/x]$. That is, we replace x by the CPS value of $e_1^{\div}$, which corresponds to the value we obtain by evaluating the source term e_1 in an empty context. This is justified by [≡-Cont], as we will see in Section 4.6. Now, we can conclude that the application $v_0\ v_1\ k$ is well-typed.

The translation of Bowman et al. was later extended to a calculus with inductive datatypes and the shift/reset operators [45]. In the presence of control effects, we can no longer translate every term using a polymorphic answer type. However, the tricks of Bowman et al. still apply to the shift/reset-calsulus, since it has a purity restriction on type dependency. This means, any term appearing in a type can be translated into a polymorphic computation, from which we can extract a CPS value.

The polymorphism-based CPS translation is also used in the dependent $\lambda\mu\tilde{\mu}$-calculus of Miquey [121]. The type preservation proof again relies on the constrained dependency of the source language, which guarantees that types depend only on NEF terms.

4.3 Our Solution: Selective Translation

The polymorphic answer type translation solved the 15-year-old problem, but the solution is not completely satisfactory, because the type preservation argument relies heavily on impredicativity and parametricity. Recall how we proved type preservation of snd $e : B[\mathsf{fst}\ e/x]$: to obtain the CPS value of e, we ran $e^{\div}$ with the identity continuation by instantiating its answer type to $\Sigma\, x : A^{\div}. B^{\div}$, where

$A^{\div} = \Pi\, \alpha : *.\, (A^{+} \to \alpha) \to \alpha$. The instantiated answer type has a universal quantification, which means, for the type instantiation to be valid, we must allow Π types to quantify over types including the type being formed. That is, the base kind $*$ has to be impredicative.

Assuming impredicativity restricts scalability of the CPS translation. For instance, Agda is a predicative language, hence it does not admit type instantiation that makes the proof of $\mathsf{snd}\ e$ go through. Coq has an impredicative sort Prop, which is the type of type of proof terms, but all other sorts, like Set and Type_1, are predicative; in particular, disallowing impredicative sorts at non-bottom levels is mandatory for the language to be consistent [84]. This means the value extraction technique does not apply to anything other than proofs and atomic data—that is to say, we cannot even make the call-by-value translation of $(\lambda\, x : \mathbb{N}.\, x)\ 0$ type-preserving!

Parametricity, which justifies the [$\equiv$-Cont] rule, is not a default feature, either. Boulier et al. [30] gives a syntactical translation of a CC-like calculus, where the target can prove the existence of a polymorphic function $f : \Pi\, \alpha : *.\, \alpha \to \alpha$ that is not an identity function. In our context, this result implies that not every variant of type theory admits addition of the equivalence rule [$\equiv$-Cont].

So the question is: can we have a type-preserving CPS translation that does not require impredicativity or parametricity? The answer is yes and no, depending on in what context we CPS translate programs. CPS is often used as an intermediate language of compilers [6], because we can specify the order of evaluation by making the continuation of every piece of code explicit. In this context, it seems inevitable to assume impredicativity and parametricity, because every single term has to be converted into a CPS computation, whose value can only be accessed via the interface $e^{\div}\ A^{+}\ \mathsf{id}_{A^{+}}$.

On the other hand, if we use the translation as an earsure of control operators—that is, if all we want is a program that simulates control effects using pure λ-terms—then we do not have to CPS translate everything; we only need to turn impure terms, which actually uses their continuation, into CPS. Such a non-uniform translation is called a *selective CPS translation.* Selective translations have been studied as a practical means to support control operators [140, 12]. We show that selective translations are particularly well-suited when working with dependent types, since they give us type preservation for free, as long as the source language does not allow types dependent on impure terms.

To see how exactly a selective translation works, let us translate a call-by-value function application. Suppose we have a Dellina- application $\mathsf{e_0}\ \mathsf{e_1}$, where the function $\mathsf{e_0}$ is an impure term, while the argument $\mathsf{e_1}$ and the body of the function are both pure. That is, we have the following derivation:

$$\frac{\Gamma \vdash \mathsf{e_0} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}[\alpha, \beta] \qquad \Gamma \vdash \mathsf{e_1} : \mathsf{A}}{\Gamma \vdash \mathsf{e_0}\ \mathsf{e_1} : \mathsf{B}[\mathsf{e_1}/\mathsf{x}][\alpha, \beta]}\ (\text{E-DApp})$$

The application is translated as follows (note that we use a red font to typeset the target terms); for comparison, we also repeat Bowman et al.'s call-by-value translation of a pure application:

$\boxed{\text{Selective Translation}}$

$$\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^{+} \to \alpha^{+}.\,\mathsf{e_0}^{\div}\ (\lambda\,\mathbf{v_0} : \mathbf{\Pi\,x} : \mathbf{A}^{+}.\,\mathbf{B}^{+}.\,\mathbf{k}\ (\mathbf{v_0}\ \mathsf{e_1}^{+}))$$

$\boxed{\text{Unselective Translation (Bowman et al.)}}$

$$\begin{aligned}&\lambda\,\alpha : *.\,\lambda\,k : (B[e_1/x])^{+} \to \alpha.\\ &\quad e_0^{\div}\ \alpha\ (\lambda\,v_0 : \Pi\,x : A^{+}.\,\Pi\,\alpha : *.\,(B^{+} \to \alpha) \to \alpha.\,e_1^{\div}\ @\ \alpha\ (\lambda\,v_1 : A^{+}.\,v_0\ v_1\ k))\end{aligned}$$

When selectively translating terms, we follow two principles: (i) turn every impure term into $\lambda\,\mathbf{k}.\,\mathbf{e}$; and (ii) keep every pure term in direct style. That is, we apply a computation translation $^{\div}$ to impure terms, and a value translation $^{+}$ to pure terms. Now, if we look at the derivation of $\mathsf{e_0}\ \mathsf{e_1}$, we see that the whole application is an impure term, hence it must be applied the computation translation, and be turned into a continuation-awaiting function $\lambda\,\mathbf{k}.\,\mathbf{e}$. The impureness is brought by the function $\mathsf{e_0}$, so $\mathsf{e_0}$ also has to be applied the computation translation, and the result $\mathsf{e_0}^{\div}$ must be passed a continuation $\lambda\,\mathbf{v_0}.\,\mathbf{e_0}$. The argument $\mathsf{e_1}$, on the other hand, is a pure term. This means that we should apply the value translation $^{+}$ to $\mathsf{e_1}$, which gives us a direct-style term. Therefore, in the CPS image of the application, we do not have the familiar CPS pattern $\mathsf{e_1}^{\div}\ (\lambda\,\mathbf{v_1}.\,\mathbf{e_1})$; instead, we directly pass $\mathsf{e_1}^{+}$ to $\mathbf{v_0}$, which represents the CPS value of the function $\mathsf{e_0}$. Lastly, we need to figure out how to use the top-level continuation $\mathbf{k}$ to perform the computation that happens after the application $e_0\ e_1$. In this case, the right way to continue evaluation is to apply $\mathbf{k}$ to the application $\mathbf{v_0}\ {e_1}^+$. Notice that we are using $\mathbf{k}$ differently from the unselective translation, where the application is the other way around, namely $\mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k}$. This difference is due to the fact that we do not translate the body of e_0, which, as the derivation suggests, is a pure term. While this body is not present syntactically, we can see how it is translated if we look at the type of $\mathbf{v_0}$: the co-domain B^+ has no arrow representing the demand for a continuation. In contrast, the unselective translation tells us that the co-domain of v_0 is $B^\div = \Pi\, \alpha : *.\, (B^+ \to \alpha) \to \alpha$, *i.e.*, v_0 is a function that requires an extra argument representing a continuation.

Having seen how to selectively translate application, let us look at how selectiveness helps us type CPS-translated application. Our goal is to show:

$$\lambda\, \mathbf{k} : (B[e_1/x])^+ \to \alpha^+.\, {e_0}^\div\ (\lambda\, \mathbf{v_0} : \mathbf{\Pi\, x} : \mathbf{A}^+.\, \mathbf{B}^+.\, \mathbf{k}\ (\mathbf{v_0}\ {e_1}^+))$$

has type

$$((B[e_1/x])^+ \to \alpha^+) \to \beta^+$$

in the environment Γ^+. By the induction hypothesis, we have

$$\Gamma^+ \vdash {e_0}^\div : (\mathbf{\Pi\, x} : \mathbf{A}^+.\, \mathbf{B}^+ \to \alpha^+) \to \beta^+ \quad \text{and} \quad \Gamma^+ \vdash {e_1}^+ : A^+$$

The interesting part is the last application $\mathbf{k}\ (\mathbf{v_0}\ {e_1}^+)$. By the application rule of the target language, we know $\mathbf{v_0}\ {e_1}^+ : B^+[{e_1}^+/\mathbf{x}]$. It suffices to show that $B^+[{e_1}^+/\mathbf{x}]$ is equivalent to the domain of $\mathbf{k}$, namely $(B[e_1/x])^+$. As we will see in Section 4.6, it is fairly easy to show this is the case, thanks to the selectiveness. Now, we know $\mathbf{k}\ (\mathbf{v_0}\ {e_1}^+) : \alpha^+$, and the goal follows by the application and abstraction rules.

4.4 Target Language

As a first step to defining a CPS translation of Dellina-, we formalize the target language of the translation. The language is roughly the pure subset of Dellina- featuring a call-by-value semantics. Below we define the syntax, reduction, equivalence, and typing rules.

$$\begin{array}{llll}
\text{Environments} & \mathbf{\Gamma} & ::= & \bullet \mid \mathbf{\Gamma}, \mathbf{x} : \mathbf{A} \mid \mathbf{\Gamma}, \mathbf{e_1} \equiv \mathbf{e_2} \\
\text{Kinds} & \kappa & ::= & * \mid \square \\
\text{Types} & \mathbf{A} & ::= & \mathbf{Unit} \mid \mathbb{N} \mid \mathbf{L}\ \mathbf{e} \mid \mathbf{\Pi\, x : A.\, B} \\
\text{Values} & \mathbf{v} & ::= & \mathbf{x} \mid \lambda\, \mathbf{x : A.\, e} \mid \mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi x : A.B}}\ \mathbf{x.\, e} \mid \mathbf{z} \mid \mathbf{suc\ v} \mid \mathbf{nil} \mid \mathbf{::\ v\ v\ v} \\
\text{Terms} & \mathbf{e} & ::= & \mathbf{v} \mid \mathbf{e\ e} \mid \mathbf{suc\ e} \mid \mathbf{::\ e\ e\ e} \\
& & \mid & \mathbf{pm\ e\ as\ x\ in}\ \mathbb{N}\ \mathbf{ret\ P\ with\ z} \to \mathbf{e} \mid \mathbf{suc\ n} \to \mathbf{e} \\
& & \mid & \mathbf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \to \mathbf{e} \mid \mathbf{::\ m\ h\ t} \to \mathbf{e}
\end{array}$$

Figure 4.1: Target Syntax

4.4.1 Syntax

We show the the syntax of the target language in Figure 4.1. We use a red, serif font to typeset target expressions. Our first observation is that typing environments can be extended with equivalence information $\mathbf{e_1} \equiv \mathbf{e_2}$. As we will see in Section 4.6, this is necessary for the CPS image of pattern matching to be well-typed. We next find that function types $\mathbf{\Pi\, x : A.\, B}$ are in their standard form; that is, they do not carry effect annotations. This should not be surprising, because the target language has no control effects. Thirdly, the definition of values and computations tells us that the target language is call-by-value. It is a common design decision to equip the target with a call-by-value semantics when CPS translating call-by-value control operators. The reason is that the translation does not fix the order of evaluation in the presence of control effects: if the translation is unselective, it produces nested function application in the `shift` and `reset` cases, and if the translation is selective, it yields potentially nested function calls in the application cases with a pure argument.

4.4.2 Reduction and Equivalence

As in Dellina-, we define runtime evaluation and typechecking-time reduction of the target language using distinct sets of rules. In Figure 4.2, we see one kind of evaluation contexts $\mathbf{E}$, which have the exactly same definition as Dellina-'s pure contexts (*i.e.*, F). On the other hand, reduction rules have been extended with the β_Ω-rule, which allows substitution of non-value function arguments for variables in a redex position. These rules are commonly used when reasoning about CPS

$\boxed{\text{Evaluation Contexts } \mathbf{E}}$

$$\begin{array}{rcl} \mathbf{E} & ::= & [\,] \mid \mathbf{E}\ \mathbf{e} \\ \mid \mathbf{v}\ \mathbf{E} & & \\ & \mid & \mathbf{pm}\ \mathbf{E}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with}\ \mathbf{z} \to \mathbf{e} \mid \mathbf{suc}\ \mathbf{n} \to \mathbf{e} \\ & \mid & \mathbf{pm}\ \mathbf{E}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbf{L}\ \mathbf{a}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with}\ \mathbf{nil} \to \mathbf{e} \mid ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to \mathbf{e} \end{array}$$

$\boxed{\text{Reduction Rules } \mathbf{e} \ \triangleright\ \mathbf{e}'}$

$$\begin{array}{rcl} (\lambda\,\mathbf{x}.\,\mathbf{e})\ \mathbf{v} & \triangleright_{\beta} & \mathbf{e}[\mathbf{v}/\mathbf{x}] \\ (\lambda\,\mathbf{x}.\,\mathbf{E}[\mathbf{x}])\ \mathbf{e} & \triangleright_{\beta\Omega} & \mathbf{E}[\mathbf{e}] \ \text{ if } \mathbf{x} \notin FV(\mathbf{E}) \\ (\mathbf{rec}\ \mathbf{f}\ \mathbf{x}.\,\mathbf{e})\ \mathbf{v} & \triangleright_{\mu} & \mathbf{e}[\mathbf{rec}\ \mathbf{f}\ \mathbf{x}.\,\mathbf{e}/\mathbf{f}, \mathbf{v}/\mathbf{x}] \\ \begin{array}{r}\mathbf{pm}\ \mathbf{z}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with} \\ \mathbf{z} \to \mathbf{e_1} \mid \mathbf{suc}\ \mathbf{n} \to \mathbf{e_2}\end{array} & \triangleright_{\iota} & \mathbf{e_1} \\ \begin{array}{r}\mathbf{pm}\ \mathbf{suc}\ \mathbf{v}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with} \\ \mathbf{z} \to \mathbf{e_1} \mid \mathbf{suc}\ \mathbf{n} \to \mathbf{e_2}\end{array} & \triangleright_{\iota} & \mathbf{e_2}[\mathbf{v}/\mathbf{n}] \\ \begin{array}{r}\mathbf{pm}\ \mathbf{nil}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbf{L}\ \mathbf{a}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with} \\ \mathbf{nil} \to \mathbf{e_1} \mid ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to \mathbf{e_2}\end{array} & \triangleright_{\iota} & \mathbf{e_1} \\ \begin{array}{r}\mathbf{pm}\ ::\ \mathbf{v_0}\ \mathbf{v_1}\ \mathbf{v_2}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbf{L}\ \mathbf{a}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with} \\ \mathbf{nil} \to \mathbf{e_1} \mid ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to \mathbf{e_2}\end{array} & \triangleright_{\iota} & \mathbf{e_2}[\mathbf{v_0}/\mathbf{m}, \mathbf{v_1}/\mathbf{h}, \mathbf{v_2}/\mathbf{t}] \end{array}$$

$\boxed{\text{Single-step Evaluation}}$

$$\frac{\mathbf{e}\ \triangleright\ \mathbf{e}'}{\mathbf{E}[\mathbf{e}]\ \triangleright\ \mathbf{E}[\mathbf{e}']}\ [\text{R-Eval}]$$

$\boxed{\text{Multi-step Evaluation } \mathbf{e}\ \triangleright^{\star}\ \mathbf{e}'}$

$$\frac{}{\mathbf{e}\ \triangleright^{\star}\ \mathbf{e}}\ [\text{RS-Refl}] \qquad \frac{\mathbf{e_0}\ \triangleright^{\star}\ \mathbf{e_1} \qquad \mathbf{e_1}\ \triangleright^{\star}\ \mathbf{e_2}}{\mathbf{e_0}\ \triangleright^{\star}\ \mathbf{e_2}}\ [\text{RS-Trans}]$$

Figure 4.2: Target Evaluation and Reduction Rules

programs, and are compatible with the call-by-value semantics [100], because a computation in a redex position must be evaluated to a value.

We next extend the reduction relation to parallel reduction (Figures 4.3 - 4.4), which we use to define the equivalence rule [$\equiv$] (Figure 4.5). However, in the target language, this is not the only rule for deriving equivalence: we also have a pair of η-rules, which we use in the proof of type preservation. The addition of the η-rules require explicit congruence and transitivity rules. Note that congruence rules simply state that two expressions are equivalent if all of their subexpressions are equivalent: *e.g.*, $e_0\ e_1 \equiv e'_0\ e'_1$ if $e_0 \equiv e'_0$ and $e_1 \equiv e'_1$.

Remark We do not include η-reduction ($\lambda\,\mathbf{x} : \mathbf{A}.\,\mathbf{e}\ \mathbf{x} \ \triangleright\ \mathbf{e}$) in the rules for runtime/typechecking-time reduction. The reason is that η-reduction is known to behave badly in dependently typed languages. In particular, adding η-reduction results in the loss of preservation [160], and makes confluence dependent on normalization of the language[1] [80, 81].

4.4.3 Typing

We now look at typing rules, which are defined in Figures 4.6 - 4.8. Most rules are instances of Dellina- rules where all effect annotations are empty. As the language is effect-free, there is no need to separate dependent and non-dependent rules for application and pattern matching. However, we still need the well-formedness premises of result types, because substitution of values is unsafe under a call-by-value semantics. A closer look at [E-MatchN] and [E-MatchL] reveals the fact that we reason about target parttern matching with some additional information. These rules say: when we type check the branches, we extend the context not only with the constructor arguments, but also with equivalence information. For instance, when we have a pattern matching construct that inspects a natural number $\mathbf{e}$, we type check the zero branch $\mathbf{e_1}$ with $\mathbf{e} \equiv \mathbf{z}$, and the suc branch $\mathbf{e_2}$ with $\mathbf{e} \equiv \mathbf{suc\ n}$. That is, we assume that the actual scrutinee $\mathbf{e}$ is equivalent to the

[1]Nederpelt [125] was the first to identify the issue with η and confluence on "pseudo-terms", *i.e.*, terms that have not applied type checking. Specifically, he used the following counterexample:

$$\lambda\,\mathbf{x} : \mathbf{A}.\,\mathbf{x} \lhd_\beta \lambda\,\mathbf{x} : \mathbf{A}.\,(\lambda\,\mathbf{y} : \mathbf{B}.\,\mathbf{y})\ \mathbf{x} \rhd_\eta \lambda\,\mathbf{y} : \mathbf{B}.\,\mathbf{y} \quad \text{where } \mathbf{A} \not\equiv \mathbf{B}$$

$$\frac{}{\mathbf{t} \rhd_p \mathbf{t}}\ [\text{P-Refl}]$$

$$\frac{\mathbf{e} \rhd_p \mathbf{e}'}{\mathbf{L}\ \mathbf{e} \rhd_p \mathbf{L}\ \mathbf{e}'}\ [\text{P-List}]$$

$$\frac{\mathbf{A} \rhd_p \mathbf{A}' \qquad \mathbf{B} \rhd_p \mathbf{B}'}{\mathbf{\Pi\, x : A.\, B} \rhd_p \mathbf{\Pi\, x : A'.\, B'}}\ [\text{P-Pi}]$$

$$\frac{\mathbf{A} \rhd_p \mathbf{A}' \qquad \mathbf{e} \rhd_p \mathbf{e}'}{\boldsymbol{\lambda}\, \mathbf{x : A.\, e} \rhd_p \boldsymbol{\lambda}\, \mathbf{x : A'.\, e'}}\ [\text{P-Abs}]$$

$$\frac{\mathbf{\Pi\, x : A.\, B} \rhd_p \mathbf{\Pi\, x : A'.\, B'} \qquad \mathbf{e} \rhd_p \mathbf{e}'}{\mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi\, x : A.\, B}}\ \mathbf{x.\, e} \rhd_p \mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi\, x : A'.\, B'}}\ \mathbf{x.\, e'}}\ [\text{P-Rec}]$$

$$\frac{\mathbf{e_0} \rhd_p \mathbf{e_0'} \qquad \mathbf{e_1} \rhd_p \mathbf{e_1'}}{\mathbf{e_0}\ \mathbf{e_1} \rhd_p \mathbf{e_0'}\ \mathbf{e_1'}}\ [\text{P-App}]$$

$$\frac{\mathbf{e_0} \rhd_p \mathbf{e_0'} \qquad \mathbf{v_1} \rhd_p \mathbf{v_1'}}{(\boldsymbol{\lambda}\, \mathbf{x : A.\, e_0})\ \mathbf{e_1} \rhd_p \mathbf{e_0'}[\mathbf{v_1'}/\mathbf{x}]}\ [\text{P-AppBeta}]$$

$$\frac{\begin{array}{c}\mathbf{\Pi\, x : A.\, B} \rhd_p \mathbf{\Pi\, x : A'.\, B'} \\ \mathbf{e_0} \rhd_p \mathbf{e_0'} \qquad \mathbf{v_1} \rhd_p \mathbf{v_1'}\end{array}}{(\mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi\, x : A.\, B}}\ \mathbf{x.\, e_0})\ \mathbf{v_1} \rhd_p \mathbf{e_0'}[\mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi\, x : A.\, B}}\ \mathbf{x.\, e_0'}/\mathbf{f}, \mathbf{v_1'}/\mathbf{x}]}\ [\text{P-AppMu}]$$

$$\frac{\mathbf{E} \rhd_p \mathbf{E}' \qquad \mathbf{e} \rhd_p \mathbf{e}'}{(\boldsymbol{\lambda}\, \mathbf{x : A.\, E}[\mathbf{x}])\ \mathbf{e} \rhd_p \mathbf{E}'[\mathbf{e}]}\ [\text{P-AppBetaOmega}]$$

Figure 4.3: Target Parallel Reduction (Types and λ-Terms)

$$\frac{\mathbf{e} \rhd_p \mathbf{e'}}{\mathbf{suc\ e} \rhd_p \mathbf{suc\ e'}}\ [\text{P-Suc}]$$

$$\frac{\mathbf{e_0} \rhd_p \mathbf{e'_0} \quad \mathbf{e_1} \rhd_p \mathbf{e'_1} \quad \mathbf{e_2} \rhd_p \mathbf{e'_2}}{\mathbf{::\ e_0\ e_1\ e_2} \rhd_p \mathbf{::\ e'_0\ e'_1\ e'_2}}\ [\text{P-Cons}]$$

$$\frac{\mathbf{e} \rhd_p \mathbf{e'} \quad \mathbf{P} \rhd_p \mathbf{P'} \quad \mathbf{e_1} \rhd_p \mathbf{e'_1} \quad \mathbf{e_2} \rhd_p \mathbf{e'_2}}{\begin{array}{l}\mathbf{pm\ e\ as\ x\ in\ \mathbb{N}\ ret\ P\ with\ z \to e_1 \,|\, suc\ n \to e_2} \rhd_p \\ \mathbf{pm\ e'\ as\ x\ in\ \mathbb{N}\ ret\ P'\ with\ z \to e'_1 \,|\, suc\ n \to e'_2}\end{array}}\ [\text{P-MatchN}]$$

$$\frac{\mathbf{e_1} \rhd_p \mathbf{e'_1}}{\mathbf{pm\ z\ as\ x\ in\ \mathbb{N}\ ret\ P\ with\ z \to e_1 \,|\, suc\ n \to e_2} \rhd_p \mathbf{e'_1}}\ [\text{P-MatchZero}]$$

$$\frac{\mathbf{v} \rhd_p \mathbf{v'} \quad \mathbf{e_2} \rhd_p \mathbf{e'_2}}{\mathbf{pm\ suc\ v\ as\ x\ in\ \mathbb{N}\ ret\ P\ with\ z \to e_1 \,|\, suc\ n \to e_2} \rhd_p \mathbf{e'_2[v'/n]}}\ [\text{P-MatchSuc}]$$

$$\frac{\mathbf{e} \rhd_p \mathbf{e'} \quad \mathbf{P} \rhd_p \mathbf{P'} \quad \mathbf{e_1} \rhd_p \mathbf{e'_1} \quad \mathbf{e_2} \rhd_p \mathbf{e'_2}}{\begin{array}{l}\mathbf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil \to e_1 \,|\, ::\ m\ h\ t \to e_2} \rhd_p \\ \mathbf{pm\ e'\ as\ x\ in\ L\ a\ ret\ P\ with\ nil \to e'_1 \,|\, ::\ m\ h\ t \to e'_2}\end{array}}\ [\text{P-MatchL}]$$

$$\frac{\mathbf{e_1} \rhd_p \mathbf{e'_1}}{\mathbf{pm\ nil\ as\ x\ in\ L\ a\ ret\ P'\ with\ nil \to e_1 \,|\, ::\ m\ h\ t \to e_2} \rhd_p \mathbf{e'_1}}\ [\text{P-MatchNil}]$$

$$\frac{\mathbf{v_0} \rhd_p \mathbf{v'_0} \quad \mathbf{v_1} \rhd_p \mathbf{v'_1} \quad \mathbf{v_2} \rhd_p \mathbf{v'_2} \quad \mathbf{e_2} \rhd_p \mathbf{e'_2}}{\begin{array}{r}\mathbf{pm\ ::\ v_0\ v_1\ v_2\ as\ x\ in\ L\ a\ ret\ P\ with\ nil \to e_1 \,|\, ::\ m\ h\ t \to e_2} \rhd_p \\ \mathbf{e'_2[v'_0/m, v'_1/h, v'_2/t]}\end{array}}\ [\text{P-MatchCons}]$$

$$\frac{}{\mathbf{t} \rhd_p^\star \mathbf{t}}\ [\text{PS-Refl}] \qquad \frac{\mathbf{t_1} \rhd_p \mathbf{t_1} \quad \mathbf{t_1} \rhd_p^\star \mathbf{t_2}}{\mathbf{t_1} \rhd_p^\star \mathbf{t_2}}\ [\text{PS-Trans}]$$

Figure 4.4: Target Parallel Reduction (Inductive Data, Reflexivity, Transitivity)

$$\frac{\mathbf{t_1} \rhd_p \mathbf{t} \quad \mathbf{t_1} \rhd_p \mathbf{t}}{\mathbf{0} \equiv \mathbf{1}}\ [\equiv]$$

$$\frac{\mathbf{e} \rhd_p \lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{e_0} \quad \mathbf{e'} \rhd_p \mathbf{v_1} \quad \mathbf{e_0} \equiv \mathbf{v_1}\ \mathbf{x}}{\mathbf{e} \equiv \mathbf{e'}}\ [\equiv\text{-}\eta_1]$$

$$\frac{\mathbf{e} \rhd_p \mathbf{v_0} \quad \mathbf{e'} \rhd_p \lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{e_1} \quad \mathbf{v_0}\ \mathbf{x} \equiv \mathbf{e_1}}{\mathbf{e} \equiv \mathbf{e'}}\ [\equiv\text{-}\eta_2]$$

$+$ congruence and transitivity

Figure 4.5: Target Equivalence

$\boxed{\text{Well-formed Environments} \vdash \mathbf{\Gamma}}$

$$\frac{}{\vdash \bullet}\ [\text{G-Empty}] \qquad \frac{\vdash \mathbf{\Gamma} \quad \mathbf{\Gamma} \vdash \mathbf{A} : *}{\vdash \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}}\ [\text{G-Ext}]$$

$$\frac{\vdash \mathbf{\Gamma} \quad \mathbf{\Gamma} \vdash \mathbf{e_1} : \mathbf{A} \quad \mathbf{\Gamma} \vdash \mathbf{e_2} : \mathbf{A} \quad \mathbf{e_1} \equiv \mathbf{e_2}}{\vdash \mathbf{\Gamma}, \mathbf{e_1} \equiv \mathbf{e_2}}\ [\text{G-ExtEq}]$$

$\boxed{\text{Well-formed Kinds } \mathbf{\Gamma} \vdash \kappa : \square}$

$$\frac{\vdash \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash * : \square}\ [\text{K-Star}]$$

$\boxed{\text{Well-formed Types } \mathbf{\Gamma} \vdash \mathbf{A} : *}$

$$\frac{\vdash \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbf{Unit} : *}\ [\text{T-Unit}] \qquad \frac{\vdash \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbb{N} : *}\ [\text{T-Nat}] \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{e} : \mathbb{N}}{\mathbf{\Gamma} \vdash \mathbf{L}\ \mathbf{e} : *}\ [\text{T-List}]$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{A} : * \quad \mathbf{\Gamma}, \mathbf{x} : \mathbf{A} \vdash \mathbf{B} : *}{\mathbf{\Gamma} \vdash \Pi\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} : *}\ [\text{T-Pi}]$$

Figure 4.6: Target Well-formed Environements, Kinds, and Types

$\boxed{\text{Well-typed Terms } \Gamma \vdash e : A}$

$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x : A}\text{[E-Var]} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : \Pi x : A.\, B}\text{[E-Abs]}$$

$$\frac{\Gamma, f : \Pi x : A.\, B, x : A \vdash e : B \quad \Gamma \vdash \Pi x : A.\, B : * \quad \text{guard}(f, x, e, \{\})}{\Gamma \vdash \mathbf{rec}\ f_{\Pi x : A.\, B}\ x.\, e : \Pi x : A.\, B}\text{[E-Rec]}$$

$$\frac{\Gamma \vdash e_0 : \Pi x : A.\, B \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash B[e_1/x] : *}{\Gamma \vdash e_0\ e_1 : B[e_1/x]}\text{[E-App]}$$

Figure 4.7: Target Typing Rules (λ-terms)

current pattern. In [E-MatchL], we further use an equivalence assumption on the length index: assume $n \equiv z$ and $e \equiv \mathbf{nil}$ in the empty branch, and $n \equiv \mathbf{suc}\ m$ and $e \equiv\ ::\ m\ h\ t$ in the cons branch.

The purpose of adding these equivalences is to make the type preservation proof of the CPS translation go through. The relevant case is when we have a Dellina- pattern matching that inspects a pure term and returns an impure computation. Since the branches are impure, we translate the whole construct into a CPS computation $\lambda k.\, e$, and distribute k to each branch. However, this results in a type mismatch: while the translated branches have a type dependent on the corresponding pattern, the continuation k has a type dependent on the scrutinee. To solve this mismatch, we need to assume in each branch that the scrutinee is equivalent to the current pattern.

The need for equivalence information motivates us to define a new formation rule [G-ExtEq] for typing environments, which allows extension by an assumption of the form $e_1 \equiv e_2$. As can be easily seen, the rule does not exclude inconsistent assumptions like $z \equiv \mathbf{suc}\ z$, and indeed, the assumptions introduced by [E-MatchN] and [E-MatchL] can be inconsistent. For instance, when type checking

$$\mathbf{pm}\ z\ \mathbf{as}\ x\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ P\ \mathbf{with}\ z \to e_1 \,|\, \mathbf{suc}\ n \to e_2$$

$$\frac{\vdash \Gamma}{\Gamma \vdash () : \mathbf{Unit}}\ [\text{E-Unit}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{z} : \mathbb{N}}\ [\text{E-Zero}] \qquad \frac{\Gamma \vdash \mathbf{e} : \mathbb{N}}{\Gamma \vdash \mathbf{suc}\ \mathbf{e} : \mathbb{N}}\ [\text{E-Suc}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{nil} : \mathbf{L}\ \mathbf{z}}\ [\text{E-Nil}] \qquad \frac{\Gamma \vdash \mathbf{e_0} : \mathbb{N} \quad \Gamma \vdash \mathbf{e_1} : \mathbb{N} \quad \Gamma \vdash \mathbf{e_2} : \mathbf{L}\ \mathbf{e_0}}{\Gamma \vdash\ ::\ \mathbf{e_0}\ \mathbf{e_1}\ \mathbf{e_2} : \mathbf{L}\ (\mathbf{suc}\ \mathbf{e_0})}\ [\text{E-Cons}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{e} : \mathbb{N} \quad \Gamma, \mathbf{x} : \mathbb{N} \vdash \mathbf{P} : * \\ \Gamma, \mathbf{e} \equiv \mathbf{z} \vdash \mathbf{e_1} : \mathbf{P}[\mathbf{z}/\mathbf{x}] \quad \Gamma, \mathbf{n} : \mathbb{N}, \mathbf{e} \equiv \mathbf{suc}\ \mathbf{n} \vdash \mathbf{e_2} : \mathbf{P}[\mathbf{suc}\ \mathbf{n}/\mathbf{x}] \\ \Gamma \vdash \mathbf{P}[\mathbf{e}/\mathbf{x}] : *\end{array}}{\Gamma \vdash \mathbf{pm}\ \mathbf{e}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with}\ \mathbf{z} \rightarrow \mathbf{e_1} \mid \mathbf{suc}\ \mathbf{n} \rightarrow \mathbf{e_2} : \mathbf{P}[\mathbf{e}/\mathbf{x}]}\ [\text{E-MatchN}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{e} : \mathbf{L}\ \mathbf{n} \quad \Gamma, \mathbf{a} : \mathbb{N}, \mathbf{x} : \mathbf{L}\ \mathbf{a} \vdash \mathbf{P} : * \\ \Gamma, \mathbf{n} \equiv \mathbf{z}, \mathbf{e} \equiv \mathbf{nil} \vdash \mathbf{e_1} : \mathbf{P}[\mathbf{z}/\mathbf{a}, \mathbf{nil}/\mathbf{x}] \\ \Gamma, \mathbf{m} : \mathbb{N}, \mathbf{h} : \mathbb{N}, \mathbf{t} : \mathbf{L}\ \mathbf{m}, \mathbf{n} \equiv \mathbf{suc}\ \mathbf{m}, \mathbf{e} \equiv\ ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \vdash \mathbf{e_2} : \mathbf{P}[\mathbf{suc}\ \mathbf{m}/\mathbf{a}, ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t}/\mathbf{x}] \\ \Gamma \vdash \mathbf{P}[\mathbf{n}/\mathbf{a}, \mathbf{e}/\mathbf{x}] : *\end{array}}{\Gamma \vdash \mathbf{pm}\ \mathbf{e}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbf{L}\ \mathbf{a}\ \mathbf{ret}\ \mathbf{P}\ \mathbf{with}\ \mathbf{nil} \rightarrow \mathbf{e_1} \mid\ ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \rightarrow \mathbf{e_2} : \mathbf{P}[\mathbf{n}/\mathbf{a}, \mathbf{e}/\mathbf{x}]}\ [\text{E-MatchL}]$$

$$\frac{\Gamma \vdash \mathbf{e} : \mathbf{A} \quad \Gamma \vdash \mathbf{B} : * \quad \mathbf{A} \equiv \mathbf{B}}{\Gamma \vdash \mathbf{e} : \mathbf{B}}\ [\text{E-Conv}]$$

$$\frac{\mathbf{e_1} \equiv \mathbf{e_2} \in \Gamma \quad \mathbf{e_1} \equiv \mathbf{c_i}\ \overline{\mathbf{a}} \quad \mathbf{e_2} \equiv \mathbf{c_j}\ \overline{\mathbf{b}} \quad \mathbf{c_i} \neq \mathbf{c_j}}{\Gamma \vdash \mathbf{e} : \mathbf{A}}\ [\text{E-InCon}]$$

Figure 4.8: Target Typing Rules (Inductive Data and Conversion)

we would assume $\mathbf{z} \equiv \mathbf{suc}\ \mathbf{n}$ in $\mathbf{e_2}$. With this assumption, type checking of $\mathbf{e_2}$ no longer makes any sense, therefore we "skip" this process via a new typing rule [E-InCon] [99]. The rule tells us that, if $\boldsymbol{\Gamma}$ contains an inconsistent assumption, we may derive *any* conclusion $\mathbf{e} : \mathbf{A}$. The inconsistency is defined in terms of head constructors; specifically, we say $\mathbf{e_1} \equiv \mathbf{e_2}$ is inconsistent if $\mathbf{e_1}$ and $\mathbf{e_2}$ reduce to $\mathbf{c_i}\ \overline{\mathbf{a}}$ and $\mathbf{c_j}\ \overline{\mathbf{b}}$, where $\mathbf{c_i}$ and $\mathbf{c_j}$ are different constructors, and $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$ are sequences of arguments. From a logical point of view, [E-InCon] can be viewed as the *principle of explosion*, also known as *Ex Falso Quodlibet (EFQ)*.

The introduction of [E-InCon] comes at the cost of giving up termination and other metatheoretic properties (such as regularity and canonical forms). Nevertheless, we believe that the fragment we are interested in—namely the image of our CPS translation—has the good properties. Intuitively, the reason is that there is no way to use inconsistent equivalence in the source language (remember that our equivalence is sound), and that the translation is defined in a way that it never uses the assumed equivalence to do anything bad.

4.5 CPS Translation

Based on the idea we sketched in Section 4.3, we design a selective CPS translation of Dellina-. Unlike ordinary CPS translations, which are defined by indution on the structure, our selective translation is defined on the typing derivation. One reason is that our target language features annotated abstractions, which means we must give an appropriate type annotation to all variables introduced by the translation. Suppose we have an application $\mathsf{e}_0\ \mathsf{e}_1$ where all subterms are impure. In its CPS image, we need to generate annotations for variables $\mathbf{k}$, $\mathbf{v_0}$, and $\mathbf{v_1}$:

$$\lambda\,\mathbf{k} : ?_1 \to ?_2.\ \mathsf{e}_0{}^{\div}\ (\lambda\,\mathbf{v_0} : ?_3.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : ?_4.\ \mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k}))$$

Since a CPS translation turns evaluation contexts in the source language into functions in the target language, $?_1$, $?_3$, and $?_4$ must be (the CPS translation of) the type of $\mathsf{e}_0\ \mathsf{e}_1$, e_0, and e_1. We also know that $?_2$ must be (the CPS translation of) the initial answer type of the whole application. None of these are available from the source term $\mathsf{e}_0\ \mathsf{e}_1$, but if we look at the *derivation* of this application, namely

$$\frac{\Gamma \vdash e_0 : (A \to B[\alpha, \beta])[\gamma, \delta] \qquad \Gamma \vdash e_1 : A[\beta, \gamma]}{\Gamma \vdash e_0\ e_1 : B[\alpha, \delta]} \ (\text{E-NDApp})$$

then we can obtain all the information we need to generate annotations.

The second reason we define the translation on the derivation is that we translate pure and impure terms in distinct ways. Recall that we have a value translation, which applies to pure terms, and a computation translation, which applies to impure terms. If we decide applicability of these translations using syntatic information, we would be able to apply the value translation only to syntactically pure terms, such as variables and functions. However, in Dellina-, we have other forms of pure terms as well, including application and pattern matching. Then, what will happen is that, some source terms, which are judged pure and allowed to appear in a type, would be applied the computation translation. This would break type preservation, since the property relies heavily on the fact that pure terms are kept in direct style.

For these reasons, we define one CPS image for each "instance" of the typing rules from Section 3.3. By "instance", we mean the possible combinations of the effect annotations of subterms: *e.g.*, in the case of function application, each of the function, the argument, and the body of the function can be pure or impure, hence we define $2^3 = 8$ different CPS images.

Note that we will use the following abbreviation throughout the translation:

$$\begin{aligned}
\Gamma^+ &\overset{\text{def}}{\equiv} \mathbf{\Gamma} \ \text{ where} \vdash \Gamma \overset{+}{\rightsquigarrow} \mathbf{\Gamma} \\
\kappa^+ &\overset{\text{def}}{\equiv} \kappa \ \text{ where } \kappa \vdash\ : \square \overset{+}{\rightsquigarrow} \kappa \\
A^+ &\overset{\text{def}}{\equiv} \mathbf{A} \ \text{ where } \Gamma \vdash A : * \overset{+}{\rightsquigarrow} \mathbf{A} \\
e^+ &\overset{\text{def}}{\equiv} \mathbf{e} \ \text{ where } \Gamma \vdash e : A \overset{+}{\rightsquigarrow} \mathbf{e} \\
e^\div &\overset{\text{def}}{\equiv} \mathbf{e} \ \text{ where } \Gamma \vdash e : A[\alpha, \beta] \overset{\div}{\rightsquigarrow} \mathbf{e}
\end{aligned}$$

Now, let us walk through the CPS translation. The translation of typing environments (Figure 4.9) is a mapping of the value translation $^+$. This reflects the fact that Dellina- is a call-by-value language, where variables are pure values. Since pure terms are never be applied the computation translation, their type are never doubly negated either.

$$\frac{}{\vdash \bullet}\ (\textsc{G-Empty}) \overset{+}{\rightsquigarrow} \bullet$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash A : *}{\vdash \Gamma, x : A}\ (\textsc{G-Ext}) \overset{+}{\rightsquigarrow} \Gamma^+, \mathbf{x} : A^+$$

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \square}\ (\textsc{K-Star}) \overset{+}{\rightsquigarrow} *$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Unit} : *}\ (\textsc{T-Unit}) \overset{+}{\rightsquigarrow} \mathbf{Unit}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : *}\ (\textsc{T-Nat}) \overset{+}{\rightsquigarrow} \mathbb{N}$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{L}\ e : *}\ (\textsc{T-List}) \overset{+}{\rightsquigarrow} \mathbf{L}\ e^+$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \quad \Gamma, x : A \vdash \rho}{\Gamma \vdash \Pi x : A.\, B : *}\ (\textsc{T-Pi})$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\, x} : A^+.\, B^+ \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\, x} : A^+.\, (B^+ \to \alpha^+) \to \beta^+ \text{ if } \rho = [\alpha, \beta]$$

Figure 4.9: CPS Translation of Environments, Kinds, and Types

Next, we look at the translation of kinds and types. The translation of (K-Star), (T-Unit), and (T-Nat) is trivial. The source list type $\mathsf{L}\ \mathsf{e}$ is converted into the target list type $\mathbf{L}\ \mathsf{e}^+$. We find that the index is applied the value translation $^+$. The applicability of $^+$ is guaranteed by the source rule (T-List), which has a premise $\Gamma \vdash \mathsf{e} : \mathbb{N}$ requiring e to be a pure natural number.

Function types are translated differently depending on whether the effect annotation ρ is empty or not. When ρ is empty, *i.e.*, when the function type takes the form $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, we convert the type to $\mathbf{\Pi\,x} : \mathsf{A}^+.\,\mathsf{B}^+$. The use of $^+$ on the domain A comes from the call-by-value semantics of Dellina-. We use the same $^+$-translation for co-domain B as well, because the type is inhabited by functions having a pure body. Since we translate pure terms into direct style, the post-translation term must have a non-negated type. In contrast, when ρ is non-empty, *i.e.*, when the type takes the form $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}[\alpha, \beta]$, we convert it to $\mathbf{\Pi\,x} : \mathsf{A}^+.\,(\mathsf{B}^+ \to \alpha^+) \to \beta^{+}$[2]. While the domain is translated the same way as before, the co-domain has two additional arrows. These arrows reflect the fact that the source type is inhabited by functions with an impure body, which must be applied the computation translation $^\div$. Since $^\div$ introduces a new λ for receiving a continuation, the resulting term has a doubly negated type $(\mathsf{B}^+ \to \alpha^+) \to \beta^+$.

As stated earlier, terms are translated into two distinct forms: when e is a pure term, we translate it into a direct-style term using $^+$, and when e is impure, we convert it into CPS using $^\div$. An implication of this design strategy is that values are uniformly applied the $^+$-translation. We may, however, use the $\div$-translation when we recurse on the body of functions (see the second case of (E-Abs) and (E-Rec)). This brings the two arrows into the CPS image of impure function types.

It is worth spending some time comparing different ways of translating an application $\mathsf{e}_0\ \mathsf{e}_1$ (Figure 4.11). We have eight variants in total, but all of these are defined in a systematic way. The overall structure is a direct-style application if

[2]As we can see from this type, our translation forces impure functions to take in their argument first and a continuation later. This translation is called Plotkin-style [60]. A different option would be to adopt a Fischer-style translation, where the continuation comes before the argument. When using a fixed answer type, the latter turns an arrow type into $(B^+ \to \bot) \to (A^+ \to \bot)$, which is sometimes preferred for its symmetric structure [162]. However, Fischer-style translations do not work for dependent calculi, because the co-domain B may refer to the argument variable of type A.

$$\frac{\vdash \Gamma \quad \mathsf{x} : \mathsf{A} \in \Gamma \text{ or } \mathsf{x} = \mathsf{e} : \mathsf{A} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{A}} \text{ (E-VAR)} \overset{+}{\rightsquigarrow} \mathbf{x}$$

$$\frac{\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho} \text{ (E-ABS)}$$

$$\overset{+}{\rightsquigarrow} \lambda \mathbf{x} : \mathsf{A}^{+}.\, \mathsf{e}^{+} \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \lambda \mathbf{x} : \mathsf{A}^{+}.\, \mathsf{e}^{\div} \text{ if } \rho = [\alpha, \beta]$$

$$\frac{\Gamma, \mathsf{f} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B} \quad \text{guard}(\mathsf{f}, \mathsf{x}, \mathsf{e}, \{\})}{\Gamma \vdash \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho}\, \mathsf{x}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho} \text{ (E-REC)}$$

$$\overset{+}{\rightsquigarrow} \mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi x} : \mathsf{A}^{+}.\, \mathsf{B}^{+}}\ \mathbf{x}.\, \mathsf{e}^{+} \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \mathbf{rec}\ \mathbf{f}_{\mathbf{\Pi x} : \mathsf{A}^{+}.\, (\mathsf{B}^{+} \to \alpha^{+}) \to \beta^{+}}\ \mathbf{x}.\, \mathsf{e}^{\div} \text{ if } \rho = [\alpha, \beta]$$

Figure 4.10: CPS Translation of Terms (Values)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_0 : (\Pi\,\mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \tau)\ \rho \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{A} \\ \Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : * \qquad \nu = \mathrm{comp}(\rho, \tau[\mathsf{e}_1/\mathsf{x}]) \qquad \Gamma \vdash \nu\end{array}}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}]\ \nu}\ \text{(E-DApp)}$$

$\overset{+}{\rightsquigarrow}$ $\mathsf{e}_0{}^+\ \mathsf{e}_1{}^+$
if $\rho = \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e}_1/\mathsf{x}])^+ \to (\alpha[\mathsf{e}_1/\mathsf{x}])^+.\ \mathsf{e}_0{}^+\ \mathsf{e}_1{}^+\ \mathbf{k}$
if $\rho = \epsilon, \tau = [\alpha, \beta]$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e}_1/\mathsf{x}])^+ \to \alpha^+.\ \mathsf{e}_0{}^\div\ (\lambda\,\mathbf{v_0} : \mathbf{\Pi\,x} : \mathsf{A}^+.\ \mathsf{B}^+.\ \mathbf{k}\ (\mathbf{v_0}\ \mathsf{e}_1{}^+))$
if $\rho = [\alpha, \beta], \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e}_1/\mathsf{x}])^+ \to (\alpha[\mathsf{e}_1/\mathsf{x}])^+.\ \mathsf{e}_0{}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \to (\mathsf{B}^+ \to \alpha^+) \to \beta^+.\ \mathbf{v_0}\ \mathsf{e}_1{}^+\ \mathbf{k})$
if $\rho = [\beta[\mathsf{e}_1/\mathsf{x}], \gamma], \tau = [\alpha, \beta]$

$$\frac{\Gamma \vdash \mathsf{e}_0 : (\mathsf{A} \to \mathsf{B}\ \tau)\ \rho \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{A}\ \sigma \qquad \nu = \mathrm{comp}(\rho, \sigma, \tau)}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}\ \nu}\ \text{(E-NDApp)}$$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : \mathsf{B}^+ \to \alpha^+.\ \mathsf{e}_1{}^\div\ (\lambda\,\mathbf{v_1} : \mathsf{A}^+.\ \mathbf{k}\ (\mathsf{e}_0{}^+\ \mathbf{v_1}))$
if $\rho = \epsilon, \sigma = [\alpha, \beta], \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : \mathsf{B}^+ \to \alpha^+.\ \mathsf{e}_1{}^\div\ (\lambda\,\mathbf{v_1} : \mathsf{A}^+.\ \mathsf{e}_0{}^+\ \mathbf{v_1}\ \mathbf{k})$
if $\rho = \epsilon, \sigma = [\beta, \gamma], \tau = [\alpha, \beta]$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : \mathsf{B}^+ \to \alpha^+.\ \mathsf{e}_0{}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \to \mathsf{B}^+.\ \mathsf{e}_1{}^+\ (\lambda\,\mathbf{v_1} : \mathsf{A}^+.\ \mathbf{k}\ (\mathbf{v_0}\ \mathbf{v_1})))$
if $\rho = [\beta, \gamma], \sigma = [\alpha, \beta], \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : \mathsf{B}^+ \to \alpha^+.\ \mathsf{e}_0{}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \to (\mathsf{B}^+ \to \alpha^+) \to \beta^+.\ \mathsf{e}_1{}^\div\ (\lambda\,\mathbf{v_1} : \mathsf{A}^+.\ \mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k}))$
if $\rho = [\gamma, \delta], \sigma = [\beta, \gamma], \tau = [\alpha, \beta]$

Figure 4.11: CPS Translation of Terms (Application)

$\mathsf{e_0}\ \mathsf{e_1}$ is pure as a whole; otherwise the CPS image takes the form $\lambda\,\mathbf{k}.\,\mathbf{e}$. When recursing on the subterm $\mathsf{e_i}$ (where $i = 0, 1$), we use $^+$ if it is pure and $^\div$ if it is impure. In the latter case, we supply $\mathsf{e_i}^\div$ with a continuation $\lambda\,\mathbf{v_i}.\,\mathbf{e}$. The presence of this continuation corresponds to the need of $\mathsf{e_i}$ for a delimited context—remember that evaluation of impure terms is only possible when they are surrounded by a context and a `reset`.

Turning our attention to the use of the top-level continuation $\mathbf{k}$, we find another shared pattern: when the function's body is pure (*i.e.*, when $\tau = \epsilon$), the last application takes the form $\mathbf{k}\ (\mathbf{e_0}\ \mathbf{e_1})$, whereas when the body is impure (*i.e.*, when $\tau = [\alpha, \beta]$), the application looks like $\mathbf{e_0}\ \mathbf{e_1}\ \mathbf{k}$. The contrast is a natural consequence of our selective translation on functions: in the pure case, the body of $\mathbf{e_0}$ must be a direct-style term, which does not require a continuation, whereas in the impure case, the body is a CPS computation, which does require a continuation.

Inductive data and pattern matching (Figures 4.12 – 4.14) are defined along the same lines as application. The all-pure instances of the typing rules are translated to a direct-style term, just like a pure application. Other instances are converted into a CPS computation $\lambda\,\mathbf{k}.\,\mathbf{e}$, and in the pattern matching cases, we again see two different uses of the continuation $\mathbf{k}$. Recall the semantics of a pattern matching: it ultimately evaluates to one of the brances $\mathsf{e_i}$ ($i = 1, 2$). This means, we have to find an appropriate way to compose the translation of $\mathsf{e_i}$ and $\mathbf{k}$. Similarly to the application cases, the right use of $\mathbf{k}$ is $\mathbf{k}\ \mathsf{e_i}^+$ when $\mathsf{e_i}$ is pure, and $\mathsf{e_i}^\div\ \mathbf{k}$ when $\mathsf{e_i}$ is impure. Note that this composition has the effect of changing the return type from P to α^+ or β^+, which represents the type of the *answer* we eventually obtain.

The translation of control operators (Figure 4.15) exhibits their semantics in a clear and intuitive way. Since a `shift` construct is impure regardless of its body, the only applicable translation is the computation translation. Observe that, when the body e is impure, we apply $\mathsf{e}^\div$ to the identity function $\lambda\,\mathbf{v} : \mathsf{B}^+.\,\mathbf{v}$. This corresponds to the outer `reset` in the reduct of a `shift`-redex. The `shift`-bound continuation k is turned into a λ-bound continuation $\mathbf{k}$, which has a function type whose codomain is not doubly negated. This reflects the fact that continuations are pure functions, which is guaranteed by the inner `reset` in the reduct of a `shift`-redex.

By contrast, a `reset` construct is always a pure term, hence the only applicable translation is the value translation. Similarly to `shift`, we apply $\mathsf{e}^\div$ to the identity

$$\frac{\vdash \Gamma}{\Gamma \vdash () : \mathsf{Unit}}\ (\text{E-Unit}) \overset{+}{\leadsto} ()$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ (\text{E-Zero}) \overset{+}{\leadsto} \mathbf{z}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho}{\Gamma \vdash \mathsf{suc}\ \mathsf{e} : \mathbb{N}\ \rho}\ (\text{E-Suc})$$

$\overset{+}{\leadsto}$ $\mathbf{suc}\ \mathsf{e}^{+}$ if $\rho = \epsilon$

$\overset{\div}{\leadsto}$ $\lambda\,\mathbf{k} : \mathbb{N} \to \alpha^{+}.\ \mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathbb{N}.\ \mathbf{k}\ (\mathbf{suc}\ \mathbf{v}))$ if $\rho = [\alpha, \beta]$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nil} : \mathsf{L}\ \mathsf{z}}\ (\text{E-Nil}) \overset{+}{\leadsto} \mathbf{nil}$$

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathbb{N} \qquad \Gamma \vdash \mathsf{e}_1 : \mathbb{N}\ \rho \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{L}\ \mathsf{e}_0\ \sigma \qquad \tau = \mathrm{comp}(\rho, \sigma)}{\Gamma \vdash\ ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 : (\mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0))\ \tau}\ (\text{E-Cons})$$

$\overset{+}{\leadsto}$ $::\ \mathsf{e}_0{}^{+}\ \mathsf{e}_1{}^{+}\ \mathsf{e}_2{}^{+}$
if $\rho = \sigma = \epsilon$

$\overset{\div}{\leadsto}$ $\lambda\,\mathbf{k} : \mathbf{L}\ \mathsf{e}_0{}^{+} \to \alpha^{+}.\ \mathsf{e}_2{}^{\div}\ (\lambda\,\mathbf{v_2} : \mathbf{L}\ \mathsf{e}_0{}^{+}.\ \mathbf{k}\ (::\ \mathsf{e}_0{}^{+}\ \mathsf{e}_1{}^{+}\ \mathbf{v_2}))$
if $\rho = \epsilon, \sigma = [\alpha, \beta]$

$\overset{\div}{\leadsto}$ $\lambda\,\mathbf{k} : \mathbf{L}\ \mathsf{e}_0{}^{+} \to \alpha^{+}.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : \mathbb{N}.\ \mathbf{k}\ (::\ \mathsf{e}_0{}^{+}\ \mathbf{v_1}\ \mathsf{e}_2{}^{+}))$
if $\rho = [\alpha, \beta], \sigma = \epsilon$

$\overset{\div}{\leadsto}$ $\lambda\,\mathbf{k} : \mathbf{L}\ \mathsf{e}_0{}^{+} \to \alpha^{+}.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : \mathbb{N}.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : \mathbb{N}.\ \mathbf{k}\ (::\ \mathsf{e}_0{}^{+}\ \mathbf{v_1}\ \mathbf{v_0})))$
if $\rho = [\beta, \gamma], \sigma = [\alpha, \beta]$

Figure 4.12: CPS Translation of Terms (Inductive Data)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N} \quad \Gamma, \mathsf{x} : \mathbb{N} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{x}] \quad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ n}/\mathsf{x}]\ \rho[\mathsf{suc\ n}/\mathsf{x}] \\ \Gamma \vdash \mathsf{P}[\mathsf{e}/\mathsf{x}] : * \quad \Gamma \vdash \rho[\mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}[\mathsf{e}/\mathsf{x}]\ \rho[\mathsf{e}/\mathsf{x}]} \text{(E-DMatchN)}$$

$$\overset{+}{\rightsquigarrow} \mathbf{pm}\ \mathsf{e}^+ \ \mathbf{as\ x\ in}\ \mathbb{N}\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with\ z} \rightarrow \mathsf{e_1}^+ \mid \mathbf{suc\ n} \rightarrow \mathsf{e_2}^+$$
$$\text{if } \rho = \epsilon$$

$$\overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : (\mathsf{P}[\mathsf{e}/\mathsf{x}])^+ \rightarrow (\alpha[\mathsf{e}/\mathsf{x}])^+.$$
$$\mathbf{pm}\ \mathsf{e}^+\ \mathbf{as\ x\ in}\ \mathbb{N}\ \mathbf{ret}\ \beta^+\ \mathbf{with\ z} \rightarrow \mathsf{e_1}^{\div}\ \mathbf{k} \mid \mathbf{suc\ n} \rightarrow \mathsf{e_2}^{\div}\ \mathbf{k}$$
$$\text{if } \rho = [\alpha, \beta]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho \quad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \quad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ \_\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}\ \tau} \text{(E-NDMatchN)}$$

$$\overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{P}^+ \rightarrow \alpha^+.$$
$$\mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathbb{N}.\,\mathbf{pm\ v\ as\ \_\ in}\ \mathbb{N}\ \mathbf{ret}\ \alpha^+\ \mathbf{with\ z} \rightarrow \mathbf{k}\ \mathsf{e_1}^+ \mid \mathbf{suc\ n} \rightarrow \mathbf{k}\ \mathsf{e_2}^+)$$
$$\text{if } \rho = [\alpha, \beta], \sigma = \epsilon$$

$$\overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{P}^+ \rightarrow \alpha^+.$$
$$\mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathbb{N}.\,\mathbf{pm\ v\ as\ \_\ in}\ \mathbb{N}\ \mathbf{ret}\ \beta^+\ \mathbf{with\ z} \rightarrow \mathsf{e_1}^{\div}\ \mathbf{k} \mid \mathbf{suc\ n} \rightarrow \mathsf{e_2}^{\div}\ \mathbf{k})$$
$$\text{if } \rho = [\beta, \gamma], \sigma = [\alpha, \beta]$$

Figure 4.13: CPS Translation of Terms (Pattern Matching on Natural Numbers)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{L}\ \mathsf{n} \quad \Gamma, \mathsf{a} : \mathbb{N}, \mathsf{x} : \mathsf{L}\ \mathsf{a} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}] \\ \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc}\ \mathsf{m}/\mathsf{a}, ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t}/\mathsf{x}]\ \rho[\mathsf{suc}\ \mathsf{m}/\mathsf{a}, ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t}/\mathsf{x}] \\ \Gamma \vdash \mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}] : * \quad \Gamma \vdash \rho[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]\ \rho[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]}\ \text{(E-DMATCHL)}$$

$$\overset{+}{\rightsquigarrow}\ \mathbf{pm}\ \mathsf{e}^{+}\ \mathbf{as\ x\ in\ L\ a\ ret}\ \mathsf{P}^{+}\ \mathbf{with\ nil} \to \mathsf{e_1}^{+} \mid ::\ \mathbf{m\ h\ t} \to \mathsf{e_2}^{+}$$
$$\text{if } \rho = \epsilon$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\,\mathbf{k} : (\mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}])^{+} \to (\alpha[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}])^{+}.$$
$$\mathbf{pm}\ \mathsf{e}^{+}\ \mathbf{as\ x\ in\ L\ a\ ret}\ \beta^{+}\ \mathbf{with\ nil} \to \mathsf{e_1}^{\div}\ \mathbf{k} \mid ::\ \mathbf{m\ h\ t} \to \mathsf{e_2}^{\div}\ \mathbf{k}$$
$$\text{if } \rho = [\alpha, \beta]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : (\mathsf{L}\ \mathsf{n})\ \rho \quad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \quad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{P}\ \tau}\ \text{(E-NDMATCHL)}$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\,\mathbf{k} : \mathsf{P}^{+} \to \alpha^{+}.$$
$$\mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathbb{N}.\ \mathbf{pm\ v\ as}\ \_\ \mathbf{in\ L}\ \_\ \mathbf{ret}\ \alpha^{+}\ \mathbf{with\ nil} \to \mathbf{k}\ \mathsf{e_1}^{+} \mid ::\ \mathbf{m\ h\ t} \to \mathbf{k}\ \mathsf{e_2}^{+})$$
$$\text{if } \rho = [\alpha, \beta], \sigma = \epsilon$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\,\mathbf{k} : \mathsf{P}^{+} \to \alpha^{+}.$$
$$\mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathbb{N}.\ \mathbf{pm\ v\ as}\ \_\ \mathbf{in\ L}\ \_\ \mathbf{ret}\ \beta^{+}\ \mathbf{with\ nil} \to \mathsf{e_1}^{\div}\ \mathbf{k} \mid ::\ \mathbf{m\ h\ t} \to \mathsf{e_2}^{\div}\ \mathbf{k})$$
$$\text{if } \rho = [\beta, \gamma], \sigma = [\alpha, \beta]$$

Figure 4.14: CPS Translation of Terms (Pattern Matching on Lists)

$$\frac{\begin{array}{c}\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \beta \quad \text{or} \quad \Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta] \\ \Gamma \vdash \beta : * \end{array}}{\Gamma \vdash \alpha : \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha .\, \mathsf{e}[\mathsf{A}, \beta]} \ (\text{E-Shift})$$

$$\overset{\div}{\rightsquigarrow} \ \lambda\, \mathbf{k} : \mathsf{A}^{+} \to \alpha^{+} .\, \mathsf{e}^{+} \text{ if } \mathsf{e} \text{ pure}$$

$$\overset{\div}{\rightsquigarrow} \ \lambda\, \mathbf{k} : \mathsf{A}^{+} \to \alpha^{+} .\, \mathsf{e}^{+} \ (\lambda\, \mathbf{v} : \mathsf{B}^{+} .\, \mathbf{v}) \text{ if } \mathsf{e} \text{ impure}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \quad \text{or} \quad \Gamma \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \mathsf{A}]}{\Gamma \vdash \langle \mathsf{e} \rangle : \mathsf{A}} \ (\text{E-Reset})$$

$$\overset{+}{\rightsquigarrow} \ \mathsf{e}^{+} \text{ if } \mathsf{e} \text{ pure}$$

$$\overset{+}{\rightsquigarrow} \ \mathsf{e}^{\div} \ (\lambda\, \mathbf{v} : \mathsf{B}^{+} .\, \mathbf{v}) \text{ if } \mathsf{e} \text{ impure}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \qquad \Gamma \vdash \mathsf{B} : * \qquad \Gamma \vdash \sigma \\ \mathsf{A} \equiv \mathsf{B} \qquad \rho \equiv \sigma \end{array}}{\Gamma \vdash \mathsf{e} : \mathsf{B}\ \sigma} \ (\text{E-Conv})$$

$$\overset{+}{\rightsquigarrow} \ \mathsf{e}^{+} \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \ \mathsf{e}^{\div} \text{ if } \rho = [\alpha, \beta]$$

Figure 4.15: CPS Translation of Terms (Control Opertaors and Conversion)

function when the body e is impure, which corresponds to running e in an empty context.

Lastly, we have a very simple translation for the conversion rule. The CPS image is exactly the same as the translation of the pre-conversion derivation, since type conversion has no computational impact.

Remark on Order of Evaluation As we noted earlier, selective CPS translations do not fix the order of evaluation by themselves. This is the case in our translation as well. Consider the translation of an application $\mathsf{e}_0\ \mathsf{e}_1$, where e_0 and e_1 are both pure, but the body of e_0 is impure. In the CPS image, we will have an application $\mathbf{k}\ (\mathsf{e}_0{}^{+}\ \mathsf{e}_1{}^{+})$, where $\mathbf{k}$ is called in a non-tail position. Furthermore, our translation does not preserve the left-to-right evaluation of the source language. For instance, if we have an impure application $\mathsf{e}_0\ \mathsf{e}_1$, where e_0 is pure but e_1 is impure, we would obtain a CPS computation of the form $\lambda\,\mathbf{k}.\,\mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v}_1.\,\mathbf{e})$, which would force evaluation of $\mathsf{e}_1{}^{\div}$ to happen before evaluation of $\mathsf{e}_0{}^{+}$ in $\mathbf{e}$.

4.6 Proof of Type Preservation

In this section, we prove that our selective CPS translation is type-preserving, that is, well-typed Dellina- programs are translated to well-typed target programs. The proof is staged as follows. First, we prove *compositionality* (Lemma 4.6.1), which states that the translation commutes with substitution. Using compositionality, we next prove *correctness* (Lemma 4.6.3), which tells us that two source expressions related by reduction are mapped to two equivalent target expressions. Correctness then gives us *computational soundness* (Lemma 4.6.4) [30], which says that equivalent terms are mapped to equivalent terms. These allow us to prove the main theorem: the translation preserves typing (Theorem 4.6.1).

The reason we cannot directly prove type preservation is that our translation is defined on the derivation, not the syntax. This means we have to deal with the type conversion case. Suppose we have a pure term e whose type has converted from A to B. In the type preservation proof, we have to show $\mathsf{e} : \mathsf{B}^{+}$, whereas the induction hypothesis gives us $\mathsf{e} : \mathsf{A}^{+}$. For this case to go through, we need computational soundness, that is, whenever we have $\mathsf{A} \equiv \mathsf{B}$ in the source, we have

$A^+ \equiv B^+$ in the target. As pointed out by Barthe et al. [23], this staging is not possible when we use a typed equivalence, because that would require us to show $A^+ : *$ and $B^+ : *$, which is exactly the type preservation statement! This is why we use an untyped equivalence in the source and target languages. While typed and untyped variants of equivalence are suited for different purposes, they have proven equivalent in Pure Type Systems [81].

4.6.1 Compositionality

We begin by proving compositionality: we can either first perform substitution and then translate the result, or first translate the expression and then perform substitution.

Lemma 4.6.1 (Compositionality)**.** *Suppose* e' *is a pure term. Then, the following equalities hold:*

1. $(A[e'/x'])^+ = A^+[e'^+/\mathbf{x}']$
2. $(e[e'/x'])^+ = e^+[e'^+/\mathbf{x}']$
3. $(e[e'/x'])^\div = e^\div[e'^+/\mathbf{x}']$

Proof. The proof is by mutual induction on the derivation of A and e. We show some representative cases.

Case 1: (T-Unit), (T-Nat)
These cases are trivial.

Case 2: (T-List)

$$\begin{aligned}
((\mathsf{L}\ e_1)[e'/x'])^+ &= (\mathsf{L}\ (e_1[e'/x']))^+ && \text{by substitution}\\
&= \mathbf{L}\ (e_1[e'/x'])^+ && \text{by translation}\\
&= \mathbf{L}\ ({e_1}^+[e'^+/\mathbf{x}']) && \text{by IH on } e_1\\
&= (\mathbf{L}\ {e_1}^+)[e'^+/\mathbf{x}'] && \text{by substitution}
\end{aligned}$$

Case 3: (T-Pi) where $\rho = \epsilon$

$$\begin{aligned}
((\Pi\, x : A.\, B)[e'/x'])^{+} &= (\Pi\, x : A[e'/x'].\, B[e'/x'])^{+} && \text{by substitution}\\
&= \mathbf{\Pi}\, \mathbf{x} : (A[e'/x'])^{+}.\, (B[e'/x'])^{+} && \text{by translation}\\
&= \mathbf{\Pi}\, \mathbf{x} : A^{+}[e'^{+}/\mathbf{x}'].\, B^{+}[e'^{+}/\mathbf{x}'] && \text{by IH on } A \text{ and } B\\
&= (\mathbf{\Pi}\, \mathbf{x} : A^{+}.\, B^{+})[e'^{+}/\mathbf{x}'] && \text{by substitution}
\end{aligned}$$

Case 4: (E-Var)

Sub-Case 1: $e = x'$

$$\begin{aligned}
(x'[e'/x'])^{+} &= e'^{+} && \text{by substitution}\\
&= \mathbf{x}'[e'^{+}/\mathbf{x}'] && \text{by substitution}\\
&= x'^{+}[e'^{+}/\mathbf{x}'] && \text{by translation}
\end{aligned}$$

Sub-Case 2: $e = y$ where $y \neq x'$

$$\begin{aligned}
(y[e'/x'])^{+} &= y^{+} && \text{by substitution}\\
&= \mathbf{y} && \text{by translation}\\
&= \mathbf{y}[e'^{+}/\mathbf{x}'] && \text{by substitution}
\end{aligned}$$

Case 5: (E-Abs) where $\rho = [\alpha, \beta]$

$$\begin{aligned}
((\lambda\, x : A.\, e)[e'/x'])^{+} &= (\lambda\, x : A[e'/x'].\, e[e'/x'])^{+} && \text{by substitution}\\
&= \lambda\, \mathbf{x} : (A[e'/x'])^{+}.\, (e[e'/x'])^{+} && \text{by translation}\\
&= \lambda\, \mathbf{x} : A^{+}[e'^{+}/\mathbf{x}'].\, e^{+}[e'^{+}/\mathbf{x}'] && \text{by IH on } A \text{ and } e\\
&= (\lambda\, \mathbf{x} : A^{+}.\, e^{+})[e'^{+}/\mathbf{x}'] && \text{by substitution}
\end{aligned}$$

Case 6: (E-DApp) where $\rho = [\beta[e_1/x], \gamma], \tau = [\alpha, \beta]$

$$\begin{aligned}
((\mathsf{e}_0\ \mathsf{e}_1)[\mathsf{e}'/\mathsf{x}'])^{\div} &= (\mathsf{e}_0[\mathsf{e}'/\mathsf{x}']\ \mathsf{e}_1[\mathsf{e}'/\mathsf{x}'])^{\div} && \text{by substitution}\\
&= \lambda\,\mathbf{k}.\,(\mathsf{e}_0[\mathsf{e}'/\mathsf{x}'])^{\div}\ (\lambda\,\mathbf{v_0}.\,\mathbf{v_0}\ (\mathsf{e}_1[\mathsf{e}'/\mathsf{x}'])^{+}\ \mathbf{k}) && \text{by translation}\\
&= \lambda\,\mathbf{k}.\,\mathsf{e}_0{}^{\div}[\mathsf{e}'^{+}/\mathbf{x}']\ (\lambda\,\mathbf{v_0}.\,\mathbf{v_0}\ \mathsf{e}_1{}^{+}[\mathsf{e}'^{+}/\mathbf{x}']\ \mathbf{k}) && \text{by IH on } \mathsf{e}_0 \text{ and } \mathsf{e}_1\\
&= (\lambda\,\mathbf{k}.\,\mathsf{e}_0{}^{\div}\ (\lambda\,\mathbf{v_0}.\,\mathbf{v_0}\ \mathsf{e}_1{}^{+}\ \mathbf{k}))[\mathsf{e}'^{+}/\mathbf{x}'] && \text{by substitution}
\end{aligned}$$

□

Remark In a polymorphic answer type translation, the compositionality property is stated as follows:

$$(\mathsf{e}[\mathsf{e}'/\mathsf{x}'])^{\div} \equiv \mathsf{e}^{\div}[\mathsf{e}'^{\div}\ \mathsf{A}'^{+}\ \mathbf{id}_{\mathsf{A}'^{+}}/\mathbf{x}']$$

While our compositionality lemma follows directly by substitution, proving the above statement is much more involved; in particular, it requires the [≡-CONT] rule of Bowman et al. [34], as shown below:

$$\begin{aligned}
(\mathsf{x}'[\mathsf{e}'/\mathsf{x}'])^{\div} &= \mathsf{e}'^{\div} && \text{by substitution}\\
&\equiv \lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathsf{e}'^{\div}\ @\ \alpha\ (\lambda\,\mathbf{x}.\,\mathbf{k}\ \mathbf{x}) && \text{by } \eta \text{ and semantics of } @\\
&\equiv \lambda\,\alpha.\,\lambda\,\mathbf{k}.\,(\lambda\,\mathbf{x}.\,\mathbf{k}\ \mathbf{x})\ (\mathsf{e}'^{\div}\ \mathsf{A}'^{+}\ \mathbf{id}_{\mathsf{A}'^{+}}) && \text{by [}\equiv\text{-CONT]}\\
&= (\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,(\lambda\,\mathbf{x}.\,\mathbf{k}\ \mathbf{x})\ \mathbf{x}')[\mathsf{e}'^{\div}\ \mathsf{A}'^{+}\ \mathbf{id}_{\mathsf{A}'^{+}}/\mathbf{x}'] && \text{by substitution}\\
&\triangleright_{\beta} (\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{k}\ \mathbf{x}')[\mathsf{e}'^{\div}\ \mathsf{A}'^{+}\ \mathbf{id}_{\mathsf{A}'^{+}}/\mathbf{x}'] &&\\
&= \mathsf{x}'^{\div}[\mathsf{e}'^{\div}\ \mathsf{A}'^{+}\ \mathbf{id}_{\mathsf{A}'^{+}}/\mathbf{x}'] && \text{by translation}
\end{aligned}$$

4.6.2 Correctness and Computational Soundness

Lemma 4.6.2 (Correctness w.r.t. Single-step Reduction)**.**

1. *If* $\Gamma \vdash \mathsf{A} : *$ *and* $\mathsf{A}\ \triangleright_p\ \mathsf{A}'$, *then* $\mathsf{A}^{+} \equiv \mathsf{A}'^{+}$.
2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $\mathsf{e}\ \triangleright_p\ \mathsf{e}'$, *then* $\mathsf{e}^{+} \equiv \mathsf{e}'^{+}$.
3. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]$ *and* $\mathsf{e}\ \triangleright_p\ \mathsf{e}'$, *then* $\mathsf{e}^{\div} \equiv \mathsf{e}'^{\div}$.

Proof. The proof is by mutual induction on the derivation of A and e. For all typing rules, the reflexive case ($\mathsf{t} \rhd_p \mathsf{t}$) is trivial, so we only show the cases where t actually takes step.

Case 1: (T-List)
The only way the list type $\mathsf{L}\ \mathsf{e}$ takes step is via (P-List).

$$\begin{aligned}
(\mathsf{L}\ \mathsf{e})^+ &= \mathbf{L}\ \mathsf{e}^+ && \text{by translation}\\
&\equiv \mathbf{L}\ \mathsf{e}'^+ && \text{by IH on } \mathsf{e}\\
&= (\mathsf{L}\ \mathsf{e}')^+ && \text{by translation}
\end{aligned}$$

Case 2: (T-Pi)
This case can be easily proven using induction hypothesis on the subcomponents.

Case 3: (E-Var)

Sub-Case 1: $\mathsf{x}' \rhd_p \mathsf{v}'$ by (P-VarDelta)

$$\begin{aligned}
\mathsf{x}^+ &= \mathbf{x} && \text{by translation}\\
&\rhd_\delta \mathsf{v}^+ && \text{by } \mathsf{v}^+ \in \Gamma^+
\end{aligned}$$

Case 4: (E-DApp) where $\rho = \epsilon, \tau = [\alpha, \beta]$

Sub-Case 1: $\mathsf{e}_0\ \mathsf{e}_1 \rhd_p \mathsf{e}_0'\ \mathsf{e}_1'$ by (P-App)

$$\begin{aligned}
(\mathsf{e}_0\ \mathsf{e}_1)^\div &= \lambda\,\mathbf{k}.\, \mathsf{e}_0{}^+\ \mathsf{e}_1{}^+\ \mathbf{k} && \text{by translation}\\
&\equiv \lambda\,\mathbf{k}.\, \mathsf{e}_0'{}^+\ \mathsf{e}_1'{}^+\ \mathbf{k} && \text{by IH on } \mathsf{e}_0 \text{ and } \mathsf{e}_1\\
&= (\mathsf{e}_0'\ \mathsf{e}_1')^\div && \text{by translation}
\end{aligned}$$

Sub-Case 2: $(\lambda\,\mathsf{x}.\,\mathsf{e}_0)\ \mathsf{v}_1 \rhd_p \mathsf{e}_0'[\mathsf{v}_1'/\mathsf{x}]$ by (P-AppBeta)

$$\begin{aligned}
((\lambda x.\, e_0)\ v_1)^+ &= \lambda \mathbf{k}.\, (\lambda x : A'.\, e_0)^+\ {v_1}^+\ \mathbf{k} && \text{by translation}\\
&= \lambda \mathbf{k}.\, (\lambda \mathbf{x}.\, {e_0}^{\div})\ {v_1}^+\ \mathbf{k} && \text{by translation}\\
&\equiv \lambda \mathbf{k}.\, (\lambda \mathbf{x}.\, {e'_0}^{\div})\ {v'_1}^+\ \mathbf{k} && \text{by IH on } e_0 \text{ and } v_1\\
&\rhd_\beta \lambda \mathbf{k}.\, ({e'_0}^{\div}[{v'_1}^+/\mathbf{x}])\ \mathbf{k} &&\\
&\equiv {e'_0}^{\div}[{v'_1}^+/\mathbf{x}] && \text{by } \eta\\
&= (e'_0[v'_1/x])^{\div} && \text{by compositionality}
\end{aligned}$$

Case 5: (E-Reset)

Sub-Case 1: $\langle e \rangle \ \rhd_p\ \langle e' \rangle$ by (P-Reset)

$$\begin{aligned}
\langle e \rangle^+ &= e^{\div}\ (\lambda \mathbf{v}.\, \mathbf{v}) && \text{by translation}\\
&\equiv e'^{\div}\ (\lambda \mathbf{v}.\, \mathbf{v}) && \text{by IH on } e\\
&= \langle e' \rangle^+ && \text{by translation}
\end{aligned}$$

Sub-Case 2: $\langle F[\mathcal{S}k.\, e] \rangle \ \rhd_p\ \langle e'[\lambda x.\, F'[x]/k] \rangle$ by (P-ResetS)

Similarly to the preservation theorem, we prove this case by decomposing the `shift`-reduction into smaller ones.

Sub-Sub-Case 1: (P-SApp1) where the function body, e, and e_1 are all pure

$$\begin{aligned}
((\mathcal{S}k.\, e)\ e_1)^{\div} &= \lambda \mathbf{k}'.\, (\mathcal{S}k.\, e)^{\div}\ (\lambda \mathbf{v_0}.\, \mathbf{k}'\ (\mathbf{v_0}\ {e_1}^+)) && \text{by translation}\\
&= \lambda \mathbf{k}'.\, (\lambda \mathbf{k}.\, e^+)\ (\lambda \mathbf{v_0}.\, \mathbf{k}'\ (\mathbf{v_0}\ {e_1}^+)) && \text{by translation}\\
&\equiv \lambda \mathbf{k}'.\, (\lambda \mathbf{k}.\, e'^+)\ (\lambda \mathbf{v_0}.\, \mathbf{k}'\ (\mathbf{v_0}\ {e'_1}^+)) && \text{by IH on } e \text{ and } e'_1\\
&\rhd_\beta \lambda \mathbf{k}'.\, e'^+[\lambda \mathbf{v_0}.\, \mathbf{k}'\ (\mathbf{v_0}\ {e'_1}^+)/\mathbf{k}] &&\\
&= \lambda \mathbf{k}'.\, (e'[\lambda v_0.\, \langle k'\ (v_0\ e'_1)\rangle/k])^+ && \text{by compositionality}\\
&= (\mathcal{S}k'.\, e'[\lambda v_0.\, \langle k'\ (v_0\ e'_1)\rangle/k])^{\div} && \text{by translation}
\end{aligned}$$

Sub-Sub-Case 2: (P-SApp2) where the function body and e are impure

$$\begin{aligned}
(\mathsf{v}_0\ (\mathcal{S}\mathsf{k}.\,\mathsf{e}))^{\div} &= \lambda\,\mathbf{k}'.\,(\mathcal{S}\mathsf{k}.\,\mathsf{e})^{\div}\ (\lambda\,\mathbf{v_1}.\,{\mathsf{v}_0}^{+}\ \mathbf{v_1}\ \mathbf{k}') && \text{by translation}\\
&= \lambda\,\mathbf{k}'.\,(\lambda\,\mathbf{k}.\,\mathsf{e}^{\div}\ \mathbf{id})\ (\lambda\,\mathbf{v_1}.\,{\mathsf{v}_0}^{+}\ \mathbf{v_1}\ \mathbf{k}') && \text{by translation}\\
&\equiv \lambda\,\mathbf{k}'.\,(\lambda\,\mathbf{k}.\,\mathsf{e}'^{\div}\ \mathbf{id})\ (\lambda\,\mathbf{v_1}.\,\mathsf{v}_0'^{+}\ \mathbf{v_1}\ \mathbf{k}') && \text{by IH on } \mathsf{e} \text{ and } \mathsf{v}_0\\
&\rhd_{\beta}\ \lambda\,\mathbf{k}'.\,(\mathsf{e}'^{\div}\ \mathbf{id})[\lambda\,\mathbf{v_1}.\,\mathsf{v}_0'^{+}\ \mathbf{v_1}\ \mathbf{k}'/\mathbf{k}]\\
&\lhd_{\beta}\ \lambda\,\mathbf{k}'.\,(\mathsf{e}'^{\div}\ \mathbf{id})[\lambda\,\mathbf{v_1}.\,(\mathsf{v}_0'\ \mathsf{v}_1)^{\div}\ \mathbf{k}'/\mathbf{k}]\\
&\equiv \lambda\,\mathbf{k}'.\,\mathsf{e}'^{\div}[\lambda\,\mathbf{v_1}.\,(\mathsf{v}_0'\ \mathsf{v}_1)^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{k}'\ \mathbf{v})/\mathbf{k}]\ \mathbf{id} && \text{by } \eta\\
&\lhd_{\beta_\Omega}\ \lambda\,\mathbf{k}'.\,\mathsf{e}'^{\div}[\lambda\,\mathbf{v_1}.\,(\mathsf{v}_0'\ \mathsf{v}_1)^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{id}\ (\mathbf{k}'\ \mathbf{v}))/\mathbf{k}]\ \mathbf{id}\\
&\lhd_{\beta}\ \lambda\,\mathbf{k}'.\,\mathsf{e}'^{\div}[\lambda\,\mathbf{v_1}.\,(\lambda\,\mathbf{k}''.\,(\mathsf{v}_0'\ \mathsf{v}_1)^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{k}''\ (\mathbf{k}'\ \mathbf{v})))\ \mathbf{id}/\mathbf{k}]\ \mathbf{id}\\
&= \lambda\,\mathbf{k}'.\,\mathsf{e}'^{\div}[\lambda\,\mathbf{v_1}.\,(\mathsf{k}'\ (\mathsf{v}_0'\ \mathsf{v}_1))^{\div}\ \mathbf{id}/\mathbf{k}]\ \mathbf{id} && \text{by translation}\\
&= \lambda\,\mathbf{k}'.\,\mathsf{e}'^{\div}[(\lambda\,\mathsf{v}_1.\,\langle\mathsf{k}'\ (\mathsf{v}_0'\ \mathsf{v}_1)\rangle)^{+}/\mathbf{k}]\ \mathbf{id} && \text{by translation}\\
&= \lambda\,\mathbf{k}'.\,(\mathsf{e}'[\lambda\,\mathsf{v}_1.\,\langle\mathsf{k}'\ (\mathsf{v}_0'\ \mathsf{v}_1)\rangle/\mathsf{k}])^{\div}\ \mathbf{id} && \text{by compositionality}\\
&= (\mathcal{S}\mathsf{k}'.\,\mathsf{e}'[\lambda\,\mathsf{v}_1.\,\langle\mathsf{k}'\ (\mathsf{v}_0'\ \mathsf{v}_1)\rangle/\mathsf{k}])^{\div} && \text{by translation}
\end{aligned}$$

Sub-Sub-Case 3: (P-SEmpty) where e impure

$$\begin{aligned}
\langle\mathcal{S}\mathsf{k}.\,\mathsf{e}\rangle^{+} &= (\mathcal{S}\mathsf{k}.\,\mathsf{e})^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by translation}\\
&= (\lambda\,\mathbf{k}.\,\mathsf{e}^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{v}))\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by translation}\\
&\equiv (\lambda\,\mathbf{k}.\,\mathsf{e}'^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{v}))\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by IH on } \mathsf{e}\\
&\rhd_{\beta}\ (\mathsf{e}'^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{v}))[\lambda\,\mathbf{v}.\,\mathbf{v}/\mathbf{k}]\\
&= (\mathsf{e}'^{\div}[\lambda\,\mathbf{v}.\,\mathbf{v}/\mathbf{k}])\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by substitution}\\
&= (\mathsf{e}'^{\div}[(\lambda\,\mathsf{v}.\,\mathsf{v})^{+}/\mathbf{k}])\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by translation}\\
&= (\mathsf{e}'[\lambda\,\mathsf{v}.\,\mathsf{v}/\mathsf{k}])^{\div}\ (\lambda\,\mathbf{v}.\,\mathbf{v}) && \text{by compositionality}\\
&= \langle\mathsf{e}'[\lambda\,\mathsf{v}.\,\mathsf{v}/\mathsf{k}]\rangle^{+} && \text{by translation}
\end{aligned}$$

Case 6: (P-ResetV)

$$\begin{aligned} \langle \mathsf{v} \rangle^{+} &= \mathsf{v}^{+} && \text{by translation} \\ &\equiv \mathsf{v}'^{+} && \text{by IH on } \mathsf{v} \end{aligned}$$

□

Lemma 4.6.3 (Correctness)**.**

1. *If* $\Gamma \vdash \mathsf{A} : *$ *and* $\mathsf{A} \ \triangleright^{\star} \ \mathsf{A}'$, *then* $\mathsf{A}^{+} \equiv \mathsf{A}'^{+}$.
2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $\mathsf{e} \ \triangleright^{\star} \ \mathsf{e}'$, *then* $\mathsf{e}^{+} \equiv \mathsf{e}'^{+}$.
3. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]$ *and* $\mathsf{e} \ \triangleright^{\star} \ \mathsf{e}'$, *then* $\mathsf{e}^{\div} \equiv \mathsf{e}'^{\div}$.

Proof. The proof is by induction on the length of the reduction sequence. The base case, where the length is zero, is trivial. The inductive case follows by the induction hypothesis, Lemma 4.6.3, and the transitivity of equivalence. □

Lemma 4.6.4 (Computational Soundness)**.**

1. *If* $\mathsf{A} \equiv \mathsf{A}'$, *then* $\mathsf{A}^{+} \equiv \mathsf{A}'^{+}$.
2. *If* $\mathsf{e} \equiv \mathsf{e}'$, *then* $\mathsf{e}^{+} \equiv \mathsf{e}'^{+}$.
3. *If* $\mathsf{e} \equiv \mathsf{e}'$, *then* $\mathsf{e}^{\div} \equiv \mathsf{e}'^{\div}$.

Proof. This is a direct consequence of the correctness lemma (Lemma 4.6.3). By the definition of equivalence, we know that, when we have $\mathsf{t}_1 \equiv \mathsf{t}_2$, there must be an expression t such that $\mathsf{t}_1 \ \triangleright^{\star} \ \mathsf{t}$ and $\mathsf{t}_2 \ \triangleright^{\star} \ \mathsf{t}$. By correctness of the translation, we know that ${\mathsf{t}_1}^{+} \equiv \mathsf{t}^{+}$ $({\mathsf{t}_1}^{\div} \equiv \mathsf{t}^{\div})$ and ${\mathsf{t}_2}^{+} \equiv \mathsf{t}^{+}$ $({\mathsf{t}_2}^{\div} \equiv \mathsf{t}^{\div})$. The goal follows by symmetry and transitivity of equivalence. □

4.6.3 Type Preservation

Theorem 4.6.1 (Type Preservation)**.**

1. *If* $\vdash \Gamma$, *then* $\vdash \Gamma^{+}$.

2. *If* $\Gamma \vdash \kappa : \square$, *then* $\Gamma^+ \vdash \kappa^+ : \square$.

3. *If* $\Gamma \vdash A : *$, *then* $\Gamma^+ \vdash A^+ : *$.

4. *If* $\Gamma \vdash e : A$, *then* $\Gamma^+ \vdash e^+ : A^+$.

5. *If* $\Gamma \vdash e : A[\alpha, \beta]$, *then* $\Gamma^+ \vdash e^{\div} : (A^+ \rightarrow \alpha^+) \rightarrow \beta^+$.

Proof. The proof is by mutual induction on the derivation of Γ, κ, A, and e. We show some representative cases.

Case 1: (G-Empty)
This case is trivial.

Case 2: (G-Ext)
Our goal is to show

$$\vdash \Gamma^+, \mathbf{x} : A^+$$

By the induction hypothesis, we have

$$\vdash \Gamma^+ \quad \text{and} \quad \Gamma^+ \vdash A^+ : *$$

The goal easily follows by [G-Ext].

Case 3: (K-Star), (T-Unit), (T-Nat), (E-Unit), (E-Zero), (E-Nil)
These cases easily follow by the induction hypothesis on Γ.

Case 4: (T-List)
Our goal is to show

$$\Gamma^+ \vdash \mathbf{L}\ e^+ : *$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash e^+ : \mathbb{N}$$

The goal immediately follows by [T-List].

Case 5: (T-Pi) where $\rho = \epsilon$
Our goal is to show

$$\Gamma^+ \vdash \mathbf{\Pi\, x} : A^+ . B^+ : *$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash A^+ : * \quad \text{and} \quad \Gamma^+, \mathbf{x} : A^+ \vdash B^+ : *$$

which imply the goal. The case where $\rho = [\alpha, \beta]$ can be shown in a similar way, with two more invocations of the induction hypothesis on the answer types α and β.

Case 6: (E-Var)
Our goal is to show

$$\Gamma^+ \vdash \mathbf{x} : A^+$$

By the induction hypothesis, we have

$$\vdash \Gamma^+$$

By the definition of the translation, we also know that the declaration $\mathbf{x} : A^+$ is in Γ^+. The goal now follows by [E-Var].

Case 7: (E-Abs) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\Gamma^+ \vdash \lambda \mathbf{x} : A^+ . e^{\div} : \mathbf{\Pi}\, \mathbf{x} : A^+ . (B^+ \rightarrow \alpha^+) \rightarrow \beta^+$$

By the induction hypothesis, we have

$$\Gamma^+, \mathbf{x} : A^+ \vdash e^{\div} : (B^+ \rightarrow \alpha^+) \rightarrow \beta^+$$

The goal easily follows by [E-Abs].

Case 8: (E-Rec)
This case is analogous to (E-Abs), but we have to make sure that the guardedness is preserved by the translation. Recall how the guard condition is defined: a recursive function f is guarded if its body destructs the argument x via pattern matching, and in each branch, the function is called only with a pattern variable representing a recursive argument. This means, the only possible form of a guarded recursive call is $f\ y$, where $y = n$ when f recurses on a natural number and $y = t$ when f recurses on a list. Then, to show preservation of guardedness, it suffices to show that the translation does not change the shape of the application $f\ y$. We can

easily see that the requirement is satisfied: if the body of f is a pure term, $\mathsf{f}\ \mathsf{y}$ is translated to $\mathbf{f}\ \mathbf{y}$ via the $^+$-translation, and otherwise, it is converted to $\lambda\,\mathbf{k}.\,\mathbf{f}\ \mathbf{y}\ \mathbf{k}$.

Case 9: (E-DApp) where $\rho = \tau = \epsilon$
Our goal is to show

$$\Gamma^+ \vdash \mathsf{e_0}^+\ \mathsf{e_1}^+ : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash \mathsf{e_0}^+ : \mathbf{\Pi}\,\mathbf{x} : \mathsf{A}^+.\,\mathsf{B}^+ \ \text{ and }\ \Gamma^+ \vdash \mathsf{e_1}^+ : \mathsf{A}^+$$

By compositionality (Lemma 4.6.1), we know that $(\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ = \mathsf{B}^+[\mathsf{e_1}^+/\mathbf{x}]$. The goal easily follows by [E-App].

Case 10: (E-NDApp) where $\rho = [\gamma, \delta], \sigma = [\beta, \gamma], \tau = [\alpha, \beta]$
Our goal is to show

$$\Gamma^+ \vdash \lambda\,\mathbf{k} : \mathsf{B}^+ \to \alpha^+.\,\mathsf{e_0}^{\div}\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \to (\mathsf{B}^+ \to \alpha^+) \to \beta^+.\,\mathsf{e_1}^{\div}\ (\lambda\,\mathbf{v_1} : \mathsf{A}^+.\,\mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k})) : \\ (\mathsf{B}^+ \to \alpha^+) \to \delta^+$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash \mathsf{e_0}^{\div} : ((\mathsf{A}^+ \to (\mathsf{B}^+ \to \alpha^+) \to \beta^+) \to \gamma^+) \to \delta^+ \ \text{ and }\ \Gamma^+ \vdash \mathsf{e_1}^{\div} : (\mathsf{A}^+ \to \beta^+) \to \gamma^+$$

Using [E-App], we can derive

$$\Gamma^+, \mathbf{k} : \mathsf{B}^+ \to \alpha^+, \mathbf{v_0} : \mathsf{A}^+ \to (\mathsf{B}^+ \to \alpha^+) \to \beta^+, \mathbf{v_1} : \mathsf{A}^+ \vdash \mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k} : \beta^+$$

This implies that the application of $\mathsf{e_1}^{\div}$ to its continuation is well-typed, and that the result type is γ^+. Then we know that the application of $\mathsf{e_0}^{\div}$ to its continuation is also well-typed, and that the result type is δ^+. Now the goal follows by [E-Abs].

Case 11: (E-Cons) where $\rho = \sigma = \tau = \epsilon$
Our goal is to show

$$\Gamma^+ \vdash ::\ \mathsf{e_0}^+\ \mathsf{e_1}^+\ \mathsf{e_2}^+ : (\mathsf{L}\ \mathsf{e_0})^+$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash {e_0}^+ : \mathbb{N}, \quad \Gamma^+ \vdash {e_1}^+ : \mathbb{N}, \quad \text{and} \quad \Gamma^+ \vdash {e_2}^+ : \mathbf{L}\ {e_0}^+$$

The goal easily follows by [E-Cons].

Case 12: (E-DMatchN) where $\rho = \epsilon$
Our goal is to show

$$\Gamma^+ \vdash \mathbf{pm}\ e^+\ \mathbf{as\ x\ in}\ \mathbb{N}\ \mathbf{ret}\ P^+\ \mathbf{with\ z} \rightarrow {e_1}^+ \,|\, \mathbf{suc\ n} \rightarrow {e_2}^+ : P^+[e^+/\mathbf{x}]$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash e^+ : \mathbb{N}\ , \quad \Gamma^+ \vdash {e_1}^+ : (P[z/x])^+\ , \quad \text{and} \quad \Gamma^+, \mathbf{n} : \mathbb{N} \vdash {e_2}^+ : (P[\mathsf{suc}\ n/x])^+$$

By compositionality (Lemma 4.6.1), we know that $(P[z/x])^+ = P^+[\mathbf{z}/\mathbf{x}]$ and similarly $(P[\mathsf{suc}\ n/x])^+ = P^+[\mathbf{suc\ n}/\mathbf{x}]$. The goal easily follows by [E-MatchN].

Case 13: (E-DMatchL) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\begin{aligned}&\Gamma^+ \vdash \lambda\mathbf{k} : (P[n/a, e/x])^+ \rightarrow (\alpha[n/a, e/x])^+. \\ &\qquad\quad \mathbf{pm}\ e^+\ \mathbf{as\ x\ in\ L\ a\ ret}\ \beta^+\ \mathbf{with\ z} \rightarrow {e_1}^{\div}\ \mathbf{k} \,|\, \mathbf{suc\ n} \rightarrow {e_2}^{\div}\ \mathbf{k} : \\ &\quad ((P[n/a, e/x])^+ \rightarrow (\alpha[n/a, e/x])^+) \rightarrow (\beta[n/a, e/x])^+\end{aligned}$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash e^+ : \mathbf{L}\ n^+\ ,$$

$$\Gamma^+ \vdash {e_1}^{\div} : ((P[z/a, \mathsf{nil}/x])^+ \rightarrow (\alpha[z/a, \mathsf{nil}/x])^+) \rightarrow (\beta[z/a, \mathsf{nil}/x])^+\ , \quad \text{and}$$

$$\begin{aligned}&\Gamma^+, \mathbf{m} : \mathbb{N}, \mathbf{h} : \mathbb{N}, \mathbf{t} : \mathbf{L}\ \mathbf{m} \vdash {e_2}^{\div} : \\ &((P[\mathsf{suc}\ m/a, ::\ m\ h\ t/x])^+ \rightarrow (\alpha[\mathsf{suc}\ m/a, ::\ m\ h\ t/x])^+) \rightarrow (\beta[\mathsf{suc}\ m/a, ::\ m\ h\ t/x])^+\end{aligned}$$

Similarly to the (E-DMatchN) case, we know that $(P[\mathsf{suc}\ m/a, ::\ m\ h\ t/x])^+ = P^+[\mathbf{suc\ m}/\mathbf{a}, ::\ \mathbf{m\ h\ t}/\mathbf{x}]$ by compositionality (Lemma 4.6.1). Our main task is to show that the applictaion $\mathbf{e_i}^{\div}\ \mathbf{k}$ $(i = 1, 2)$ is well-typed. Let us focus our attention to the first branch $\mathsf{e_1}^{\div}\ \mathbf{k}$. The induction hypothesis tells us that $\mathsf{e_1}^{\div}$ expects a continuation whose domain depends on $\mathbf{z}$ and $\mathbf{nil}$, while the actual continuation $\mathbf{k}$ has a domain dependent on n^+ and e^+. So, how can we prove that the application $\mathsf{e_1}^{\div}\ \mathbf{k}$ is well-typed? The answer is to use the equivalence assumptions of the target typing rule [E-Match]. The rule makes the equivalences $\mathsf{n}^+ \equiv \mathbf{z}$ and $\mathsf{e}^+ \equiv \mathbf{nil}$ available for type checking, bridging the gap between the required and actual domains of the continuation. By applying the similar reasoning to the other branch $\mathsf{e_2}^{\div}\ \mathbf{k}$, we can show that the translated pattern matching is well-typed.

Case 14: (E-Shift) where body impure
Our goal is to show

$$\Gamma^+ \vdash \lambda\,\mathbf{k} : \mathsf{A}^+ \rightarrow \alpha^+ .\, \mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathsf{B}^+ .\, \mathbf{v}) : (\mathsf{A}^+ \rightarrow \alpha^+) \rightarrow \beta^+$$

By the induction hypothesis, we have

$$\Gamma^+, \mathbf{k} : \mathsf{A}^+ \rightarrow \alpha^+ \vdash \mathsf{e}^{\div} : (\mathsf{B}^+ \rightarrow \mathsf{B}^+) \rightarrow \beta^+$$

Using [E-Abs] and [E-App], we can derive

$$\Gamma^+, \mathbf{k} : \mathsf{A}^+ \rightarrow \alpha^+ \vdash \mathsf{e}^{\div}\ (\lambda\,\mathbf{v} : \mathsf{B}^+ .\, \mathbf{v}) : \beta^+$$

The goal now follows by [E-Abs].

Case 15: (E-Reset) where body pure
Our goal is to show

$$\Gamma^+ \vdash \mathsf{e}^+ : \mathsf{A}^+$$

This immediately follows by the induction hypothesis on e.

Case 16: E-Conv where $\rho = [\alpha, \beta]$
Our goal is to show

$$\Gamma^+ \vdash \mathsf{e}^{\div} : (\mathsf{B}^+ \rightarrow \alpha'^+) \rightarrow \beta'^+$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash \mathsf{e}^{\div} : (\mathsf{A}^+ \rightarrow \alpha^+) \rightarrow \beta^+$$

By computational soundness (Lemma 4.6.4), we obtain $\mathsf{A}^+ \equiv \mathsf{B}^+$, $\alpha^+ \equiv \alpha'^+$, and $\beta^+ \equiv \beta'^+$. The goal now follows by [E-Conv].

□

Remark on Preservation of Guardedness If our translation was unselective, type preservation would require a weaker notion of guardedness in the target language. To see why this is the case, consider a Dellina- recursive call $\mathsf{f}\ \mathsf{y}$. An unselective translation would convert it to

$$\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,(\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{k}\ \mathbf{f})\ (\lambda\,\mathbf{v_0}.\,(\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{k}\ \mathbf{y})\ (\lambda\,\mathbf{v_1}.\,\mathbf{v_0}\ \mathbf{v_1}\ \alpha\ \mathbf{k}))$$

when the function has a pure body, and to

$$\lambda\,\mathbf{k}.\,(\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{k}\ \mathbf{f})\ (\lambda\,\mathbf{v_0}.\,(\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{k}\ \mathbf{y})\ (\lambda\,\mathbf{v_1}.\,\mathbf{v_0}\ \mathbf{v_1}\ \mathbf{k}))$$

when the function has an impure body. Notice that the function $\mathbf{f}$ is not fully applied in either case. This means we are unable to determine from the surface syntax whether $\mathbf{f}$ is called on a smaller argument.

The reason we obtain unapplied functions is that an unselective translation turns variables, which cannot have control effects, into suspended computations. On the other hand, if we reduce all redexes introduced by the translation (called *administrative redexes*), we will obtain $\lambda\,\alpha.\,\lambda\,\mathbf{k}.\,\mathbf{f}\ \mathbf{y}\ \alpha\ \mathbf{k}$ or $\lambda\,\mathbf{k}.\,\mathbf{f}\ \mathbf{y}\ \mathbf{k}$, which meets the guard condition. Based on this observation, Cong and Asai [45] recover preservation of guardedness by weakening the guard condition of the target language, which inspects CPS-translated recursive calls after administrative reductions.

4.7 Related Work

Unselective CPS Translations for Dependently Typed Languages The first work on typed CPS translations is due to Meyer and Wand [119], who studied a call-by-value, fixed-answer type translation of the simply typed λ-calculus. Subsequent work by Milner et al. [120] lifted the result to the full Standard ML, by extending the translation with polymorphism and undelimited control via `call/cc`.

In the dependent types community, CPS translations have been studied mainly in purely theoretical settings [23, 24, 121, 34]. The only exception, which accounts for a reasonably large language, is the work by Xi and Schürmann [175], who give a call-by-value translation of Dependent ML. Despite the difficulties with call-by-value translations discussed in Section 4.1, Xi and Schürmann managed to make their translation type-preserving, but the result largely hinges on the restricted dependency of the source language. As mentioned in Section 2.4, Dependent ML has a separate index language, which can only express a restricted form of computations. This language design makes it possible to keep type indices intact through the translation. A similar translation can be found in Ilik's work on extending predicate logic with delimited control [96]: he designs the calculus in such a way that types may depend only on "individuals" (constants), and defines a double-negation translation that has no effect on individuals. While these translations greatly simplify the type preservation argument, they do not scale to full-spectrum dependent languages, where type indices are generated by the same language as user programs.

Selective CPS Translations for Simply Typed Languages We adopt a selective CPS translation to preserve dependent types, but the original motivation for selectively translating programs comes from the performance side. Rompf et al. [140] support `shift` and `reset` in the Scala programming language via a selective CPS translation. To evaluate the performance, they compare the running time of continuations benchmarks with Kilim, a Java library that implements continuations via stack inspection [154]. The results show that Scala continuations are faster than Java continuations; in certain cases the difference amounts to more than a factor of seven. They also assess the efficiency of direct-style programming with `shift` and `reset` using the famous same-fringe problem, which inspects whether two binary trees have the same leaves in the same order. Among different Scala implementations, the `shift`/`reset`-based one runs reasonably fast, but not the fastest, due to the lazy semantics of a certain alternative means. On the other hand, when running programs with a memory restriction, many non-continuation implementations cannot even produce an answer, showing the economical advantage of continuation-based approaches.

Asai and Uehara [12] implement a Rompf-like type-and-effect system for `shift` and `reset`, and a selective CPS translation, in the OCaml language. By comparing selectively and unselectively translated programs, they observed a significant impact of selectiveness in the cases involving frequent uses of `shift`-captured continuations. For instance, in the N-Queen problem, which runs the same continuation multiple times with different inputs, the selective translation showed a 20-30% speedup. This result is due to the fact that a selective translation converts continuations into functions having a direct-style body—remember that continuations are *pure* functions.

Avoiding Induction on Derivations We defined our CPS translation by induction on the derivation. This design principle required us to establish equivalence preservation before type preservation, which required us to adopt an untyped equivalence. As an alternative approach, Barthe et al. [23] define a call-by-name CPS translation of the Calculus of Constructions [50], by induction on the syntax of the source term. To make this induction work, they give up explicitly typed abstractions, and use *domain-free* abstractions $\lambda\, x.\, e$ instead. Thus, we no longer need to generate type annotations for variables introduced by the translation.

In an effect-free setting, discarding type annotations has one single effect: type checking becomes hard. In an effectful setting, on the other hand, it further breaks uniqueness of the CPS image of syntactically equivalent terms. Recall from Section 3.2 that the function

$$\lambda\, \mathsf{f}.\, \lambda\, \mathsf{g}.\, ::\ 1\ 2\ \langle \mathsf{f}\ 3 + \mathsf{g}\ 4 \rangle$$

may be assigned either of the following types:

$$(\mathbb{N} \to \mathbb{N}[\mathbb{N}, \mathsf{L}\ 1]) \to (\mathbb{N} \to \mathbb{N}) \to \mathsf{L}\ 2 \quad \text{and} \quad (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}[\mathbb{N}, \mathsf{L}\ 1]) \to \mathsf{L}\ 2$$

When the function is given the former type, the derivation concludes f as an impure function, hence its CPS image would look like:

$$\lambda\, \mathbf{f}.\, \lambda\, \mathbf{g}.\, ::\ \mathbf{1}\ \mathbf{2}\ (\lambda\, \mathbf{k}.\, (\lambda\, \mathbf{k}.\, \mathbf{f}\ \mathbf{3}\ \mathbf{k})\ (\lambda\, \mathbf{v_1}.\, \mathbf{v_1} + \mathbf{g}\ \mathbf{4}))\ (\lambda\, \mathbf{v}.\, \mathbf{v})$$

On the other hand, when the function is given the latter type, the derivation concludes g as an impure function, hence we obtain:

$$\lambda\, \mathbf{f}.\, \lambda\, \mathbf{g}.\, ::\ \mathbf{1}\ \mathbf{2}\ (\lambda\, \mathbf{k}.\, (\lambda\, \mathbf{k}.\, \mathbf{g}\ \mathbf{4}\ \mathbf{k})\ (\lambda\, \mathbf{v_2}.\, \mathbf{f}\ \mathbf{3} + \mathbf{v_2}))\ (\lambda\, \mathbf{v}.\, \mathbf{v})$$

Obviously, these terms do not reduce to a common value. What this implies is that computational soundness does not hold, which in turn implies the failure of type preservation.

Challenges with Commutative Cuts The difficulty of CPS translating pattern matching is closely related to the challenges with *commutative cuts* [31] (*aka commuting conversions* [85]). Commutative cuts are one of the common optimization techniques of compilers, which expose redexes hidden in nested branching constructs (such as conditionals and pattern matching):

```
(* source program *)
if e1 && e2 then e3 else e4

(* desugaring *)
if (if e1 then e2 else false) then e3 else e4

(* commutative cut *)
if e1 then (if e2 then e3 else e4) else (if false then e3 else e4)

(* if-reduction *)
if e1 then (if e2 then e3 else e4) else e4
```

Given the source program, we first desugar the `&&` operator, obtaining a program that contains nested `if` expressions. We next apply a commutative cut, *i.e.*, we move the outer `if`-context to the branches `e2` and `false`. Then, we can see that the resulting program has a redex `if false ...`, which was not present in the source program. Thus, commutative cuts help us reduce runtime reductions.

Interestingly, what is happening in this example is exactly what our CPS translation does for pattern matching with impure branches, that is, both transformations distribute the surroundings of a branching construct to its branches. This means commutative cuts suffer from the same typing issue with the CPS translation of pattern matching: in a dependently typed setting, the copied context would expect a term of type `P(e1)`, whereas the branches `e3` and `e4` have types `P(true)` and `P(false)`, respectively. As a solution, Sozeau [153] proposes to type check the resulting branches with assumed equality proofs on type indices, which essentially play the same role as the equivalence assumptions in our target typing rule of pattern matching.

Remark on Lafont-Streicher-Reus CPS As Petrolo points out [133], standard CPS translations comprise a notion of values in order to make evaluation order explicit. On the other hand, there is a variant of CPS translation, called the Lafont-Streicher-Reus translation [109], that is based on the notion of continuations. The continuation-oriented nature of the LSR translation gives us several nice properties that a standard CPS translation lacks: *e.g.*, it validates the full η-reduction. The translation is also said to be well-suited for dependent calculi [132], as it simplifies the proof of type preservation. This is not a surprising result, because all the difficulties with dependent CPS comes from the treatment of values—recall the discussion from Section 4.1.

Chapter 5

The Dellina Language

In the preceding chapters, we saw how to deal with delimited continuations and dependent types in a restricted language Dellina-. Now we extend the language with advanced features, and thus obtain the full language Dellina. The features to be discussed include polymorphism and type operators (Section 5.1), a hierarchy of universes (Section 5.2), user-defined inductive types (Section 5.3), and local definitions via dependent `let` (Section 5.4). These are all essential ingredients in the mainstream dependently typed languages, but they have not yet been formally studied in combination with control effects. To show the practical impact of our language, we build an evaluator for a small object language (Section 5.5), which uses dependent types to ensure well-typedness of programs and control operators to simulate efficient exception raising.

5.1 Polymorphism and Type Operators

Barendregt's lambda cube has three axes: polymorphism, type operators, and dependency. In Dellina-, we partly supported the last feature by including a primitive constant `L` that forms a dependent type. The goal of this section is to fully support all the three features, making our language as expressive as the Calculus of Constructions (CC).

For simplicity, we will exclude primitive inductive types (`Unit`, $\mathbb{N}$, and `L n`) and recursive functions from the language, hence the resulting language is essentially a `shift`/`reset`-extension of CC.

$$\begin{array}{llcl}
\text{Environments} & \Gamma & ::= & \bullet \mid \Gamma, x : A \mid \Gamma, \alpha : \kappa \\
\text{Kinds} & \kappa & ::= & * \mid \Pi x : A.\, \kappa \mid \Pi \alpha : \kappa.\, \kappa \\
\text{Type-level Values} & V & ::= & \alpha \mid \lambda x : A.\, B \mid \lambda \alpha : \kappa.\, B \mid \Pi x : A.\, B\ \rho \mid \Pi \alpha : \kappa.\, B\ \rho \\
\text{Types} & A & ::= & V \mid A\ e \mid A\ B \\
\text{Term-level Values} & v & ::= & x \mid \lambda x : A.\, e \mid \lambda \alpha : \kappa.\, e \\
\text{Terms} & e & ::= & v \mid e\ e \mid e\ A \mid \mathcal{S}k : A \rightarrow \alpha.\, e \mid \langle e \rangle
\end{array}$$

Figure 5.1: Syntax of CC + `shift`/`reset`

5.1.1 Specification

5.1.1.1 Syntax

In Figure 5.1, we present the updated syntax. Readers familiar with PTS might find that the presentation is rather non-standard, in that kinds, types, terms are defined as separate syntactic categories. We make this distinction because we would like to be explicit about which category can have control effects and which cannot. As Barthe et al. [23] show, the two presentations are equivalent.

Let us go through the syntactic categories one by one. Typing environments have a new extension form $\Gamma, \alpha : \kappa$, which adds a type variable α and its kind κ. This extension is used when *e.g.* type checking the body of a type abstraction.

In the definition of kinds, we find functional kinds $\Pi x : A.\, \kappa$ and $\Pi \alpha : \kappa.\, \kappa'$ quantifying over term and type variables. These kinds represent the type of type-level functions $\lambda x : A.\, B$ and $\lambda \alpha : \kappa.\, B$, respectively. Notice that we do not have impure functional kinds, since the type language does not include control operators.

Having type-level functions means being able to compute at the level of types. This gives rise to the notion of type-level values and computations[1]. The former, which we evoke using the metavariable V, consist of variables, abstractions, and (pure and impure) function types. Non-value types include two forms of application, differing in whether the argument is a term or a type. Note that value types may have non-value types and terms as their subcomponents: for instance, $\Pi x : A.\, (B\ C)$ is classified as a value. Note also that the kind of polymorphism supported here is impredicative, since we do not restrict the domain of type ab-

[1]What we call type-level values/computations should not be confused with value/computation types from the CBPV literature, which denote the type of pure/impure terms.

$\boxed{\text{Evaluation Contexts } \mathsf{E}, \mathsf{F}}$

$$\begin{array}{rcl} \mathsf{E} & ::= & [\,] \mid \mathsf{E}\ \mathsf{e} \mid \mathsf{E}\ \mathsf{A} \mid \mathsf{v}\ \mathsf{E} \mid \mathsf{V}\ \mathsf{E} \mid \langle \mathsf{E} \rangle \\ \mathsf{F} & ::= & [\,] \mid \mathsf{F}\ \mathsf{e} \mid \mathsf{F}\ \mathsf{A} \mid \mathsf{v}\ \mathsf{F} \mid \mathsf{V}\ \mathsf{F} \end{array}$$

$\boxed{\text{Reduction Rules } \mathsf{e} \ \triangleright\ \mathsf{e}'}$

$$\begin{array}{rcl} (\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{B})\ \mathsf{v} & \triangleright_\beta & \mathsf{B}[\mathsf{v}/\mathsf{x}] \\ (\lambda\, \alpha : \kappa.\, \mathsf{A})\ \mathsf{V} & \triangleright_\beta & \mathsf{A}[\mathsf{V}/\alpha] \\ (\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e})\ \mathsf{v} & \triangleright_\beta & \mathsf{e}[\mathsf{v}/\mathsf{x}] \\ (\lambda\, \alpha : \kappa.\, \mathsf{e})\ \mathsf{V} & \triangleright_\beta & \mathsf{e}[\mathsf{V}/\alpha] \\ \langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle & \triangleright_{\mathcal{S}} & \langle \mathsf{e}[\lambda\, \mathsf{x} : \langle \mathsf{F}[\mathsf{x}] \rangle.\, /\mathsf{k}] \rangle \\ \langle \mathsf{v} \rangle & \triangleright_{\mathcal{R}} & \mathsf{v} \end{array}$$

Figure 5.2: Evaluation of CC + `shift`/`reset`

straction to monomorphic ($\Pi\,\alpha$-free) types.

Lastly, we extend terms with type abstraction and instantiation constructs. We classify type abstraction $\lambda\,\alpha : *.\, \mathsf{e}$ as a value, just like ordinary term abstraction. This is in contrast to ML-style polymorphism, where type abstraction is a non-value and runtime evaluation goes under the binder [67].

5.1.1.2 Reduction

We next extend evaluation contexts and reduction rules so that types may also participate in computation (Figure 5.2). The main change is that context holes can now be plugged with a type, and β-reduction can also result in a type. As we can see from the figure, both type-level application and type instantiation are evaluated in a call-by-value manner: we reduce the argument to either a term-level value v or a type-level value V, and then perform substitution.

5.1.1.3 Typing

Typing rules for the new constructs can be obtained by making trivial modifications to the Dellina- rules for variables, function types, abstractions, and application. Among the rules in Figures 5.3 and 5.4, (T-App) tells us that an application of a

$\boxed{\text{Well-formed Environments } \vdash \Gamma}$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \kappa : \Box}{\vdash \Gamma, \alpha : \kappa} \text{(G-ExtT)}$$

$\boxed{\text{Well-formed Kinds } \Gamma \vdash \kappa : \Box}$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash \kappa : \Box}{\Gamma \vdash \Pi x : A.\, \kappa : \Box} \text{(K-Pi)} \qquad \frac{\Gamma \vdash \kappa : \Box \quad \Gamma, \alpha : \kappa \vdash \kappa' : \Box}{\Gamma \vdash \Pi \alpha : \kappa.\, \kappa' : \Box} \text{(K-PiK)}$$

$\boxed{\text{Well-formed Types } \Gamma \vdash A : \kappa}$

$$\frac{\vdash \Gamma \quad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \text{(T-Var)}$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \lambda x : A.\, B : *} \text{(T-Abs)} \qquad \frac{\Gamma \vdash \kappa : \Box \quad \Gamma, \alpha : \kappa \vdash B : *}{\Gamma \vdash \lambda \alpha : \kappa.\, B : *} \text{(T-AbsK)}$$

$$\frac{\Gamma \vdash A : \Pi x : A'.\, \kappa \quad \Gamma \vdash e : A' \quad \Gamma \vdash \kappa[e/x] : \Box}{\Gamma \vdash A\ e : \kappa[e/x]} \text{(T-App)}$$

$$\frac{\Gamma \vdash A : \Pi \alpha : \kappa.\, \kappa' \quad \Gamma \vdash B : \kappa \quad \Gamma \vdash \kappa'[B/\alpha] : \Box}{\Gamma \vdash A\ B : \kappa'[B/\alpha]} \text{(T-Inst)}$$

$$\frac{\Gamma \vdash \kappa : \Box \quad \Gamma, \alpha : \kappa \vdash B : * \quad \Gamma, \alpha : \kappa \vdash \rho}{\Gamma \vdash \Pi \alpha : \kappa.\, B\ \rho : *} \text{(T-PiK)}$$

$$\frac{\Gamma \vdash A : \kappa \quad \Gamma \vdash \kappa' : \Box \quad \kappa \equiv \kappa'}{\Gamma \vdash A : \kappa'} \text{(T-Conv)}$$

Figure 5.3: Typing Rules for Polymorphic Types and Type Operators

$\boxed{\text{Well-typed Terms } \Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho}$

$$\frac{\Gamma, \alpha : \kappa \vdash \mathsf{e} : \mathsf{B}\ \rho}{\Gamma \vdash \lambda\, \alpha : \kappa. \mathsf{e} : \Pi\, \alpha : \kappa. \mathsf{B}\ \rho}\ (\text{E-AbsK})$$

$$\frac{\Gamma \vdash \mathsf{e} : (\Pi\, \alpha : \kappa. \mathsf{B}\ \sigma)\ \rho \qquad \Gamma \vdash \mathsf{A} : \kappa \qquad \Gamma \vdash \mathsf{B}[\mathsf{A}/\alpha] : * \qquad \Gamma \vdash \sigma[\mathsf{A}/\alpha] \qquad \tau = \mathrm{comp}(\rho, \sigma[\mathsf{A}/\alpha])}{\Gamma \vdash \mathsf{e}\ \mathsf{A} : \mathsf{B}[\mathsf{A}/\alpha]\ \tau}\ (\text{E-Inst})$$

Figure 5.4: Typing Rules for Polymorphic Functions and Type Instantiation

type A to a term e is valid only if the term is pure. We also have a well-formedness certificate of the result kind $\kappa[\mathsf{e}/\mathsf{x}]$, which we need for proving regularity. Similar premises can be found in (T-Inst) and (E-Inst), which account for type instantiation. Unlike Dellina-, we now have types whose kind is not just a constant kind $*$, hence we incorporate a conversion rule (T-Conv) for kind casting, making more type-level computations type check.

5.1.2 Metatheory

In this subsection, we examine the metatheoretic properties of the extended fragment. As we saw, extension by polymorphism and type operators gives rise to substitution that closes off type variables, hence we must restate all substitution-related properties. For instance, the substitution lemma is extended with the following clauses, assuming $\Gamma \vdash \mathsf{V} : \kappa$:

3. If $\vdash \Gamma, \alpha : \kappa, \Gamma'$, then $\vdash \Gamma, \Gamma'[\mathsf{V}/\alpha]$.

4. If $\Gamma, \alpha : \kappa, \Gamma' \vdash \mathsf{t} : \mathsf{T}\ \rho$, then $\Gamma, \Gamma'[\mathsf{V}/\alpha] \vdash \mathsf{t}[\mathsf{V}/\alpha] : \mathsf{T}[\mathsf{V}/\alpha]\ \rho[\mathsf{V}/\alpha]$.

Using this extended lemma, we can prove the type instantiation case of the preservation theorem, just as we did for term-level application:

Proof.

Case 1: (T-Inst)

Sub-Case 1: $(\lambda\,\alpha : \kappa_0.\, \mathsf{A})\ \mathsf{V}\ \rhd_p\ \mathsf{A}'[\mathsf{V}'/\alpha]$
Our goal is to show

$$\Gamma \vdash \mathsf{A}'[\mathsf{V}'/\alpha] : \kappa'[\mathsf{V}/\alpha]$$

By the induction hypothesis, we have

$$\Gamma \vdash \lambda\,\alpha : \kappa_0'.\, \mathsf{A}' : \Pi\,\alpha : \kappa.\, \kappa'\ ,\quad \Gamma \vdash \mathsf{V}' : \kappa\ ,\quad \text{and}\quad \Gamma \vdash \kappa[\mathsf{V}'/\alpha] : \square$$

We also have the following facts:

1. $\Pi\,\alpha : \kappa.\, \kappa' \equiv \Pi\,\alpha : \kappa_0'.\, \kappa_1$ (by inversion for λ)
2. $\Gamma,\, \alpha : \kappa_0' \vdash \mathsf{A} : \kappa_1$ (by inversion for λ)
3. $\kappa \equiv \kappa_0'$ and $\kappa' \equiv \kappa_1$ (by item 1 and injectivity of Π (Lemma 3.4.6))
4. There is a subderivation of $\Gamma \vdash \kappa_0' : \square$ (by item 2 and Lemma 3.4.10)

By items 3, 4, and (E-Conv), we can derive $\Gamma \vdash \mathsf{V}' : \kappa_0'$. Then, by item 2 and the substitution lemma (Lemma 5.4.4), we obtain

$$\Gamma \vdash \mathsf{A}'[\mathsf{V}'/\alpha] : \kappa_1[\mathsf{V}'/\alpha]$$

Item 2 and Lemma 3.4.7 further give us $\kappa'[\mathsf{V}'/\alpha] \equiv \kappa_1[\mathsf{V}'/\alpha]$. These imply

$$\Gamma \vdash \mathsf{A}'[\mathsf{V}'/\alpha] : \kappa'[\mathsf{V}'/\alpha]$$

Now, Lemma 3.4.2 tells us that $\kappa'[\mathsf{V}/\alpha]\ \rhd_p\ \kappa'[\mathsf{V}'/\alpha]$. Using these facts, and the well-formedness premises of the result type, and (E-Conv), we can derive $\Gamma \vdash \mathsf{A}'[\mathsf{V}'/\alpha] : \kappa'[\mathsf{V}/\alpha]$ as desired.

$\square$

5.1.3 CPS Translation

In Figure 5.5, we present the CPS translation for the polymorphism and type operators fragment. There is nothing surprising in the translation, but it is worth noting that a type-level application ($\mathsf{A}\ \mathsf{e}$ or $\mathsf{A}\ \mathsf{B}$) is never translated into CPS,

$$\frac{\vdash \Gamma \quad \Gamma \vdash \kappa : \square}{\vdash \Gamma, \alpha : \kappa} \text{ (G-ExtT)} \overset{+}{\rightsquigarrow} \Gamma^+, \alpha : \kappa^+$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash \kappa : \square}{\Gamma \vdash \Pi x : A.\, \kappa : \square} \text{ (K-Pi)} \overset{+}{\rightsquigarrow} \mathbf{\Pi}\, \mathbf{x} : A^+.\, \kappa^+$$

$$\frac{\Gamma \vdash \kappa : \square \quad \Gamma, \alpha : \kappa \vdash \kappa' : \square}{\Gamma \vdash \Pi \alpha : \kappa.\, \kappa' : \square} \text{ (K-PiK)} \overset{+}{\rightsquigarrow} \mathbf{\Pi}\, \alpha : \kappa^+.\, \kappa'^+$$

$$\frac{\vdash \Gamma \quad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \text{ (T-Var)} \overset{+}{\rightsquigarrow} \alpha$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \lambda x : A.\, B : *} \text{ (T-Abs)} \overset{+}{\rightsquigarrow} \lambda\, \mathbf{x} : A^+.\, B^+$$

$$\frac{\Gamma \vdash \kappa : \square \quad \Gamma, \alpha : \kappa \vdash B : *}{\Gamma \vdash \lambda \alpha : \kappa.\, B : *} \text{ (T-AbsK)} \overset{+}{\rightsquigarrow} \lambda\, \alpha : \kappa^+.\, B^+$$

$$\frac{\Gamma \vdash A : \Pi x : A'.\, \kappa \quad \Gamma \vdash e : A' \quad \Gamma \vdash \kappa[e/x] : \square}{\Gamma \vdash A\ e : \kappa[e/x]} \text{ (T-App)} \overset{+}{\rightsquigarrow} A^+\ e^+$$

$$\frac{\Gamma \vdash A : \Pi \alpha : \kappa.\, \kappa' \quad \Gamma \vdash B : \kappa \quad \Gamma \vdash \kappa'[B/\alpha] : \square}{\Gamma \vdash A\ B : \kappa'[B/\alpha]} \text{ (T-Inst)} \overset{+}{\rightsquigarrow} A^+\ B^+$$

$$\frac{\Gamma \vdash \kappa : \square \quad \Gamma, \alpha : \kappa \vdash B : * \quad \Gamma, \alpha : \kappa \vdash \rho}{\Gamma \vdash \Pi \alpha : \kappa.\, B\ \rho : *} \text{ (T-PiK)}$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi}\, \alpha : \kappa^+.\, B^+ \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi}\, \alpha : \kappa^+.\, (B^+ \to \alpha^+) \to \beta^+ \text{ if } \rho = [\alpha, \beta]$$

$$\frac{\Gamma \vdash A : \kappa \quad \Gamma \vdash \kappa' : \square \quad \kappa \equiv \kappa'}{\Gamma \vdash A : \kappa'} \text{ (T-Conv)} \overset{+}{\rightsquigarrow} A^+$$

Figure 5.5: CPS Translation for Polymorphic Types and Type Operators

$$\frac{\Gamma, \alpha : \kappa \vdash \mathsf{e} : \mathsf{B}\ \rho}{\Gamma \vdash \lambda\, \alpha : \kappa.\, \mathsf{e} : \Pi\, \alpha : \kappa.\, \mathsf{B}\ \rho}\ \text{(E-AbsK)}$$

$$\overset{+}{\rightsquigarrow}\ \lambda\, \alpha : \kappa^{+}.\, \mathsf{e}^{+} \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow}\ \lambda\, \alpha : \kappa^{+}.\, \mathsf{e}^{\div} \text{ if } \rho = [\alpha', \beta]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \Pi\, \alpha : \kappa.\, \mathsf{B}\ \sigma\ \rho \quad \Gamma \vdash \mathsf{A} : \kappa \\ \Gamma \vdash \mathsf{B}[\mathsf{A}/\alpha] : * \quad \Gamma \vdash \rho[\mathsf{A}/\alpha] \quad \tau = \mathrm{comp}(\rho, \sigma[\mathsf{A}/\alpha])\end{array}}{\Gamma \vdash \mathsf{e}\ \mathsf{A} : \mathsf{B}[\mathsf{A}/\alpha]\ \tau}\ \text{(E-Inst)}$$

$$\overset{+}{\rightsquigarrow}\ \mathsf{e}^{+}\ \mathsf{A}^{+}$$
$$\text{if } \rho = \sigma = \epsilon$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\, \mathbf{k} : \mathsf{B}^{+}[\mathsf{A}^{+}/\alpha] \rightarrow \alpha'^{+}.\, \mathsf{e}^{\div}\ (\lambda\, \mathbf{v} : \mathbf{\Pi}\, \alpha : \kappa^{+}.\, \mathsf{B}^{+}.\, \mathbf{k}\ (\mathbf{v}\ \mathsf{A}^{+}))$$
$$\text{if } \rho = \epsilon, \sigma = [\alpha', \beta]$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\, \mathbf{k} : \mathsf{B}^{+}[\mathsf{A}^{+}/\alpha] \rightarrow \alpha'^{+}.\, \mathsf{e}^{+}\ \mathsf{A}^{+}\ \mathbf{k}$$
$$\text{if } \rho = [\alpha', \beta], \sigma = \epsilon$$

$$\overset{\div}{\rightsquigarrow}\ \lambda\, \mathbf{k} : \mathsf{B}^{+}[\mathsf{A}^{+}/\alpha] \rightarrow \alpha'^{+}[\mathsf{A}^{+}/\alpha].\, \mathsf{e}^{\div}\ (\lambda\, \mathbf{v} : \mathbf{\Pi}\, \alpha : \kappa^{+}.\, \mathsf{B}^{+}.\, \mathsf{v}\ \mathsf{A}^{+}\ \mathbf{k})$$
$$\text{if } \rho = [\beta, \gamma], \sigma = [\alpha', \beta]$$

Figure 5.6: CPS Translation for Polymorphic Functions and Type Instantiation

because all its constituents (function, body of function, and argument) are pure expressions.

Similarly to the substitution lemma discussed above, we extend the compositionality lemma with several additional cases. Note that we need the first item because substitution now operates on kinds as well.

1. $(\kappa[\mathsf{e}'/\mathsf{x}'])^{+} = \kappa^{+}[\mathsf{e}'^{+}/\mathbf{x}']$
2. $(\kappa[\mathsf{A}'/\alpha])^{+} = \kappa^{+}[\mathsf{A}'^{+}/\alpha]$
3. $(\mathsf{A}[\mathsf{A}'/\alpha])^{+} = \mathsf{A}^{+}[\mathsf{A}'^{+}/\alpha]$
4. $(\mathsf{e}[\mathsf{A}'/\alpha])^{+} = \mathsf{e}^{+}[\mathsf{A}'^{+}/\alpha]$
5. $(\mathsf{e}[\mathsf{A}'/\alpha])^{\div} = \mathsf{e}^{\div}[\mathsf{A}'^{+}/\alpha]$

This allows us to prove computational soundness of the new β-reduction $(\lambda\,\alpha.\,\mathsf{t})\ \mathsf{A} \triangleright \mathsf{t}[\mathsf{A}/\alpha]$. Here is the proof of the case where t is an impure term e and the β-redex is derived by (E-Inst):

Proof.

$$\begin{aligned}
((\lambda\,\alpha.\,\mathsf{e})\ \mathsf{V})^{\div} &= \lambda\,\mathbf{k}.\,(\lambda\,\alpha.\,\mathsf{e})^{+}\ \mathsf{V}^{+}\ \mathbf{k} && \text{by translation}\\
&= \lambda\,\mathbf{k}.\,(\lambda\,\alpha.\,\mathsf{e}^{\div})\ \mathsf{V}^{+}\ \mathbf{k} && \text{by translation}\\
&\triangleright_{\beta} \lambda\,\mathbf{k}.\,(\mathsf{e}^{\div}[\mathsf{V}^{+}/\alpha])\ \mathbf{k} && \\
&\equiv \mathsf{e}^{\div}[\mathsf{V}^{+}/\alpha] && \text{by } \eta\\
&= (\mathsf{e}[\mathsf{V}/\alpha])^{\div} && \text{by compositionality}
\end{aligned}$$

□

5.1.4 Remark on Polymorphically Typed Shift and Reset

Asai and Kameyama [10] study `shift` and `reset` in the presence of *let-polymorphism*[2]. It is well-known that naïve combination of polymorphism and control effects is un-

[2]A polymorphic `let` expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ allows using the variable x in different contexts. For instance, $\mathsf{let}\ f = \lambda\,x.\,x\ \mathsf{in}\ \mathsf{if}\ f\ \mathsf{true}\ \mathsf{then}\ f\ 1\ \mathsf{else}\ f\ 2$ is well-typed in a language featuring let-polymorphism; notice that the type of f is instantiated to $\mathsf{bool} \to \mathsf{bool}$ in the first call and $\mathsf{int} \to \mathsf{int}$ in the second and third calls.

sound: we can easily build a backtracking function that behaves monomorphically but is wrongly given a polymorphic type [91]. A popular approach to recovering soundness is to impose a *value restriction* [169], that is, we restrict the definition of a `let`-bound variable to a value. Asai and Kameyama, on the other hand, use a more generous *purity restriction*, which restricts definitions to pure terms. The resulting calculus is sound, and enjoys other properties such as existence of principal types.

As an extension, Asai and Kameyama further consider a `shift`/`reset`-calculus with impredicative polymorphism *à la* Girard's System F [84]. They give two variants of their calculus: one with the standard notion of polymorphism, and the other with an ML-like treatment of type abstraction. As noted above, the two variants differ in the reduction semantics, hence their CPS translations are also different, but both of them have proven type- and meaning-preserving [10].

Biernacka et al. [28] prove the normalization property of System F extended with `shift` and `reset`. Normalization of calculi with control is often proved by defining a CPS translation into a normalizing language and showing preservation of reduction and typing, as in the work by Harper and Lillibridge [92] and Asai and Kameyama [10]. Biernacka et al. take a different approach: they refine Girard's reducibility method [84] by defining reducibility predicates on evaluation contexts. Intuitively, the predicates tell us that an context is "good" if it yields a good expression when plugged with a good value.

5.2 Prop, Set, and Universe Hierarchy

We next enrich our language with an infinite hierarchy of *sorts*. So far, we have been dealing with two kinds: $*$, which is the type of types, and $\square$, which is the type of $*$. The sort hierarchy is a generalization of this "type-of" relation, allowing us to discuss the type of $\square$ and any other expressions at a higher level.

When incorporating sorts, we must decide which sorts are predicative, and which are not. As shown by Girard [84], this is a quite delicate issue; in particular, allowing impredicative sorts at non-bottom levels (as in System U [84]) results in an inconsistent calculus, where one can prove a type-theoretic variant of Russell's paradox (*aka* Girard's paradox) [48].

Values	v, V	$::=$	$\mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type_i} \mid \mathsf{x} \mid \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e} \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$
Expressions	$\mathsf{s}, \mathsf{A}, \mathsf{e}$	$::=$	$\mathsf{v} \mid \mathsf{e}\ \mathsf{e} \mid \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} \mid \langle \mathsf{e} \rangle$

Figure 5.7: Collapsed Syntax

We equip Dellina with a sort hierarchy in the style of Calculus of Inductive Constructions (CIC) [168]. This extension makes everything look like an element of the term language, where we have application, pattern matching, and all sorts of interesting computations. However, there is one proviso: control operators are available only at the bottom level. We do not allow type-level control effects because types are generally erased after the type checking phase; if types had control effects, the erasure would fail to preserve the behavior of the source program. As it turns out, even with the restricted use of control operators, we need to be careful about the "answer sort" of impure terms in order to obtain a type-preserving CPS translation.

5.2.1 Specification

5.2.1.1 Syntax

In languages featuring a sort hierarchy, we usually do not distinguish between types and terms. Therefore, we switch our syntax to a collapsed one (Figure 5.7), where sorts, Π-types, and λ-terms are all defined in one single category. We have two kinds of bottom sorts, Prop and Set. As in Coq, Prop is the type of propositions (*e.g.*, $0 = 0$), which are inhabited by proofs, whereas Set is the type of datatypes (*e.g.*, $\mathbb{N}$), which are inhabited by computations. The two sorts differ in their predicativity: the former is impredicative, while the latter is predicative. Higher sorts, also called *universes*, take the form $\mathsf{Type_i}$, where i ranges over non-zero natural numbers. These are all predicative, which is a mandatory design strategy for maintaining consistency. As a convention, we will write S to mean the set of sorts and universes, that is, $\mathsf{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type_i}\}$.

5.2.1.2 Subtyping

In the presence of a universe hierarchy, one sometimes want to regard a term of type $\mathsf{Type_i}$ as being of type $\mathsf{Type_j}$ for any $j > i$. That is, we wish universes to be

$$\frac{\mathsf{A} \equiv \mathsf{B}}{\mathsf{A} \preceq \mathsf{B}}\ (\preceq\text{-}\equiv) \qquad \frac{\mathsf{A} \preceq \mathsf{B} \quad \mathsf{B} \preceq \mathsf{C}}{\mathsf{A} \preceq \mathsf{C}}\ (\preceq\text{-}\textsc{Trans})$$

$$\frac{}{\mathsf{Type_i} \preceq \mathsf{Type_{i+1}}}\ (\preceq\text{-}\textsc{Cum}) \qquad \frac{\mathsf{A_1} \equiv \mathsf{A_2} \quad \mathsf{B_1} \preceq \mathsf{B_2}}{\Pi\,\mathsf{x} : \mathsf{A_1}.\,\mathsf{B_1} \preceq \Pi\,\mathsf{x} : \mathsf{A_2}.\,\mathsf{B_2}}\ (\preceq\text{-}\textsc{Pi})$$

Figure 5.8: Subtyping Rules

not only stratified, but also *cumulative*. To account for cumulativity, we replace our equivalence-based convertibility with a *subtyping*-based one. In Figure 5.8, we present four subtyping rules, which are borrowed from the Extended Calculus of Constructions of Luo [114]. The central idea of the subtyping relation is captured by ($\preceq$-Cum), which tells us that inhabitants of $\mathsf{Type_i}$ are also inhabitants of $\mathsf{Type_{i+1}}$. The relation also includes the equivalence relation we have been using so far, *i.e.*, equivalent types are the subtype of each other (see rule ($\preceq$-$\equiv$)).

Rule ($\preceq$-Pi) lifts the subtyping relation to function types. It might be a good idea to first look at the subtyping relation of non-dependent arrow types: we say $A_1 \to B_1 \preceq A_2 \to B_2$ when $A_2 \preceq A_1$ and $B_1 \preceq B_2$. The definition tells us that function types are *contravariant* in their domain, and *covariant* in their co-domain[3].

Now, if we look at ($\preceq$-Pi), we see that the two domains $\mathsf{A_1}$ and $\mathsf{A_2}$ are required to be equivalent. As Luo [114] points out, making domains contravariant does not break metatheoretic properties, but it allows certain terms to have multiple types: for instance, the identity function $\lambda\,\mathsf{x} : \mathsf{Type_1}.\,\mathsf{x}$ may have types $\mathsf{Type_1} \to \mathsf{Type_i}$, $\mathsf{Prop} \to \mathsf{Type_i}$, and $\mathsf{Set} \to \mathsf{Type_i}$ for any $i \geq 1$.

Notice that ($\preceq$-Pi) does not account for function types with effect annotations. The reason is that cumulativity is available only in higher universes $\mathsf{Type_i}$, whose inhabitants are free from control effects. In other words, although we have refined the notion of convertibility in terms of subtyping, type casting of bottom-level terms is still done with reagrd to equivalence. What this implies is that, when we

[3]As a simple example showing why the domain must be contravariant, consider $f_1 = \lambda\, x : \mathsf{int}.\, 1$ and $f_2 = \lambda\, x : \mathbb{N}.\, 1$. Using f_1 in a context demanding a $\mathbb{N}$-receiving function is always safe: *e.g.*, we can put f_1 into the hole of [.] 2. However, it is unsafe to use f_2 in a context demanding a int-receiving: *e.g.*, we cannot put f_2 into the hole of [.] (–1). This means, we may cast $\mathsf{int} \to \mathbb{N}$ to $\mathbb{N} \to \mathbb{N}$, but not the other way around.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_1}\ (\textsc{Prop}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}_1}\ (\textsc{Set})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ (\textsc{Type})$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : \mathsf{Prop} \\ \Gamma, x : A \vdash \rho : (s', \mathsf{Prop}) \quad s \in S \quad s' \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash \Pi x : A.\, B\ \rho : \mathsf{Prop}}\ (\textsc{PiProp})$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : \mathsf{Set} \\ \Gamma, x : A \vdash \rho : (s', \mathsf{Set}) \quad s, s' \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash \Pi x : A.\, B\ \rho : \mathsf{Set}}\ (\textsc{PiSet})$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, x : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \Pi x : A.\, B : \mathsf{Type}_i}\ (\textsc{PiType})$$

Figure 5.9: Typing Rules of Sorts and Function Types

have a bottom-level term whose type has been casted from A to B via $A \preceq B$, the last rule used to derive the subtyping relation must be ($\preceq$-$\equiv$).

5.2.1.3 Typing

We now extend our type system. First, we add three axioms that account for the stratification of sorts (Figure 5.9). (Prop) and (Set) tell us that Prop and Set are inhabitants of Type_1. Universes Type_i reside in Type_{i+1}, as stated by (Type). Following the PTS terminology, we call e an s-*term* and A an s-*type* when $\Gamma \vdash e : A\ \rho$ and $\Gamma \vdash A : s$.

In the same figure, we define rules for function types, which characterize the three sorts as either predicative or impredicative. In (PiProp), the domain A resides in an *arbitrary* sort s, which means A may include the type being formed in

$$\frac{\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}\ \rho \quad \Gamma \vdash \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho : \mathsf{s} \quad \mathsf{s} \in \mathsf{S}}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_0 : (\mathsf{A} \rightarrow \mathsf{B}\ \tau)\ \rho \quad \Gamma \vdash \mathsf{e}_1 : \mathsf{A}\ \sigma \\ \nu = \mathrm{comp}(\rho, \sigma, \tau) \quad \Gamma \vdash \mathsf{B} : \mathsf{Prop}\ \text{ or }\ \Gamma \vdash \mathsf{B} : \mathsf{S}\end{array}}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}\ \nu}\ (\textsc{NDApp})$$

$$\frac{\begin{array}{c}\Gamma, \mathsf{k} : \mathsf{A} \rightarrow \alpha \vdash \mathsf{e} : \beta\ \text{ or }\ \Gamma, \mathsf{k} : \mathsf{A} \rightarrow \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta] \\ \Gamma \vdash \mathsf{A} : \mathsf{s} \quad \Gamma \vdash \alpha : \mathsf{s}' \quad \Gamma \vdash \beta : \mathsf{s} \quad \mathsf{s}, \mathsf{s}' \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e} : \mathsf{A}[\alpha, \beta]}\ (\textsc{Shift})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \quad \Gamma \vdash \mathsf{A} : \mathsf{s} \quad \Gamma \vdash \rho : (\mathsf{s}', \mathsf{s}) \\ \Gamma \vdash \mathsf{B} : \mathsf{s} \quad \Gamma \vdash \sigma : (\mathsf{s}', \mathsf{s}) \\ \mathsf{A} \preceq \mathsf{B} \quad \rho \preceq \sigma\end{array}}{\Gamma \vdash \mathsf{e} : \mathsf{B}\ \sigma}\ (\textsc{Conv})$$

Figure 5.10: Typing Rules of Shift and Conversion Rules

the conclusion. In (PiSet), A is less flexible about its universe level: it must reside in Prop or Set, which cannot contain the function type we are forming. Similarly, (PiType) requires that the domain A lives in the same level as the co-domain B. Note that the rule allows building a type where A resides in some lower universe, thanks to the ($\preceq$-Cum) rule. If we did not have the subtyping rules, we would have to tweak (PiType) to allow functions across universe levels, which would make the rule less elegant.

Shifting our attention to the effect annotations in (PiProp) and (PiSet), we find a new form of judgment $\Gamma, \mathsf{x} : \mathsf{A} \vdash \rho : (\mathsf{s}', \mathsf{s})$, associated with a requirement on s'. These premises state that, when ρ is a non-empty annotation $[\alpha, \beta]$, the final answer type β must reside in s, *i.e.*, the same sort as B, while the initial answer type α may live in a different sort s (but at the same level). Note that the judgment trivially holds when ρ is empty. As we will see in Section 5.2.3, the constraints on the sort of answer types is necessary for making the CPS translation type-preserving.

Figure 5.10 includes a refined version of the typing rules for abstraction, non-dependent application, `shift`, and type conversion. In (Abs), we see an extra premise $\Gamma \vdash \Pi x : A.\, B\ \rho : s$, stating that the type to be given to the abstraction is well-formed. We need this presmise because not all combinations of A and B are valid: *e.g.*, when $A = \mathsf{Type}_1$ and $B = \mathsf{Set}$, we have $\Gamma, x : \mathsf{Type}_1 \vdash \mathbb{N} : \mathsf{Set}$, but the type $\Pi x : \mathsf{Type}_1.\, \mathsf{Set}$ of the function $\lambda x : \mathsf{Type}_1.\, \mathbb{N}$ violates the predicativity of Set. We also find that the name of the typing rule lacks the "T-" or "E-" prefix we had before. What this means is that we may use (Abs) for abstractions receiving and returning expressions at any universe level. However, the effect annotation ρ must be empty when e is not a Prop- or Set-term[4]. We make similar modifications to rules for variables and application. Furthermore, in (NDApp), which derives a non-dependent application, we require that the result type is either a Prop-type or a Set-type. The reason is that having an impure argument in non-term-level application results in invalid dependency. This design principle is already present in the type system from the previous section: remember that type-level application $A\ e$ has one single rule that requires a pure e.

We next look at (Shift) and (Conv). The former rule introduces control effects, hence we restrict the universe of answer types along the same lines as we do in (PiProp) and (PiSet). The conversion rule allows casting the type of e from A to B if A is a subtype of B. The additional premises on the original type and annotation are used to ensure that casting does not change their universe, which would break the restriction imposed by rules for function types and `shift` constructs.

5.2.2 Metatheory

We have seen that the introduction of a sort hierarchy has the following impacts on the type system:

1. New axioms for sorts, like (Prop);

2. Extra premises in the existing rules, such as (Abs) and (Shift); and

[4]The purity of non-term-level functions is guaranteed by the well-formedness of $\Pi x : A.\, B\ \rho$: if $\rho = [\alpha, \beta]$, the function type must be derived by (Pi-Prop) or (Pi-Set), which ensure that B and β reside in the same, bottom-level sort.

Axioms derive an atomic expression, hence the metatheoretic properties trivially scale to these rules. The additional premises require some refinement to the statement of theorems as well as the proofs. For instance, to incorporate the universe requirement of (Shift), we have to modify the regularity statement in the following way:

Theorem 5.2.1 (Regularity)**.**

1. *If* $\Gamma \vdash \mathsf{t} : \mathsf{T}$*, then there exists some* $\mathsf{s} \in \mathsf{S}$ *such that* $\Gamma \vdash \mathsf{T} : \mathsf{s}$*.*
2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]$*, then there exists some* $\mathsf{s}, \mathsf{s}' \in \{\mathsf{Prop}, \mathsf{Set}\}$ *such that* $\Gamma \vdash \mathsf{A} : \mathsf{s}$*,* $\Gamma \vdash \alpha : \mathsf{s}'$*, and* $\Gamma \vdash \beta : \mathsf{s}$*.*

The theorem has a separate clause where the subject e is an impure term. Recall that, in Dellina-, there is only one single kind $*$, and regularity simply states that the types A, α, and β are well-formed. Now, the theorem states that the three types reside in the bottom sort, and in particular, A and β live in the same sort.

The well-formedness premise of (Abs) requires some modification to the preservation proof, as shown below:

Proof.

Case 1: (Abs)
Our goal is to show

$$\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}'.\, \mathsf{e}' : \Pi \mathsf{x} : \mathsf{A}.\, \mathsf{B}\ \rho$$

As before, we have

$$\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e}' : \mathsf{B}\ \rho$$

by the induction hypothesis, from which we can derive

$$\Gamma, \mathsf{x} : \mathsf{A}' \vdash \mathsf{e}' : \mathsf{B}\ \rho$$

using Lemma 3.4.10, the induction hypothesis, and Lemma 3.4.12. The next step is to apply (Abs), but since the rule now requires well-formedness of the resulting function type, we must show

$$\Gamma \vdash \Pi x : A'.\, B\ \rho : s$$

This is however trivial, since the derivation of the pre-reduction abstraction $\lambda x : A.\, e$ has a well-formedness premise $\Pi x : A.\, B\ \rho$, and the induction hypothesis on this premise gives us what we need. Thus we obtain

$$\Gamma \vdash \lambda x : A'.\, e' : \Pi x : A'.\, B\ \rho$$

and the proof is completed by (Conv). Note that the last step, application of (Conv), has been simplified thanks to the well-formedness premise of the original type.

□

The new notion of convertibility requires some care when proving the progress theorem. Recall the discussion from Section 3.4.6: to prove progress, we need the canonical forms lemma, which holds only if type casting preserves the shape of the original type. Since we now use the subtyping relation to cast types, we must show that it never relates two types having distinct head constructors. It is not hard to see that this is the case. As we have no inductive types, the only head constructor of the language is Π. The subtyping relation $C \preceq \Pi x : A.\, B$ may be derived by either ($\preceq$-$\equiv$) or ($\preceq$-Pi): in the former case, C must reduce to a function type, and in the latter case, C must itself be a function type. Therefore, when we have a value $v : \Pi x : A.\, B$ derived by (Conv), we know that the pre-conversion type must be convertible with a function type, *i.e.*, the induction hypothesis applies.

When incorporating user-defined inductive types, on the other hand, we must be careful of their interaction with subtyping. We will provide some more details in Section 5.3.1.

5.2.3 CPS Translation

In Figure 5.11, we present the CPS translation of sorts and function types. These expressions have no computation translation, because they are free from control effects. The value and computation translations of Prop- and Set-terms remain the same as before.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_1}\ (\textsc{Prop}) \overset{+}{\rightsquigarrow} \mathbf{Prop}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}_1}\ (\textsc{Set}) \overset{+}{\rightsquigarrow} \mathbf{Set}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ (\textsc{Type}) \overset{+}{\rightsquigarrow} \mathbf{Type_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{s} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Prop} \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \rho : (\mathsf{s}', \mathsf{Prop}) \qquad \mathsf{s}, \mathsf{s}' \in \mathsf{S}\end{array}}{\Gamma \vdash \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho : \mathsf{Prop}}\ (\textsc{PiProp})$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\,x} : \mathsf{A}^+.\,\mathsf{B}^+ \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\,x} : \mathsf{A}^+.\,(\mathsf{B}^+ \to \alpha^+) \to \beta^+ \text{ if } \rho = [\alpha, \beta]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{s} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Set} \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \rho : (\mathsf{s}', \mathsf{Set}) \qquad \mathsf{s}, \mathsf{s}' \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}\ \rho : \mathsf{Set}}\ (\textsc{PiSet})$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\,x} : \mathsf{A}^+.\,\mathsf{B}^+ \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \mathbf{\Pi\,x} : \mathsf{A}^+.\,(\mathsf{B}^+ \to \alpha^+) \to \beta^+ \text{ if } \rho = [\alpha, \beta]$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}_i \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Type}_i}{\Gamma \vdash \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} : \mathsf{Type}_i}\ (\textsc{PiType}) \overset{+}{\rightsquigarrow} \mathbf{\Pi\,x} : \mathsf{A}^+.\,\mathsf{B}^+$$

Figure 5.11: CPS Translation of Sorts and Function Types

The main work is to establish the type preservation property. First, since conversion uses subtyping, we must show that the translation preserves the subtyping relation:

Lemma 5.2.1 (Preservation of Subtyping)**.** *If* $\mathsf{A} \preceq \mathsf{B}$*, then* $\mathsf{A}^+ \preceq \mathsf{B}^+$*.*

Proof. The proof is by induction on the derivation of $\mathsf{A} \preceq \mathsf{B}$.

Case 1: ($\preceq$-Cum)
This case is trivial, since $\mathsf{Type_i}$ is simply mapped to $\mathbf{Type_i}$ by the translation.

Case 2: ($\preceq$-Trans)
This case follows by the induction hypothesis.

Case 3: ($\preceq$-$\equiv$)
This case can be proved in the same way as the computational soundness lemma (Lemma 4.6.4).

Case 4: ($\preceq$-Pi)
This case follows by the induction hypothesis.

□

The type preservation proof is mostly straightforward, but we would like to show one case that deal with function types, because the proof relies on the sort requirements in the source typing rules.

Proof.

Case 1: (PiSet) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\Gamma^+ \vdash \mathbf{\Pi\, x} : \mathsf{A}^+.\, (\mathsf{B}^+ \to \alpha^+) \to \beta^+ : \mathbf{Set}$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash \mathsf{A}^+ : \mathbf{s}\ , \quad \Gamma^+, \mathbf{x} : \mathsf{A}^+ \vdash \mathsf{B}^+ : \mathbf{Set}\ ,$$

$$\Gamma^+, \mathbf{x} : \mathsf{A}^+ \vdash \alpha^+ : \mathbf{s'}\ , \quad \text{and} \quad \Gamma^+, \mathbf{x} : \mathsf{A}^+ \vdash \beta^+ : \mathbf{Set}$$

where $\mathbf{s}, \mathbf{s}' \in \{\mathbf{Prop}, \mathbf{Set}\}$. Using either [PiProp] or [PiSet], we can derive $\Gamma^+, \mathbf{x} : A^+ \vdash B^+ \rightarrow \alpha^+ : \mathbf{s}'$. Then, using [PiSet], we obtain $\Gamma^+, \mathbf{x} : A^+ \vdash (B^+ \rightarrow \alpha^+) \rightarrow \beta^+ : \mathbf{Set}$. The goal follows by one more application of [PiSet].

Note that the application of the target rule [PiSet] works because the source rule (PiSet) requires $\beta : \mathsf{Set}$. If $\beta : \mathsf{Prop}$, we would obtain $(B^+ \rightarrow \alpha^+) \rightarrow \beta^+ : \mathbf{Prop}$, because the translation has turned the ultimate consequence into $\beta^+ : \mathbf{Prop}$. This would break the type preservation statement.

$\square$

5.2.4 Remarks

PTS with Delimited Control In their unpublished work, Boutillier and Herbelin [31] extend PTS with `shift` and `reset`. Unlike Dellina, their calculus allows control effects at *any* level, and has one single rule for function application where the argument is a pure term. What we found interesting is the formation rule of impure function types:

$$\frac{\begin{array}{cccc}\Gamma \vdash A : p & \Gamma, x : A \vdash B : r & \Gamma, x : A \vdash \alpha : s_1 & \Gamma, x : A \vdash \beta : s_2 \\ (p, r, s) \in \mathcal{R} & (r, s_1, q) \in \mathcal{R} & (q, s_2, o) \in \mathcal{R} & (p, o, s) \in \mathcal{R}\end{array}}{\Gamma \vdash \Pi\, x : A.\, B[\alpha, \beta] : s}$$

The rule has four premises on the sorts of types A, B, α, and β. The notation $(p, r, s) \in \mathcal{R}$ means (i) it is legal to form a function type whose domain and codomain reside in sorts p and r, respectively; and (ii) the resulting funtion type resides in sort s[5]. Hence, the latter three premises serve as the well-formedness certificates of $B \rightarrow \alpha$, $(B \rightarrow \alpha) \rightarrow \beta$, and $\Pi\, x : A.\, (B \rightarrow \alpha) \rightarrow \beta$. Now, we can see that the premises are a generalization of $\Gamma, \mathsf{x} : \mathsf{A} \vdash \rho : (\mathsf{s}', \mathsf{Prop}/\mathsf{Set}))$ and $\mathsf{s}' \in \{\mathsf{Prop}, \mathsf{Set}\}$ in rules (Pi-Prop) and (Pi-Set). That is, the premises ensure well-formedness of the CPS counterpart of the function type. Unfortunately, it is unclear what properties their calculus enjoys, and whether their translation preserves typing.

[5]The rules (PiProp), (PiSet), and (PiType) ensure $\mathsf{r} = \mathsf{s}$ for any well-formed Dellina function type. This is one of the requirements for a PTS to be *persistent* [22].

Different Formulations of Universes The formulation of the $\mathsf{Type_i}$ hierarchy presented here is called *Russell-style* universes, where a universe is the type of some type. An alternative choice is *Tarski-style* universes, where a universe is a type accompanied by an "interpretation" that allows viewing its inhabitants as types. More precisely, Tarski-style universes are introduced by the following rules:

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type_i} : \mathsf{Type}}\ (\textsc{Typei}) \qquad \frac{\Gamma \vdash A : \mathsf{Type_i}}{\Gamma \vdash T_i(A) : \mathsf{Type}}\ (\textsc{Ti})$$

The first rule states that any sort $\mathsf{Type_i}$ is a type, whereas the second rule says that any term A of type $\mathsf{Type_i}$ can be used as a type. We can view $\mathsf{Type_i}$ as a code for types and $T_i()$ as a decoding function [130].

If we incorporate Tarski-style universes into Dellina, we must make sure that we will never use effectful terms as types. This can be done by requiring a pure derivation in the premise of the (Ti) rule.

The two formulations of universes have their own pros and cons. Russell-style universes are easy to use, but they are not compatible with elimination rules of universe-polymorphic inductive types (which we do not support in Dellina) [116]. Tarski-style universes are more formal and enjoy nice properties when combined with inductive types, but the presentation is less intuitive.

Type-returning CPS Translation In Dellina, we only handle continuations that return a term-level expression, but there are situations where type-returning continuations would be useful. As Swamy et al. [157] show, *Dijkstra monads* [63] help us reason about effectful programs in dependently typed languages. Intuitively, Dijkstra monads express the behavior of a program by relating its pre- and post-conditions. Recently, Ahman et al. [1] found that it is possible to generate Dijkstra monads from user-defined monads via a CPS translation. The idea is to turn a computation into a predicate transformer: for instance, a state monad of type $s \to a * s$ is translated to a function of type $s \to (s * a \to \mathsf{Type}) \to \mathsf{Type}$, which takes in an input state and a post-condition predicate, and returns a pre-condition. Observe that the answer type of the CPS translation is Type, which stands for Prop or Set of Dellina. This is because pre- and post-conditions are type-returning functions. The CPS translation makes it possible to add new effects without manually specifying their Dijkstra-counterparts, solving one issue left open in the earlier

$$
\begin{array}{llll}
\text{Telescopes} & \Delta & ::= & () \mid (\mathsf{x} : \mathsf{A})\Delta \\
\text{Expression Lists} & \overline{\mathsf{e}} & ::= & () \mid \mathsf{e}\ \overline{\mathsf{e}} \\
\text{Signatures} & \Psi & ::= & \bullet \mid \Psi,\ \mathsf{Ind}(\mathsf{D} : \mathsf{A},\ \{\mathsf{c_i} : \mathsf{C_i}\}) \\
\text{Values} & \mathsf{V}, \mathsf{v} & ::= & \ldots \mid \mathsf{D}\ \overline{\mathsf{e}} \mid \mathsf{C}\ \overline{\mathsf{e}} \mid \mathsf{c}\ \overline{\mathsf{v}} \\
\text{Expressions} & \mathsf{s}, \mathsf{A}, \mathsf{e} & ::= & \ldots \mid \mathsf{c_i}\ \overline{\mathsf{e_i}} \\
 & & \mid & \mathsf{pm\ e\ as\ x\ in\ D}\ \overline{\mathsf{a}}\ \mathsf{ret\ P\ with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \to \mathsf{e_i}\}
\end{array}
$$

Figure 5.12: Syntax of Inductive Datatypes

version of the F* language [156]. On the other hand, since the answer type is a sort rather than a type, it is unclear what the logical interpretation of such a CPS translation would be.

5.3 Inductive Datatypes

With Prop, Set, and universes at hand, we are now ready to extend the language with user-defined inductive datatypes. This is a key feature to get the most of dependent types, because building correct-by-definition programs often requires fully-customized datatypes, which encode the exact specification of user data. Inductive data are manipulated by a generalized form of pattern matching that supports *large elimination.* This generalization lets us have non-term branches, making type-level computation as rich as term-level computation. The feature also comes with an interesting observation: when CPS translating a large elimination construct, selectiveness and the purity restriction again help us avoid a difficulty noticed by others [24].

5.3.1 Specification

5.3.1.1 Syntax

To accommodate inductive datatypes, we extend the syntax with several new categories (Figure 5.12). Among them, a *telescope* is a list of variable-type associations $(\mathsf{x} : \mathsf{A})$, which represent argument types of datatype constants and constructors. Unlike typing environments, telescopes are represented as cons-lists, and each A may depend on preceding variables. Telescopes are inhabited by *expression lists* $\overline{\mathsf{e}}$, which represent sequences of arguments to type constants and constructors. As an example, the three-element list $\mathsf{1\ 2\ (::\ 0\ 3\ nil)}$, which is a valid argument sequence for the $\mathsf{::}$-constructor of Dellina-, is an inhabitant of the telescope $(\mathsf{m} : \mathbb{N})(\mathsf{h} : \mathbb{N})(\mathsf{t} : \mathsf{L\ m})$.

We will use a telescope metavariable Δ both as a sequence of binding and as a sequence of variables. Here is a list of notational abbreviations found in the reduction and typing rules below; suppose $\Delta = (\mathsf{x}_1 : \mathsf{A}_1)(\mathsf{x}_2 : \mathsf{A}_2)\ldots(\mathsf{x}_n : \mathsf{A}_n)$ and $\bar{\mathsf{e}} = \mathsf{e}_1\ \mathsf{e}_2\ \ldots\ \mathsf{e}_n$:

$$\begin{aligned}
\Gamma, \Delta &\overset{\text{def}}{\equiv} \Gamma, \mathsf{x}_1 : \mathsf{A}_1, \mathsf{x}_2 : \mathsf{A}_2, \ldots, \mathsf{x}_n : \mathsf{A}_n \\
\Pi\,\Delta.\,\mathsf{B} &\overset{\text{def}}{\equiv} \Pi\,\mathsf{x}_1 : \mathsf{A}_1.\,\Pi\,\mathsf{x}_2 : \mathsf{A}_2.\ldots\Pi\,\mathsf{x}_n : \mathsf{A}_n.\,\mathsf{B} \\
\mathsf{e}\ \Delta &\overset{\text{def}}{\equiv} \mathsf{e}\ \mathsf{x}_1\ \mathsf{x}_2\ \ldots\ \mathsf{x}_n \\
\mathsf{e}[\bar{\mathsf{e}}/\Delta] &\overset{\text{def}}{\equiv} ((\mathsf{e}[\mathsf{e}_1/\mathsf{x}_1])[\mathsf{e}_2/\mathsf{x}_2])\ldots[\mathsf{e}_n/\mathsf{x}_n]
\end{aligned}$$

A *signature* Ψ is a sequence of inductive definitions. Each inductive definition is a pair of the form $\mathsf{Ind}(\mathsf{D} : \mathsf{A}, \{\mathsf{c}_i : \mathsf{C}_i\})$, where D is a datatype constant, A is the *arity* (type) of D, c_i is the i'th constructor of D, and C_i is the type of c_i. For instance, the definition of natural numbers and indexed lists looks like:

$\boxed{\text{Natural Numbers}}$

$$\mathsf{Ind}(\mathbb{N} : \mathsf{Set}, \{\mathsf{z} : \mathbb{N};\ \mathsf{suc} : \Pi\,\mathsf{n} : \mathbb{N}.\,\mathbb{N}\})$$

$\boxed{\text{Indexed Lists}}$

$$\mathsf{Ind}(\mathsf{L} : \Pi\,\mathsf{a} : \mathbb{N}.\,\mathsf{Set}, \{\mathsf{nil} : \mathsf{L\ z};\ \mathsf{::} : \Pi\,(\mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L\ m}).\,\mathsf{L}\ (\mathsf{suc\ m})\})$$

In general, an arity is a function type of the form $\Pi\,\Delta.\,\mathsf{s}$, where Δ can be empty. Each $\mathsf{a}_i : \mathsf{A}_i$ in Δ represents the i'th index of the datatype[6], and the conclusion s tells us in which universe the datatype resides. A constructor type is also a function

[6]For simplicity, we treat all arguments to type constants as indices, but in dependently typed languages, arguments are often classified into two separate sets: parameters and indices. The difference is that parameters are fixed in the conclusion of all constructor types, while indices can vary [68]. Consider the polymorphic vector type $\mathsf{Vec\ A\ n}$, which has two constructors $\mathsf{nil_v} : \mathsf{Vec\ A\ z}$ and $\mathsf{::_v} : \Pi\,(\mathsf{m} : \mathbb{N}, \mathsf{x} : \mathsf{A}, \mathsf{xs} : \mathsf{Vec\ A\ m}).\,\mathsf{Vec\ A}\ (\mathsf{suc\ m})$. Here, A is a parameter, because both constructor types have a conclusion of the form $\mathsf{Vec\ A\ n}$, whereas n is an index, since it is instantiated to z in the $\mathsf{nil_v}$ case and $\mathsf{suc\ m}$ in the $\mathsf{::_v}$ case. Note that uniformity of parameters is not mandatory in Agda; we refer the interested reader to Cockx [41] for details.

$\boxed{\text{Evaluation Contexts } \mathsf{E}, \mathsf{F}}$

$$\begin{array}{lcl} \mathsf{E} & ::= & \ldots \mid \mathsf{c}\ \overline{\mathsf{v}}\ \mathsf{E}\ \overline{\mathsf{e}} \mid \mathsf{pm}\ \mathsf{E}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \overline{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\} \\ \mathsf{F} & ::= & \ldots \mid \mathsf{c}\ \overline{\mathsf{v}}\ \mathsf{F}\ \overline{\mathsf{e}} \mid \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \overline{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\} \end{array}$$

$\boxed{\text{Reduction Rules } \mathsf{e}\ \triangleright\ \mathsf{e'}}$

$$\Gamma \vdash \mathsf{pm}\ \mathsf{c_i}\ \overline{\mathsf{v}}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \overline{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\} \quad \triangleright_\iota \quad \mathsf{e_i}[\overline{\mathsf{v}/\mathsf{y_i}}]$$

Figure 5.13: Reduction of Inductive Data

type $\Pi\,\Delta_i.\,\mathsf{D}\ \overline{\mathsf{u_i}}$, where $\overline{\mathsf{u_i}}$ are a sequence of indices specific to the i'th constructor. Thus, a single inductive definition in fact brings *a family of* types $\mathsf{D}\ \overline{\mathsf{u_i}}$ into the language.

Expressions are extended with type constant application, data constructor application, and a general form of pattern matching. Following Sjöberg et al. [151], we require both type constants and constructors to be fully applied, in order to simplify the proof of the cannonical forms lemma. A pattern matching construct now takes the form

$$\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \overline{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\}$$

As before, the construct tells us that we are matching e against patterns $\mathsf{c_i}\ \overline{\mathsf{y_i}}$, whose return clauses are specified by $\mathsf{e_i}$. The variable x is a placeholder for the scrutinee, which may appear in the return type P. The type $\mathsf{D}\ \overline{\mathsf{a}}$ tells us that e resides in the type family $\mathsf{D}\ \overline{\mathsf{u}}$. Since the indices $\overline{\mathsf{u}}$ vary by constructor, we generalize it to a sequence of variables $\overline{\mathsf{a}}$ when scrutinizing an expression. Note that the variables are allowed to occur free in P.

5.3.1.2 Reduction and Subtyping

We next define runtime evaluation of inductive data (Figure 5.13). Given a constructor application $\mathsf{c}\ \overline{\mathsf{e}}$, we evaluate the arguments $\overline{\mathsf{e}}$ one by one, from left to right. When all arguments have reduced to values $\overline{\mathsf{v}}$, we can perform ι-reduction, by substituting $\overline{\mathsf{v}}$ for pattern variables $\overline{\mathsf{y_i}}$ in branch $\mathsf{e_i}$.

With the new reduction rules, we can discuss equivalence between inductive data and their types, and lift it to a subtyping relation via the ($\preceq$-$\equiv$) rule. The

$\boxed{\text{Telescopes } \Gamma \vdash \Delta}$

$$\frac{\vdash \Gamma}{\Gamma \vdash ()}\ (\text{T-Empty}) \qquad \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash \Delta}{\Gamma \vdash (x : A)\Delta}\ (\text{T-Ext})$$

$\boxed{\text{Expression Lists } \Gamma \vdash \bar{e} : \Delta\ \rho}$

$$\frac{\vdash \Gamma}{\Gamma \vdash () : ()}\ (\text{L-Empty})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : A \quad \Gamma \vdash \bar{e} : \Delta[e/x]\ \rho[e/x] \\ \Gamma \vdash \Delta[e/x] \quad \Gamma \vdash \rho[e/x] : (s, s') \quad s, s' \in \{\text{Prop}, \text{Set}\}\end{array}}{\Gamma \vdash e\ \bar{e} : (x : A)\Delta\ \rho}\ (\text{L-DExt})$$

$$\frac{\Gamma \vdash e : A\ \rho \quad \Gamma \vdash \bar{e} : \Delta\ \sigma \quad \tau = \text{comp}(\rho, \sigma)}{\Gamma \vdash e\ \bar{e} : (x : A)\Delta\ \tau}\ (\text{L-NDExt})$$

Figure 5.14: Well-formed Telescopes and Expression Lists

reader might further expect a ($\preceq$-Pi)-like rule, which directly gives us a subtyping relation between datatypes of the form $D\ \bar{e}$. However, we are *not* going to add such a rule, because it would break fundamental properties such as preservation and canonicity [115]. The problem stems from the incompatibility between the present formulation of universes and the reduction rule of inductive data. To avoid these unfortunate consequences, we restrict ourselves to use only equivalence when reasoning about convertibility of datatypes.

5.3.1.3 Typing

Inductive datatypes are equipped with somewhat complex typing rules. Let us begin by telescope typing (Figure 5.14). Telescopes are typed with regard to an environment Γ. When consing a new binding $x : A$, we check that A is well-formed under Γ, and the telescope Δ to be extended is well-formed under $\Gamma, x : A$. From this typing rule, we can see that extending a telescope has the effect of closing free variables.

The judgment $\Gamma \vdash \overline{e} : \Delta\ \rho$ for expression lists carries an optional effect annotation, representing how evaluation of $\overline{e}$ changes the answer type. We find that there are two extension rules. The first one, (L-DExt), is for *dependent extension*. Observe that the top-most element e is used to close off the variable x in the telescope Δ. This substitution happens in our list-consing example, repeated below:

$$1\ 2\ (::\ 0\ 3\ \mathsf{nil}) : (m : \mathbb{N})(h : \mathbb{N})(t : L\ m)$$

The third element $::\ 0\ 3\ \mathsf{nil}$ has type $(L\ m)[1/m]$, which depends on the first element 1. Since we do not allow dependency on impure terms, rule (L-DExt) imposes a purity requirement on the newly added e, and requires well-formedness of the post-substitution telescope $\Delta[e/x]$ as well as effect annotation $\rho[e/x]$.

The other extension rule, (L-NDExt), is for *non-dependent extension*. Since no type-level substitution of e takes place, e is allowed to have a non-empty effect annotation ρ, but we require that ρ composes with the effect σ of $\overline{e}$. By repeatedly extending $\overline{e}$ in this way, we can guanratee that any well-typed expression list satisfies the chaining rule we described in Section 2.1.4, that is:

1. When e_i is not the last impure argument, $\alpha_i = \beta_j$, where e_j is the closest impure argument following e_i.

2. The initial answer type α of the whole list is α_i, where e_i is the last impure argument.

3. The final answer type β of the whole list is β_i, where e_i is the first impure argument.

We next elaborate signature extension (Figure 5.15). When declaring a new datatype D, we first check that the arity $\Pi\,\Delta.\,s$ is a well-formed function type. This type checking is done in an empty environment, but we implicitly allow Δ to refer to previously declared inductive types in Ψ. We similarly check well-formedness of each constructor type $\Pi\,\Delta_i.\,D\ \overline{u_i}$, with the assumption that $D : \Pi\,\Delta.\,s$. One important thing is that a constructor type can never be an impure function type, in other words, it cannot take the form $\Pi\,\Delta_i.\,D\ \overline{u_i}\ [\alpha, \beta]$. This is the case even if the type resides in Prop or Set. The reason is that constructors are constants, not

$\boxed{\text{Signatures} \vdash \Psi}$

$$\frac{}{\vdash \bullet}\ \text{(S-Empty)}$$

$$\frac{\vdash \Psi \quad \bullet \vdash \Pi\,\Delta.\,\mathsf{s} : \mathsf{s}' \quad \mathsf{s}, \mathsf{s}' \in \mathsf{S} \quad (\bullet,\, \mathsf{D} : \Pi\,\Delta.\,\mathsf{s} \vdash \Pi\,\Delta_i.\,\mathsf{D}\ \overline{\mathsf{u_i}} : \mathsf{s})_{i=1\ldots k} \qquad \mathsf{D},\, \mathsf{c_i}\ \text{fresh} \quad \text{safe}(\mathsf{D}, \Delta_i) \quad \text{no-rec-dep}(\Delta_i)}{\vdash \Psi,\, \mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_i.\,\mathsf{D}\ \overline{\mathsf{u_i}}\})}\ \text{(S-Ext)}$$

Figure 5.15: Well-formed Signatures

functions. More specifically, an application of a constructor c to values $\overline{\mathsf{v}}$ is always a pure value $\mathsf{c}\ \overline{\mathsf{v}}$, while an application of a function $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}$ to a value v can reduce to an effectful computation $\mathsf{e}[\mathsf{v}/\mathsf{x}]$.

The second line of (S-Ext) has three requirements on the names and constructor types. The freshness condition simply requires that D and $\mathsf{c_i}$ are not used by the previously defined datatypes in Ψ. The safety condition safe(D, Δ_i) is a slightly stronger version of the *strict positivity condition* [51]. Strict positivity requires that, for every functional $\mathsf{B_i}$ in Δ_i, D does not appear in the premise of $\mathsf{B_i}$. For instance, when $\mathsf{B_i} = \Pi\,\mathsf{x} : \mathsf{B_1}.\,\mathsf{B_2}$, D should not appear in $\mathsf{B_1}$. This condition is mandatory for the language to be logically consistent: if we allowed negative occurrences of D, we can easily construct a non-terminating program without using recursion[7]. Our safety condition extends strict positivity with the following clause: when $\mathsf{B_i} = \Pi\,\mathsf{x} : \mathsf{B_1}.\,\mathsf{B_2}[\alpha, \beta]$, D does not appear in $\mathsf{B_1}$, $\mathsf{B_2}$, and α. We need this extra restriction because the CPS translation does not preserve strict positivity[8]. Recall that we translate an impure function type $\Pi\,\mathsf{x} : \mathsf{B_1}.\,\mathsf{B_2}[\alpha, \beta]$ to

[7]Here is a classic example:

```
(* a non-strictly positive datatype *)
type D = Bad : (D -> D) -> D

(* omega : D -> D *)
let omega (Bad f) = f (Bad f) in

(* this runs forever!  *)
omega (Bad omega)
```

[8]If our translation was unselective or call-by-name, and do not wish to sacrifice consistency of

$\mathbf{\Pi\, x} : \mathsf{B_1}^+ . (\mathsf{B_2}^+ \rightarrow \alpha^+) \rightarrow \beta^+$. In the CPS image, the positive occurrences of $\mathsf{B_2}$ and α have turned into negative ones, which means we have to give up either the guarantee that every Dellina inductive type has a CPS counterpart, or the consistency of the target language (that is, we turn off positivity check to make the CPS image acceptable).

The last condition no-rec-dep(Δ_i) ensures that no type in Δ_i depends on the recursive arguments of the constructor.

We will assume that there is a top-level, well-formed signature Ψ_0 containing all definitions used in the user program, and keep it implicit in most typing rules.

Having seen rules for declaring datatypes, we now look at rules for constructing and destructing data (Figure 5.16). The formation rule of inductive datatypes, (Data), is a generalization of (T-List) from Section 3.3. When D is a datatype constant of type $\Pi\,\Delta.\,\mathsf{s}$, and $\overline{\mathsf{e}}$ is an expression list inhabiting the telescope Δ, we can form a datatype $\mathsf{D}\ \overline{\mathsf{e}}$. Note that all arguments in $\overline{\mathsf{e}}$ must be free from control effects, since they appear in the type we are building. We also need the premise $\vdash\ \Gamma$ when $\overline{\mathsf{e}}$ is empty, otherwise we would not be able to prove environment regularity.

Constructor application has two rules. The first one (ConstrP) derives a pure datum: it requires a list of pure arguments $\overline{\mathsf{e}}$, and uses those arguments to close off the free variables in the result type $\mathsf{D}\ \overline{\mathsf{u_i}}$. The second rule (ConstrI) derives an impure datum: it accepts an impure expression list, as long as its impure elements appear in a non-dependent position of the constructor type. Dependency on a constructor argument can occur in two places: the type of later arguments, and the result type of the whole constructor application. Well-formedness of argument types are guaranteed by the well-typedness precondition of $\overline{\mathsf{e}}$; remember that dependent extension is only allowed when the expression to be added is pure. To make sure that the result type is also well-formed, we check that substitution of all pure expressions in $\overline{\mathsf{e}}$ closes off all free variables in $\overline{\mathsf{u_i}}$. In the typing rule, this requirement is written as $\Gamma\ \vdash\ \mathsf{D}\ \overline{\mathsf{u_i}}[\overline{\mathsf{e}}/\Delta_i]_p : \mathsf{s}$, where the substitution operation

the target language, we need more restrictions in the source language to maintain type preservation. For instance, Cong and Asai [45], who define an unselective, call-by-value CPS translation for a Dellina-like language, prohibit occurrences of D in the co-domain of pure function types. The reason is that their translation converts $\Pi\,\mathsf{x} : \mathsf{B_1}.\,\mathsf{B_2}$ into $\mathbf{\Pi\, x} : \mathsf{B_1}^+ .\, \mathbf{\Pi}\,\alpha : *. (\mathsf{B_2}^+ \rightarrow \alpha) \rightarrow \alpha$, where $\mathsf{B_2}^+$ occurs negatively. Call-by-name translations behave even worse: they insert double negation into domains as well, which means any recursive definition is turned into a non-strictly positive one.

$\boxed{\text{Expressions } \Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho}$

$$\frac{\vdash \Gamma \text{ if } \mathsf{e} = () \qquad \mathsf{D} : \Pi\,\Delta.\,\mathsf{s} \in \Psi_0 \qquad \Gamma \vdash \overline{\mathsf{e}} : \Delta}{\Gamma \vdash \mathsf{D}\ \overline{\mathsf{e}} : *}\ (\textsc{Data})$$

$$\frac{\begin{array}{c}\mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma \vdash \overline{\mathsf{e}} : \Delta_\mathsf{i} \qquad \Gamma \vdash \mathsf{D}\ \overline{\mathsf{u_i}}[\overline{\mathsf{e}}/\Delta_\mathsf{i}] : \mathsf{s}\end{array}}{\Gamma \vdash \mathsf{c_i}\ \overline{\mathsf{e}} : \mathsf{D}\ \overline{\mathsf{u_i}}[\overline{\mathsf{e}}/\Delta_\mathsf{i}]}\ (\textsc{ConstrP})$$

$$\frac{\begin{array}{c}\mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma \vdash \overline{\mathsf{e}} : \Delta_\mathsf{i}\ \rho \qquad \Gamma \vdash \mathsf{D}\ \overline{\mathsf{u_i}}[\overline{\mathsf{e}}/\Delta_\mathsf{i}]_p : \mathsf{s} \qquad \mathsf{s} \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash \mathsf{c_i}\ \overline{\mathsf{e}} : \mathsf{D}\ \overline{\mathsf{u_i}}[\overline{\mathsf{e}}/\Delta_\mathsf{i}]_p\ \rho}\ (\textsc{ConstrI})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{D}\ \overline{\mathsf{u}} \qquad \mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma,\ \Delta,\ \mathsf{x} : \mathsf{D}\ \Delta \vdash \mathsf{P} : \mathsf{s}' \qquad \text{elim-ok}(\mathsf{s}, \mathsf{s}') \\ (\Gamma,\ \Delta_\mathsf{i} \vdash \mathsf{e_i} : \mathsf{P}[\overline{\mathsf{u_i}}/\Delta, \mathsf{c_i}\ \overline{\mathsf{y_i}}/\mathsf{x}]\ \rho[\overline{\mathsf{u_i}}/\Delta, \mathsf{c_i}\ \overline{\mathsf{y_i}}/\mathsf{x}])_{i=1\ldots k} \\ \Gamma \vdash \mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}] : \mathsf{s}' \qquad \Gamma \vdash \rho[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ D}\ \Delta\ \mathsf{ret\ P\ with}\ \{\mathsf{c_i}\ \overline{\Delta_\mathsf{i}} \to \mathsf{e_i}\} : \mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]\ \rho[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]}\ (\textsc{DMatch})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{D}\ \overline{\mathsf{u}}\ \rho \qquad \mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma \vdash \mathsf{P} : \mathsf{s}' \qquad \text{elim-ok}(\mathsf{s}, \mathsf{s}') \qquad \mathsf{s}, \mathsf{s}' \in \{\mathsf{Prop}, \mathsf{Set}\} \\ (\Gamma,\ \Delta_\mathsf{i} \vdash \mathsf{e_i} : \mathsf{P}\ \sigma)_{i=1\ldots k} \qquad \tau = \text{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ D}\ \Delta\ \mathsf{ret\ P\ with}\ \{\mathsf{c_i}\ \overline{\Delta_\mathsf{i}} \to \mathsf{e_i}\} : \mathsf{P}\ \tau}\ (\textsc{NDMatch})$$

Figure 5.16: Typing Rules for Inductive Data

replaces all free variables $\mathsf{y_j}$ in Δ_i by the corresponding $\mathsf{e_j}$ when $\mathsf{e_j}$ is pure (the *p*-subscript means that the operation is substitution of *pure* terms). The last premise $\mathsf{s} \in \{\mathsf{Prop}, \mathsf{Set}\}$ ensures that impure data are only present in the bottom-level.

Rules for pattern matching are essentially a generalization of those for Dellina-pattern matching. In (DMatch), where the scrutinee is pure, the return type P and effect annotation ρ may depend on indices $\overline{\mathsf{u}}$ and the scrutinee e. In (ND-Match), we do not allow such dependency. What is new here is that, the type $\mathsf{D}\ \overline{\mathsf{u}}$ of the scrutinee e, as well as the type P of the branches $\mathsf{e_i}$, may reside in arbitrary universe levels s and $\mathsf{s'}$. However, not every combination of s and $\mathsf{s'}$ is allowed in the language; they must be related by the elim-ok(_, _) predicate defined below:

- elim-ok($\mathsf{Prop}, \mathsf{Prop}$)
- elim-ok($\mathsf{s}, \mathsf{s'}$) where $\mathsf{s} \in \{\mathsf{Set}, \mathsf{Type_i}\}$ and $\mathsf{s'} \in \mathsf{S}$

We find that elimination from Set and $\mathsf{Type_i}$ can result in an expression at any universe level, but elimination from Prop is only allowed when the branches are Prop-terms. The reason comes from the different nature of the three universes: while we distinguish two inhabitants of the same datatype, we identify all inhabitants of the same proposition, since all we care about proofs is whether they exist or not. If we allowed elimination from Prop to other universes, we would be able to use a proof differently depending on how it looks like. What this means is that it is no longer safe to erase proofs during program extraction of Coq [112], and in the presence of classical axioms, we would further run into logical inconsistency [48].

5.3.2 Metatheory

The metatheory of inductive datatypes is slightly more involved than that of the λ-calculus, because we have multi-ary constructors. We first refine the substitution lemma in order to handle a sequence of variables:

Lemma 5.3.1 (Substitution)**.** *Suppose* $\Gamma \vdash \overline{\mathsf{v}} : \Delta$*. Then, the following hold.*

1. *If* $\vdash \Gamma, \Delta, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[\overline{\mathsf{v}}/\Delta]$.

2. If $\Gamma, \Delta, \Gamma' \vdash t : T\ \rho$*, then* $\Gamma, \Gamma'[\overline{v}/\Delta] \vdash t[\overline{v}/\Delta] : T[\overline{v}/\Delta]\ \rho[\overline{v}/\Delta]$.

We next extend the regularity lemma with an additional clause for expression lists. The statement easily follows by the extension rules and the induction hypothesis.

Lemma 5.3.2 (Regularity)**.** *If* $\Gamma \vdash \overline{e} : \Delta\ \rho$*, then for each* $e_i \in \overline{e}$*, there exist some* A_i *and* ρ_i *satisfying either of the following:*

- $\Gamma \vdash e_i : A_i$ *and* $\Gamma \vdash A_i : s$
- $\Gamma \vdash e_i : A_i\ \rho_i$*,* $\Gamma \vdash A_i : s$*,* $\Gamma \vdash \rho_i : (s', s)$*, and* $s, s' \in \{\mathsf{Prop}, \mathsf{Set}\}$

Then, we can prove type soundness by generalizing the datatypes cases of the Dellina- proofs.

5.3.3 CPS Translation

Compared to the formalization of the source language, it is easier to scale the CPS translation to the datatype fragment. Let us go through the definition in Figures 5.17 – 5.18. Telescopes are translated the same way as typing environments: we simply map the value translation to all types. On the other hand, we cannot define a translation for expression lists, because they may contain impure terms, in which case the translation is not a simple mapping of $^+$ and $^\div$.

Inductive datatypes D are converted into target datatypes $\mathbf{D}$ that receive the same number of indices and have the same number of constructors. The type of indices and constructor arguments are however different in general: they are all applied the value translation.

The translation of a constructor application $c_i\ \overline{e}$ generalizes that of Dellina-indexed lists $::\ e_0\ e_1\ e_2$. Since the number of the arguments is not fixed, we can no longer list all possible variants of the translation, but we still have a simple recipe that accounts for the general case. When all arguments are pure, we translate the application to $\mathbf{c_i}\ \overline{e^+}$. When $\overline{e}$ has at least one impure term, we define the CPS image as a computation starting with $\lambda\mathbf{k}$. The body of this function is a sequence of applications $e_{i_1}{}^\div\ (\lambda\, \mathbf{v_{i_1}}.\, e_{i_2}{}^\div\ (\lambda\, \mathbf{v_{i_2}}. \ldots))$, where $e_{i_1}, e_{i_2} \ldots$ are impure arguments in e. After mapping the $^\div$-translation to every impure argument, we return the value

$$\frac{\vdash \Gamma}{\Gamma \vdash ()}\ (\textsc{T-Empty}) \overset{+}{\rightsquigarrow} ()$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash \Delta}{\Gamma \vdash (x : A)\Delta}\ (\textsc{T-Ext}) \overset{+}{\rightsquigarrow} (\mathbf{x} : A^{+})\Delta^{+}$$

$$\frac{}{\vdash \bullet}\ (\textsc{S-Empty}) \overset{+}{\rightsquigarrow} \{\}$$

$$\frac{\vdash \Psi \quad \bullet \vdash \Pi\,\Delta.\,s : s' \quad s, s' \in S \quad (\bullet, D : \Pi\,\Delta.\,s \vdash \Pi\,\Delta_i.\,D\ \overline{u_i} : s)_{i=1\ldots k}}{\vdash \Psi,\ \mathsf{Ind}(D : \Pi\,\Delta.\,s,\ \{c_i : \Pi\,\Delta_i.\,D\ \overline{u_i}\})}\ (\textsc{S-Ext})$$

$$\overset{+}{\rightsquigarrow}\ \Psi^{+},\ \mathsf{Ind}(\mathbf{D} : \mathbf{\Pi}\,\Delta^{+}.\,s^{+},\ \{\mathbf{c_i} : \mathbf{\Pi}\,\Delta_i{}^{+}.\,\mathbf{D}\ \overline{u_i{}^{+}}\})$$

$$\frac{\vdash \Gamma \quad D : \Pi\,\Delta.\,s \in \Psi_0 \quad \Gamma \vdash \overline{e} : \Delta}{\Gamma \vdash D\ \overline{e} : s}\ (\textsc{Data}) \overset{+}{\rightsquigarrow} \mathbf{D}\ \overline{e^{+}}$$

$$\frac{\begin{array}{c}\vdash \Gamma \text{ if } e = () \quad \mathsf{Ind}(D : \Pi\,\Delta.\,s,\ \{c_i : \Pi\,\Delta_i.\,D\ \overline{u_i}\}) \in \Psi_0 \\ \Gamma \vdash \overline{e} : \Delta_i \quad \Gamma \vdash D\ \overline{u_i}[\overline{e}/\Delta_i] : s\end{array}}{\Gamma \vdash c_i\ \overline{e} : D\ \overline{u_i}[\overline{e}/\Delta_i]}\ (\textsc{ConstrP}) \overset{+}{\rightsquigarrow} \mathbf{c_i}\ \overline{e^{+}}$$

$$\frac{\begin{array}{c}\vdash \Gamma \text{ if } e = () \quad \mathsf{Ind}(D : \Pi\,\Delta.\,s,\ \{c_i : \Pi\,\Delta_i.\,D\ \overline{u_i}\}) \in \Psi_0 \\ \Gamma \vdash \overline{e} : \Delta_i\ \rho \quad \Gamma \vdash D\ \overline{u_i}[\overline{e}/\Delta_i]_p : s \quad s \in \{\mathsf{Prop}, \mathsf{Set}\}\end{array}}{\Gamma \vdash c_i\ \overline{e} : D\ \overline{u_i}[\overline{e}/\Delta_i]_p\ \rho}\ (\textsc{ConstrI})$$

$$\begin{array}{l}\overset{\div}{\rightsquigarrow}\ \lambda\,\mathbf{k} : (D\ \overline{u}[\overline{e}/\Delta_i]_p)^{+} \rightarrow \alpha^{+}. \\ \quad e_{i_1}{}^{\div}\ (\lambda\,\mathbf{v_{i_1}} : B_{i_1}{}^{+}.\ e_{i_2}{}^{\div}\ (\lambda\,\mathbf{v_{i_2}} : B_{i_2}{}^{+}.\ldots\ \mathbf{k}\ (\mathbf{c_i}\ \overline{\mathrm{cpsarg}(e)}))) \\ \text{if } \rho = [\alpha, \beta]\end{array}$$

Figure 5.17: CPS Translation of Signatures and Inductive Data

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{D}\ \overline{\mathsf{u}} \qquad \mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma,\ \Delta,\ \mathsf{x} : \mathsf{D}\ \overline{\Delta} \vdash \mathsf{P} : \mathsf{s'} \qquad \text{elim-ok}(\mathsf{s}, \mathsf{s'}) \\ (\Gamma,\ \Delta_\mathsf{i} \vdash \mathsf{e_i} : \mathsf{P}[\overline{\mathsf{u_i}}/\Delta, \mathsf{c_i}\ \overline{\mathsf{y_i}}/\mathsf{x}]\ \rho[\overline{\mathsf{u_i}}/\Delta, \mathsf{c_i}\ \overline{\mathsf{y_i}}/\mathsf{x}])_{i=1\ldots k} \\ \Gamma \vdash \mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}] : \mathsf{s'} \qquad \Gamma \vdash \rho[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ D}\ \Delta\ \mathsf{ret\ P\ with}\ \{\mathsf{c_i}\ \overline{\Delta_\mathsf{i}} \to \mathsf{e_i}\} : \mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]\ \rho[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}]}\ (\text{DMatch})$$

$$\overset{+}{\leadsto}\ \mathbf{pm}\ \mathsf{e}^+\ \mathbf{as\ x\ in\ D}\ \boldsymbol{\Delta}\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with}\ \{\mathbf{c_i}\ \boldsymbol{\Delta_\mathbf{i}} \to \mathsf{e_i}^+\}$$
if $\rho = \epsilon$

$$\overset{\div}{\leadsto}\ \lambda\,\mathbf{k} : (\mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}])^+ \to (\alpha[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}])^+.$$
$$\mathbf{pm}\ \mathsf{e}^+\ \mathbf{as\ x\ in\ D}\ \boldsymbol{\Delta}\ \mathbf{ret}\ \beta^+\ \mathbf{with}\ \{\mathbf{c_i}\ \boldsymbol{\Delta_\mathbf{i}} \to \mathsf{e_i}^\div\ \mathbf{k}\}$$
if $\rho = [\alpha, \beta]$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{D}\ \overline{\mathsf{u}}\ \rho \qquad \mathsf{Ind}(\mathsf{D} : \Pi\,\Delta.\,\mathsf{s},\ \{\mathsf{c_i} : \Pi\,\Delta_\mathsf{i}.\,\mathsf{D}\ \overline{\mathsf{u_i}}\}) \in \Psi_0 \\ \Gamma \vdash \mathsf{P} : \mathsf{s'} \qquad \text{elim-ok}(\mathsf{s}, \mathsf{s'}) \qquad \mathsf{s}, \mathsf{s'} \in \{\mathsf{Prop}, \mathsf{Set}\} \\ (\Gamma,\ \Delta_\mathsf{i} \vdash \mathsf{e_i} : \mathsf{P}\ \sigma)_{i=1\ldots k} \qquad \tau = \text{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ D}\ \Delta\ \mathsf{ret\ P\ with}\ \{\mathsf{c_i}\ \overline{\Delta_\mathsf{i}} \to \mathsf{e_i}\} : \mathsf{P}\ \tau}\ (\text{NDMatch})$$

$$\overset{\div}{\leadsto}\ \lambda\,\mathbf{k} : \mathsf{P}^+ \to \alpha^+.$$
$$\mathsf{e}^\div\ (\lambda\,\mathbf{v} : \mathbf{D}\ \overline{\mathsf{u}^+}.\,\mathbf{pm\ v\ as\ \_\ in\ D}\ \boldsymbol{\Delta}\ \mathbf{ret}\ \alpha^+\ \mathbf{with}\ \{\mathbf{c_i}\ \boldsymbol{\Delta_\mathbf{i}} \to \mathbf{k}\ \mathsf{e_i}^+\})$$
if $\rho = [\alpha, \beta], \sigma = \epsilon$

$$\overset{\div}{\leadsto}\ \lambda\,\mathbf{k} : \mathsf{P}^+ \to \alpha^+.$$
$$\mathsf{e}^\div\ (\lambda\,\mathbf{v} : \mathbf{D}\ \overline{\mathsf{u}^+}.\,\mathbf{pm\ v\ as\ \_\ in\ D}\ \boldsymbol{\Delta}\ \mathbf{ret}\ \beta^+\ \mathbf{with}\ \{\mathbf{c_i}\ \boldsymbol{\Delta_\mathbf{i}} \to \mathsf{e_i}^\div\ \mathbf{k}\})$$
if $\rho = [\beta, \gamma], \sigma = [\alpha, \beta]$

Figure 5.18: CPS Translation of Inductive Data

$\mathbf{c_i}$ $\overline{\text{cpsarg}(\mathsf{e})}$ to the continuation $\mathbf{k}$. Here, the cpsarg() function turns a constructor argument into its CPS value. When the argument is pure, the function gives us a $^+$-translated argument. Otherwise, it returns a variable $\mathbf{v_i}$, which comes from the CPS pattern $\mathsf{e_i}^{\div}$ $(\lambda\,\mathbf{v_i}.\,\mathbf{e})$. For instance, if $\bar{\mathsf{e}} = \mathsf{e_1}\ \mathsf{e_2}\ \mathsf{e_3}$, where $\mathsf{e_1}$ and $\mathsf{e_3}$ are pure but $\mathsf{e_2}$ is impure, $\overline{\text{cpsarg}(e)} = \mathsf{e_1}^+\ \mathbf{v_2}\ \mathsf{e_3}^+$.

We now show that the CPS translation preserves typing. First, we check the ι-reduction case of the computational soundness property.

Proof. Suppose we reduce a pattern matching construct in the following way:

$$\mathsf{pm}\ \mathsf{c_i}\ \bar{\mathsf{v}}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \bar{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\}\ \triangleright_p\ \mathsf{e_i}[\overline{\mathsf{v}/\mathsf{y_i}}]$$

where the branches $\mathsf{e_i}$ are impure terms that change the answer type from α to β.

$$\begin{aligned}
&(\mathsf{pm}\ \mathsf{c_i}\ \bar{\mathsf{v}}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{D}\ \bar{\mathsf{a}}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \{\mathsf{c_i}\ \overline{\mathsf{y_i}} \rightarrow \mathsf{e_i}\})^+ && \\
&= \lambda\,\mathbf{k}.\,\mathbf{pm}\ \mathbf{c_i}\ \overline{\mathsf{v}^+}\ \mathbf{as}\ \mathbf{x}\ \mathbf{in}\ \mathbf{D}\ \bar{\mathbf{a}}\ \mathbf{ret}\ \beta^+\ \mathbf{with}\ \{\mathbf{c_i}\ \overline{\mathbf{y_i}} \rightarrow \mathsf{e_i}^{\div}\ \mathbf{k}\} && \text{by translation}\\
&\triangleright_\iota\ \lambda\,\mathbf{k}.\,(\mathsf{e_i}^{\div}\ \mathbf{k})[\overline{\mathsf{v}^+/\mathbf{y_i}}] && \\
&\equiv \mathsf{e_i}^{\div}[\overline{\mathsf{v}^+/\mathbf{y_i}}] && \text{by } \eta\\
&= (\mathsf{e_i}[\overline{\mathsf{v}/\mathsf{y_i}}])^{\div} && \text{by compositionality}
\end{aligned}$$

□

Thanks to the purity restriction, as well as the safety condition, the type preservation property of the CPS translation scales to the datatype fragment. Before presenting the proof, let us draw the reader's attention to two typing rules of the target language (Figure 5.19), which differ from their Dellina counterpart. In (S-Ext), we find that the safety condition safe$(\mathsf{D}, \Delta_\mathsf{i})$ has been replaced by the strict positivity condition str-pos$(\mathbf{D}, \boldsymbol{\Delta}_\mathbf{i})$. This is because we have no impure function type in the target language. In (DMatch), we see that type checking of branches $\mathbf{e_i}$ is done with a sequence of equivalence assumptions $\overline{\mathbf{u} \equiv \mathbf{u_i}}$ about indices, as well as an assumption $\mathbf{e} \equiv \mathbf{c_i}\ \overline{\mathbf{y_i}}$ about the scrutinee. As we saw earlier, these assumptions are necessary for the CPS translation to be type-preserving.

With these new rules in mind, we prove the type preservation property. The statement now has the following additional clauses accounting for signatures and

$$\frac{\vdash \mathbf{\Psi} \quad \bullet \vdash \mathbf{\Pi\,\Delta.\,s : s'} \quad \mathbf{s, s' \in S} \quad (\bullet, \mathbf{D : \Pi\,\Delta.\,s \vdash \Pi\,\Delta_i.\,D\ \overline{u_i} : s})_{i=1\ldots k} \qquad \mathbf{D}, \mathbf{c_i}\ \text{fresh} \quad \text{str-pos}(\mathbf{D}, \mathbf{\Delta_i}) \quad \text{no-rec-dep}(\mathbf{\Delta_i})}{\vdash \mathbf{\Psi}, \mathsf{Ind}(\mathbf{D : \Pi\,\Delta.\,*}, \{\mathbf{c_i : \Pi\,\Delta_i.\,D\ \overline{u_i}}\})}\ [\text{S-Ext}]$$

$$\frac{\mathbf{\Gamma \vdash e : D\ \overline{u}} \quad \mathsf{Ind}(\mathbf{D : \Pi\,\Delta.\,s}, \{\mathbf{c_i : \Pi\,\Delta_i.\,D\ \overline{u_i}}\}) \in \mathbf{\Psi_0} \qquad \mathbf{\Gamma, \Delta, x : D\ \Delta \vdash P : s'} \quad \text{elim-ok}(\mathbf{s}, \mathbf{s'}) \qquad (\mathbf{\Gamma, \Delta_i, \overline{u} \equiv \overline{u_i}, e \equiv c_i\ \overline{y_i} \vdash e_i : P[\overline{u_i}/\Delta, c_i\ \overline{y_i}/x]})_{i=1\ldots k}}{\mathbf{\Gamma \vdash pm\ e\ as\ x\ in\ D\ \Delta\ ret\ P\ with\ \{c_i\ \overline{\Delta_i} \to e_i\} : P[\overline{u}/\Delta, e/x]}}\ [\text{DMatch}]$$

Figure 5.19: Target Signature Extension and Pattern Matching

telescopes:

1. If $\vdash \Psi$, then $\vdash \Psi^+$.
2. If $\Gamma \vdash \Delta$, then $\Gamma^+ \vdash \Delta^+$.

Proof. We show three key cases: signature extension, constructor application, and pattern matching.

Case 1: (S-Ext)
Our goal is to show

$$\vdash \Psi^+, \mathsf{Ind}(\mathbf{D} : \mathbf{\Pi}\,\Delta^+.\,\mathsf{s}^+, \{\mathbf{c_i} : \mathbf{\Pi}\,\Delta_i{}^+.\,\mathbf{D}\ \overline{\mathsf{u_i}{}^+}\})$$

By the induction hypothesis, we have

$\vdash \Psi^+$, $\bullet \vdash \mathbf{\Pi}\,\Delta^+.\,\mathsf{s}^+ : \mathsf{s'}^+$, and $(\bullet, \mathbf{D} : \mathbf{\Pi}\,\Delta^+.\,\mathsf{s}^+ \vdash \mathbf{\Pi}\,\Delta_i{}^+.\,\mathbf{D}\ \overline{\mathsf{u_i}{}^+} : \mathsf{s}^+)_{i=1\ldots k}$

What remains to be shown is:

1. $\mathsf{s}^+, \mathsf{s'}^+ \in \mathbf{Set}$
2. $\mathbf{D}, \mathbf{c_i}$ fresh

3. str-pos($\mathbf{D}, \Delta_i{}^+$)

4. no-rec-dep($\Delta_i{}^+$)

Items 1, 2, 4 are trivially satisfied. Item 3 is guaranteed by the safety condition of Dellina: if Δ has an impure function type $\Pi\, x : B_1 . B_2[\alpha, \beta]$, Δ^+ will have $\mathbf{\Pi\, x} : B_1{}^+ . (B_2{}^+ \rightarrow \alpha^+) \rightarrow \beta^+$, but we know that $\mathbf{D}$ does not appear in $B_1{}^+$, $B_2{}^+$, or α^+.

Case 2: (ConstrI) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\begin{aligned}\Gamma^+ \vdash\ & \lambda\,\mathbf{k} : \mathbf{D}\ \overline{u_i{}^+}[\overline{e^+}/\mathbf{\Delta_i}]_p \rightarrow \alpha^+ . \\ & \quad e_{i_1}{}^{\div}\ (\lambda\,\mathbf{v_{i_1}} : B_{i_1}{}^+ . e_{i_2}{}^{\div}\ (\lambda\,\mathbf{v_{i_2}} : B_{i_2}{}^+ . \ldots\ \mathbf{k}\ (\mathbf{c_i}\ \overline{\text{cpsarg}(e)}))) : \\ & (\mathbf{D}\ \overline{u_i{}^+}[\overline{e}/\mathbf{\Delta_i}]_p \rightarrow \alpha^+) \rightarrow \beta^+\end{aligned}$$

It suffices to show that the last application $\mathbf{k}\ (\mathbf{c_i}\ \overline{\text{cpsarg}(e_i)})$ is well-typed. By the source typing rule (ConstrI), we know that only pure arguments e_{p_i}'s in $\overline{e}$ are substituted into the result type $D\ \overline{u_i}$; for all impure $e_{i_1}, e_{i_2}, \ldots$ in e_i, it must be the case that the variables $y_{i_1}, y_{i_2}, \ldots \in \Delta_i$ do not occur free in $D\ \overline{u_i}$. This means, the result type of $\mathbf{c_i}\ \overline{\text{cpsarg}(e)}$ only depends on $e_{p_i}{}^+$; it can never contain variables $\mathbf{v_{i_1}}, \mathbf{v_{i_2}}, \ldots$ introduced by the translation. Thus we conclude that the last application is type safe. Well-typedness of other applications follows by the induction hypothesis on impure arguments.

Case 3:
(DMatch) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\begin{aligned}& \lambda\,\mathbf{k} : (P[\overline{u}/\Delta, e/x])^+ \rightarrow (\alpha[\overline{u}/\Delta, e/x])^+ . \\ & \qquad \mathbf{pm}\ e^+\ \mathbf{as\ x\ in\ D\ \Delta\ ret}\ \beta^+\ \mathbf{with}\ \{\mathbf{c_i}\ \mathbf{\Delta_i} \rightarrow e_i{}^{\div}\ \mathbf{k}\}\end{aligned}$$

has type

$$((P[\overline{u}/\Delta, e/x])^+ \rightarrow (\alpha[\overline{u}/\Delta, e/x])^+) \rightarrow (\beta[\overline{u}/\Delta, e/x])^+$$

It suffices to show that the application $\mathsf{e_i}^{\div}\ \mathbf{k}$ is well-typed. By compositionality, we know that $(\mathsf{P}[\overline{\mathsf{u}}/\Delta, \mathsf{e}/\mathsf{x}])^{+} = \mathsf{P}^{+}[\overline{\mathbf{u}}/\Delta^{+}, \mathsf{e}^{+}/\mathbf{x}]$. Analogously to the (E-DMatchL) case from Section 4.6, $\mathsf{e_i}^{\div}$ demands a continuation whose domain depends on $\overline{\mathsf{u}^{+}}$ and e^{+}, while the domain of $\mathbf{k}$ depends on $\mathbf{\Delta}$ and $\mathbf{c_i}\ \mathbf{\Delta_i}$. Here the equivalence assumptions of the target typing rule come to the rescue: with $\overline{\mathsf{u}^{+}} \equiv \mathbf{\Delta}$ and $\mathsf{v} \equiv \mathbf{c_i}\ \mathbf{\Delta_i}$, we can conclude that the application $\mathsf{e_i}^{\div}\ \mathbf{k}$ is type-safe. The rest of the proof is completely straightforward.

$\square$

5.3.4 Encoding of Σ Types

Using dependent function types and inductive types, we can encode strong Σ types. For instance, the following pair defines a variant of Σ type that can express the existence of a datum x satisfying some predicate P:

$$\begin{aligned}&\mathsf{Ind}(\mathsf{Sigma} : \Pi\,\mathsf{A} : \mathsf{Set}.\,(\mathsf{A} \to \mathsf{Prop}) \to \mathsf{Prop},\\ &\quad \{\mathsf{exist} : \Pi\,\mathsf{A} : \mathsf{Set}.\,\Pi\,\mathsf{P} : (\mathsf{A} \to \mathsf{Prop}).\,\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{P}\ \mathsf{x} \to \mathsf{Sigma}\ \mathsf{A}\ \mathsf{P}\})\end{aligned}$$

The first and second projections are simply functions that destruct pairs constructed via exist:

$$\begin{aligned}\mathsf{fst} \overset{\text{def}}{\equiv}\ & \lambda\,\mathsf{A} : \mathsf{Set}.\,\lambda\,\mathsf{P} : \mathsf{A} \to \mathsf{Prop}.\,\lambda\,\mathsf{e} : \mathsf{Sigma}\ \mathsf{A}\ \mathsf{P}.\\ & \quad \mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{Sigma}\ \mathsf{A}\ \mathsf{P}\ \mathsf{ret}\ \mathsf{A}\ \mathsf{with}\ \mathsf{exist}\ \mathsf{A}\ \mathsf{P}\ \mathsf{a}\ \mathsf{b} \to \mathsf{a}\\ \mathsf{snd} \overset{\text{def}}{\equiv}\ & \lambda\,\mathsf{A} : \mathsf{Set}.\,\lambda\,\mathsf{P} : \mathsf{A} \to \mathsf{Prop}.\,\lambda\,\mathsf{e} : \mathsf{Sigma}\ \mathsf{A}\ \mathsf{P}.\\ & \quad \mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{Sigma}\ \mathsf{A}\ \mathsf{P}\ \mathsf{ret}\ \mathsf{P}\ (\mathsf{fst}\ \ \mathsf{A}\ \mathsf{P}\ \mathsf{x})\ \mathsf{with}\ \mathsf{exist}\ \mathsf{A}\ \mathsf{P}\ \mathsf{a}\ \mathsf{b} \to \mathsf{b}\end{aligned}$$

5.3.5 Remark on Large Elimination and CPS Translation

As reported by Barthe and Uustalu [24], it is not possible to CPS convert large elimination using a double negation translation. To see the reason, let us consider the following elimination construct:

$$\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{Set}\ \mathsf{with}\ \mathsf{z} \to \mathbb{N} \mid \mathsf{suc}\ \mathsf{n} \to \mathbb{B}$$

The reduction rule of pattern matching tells us that, to eliminate e, we first evaluate it to a constructor application, and then choose the right branch according to the constructor we obtain. This means, if we unselectively translate terms, we must have an application $\mathsf{e}^{\div}\ (\lambda\, \mathbf{v}.\, \mathbf{e})$ representing evaluation of e. Since the value of e is represented as $\mathbf{v}$, we must analyze $\mathbf{v}$ and return either $\mathbb{N}$ or $\mathbb{B}$. Therefore, a natural candidate of $\mathbf{e}$ would be $\mathbf{pm\ v\ as\ x\ in}\ \mathbb{N}\ \mathbf{ret\ Set\ with\ z} \to \mathbb{N} \mid \mathbf{suc\ n} \to \mathbb{B}$. However, this would make $\mathsf{e}^{\div}\ (\lambda\, \mathbf{v}.\, \mathbf{e})$ ill-typed: while $\mathsf{e}^{\div}$ expects a continuation returning $\bot$, it is applied to a continuation returning $\mathbf{Set}$.

This failure example should be no surprise, though, because the double negation translation is inherently designed for bottom-level terms: since $\bot$ means logical absurdity, it has type Prop, hence its inhabitants are Prop-terms. Given this fact, the reader may wonder if Bowman et al.'s translation, where the answer type is polymorphic but is fixed to $*$-type, fails in the same way. Interestingly, the answer is no, if we restrict large elimination to scrutinize a pure term. Specifically, we can translate the above pattern matching to the following well-formed type:

$$\mathbf{pm}\ \mathsf{e}^{\div}\ \mathbb{N}\ \mathbf{id}_{\mathbb{N}}\ \mathbf{as\ x\ in}\ \mathbb{N}\ \mathbf{ret\ Set\ with\ z} \to \mathbb{N} \mid \mathbf{suc\ n} \to \mathbb{B}$$

We find that the translation is almost the same as that of (DMatch) with pure branches; the only difference is that we are scrutinizing $\mathsf{e}^{\div}\ \mathbb{N}\ \mathbf{id}_{\mathbb{N}}$ instead of e^{+}. In essence, the translation presumes that we can evaluate the source term e independent of the rest of the computation, *i.e.*, analyzing its head and choosing a branch. This does not limit the applicability of the translation if the source language has no type dependent on impure terms; the purity of e ensures that we can locally evaluate it in an empty context. From this point of view, the double negation translation can be understood as giving $\mathsf{e}^{\div}$ "too much access" to the continuation.

5.4 Local Definitions

The last feature we would like to discuss is *local definitions* [143], which we incorporate by extending the language with the `let` expressions. `let` expressions are not only useful for programming, but also indispensable for compilation, since they help us express control flow while avoiding code duplication. However, this extension is not as trivial as one would expect; it requires a global modification to the reduction rules, as well as careful analysis of metatheoretic properties. The reason is that, in a dependently typed language, the binding information introduced by `let` can be used at the level of types. For instance, when f is a L 3-accepting function, we can make the program $\mathtt{let}\ x = 3 : \mathbb{N}\ \mathtt{in}\ f\ (\mathtt{mk\text{-}lst}\ x)$ type check by identifying x with 3. In this section, we show what restriction we need to support dependent `let`, and what challenges it brings into the language.

5.4.1 Specification

5.4.1.1 Syntax and Reduction

To support local definitions, we extend expressions with a new binding construct $\mathsf{let}\ \mathsf{x} = \mathsf{e} : \mathsf{A}\ \mathsf{in}\ \mathsf{e}$, and a new environment extension $\Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A}$ (Figure 5.20). In the latter construct, we require that x is Γ-*fresh* [143], *i.e.*, it must not be available in the current environment Γ.

Definitions are associated with two reduction rules δ and ζ. The latter is the standard rule for the `let` expression. The former replaces a let-bound variable with its definition, which must be a value. Since definitions are accummulated in typing environments, δ-rule must refer to Γ. Therefore, we replace the binary reduction relation $\mathsf{e} \triangleright \mathsf{e}'$ with a trinary one $\Gamma \vdash \mathsf{e} \triangleright \mathsf{e}'$, and use the new relation in all the reduction rules.

Parallel reduction is also parametrized by a typing environment, and this change gives rise to reduction of typing environments. Specifically, reduction of environments happens in (P-VarDelta), which reduces a variable to the pararell-reduct of its definition. This makes the reduction genuinely parallel, but since the definition is not a subterm of the variable, we can only obtain the reduct by reducing the environment, which means we must make the reduction relation inductive on the derivation. The definition of environment reduction is simple; we just parallel-reduce types A and definitions e.

5.4.1.2 Typing

The best way to understand the power of local definitions is to study their typing rules, which we present in Figure 5.21. Rule (G-ExtDef) allows extending an environment with a definition $\mathsf{x} = \mathsf{e} : \mathsf{A}$ when e is a pure term. The purity

$\boxed{\text{Syntax}}$

$$\begin{array}{llll} \text{Environments} & \Gamma & ::= & \ldots \mid \Gamma, x = e : A \\ \text{Expressions} & s, A, e & ::= & \ldots \mid \text{let } x = e : A \text{ in } e \end{array}$$

$\boxed{\text{Evaluation Contexts } E, F}$

$$\begin{array}{lll} E & ::= & \ldots \mid \text{let } x = E : A \text{ in } e \\ F & ::= & \ldots \mid \text{let } x = F : A \text{ in } e \end{array}$$

$\boxed{\text{Reduction Rules } \Gamma \vdash e \triangleright e'}$

$$\begin{array}{rll} \Gamma \vdash x & \triangleright_\delta & v \text{ if } x = v : A \in \Gamma \\ \Gamma \vdash \text{let } x = v : A \text{ in } e & \triangleright_\zeta & e[v/x] \end{array}$$

$\boxed{\text{Parallel Reduction } \Gamma \vdash e \triangleright_p e'}$

$$\frac{\Gamma_1, x = v : A, \Gamma_2 \;\triangleright_p\; \Gamma_1', x = v' : A', \Gamma_2'}{\Gamma_1, x = v : A, \Gamma_2 \vdash x \triangleright_p v'} \text{ (P-VarDelta)}$$

$$\frac{\Gamma \vdash e_1 \triangleright_p e_1' \quad \Gamma \vdash A \triangleright_p A' \quad \Gamma, x = e_1 : A \vdash e_2 \triangleright_p e_2'}{\Gamma \vdash \text{let } x = e_1 : A \text{ in } e_2 \triangleright_p \text{let } x = e_1' : A \text{ in } e_2'} \text{ (P-Let)}$$

$$\frac{\Gamma \vdash v_1 \triangleright_p v_1' \quad \Gamma, x = v_1 : A \vdash e_2 \triangleright_p e_2'}{\Gamma \vdash \text{let } x = v_1 : A \text{ in } e_2 \triangleright_p e_2'[v_1'/x]} \text{ (P-LetZeta)}$$

$$\frac{}{\Gamma \;\triangleright_p\; \Gamma} \text{ (P-G-Refl)} \qquad \frac{\Gamma \;\triangleright_p\; \Gamma' \quad A \;\triangleright_p\; A'}{\Gamma, x : A \;\triangleright_p\; \Gamma', x : A'} \text{ (P-G-Ext)}$$

$$\frac{\Gamma \;\triangleright_p\; \Gamma' \quad e \;\triangleright_p\; e' \quad A \;\triangleright_p\; A'}{\Gamma, x = e : A \;\triangleright_p\; \Gamma', x = e' : A'} \text{ (P-G-ExtDef)}$$

Figure 5.20: Syntax and Reduction of Definitions

$\boxed{\text{Well-formed Environments} \vdash \Gamma}$

$$\frac{\vdash \Gamma \quad \Gamma \vdash e : A \quad \Gamma \vdash A : *}{\vdash \Gamma, x = e : A}\ (\textsc{ExtDef})$$

$\boxed{\text{Well-typed Expressions } \Gamma \vdash e : A\ \rho}$

$$\frac{\vdash \Gamma \quad x : A \in \Gamma \ \text{ or } \ x = e : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \quad \Gamma, x = e_1 : A \vdash e_2 : B\ \rho \\ \Gamma \vdash B[e_1/x] : s \quad s \in S \quad \Gamma \vdash \rho[e_1/x]\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 : B[e_1/x]\ \rho[e_1/x]}\ (\textsc{DLet})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A\ \rho \quad \Gamma, x : A \vdash e_2 : B\ \sigma \\ \Gamma \vdash B : s \quad s \in \{\mathsf{Prop}, \mathsf{Set}\} \quad \Gamma \vdash \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 : B\ \tau}\ (\textsc{NDLet})$$

Figure 5.21: Typing Rules for Definitions

restriction comes from the fact that Dellina is call-by-value: recall that definitions are used in δ-reduction, which happens only when the term part of the definition is a pure value. In fact, when e is a non-value, the extension is redundant, since it can never be used during type checking. However, we do not restrict extension to the form $\mathsf{x} = \mathsf{v} : \mathsf{A}$ in order to keep the typing rule of let simple. The new environment extension necessitates a slight modification to (Var), so that we may refer to let-bound variables.

Similarly to application and pattern matching, the let expression has two typing rules (DLet) and (NDLet). In the former, the result type $\mathsf{B}[\mathsf{e}_1/\mathsf{x}]$ depends on the term part of the definition $\mathsf{x} = \mathsf{e}_1 : \mathsf{A}$, hence we check the purity of e_1, as well as the well-formedness of the result type and effect annotation. What we should pay attention to is the premise $\Gamma, \mathsf{x} = \mathsf{e}_1 : \mathsf{A} \vdash \mathsf{e}_2 : \mathsf{B}\ \rho$. Unlike (Abs), where we tyoe check the body with a binding $\mathsf{x} : \mathsf{A}$, we type check e_2 with a definition $\mathsf{x} = \mathsf{e}_1 : \mathsf{A}$. This allows us to replace the occurrences of x in B and ρ with e when e_1 is a value.

The difference in the typing environment used in let and λ implies that we may have $\mathsf{let}\ \mathsf{x} = \mathsf{e}_1 : \mathsf{A}\ \mathsf{in}\ \mathsf{e}_2$ well-typed without having $(\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}_2)\ \mathsf{e}_1$ well-typed. Consider the following term, where $\mathsf{mk\text{-}lst} : \Pi\, \mathsf{x} : \mathbb{N}.\, \mathsf{L}\ \mathsf{x}$:

$$\mathsf{let}\ \mathsf{x} = \mathsf{z} : \mathbb{N}\ \mathsf{in}\ ::\ \mathsf{z}\ 1\ (\mathsf{mk\text{-}lst}\ \mathsf{x})$$

When type checking the body $::\ \mathsf{z}\ 1\ (\mathsf{mk\text{-}lst}\ \mathsf{x})$, we find that the third argument to the $::$-constructor has type $\mathsf{L}\ \mathsf{x}$, while the constructor expects a term of type $\mathsf{L}\ \mathsf{z}$ because its first argument is z. The body is however well-typed because we have a definition $\mathsf{x} = \mathsf{z} : \mathbb{N}$ in the typing environment, and we can use it to replace the variable x in $\mathsf{L}\ \mathsf{x}$ by z via δ-reduction. If we build a similar term using a λ, namely $(\lambda\, \mathsf{x} : \mathbb{N}.\ ::\ \mathsf{z}\ 1\ (\mathsf{mk\text{-}lst}\ \mathsf{x}))\ \mathsf{z}$, the body of λ is not well-typed since x can be an arbitrary value[9].

By contrast, the non-dependent variant of let, which we derive by (NDLet), does not use the definition in an interesting way. As e_1 is an impure term, the variable x bound to it can be replaced by any value, which means we may only

[9]This difference has been used to argue the superiority of *administrative normal form (ANF)* [77] over CPS as an intermediate representation of compilers for dependently typed languages [33]. ANF makes control flow explicit by `let`-binding all intermediate results. In a dependently typed setting, this means we can make an ANF translation type preserving without the [T-Cont] rule, which turns a continuation λ into a `let`.

use the variable x at the level of terms. Therefore, in (NDLet), we type check e_2 with the binding $\mathsf{x} : \mathsf{A}$, without binding x to some specific term. We then check the well-formedness of the result type B and the effect annotation σ in the unextended environment Γ, ensuring that there is no type referring to x.

5.4.2 Metatheory

Local definitions require considerable effort to scale the metatheory of the language. The main changes are again observed in those properties that deal with substitution: we must make sure that let-bound variables are replaced by (the reduct of) their definition, not by an arbitrary value.

Lemma 5.4.1 (Substitution of Values and Parallel Reduction)**.** *If* $\Gamma, \mathsf{x} = \mathsf{v} : \mathsf{A}, \Gamma' \vdash \mathsf{t} \rhd_p \mathsf{t}'$, *then* $\Gamma, \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{t}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{t}'[\mathsf{v}/\mathsf{x}]$.

Proof. The proof is by induction on the derivation of $\Gamma, \mathsf{x} = \mathsf{v} : \mathsf{A}, \Gamma' \vdash \mathsf{t} \rhd_p \mathsf{t}'$.

Case 1: (P-VarDelta)

Sub-Case 1: $\mathsf{t} = \mathsf{x}$
Our goal is to show

$$\Gamma, \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{x}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{v}[\mathsf{v}/\mathsf{x}]$$

This is trivial because x cannot occur free in v.

Sub-Case 2: $\mathsf{t} = \mathsf{y}$ where $\mathsf{y} \neq \mathsf{x}$ and $\mathsf{y} = \mathsf{u} : \mathsf{B} \in \Gamma, \Gamma'$
Our goal is to show

$$\Gamma, \Gamma'[\mathsf{v}/\mathsf{x}] \vdash \mathsf{y}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{u}'[\mathsf{v}/\mathsf{x}]$$

If $\mathsf{y} = \mathsf{u} : \mathsf{B} \in \Gamma$, the proof is easy, since neither u or u' refers to variable x. That is, we have $\mathsf{y}[\mathsf{v}/\mathsf{x}] = \mathsf{y} \rhd_p \mathsf{u}' = \mathsf{u}'[\mathsf{v}/\mathsf{x}]$. If $\mathsf{y} = \mathsf{u} : \mathsf{B} \in \Gamma'$, the induction hypothesis on the environment implies $\mathsf{u}[\mathsf{v}/\mathsf{x}] \rhd_p \mathsf{u}'[\mathsf{v}/\mathsf{x}]$, which gives us $\mathsf{y}[\mathsf{v}/\mathsf{x}] = \mathsf{y} \rhd_p \mathsf{u}'[\mathsf{v}/\mathsf{x}]$ as desired. Note that we would get stuck in this case if the reduction did not recurse on the environment (that is, if the premise of (P-VarDelta) was $\Gamma \vdash \mathsf{v} \rhd_p \mathsf{v}'$).

□

Using these lemmas, we can prove the confluence theorem, by simultaneous induction on the derivation of $\Gamma \vdash t \rhd_p t_1$ and $\Gamma \vdash t \rhd_p t_2$.

We next extend the preservation proof to cases involving local definitions. This requires some refinements to auxiriary lemmas:

Lemma 5.4.2 (Weakening). *Suppose* $\Gamma \vdash e : A$, $\Gamma \vdash A : *$, *and* $x \notin \Gamma, \Gamma'$. *Then, the following hold.*

1. *If* $\vdash \Gamma, \Gamma'$, *then* $\vdash \Gamma, x = e : A, \Gamma'$.

2. *If* $\Gamma, \Gamma' \vdash t : T\ \rho$, *then* $\Gamma, x = e : A, \Gamma' \vdash t : T\ \rho$.

Lemma 5.4.3 (Environment Conversion).

1. *If* $\vdash \Gamma, x = v : A, \Gamma'$ *and* $\Gamma \vdash v \equiv v'$, *then* $\vdash \Gamma, x = v' : A, \Gamma'$.

2. *If* $\Gamma, x = v : A, \Gamma' \vdash e : A\ \rho$ *and* $\Gamma \vdash v \equiv v'$, *then* $\Gamma, x = v' : A, \Gamma' \vdash e : A\ \rho$.

Lemma 5.4.4 (Substitution). *Suppose* $\Gamma \vdash v \rhd_p v'$ *and* $\Gamma \vdash v' : A$. *Then, the following hold.*

1. *If* $\vdash \Gamma, x = v : A, \Gamma'$, *then* $\vdash \Gamma, \Gamma'[v'/x]$.

2. *If* $\Gamma, x = v : A, \Gamma' \vdash t : T\ \rho$, *then* $\Gamma, \Gamma'[v'/x] \vdash t[v'/x] : T[v'/x]\ \rho[v'/x]$.

Observe that the substitution lemma assumes that the reduct v' of v has the same type as v. Without this assumption, the lemma would require the preservation theorem, making the two proofs circular.

The new cases of the preservation theorem can be proved as follows:

Proof.

Case 1: (Var)

The only way where a variable takes step is via (P-VarDelta). Suppose we have $\Gamma, x = v : A, \Gamma' \vdash x \rhd_p v'$. Our goal is to show

$$\Gamma, x = v : A, \Gamma' \vdash v' : A$$

By the induction hypothesis, we have $\Gamma \vdash v' : A$. The goal easily follows by weakening (Lemma 5.4.2).

Case 2: (DLET)

Sub-Case 1: $\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 \rhd_p \mathsf{let}\ x = e'_1 : A'\ \mathsf{in}\ e'_2$ by (P-LET)
Our goal is to show

$$\Gamma \vdash \mathsf{let}\ x = e'_1 : A'\ \mathsf{in}\ e'_2 : B[e_1/x]\ \rho[e_1/x]$$

By the induction hypothesis, we have

$$\Gamma, x = e_1 : A \vdash e'_2 : B \quad , \quad \Gamma, x = e_1 : A \vdash B[e'_1/x] : s\ , \quad \text{and} \quad \Gamma, x = e_1 : A \vdash \rho[e'_1/x]$$

By environment conversion (Lemma 5.4.3), we can replace all occurrences of $x = e_1 : A$ with $x = e'_1 : A'$, and derive

$$\Gamma \vdash \mathsf{let}\ x = e'_1 : A'\ \mathsf{in}\ e'_2 : B[e'_1/x]\ \rho[e'_1/x]$$

The goal now follows by (CONV).

Sub-Case 2: $\Gamma \vdash \mathsf{let}\ x = v_1 : A\ \mathsf{in}\ e_2 \rhd_p e'_2[v'_1/x]$ by (P-LETZETA)
Our goal is to show

$$\Gamma \vdash e'_2[v'_1/x] : B[v'_1/x]\ \rho[v_1/x]$$

By the induction hypothesis, we have

$$\Gamma \vdash v'_1 : A\ , \quad \Gamma, x = v_1 : A \vdash e'_2 : B\ \rho\ ,$$

$$\Gamma, x = v_1 : A \vdash B[v'_1/x] : s\ , \quad \text{and} \quad \Gamma, x = v_1 : A \vdash \rho[v'_1/x] : (s, s')$$

By the substitution lemma, we obtain

$$\Gamma \vdash e'_2[v'_1/x] : B[v'_1/x]\ \rho[v'_1/x]$$

The goal follows by (DLET) and (CONV).

□

5.4.3 CPS Translation

The CPS translation of local definitions (Figure 5.22) can be easily defined following the recipe we have been using so far. In the translation of (EXTDEF), the term

$$\frac{\vdash \Gamma \quad \Gamma \vdash e : A \quad \Gamma \vdash A : *}{\vdash \Gamma, x = e : A}\ (\textsc{G-ExtDef}) \overset{+}{\rightsquigarrow} \Gamma^{+}, \mathbf{x} = {e_1}^{+} : A^{+}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \quad \Gamma, x = e_1 : A \vdash e_2 : B\ \rho \\ \Gamma \vdash B[e_1/x] : s \quad s \in S \quad \Gamma \vdash \rho[e_1/x]\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 : B[e_1/x]\ \rho[e_1/x]}\ (\textsc{DLet})$$

$\overset{+}{\rightsquigarrow}$ $\mathbf{let}\ \mathbf{x} = {e_1}^{+} : A^{+}\ \mathbf{in}\ {e_2}^{+}$
if $\rho = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (B[e_1/x])^{+} \rightarrow (\alpha[e_1/x])^{+}.\, \mathbf{let}\ \mathbf{x} = {e_1}^{+} : A^{+}\ \mathbf{in}\ {e_2}^{\div}\ \mathbf{k}$
if $\rho = [\alpha, \beta]$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A\ \rho \quad \Gamma, x : A \vdash e_2 : B\ \sigma \\ \Gamma \vdash B : s \quad s \in \{\mathsf{Prop}, \mathsf{Set}\} \quad \Gamma \vdash \sigma \quad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1 : A\ \mathsf{in}\ e_2 : B\ \tau}\ (\textsc{NDLet})$$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : B^{+} \rightarrow \alpha^{+}.\, {e_1}^{\div}\ (\lambda\,\mathbf{x} : A^{+}.\, \mathbf{k}\ {e_2}^{+})$
if $\rho = [\alpha, \beta], \sigma = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (B[e_1/x])^{+} \rightarrow (\alpha[e_1/x])^{+}.\, \mathbf{let}\ \mathbf{x} = {e_1}^{+} : A^{+}\ \mathbf{in}\ {e_2}^{\div}\ \mathbf{k}$
if $\rho = [\beta, \gamma], \sigma = [\alpha, \beta]$

Figure 5.22: CPS Translation of Definitions

e_1 is applied the value translation because it is pure. let expressions are translated similarly to pattern matching. The reader will find that the first translation of (NDLet) turns let into a λ, which amounts to turning a uniquely determined x into a general variable. If we allowed reference to x from B in this case, we would run into the issues discussed in Section 4.1.

Thanks to the purity restriction, we can easily show that the extended translation satisfies desired properties. First, we show the δ/ζ-reduction cases of the computational soundness proof:

Proof.

Case 1: (Var)

Sub-Case 1: $\Gamma, \mathsf{x} = \mathsf{v} : \mathsf{A}, \Gamma' \vdash \mathsf{x} \rhd_p \mathsf{v}'$ by (Var)

$$\begin{aligned} \mathsf{x}^+ &= \mathbf{x} && \text{by translation} \\ &\rhd_\delta \mathsf{v}'^+ \end{aligned}$$

Case 2: (DLet)

Sub-Case 1: $\mathsf{let}\ \mathsf{x} = \mathsf{v}_1\ \mathsf{in}\ \mathsf{e}_2 \ \rhd_p\ \mathsf{e}_2[\mathsf{v}_1'/\mathsf{x}]$ by (P-LetZeta)

$$\begin{aligned} (\mathsf{let}\ \mathsf{x} = \mathsf{v}_1\ \mathsf{in}\ \mathsf{e}_2)^\div &= \lambda\,\mathbf{k}.\,\mathbf{let}\ \mathbf{x} = \mathsf{v}_1{}^+\ \mathbf{in}\ \mathsf{e}_2{}^\div\ \mathbf{k} && \text{by translation} \\ &\equiv \lambda\,\mathbf{k}.\,\mathbf{let}\ \mathbf{x} = \mathsf{v}_1'{}^+\ \mathbf{in}\ \mathsf{e}_2'{}^\div\ \mathbf{k} && \text{by IH on } \mathsf{v}_1 \text{ and } \mathsf{e}_2 \\ &\rhd_\zeta \lambda\,\mathbf{k}.\,(\mathsf{e}_2'{}^\div\ \mathbf{k})[\mathsf{v}_1'{}^+/\mathbf{x}] \\ &= \lambda\,\mathbf{k}.\,(\mathsf{e}_2'{}^\div[\mathsf{v}_1'{}^+/\mathbf{x}])\ \mathbf{k} && \text{by substitution} \\ &\equiv \mathsf{e}_2'{}^\div[\mathsf{v}_1'{}^+/\mathbf{x}] && \text{by } \eta \\ &= (\mathsf{e}_2'[\mathsf{v}_1'/\mathsf{x}])^\div && \text{by translation} \end{aligned}$$

□

Next, we show the let case of the type preservation proof.

Proof.

Case 1: (E-DLet) where $\rho = [\alpha, \beta]$
Our goal is to show

$$\Gamma^+ \vdash \lambda \mathbf{k} : (B[e_1/x])^+ \rightarrow (\alpha[e_1/x])^+ . \, \mathbf{let} \; \mathbf{x} = {e_1}^+ : A^+ \; \mathbf{in} \; {e_2}^{\div} \; \mathbf{k} : \\ ((B[e_1/x])^+ \rightarrow (\alpha[e_1/x])^+) \rightarrow (\beta[e_1/x])^+$$

By the induction hypothesis, we have

$$\Gamma^+ \vdash {e_1}^+ : A^+ \;\; \text{and} \;\; \Gamma^+ \vdash {e_2}^{\div} : ((B[e_1/x])^+ \rightarrow (\alpha[e_1/x])^+) \rightarrow (\beta[e_1/x])^+$$

Using [App], we can derive

$$\Gamma^+, \mathbf{k} : (B[e_1/x])^+ \rightarrow (\alpha[e_1/x])^+ \vdash {e_2}^{\div} \; \mathbf{k} : (\beta[e_1/x])^+$$

The goal now follows by [DLet] and [Abs].

□

5.5 Example

With inductive datatypes and type-level computations at hand, we can now implement more interesting programs. In this section, we build a type-safe evaluator supporting efficient exception handling, using `shift`, `reset`, and dependent types in a meaningful way.

As we saw in Section 1.1, dependent types are able to express specifications of data in a precise manner. A well-known application of this ability is type-safe representation of user-defined languages [4]. Let us define a small object language $L_{\mathbb{BN}}$, which consists of booleans and numbers. To represent $L_{\mathbb{BN}}$ types and terms, we declare two datatypes Ty and Tm as follows:

$$\mathsf{Ind}(\mathsf{Ty} : \mathsf{Set}, \{\mathsf{bool} : \mathsf{Ty}; \, \mathsf{num} : \mathsf{Ty}\})$$

$$\mathsf{Ind}(\mathsf{Tm} : \mathsf{Ty} \rightarrow \mathsf{Set}, \\ \{\mathsf{bl} : \mathbb{B} \rightarrow \mathsf{Tm\ bool}; \, \mathsf{nm} : \mathbb{N} \rightarrow \mathsf{Tm\ num}; \; \ldots \; ; \, \mathsf{div} : \mathsf{Tm\ num} \rightarrow \mathsf{Tm\ num} \rightarrow \mathsf{Tm\ num}\})$$

Notice that the type of $L_{\mathbb{BN}}$ terms is indexed by an $L_{\mathbb{BN}}$ type. The index ensures that the term language produces only well-typed terms: *e.g.*, we can construct $\mathsf{div}\ (\mathsf{nm}\ 4)\ (\mathsf{nm}\ 2)$ but not $\mathsf{div}\ (\mathsf{nm}\ 4)\ (\mathsf{bl}\ \mathsf{true})$.

We next define a language Ans, which serves as the target language of the evaluation function:

$$\mathsf{Ind}(\mathsf{Ans} : \mathsf{Ty} \to \mathsf{Set},\ \{\mathsf{val} : \Pi\,\mathsf{t} : \mathsf{Ty}.\,\mathsf{interpTy}\ \mathsf{t} \to \mathsf{Ans}\ \mathsf{t};\ \mathsf{error} : \Pi\,\mathsf{t} : \mathsf{Ty}.\,\mathsf{Ans}\ \mathsf{t}\})$$

As the definition suggests, answers consist of values and errors. The datatype Ans is again indexed by an $L_{\mathbb{BN}}$ type, which is passed explicitly as an argument to the constructors. In the val case, we have a second argument representing the value returned by the evaluator (strictly speaking, the auxiriary function used in the evaluator), which has type $\mathsf{interpTy}\ \mathsf{t}$. Here, $\mathsf{interpTy}$ is a function that turns $L_{\mathbb{BN}}$ types into Dellina types:

$$\mathsf{interpTy} \overset{\text{def}}{\equiv} \lambda\,\mathsf{t} : \mathsf{Ty}.\,\mathsf{pm}\ \mathsf{t}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{Ty}\ \mathsf{ret}\ \mathsf{Set}\ \mathsf{with}\ \mathsf{bool} \to \mathbb{B} \mid \mathsf{num} \to \mathbb{N}$$

The pattern matching construct is an instance of large elimination, since it inspects a Dellina term (of type Ty) and returns a Dellina type (of type Set).

Now, we are ready to define the evaluation function eval. To simplify the top-level call, we use an auxiriary function eval' to implement the actual behavior:

$$\begin{array}{lll}
 & & \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{t}:\mathsf{Ty}.\,\mathsf{Ty} \to \mathsf{Tm}\ \mathsf{t} \to \mathsf{interpTy}\ \mathsf{t}[\mathsf{Ans}\ \mathsf{t},\mathsf{Ans}\ \mathsf{t}]}\ \mathsf{t}.\,\lambda\,\mathsf{t}' : \mathsf{Ty}.\,\lambda\,\mathsf{e} : \mathsf{Tm}\ \mathsf{t}. \\
 & & \quad \mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{Tm}\ \mathsf{t}\ \mathsf{ret}\ (\mathsf{interpTy}\ \mathsf{t})\ \mathsf{with} \\
 & & \quad\ \ \mathsf{bl}\ \mathsf{b} \to \mathcal{S}\mathsf{k} : (\mathbb{B} \to \mathsf{Ans}\ \mathsf{t}').\,\mathsf{k}\ \mathsf{b} \\
\mathsf{eval}' & \overset{\text{def}}{\equiv} & \quad \mid \mathsf{nm}\ \mathsf{n} \to \mathcal{S}\mathsf{k} : (\mathbb{N} \to \mathsf{Ans}\ \mathsf{t}').\,\mathsf{k}\ \mathsf{n} \\
 & & \quad \mid \ldots \\
 & & \quad \mid \mathsf{div}\ \mathsf{e}_1\ \mathsf{e}_2 \to \mathsf{pm}\ \mathsf{f}\ \mathsf{num}\ \mathsf{t}'\ \mathsf{e}_2\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathbb{N}\ \mathsf{with} \\
 & & \qquad\qquad\qquad \mathsf{z} \to \mathcal{S}\mathsf{k} : (\mathbb{N} \to \mathsf{Ans}\ \mathsf{t}').\,\mathsf{error}\ \mathsf{t}' \\
 & & \qquad\qquad\qquad \mid \mathsf{suc}\ \mathsf{n} \to \ldots \\
\mathsf{eval} & \overset{\text{def}}{\equiv} & \lambda\,\mathsf{t} : \mathsf{Ty}.\,\lambda\,\mathsf{e} : \mathsf{Tm}\ \mathsf{t}.\,\langle \mathsf{val}\ (\mathsf{eval}'\ \mathsf{t}\ \mathsf{t}\ \mathsf{e}) \rangle
\end{array}$$

The eval' function takes in three arguments: two $L_{\mathbb{BN}}$ types t, t' and an $L_{\mathbb{BN}}$ term e. Among the first two arguments, t is the type index of e (*i.e.*, the argument at each recursive call), whereas t' is the type index of the top-level argument. In the bl branch, the evaluator returns b, which is a Dellina boolean of type $\mathbb{B}$[10] (we again have a redundant `shift` for typing reasons). The nm branch is computed in a similar way; the difference is that the returned value is a Dellina natural number. What is interesting is the div branch: when we find a division by zero, we raise an error via the `shift` operator. The captured continuation k is a computation that builds a value, *i.e.*, it takes the form $\mathsf{val}\ \mathsf{F}$. In the exception case, we want to finish the entire computation immediately, hence we discard the continuation and return $\mathsf{error}\ \mathsf{t}'$ to the top-level.

Looking at the type annotation of f, we can see that the evaluator eval preserves types. That is, when given a top-level term of type $\mathsf{Tm}\ \mathsf{t}$, it either produces a value of type $\mathsf{Ans}\ \mathsf{t}$, or returns an error of the same type $\mathsf{Ans}\ \mathsf{t}$.

As we saw in Section 1.2, exception handling is also possible in a pure language, if we write the whole evaluator in CPS. However, programming in CPS is less convenient, and the resulting program is less efficient than the original one. Furthermore, since our evaluator is dependently typed, we would need non-standard axioms (like [T-Cont] of Bowman et al. [34]) to make the CPS evaluator well-typed. Therefore, the Dellina program above is preferable from both practical and theoretical perspectives.

[10]Since eval' returns a meta-language value instead of a user-defined datum, we say that the function is written in the *tagless-final* style [36].

Chapter 6

Call-by-name Dellina-

So far, we have been discussing the interaction between dependent types and delimited control in a call-by-value setting. The reason we chose this evaluation strategy comes from the fact that the behavior of effectful computations is easier to understand when they are eagarly evaluated. For instance, when we have a function $\lambda \mathsf{x} : \mathbb{N}.\, \mathsf{x} + \mathsf{x}$, we know that the two x's always denote the same value, regardless of the purity of the actual argument. This is not the case in a call-by-name language, where variables are replaced by a possibly effectful computation.

On the other hand, some researchers claim that dependently typed calculi are inherently call-by-name [97, 132]. The argument intuitively makes sense, because dependent constructs—such as application and pattern matching—have a type dependent on their subterm, which is not necessarily a value. This means, the typing rules are defined under the assumption that substitution of computations are always safe. When this assumption does not hold, as in Dellina, we have to manually check well-formedness of result types, by means of an additional typing premise.

Given this fact, the reader might ask: if we switch to a call-by-name semantics, couldn't we have a simpler type system and soundness proofs? It turns out that this is not quite the case if we want to allow all purely dependent types. Also, the language specification would be complicated by the semantical difference between call-by-name functions and continuations.

In this chapter, we present Dellina^n, a call-by-name variant of Dellina-. Unfortunately, we have not yet been able to prove its metatheoretic properties, nor have we established a type preservation proof of the CPS translation. However, we still think it worth showing the preliminary specification of the language; in particular, by comparing call-by-value/call-by-name effects in a dependently typed setting, we can achieve a clearer view of values and computations.

6.1 Syntax

We present the syntax of Dellina^n in Figure 6.1. There are two major changes in the definition of typing environments. First, when we extend an environment with a new variable x, we add its type A together with an effect annotation ρ. This reflects the fact that variables are potentially effectful computations in a call-by-name language.

The second change in the environment definition is the new extension form for continuation variables k. This comes from a design decision specific to Dellina^n: we distinguish between λ-bound variables and `shift`-bound variables. The distinction is commonly adopted to languages with call-by-name delimited control [27, 102, 159], and plays a key role in the CPS translation. Note that extension of a continuation variable never comes with an effect annotation, because they denote functions (but *not* call-by-name functions, as we discuss in Section 6.2).

Shifting our attention to the type language, we find that function types now take the form $\Pi\mathsf{x} : \mathsf{A}\ \rho. \mathsf{B}\ \sigma$, which has an annotation ρ representing the control effect of the argument. This change can be understood along the same line as our refinement to environment extension; in particular, it captures the fact that call-by-name β-reduction takes place before evaluating the argument to a pure value. Correspondingly, in the definition of terms, we see that abstractions have been refined to $\lambda\mathsf{x} : \mathsf{A}\ \rho. \mathsf{e}$, sharing the same binding construction with function types.

Computations now include variables x, but only those bound by a λ. `shift`-bound variables can only be used via the new construct $\mathsf{k} \hookrightarrow \mathsf{e}$, which throws a term e to a continuation variable k. This allows us to incorporate the value-accepting nature of continuations into the semantics of the language. Note that, while we cannot return a continuation by writing $\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha. \mathsf{k}$, we can still do the same thing by η-expanding the continuation, namely by writing $\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha. \lambda\mathsf{x} : \mathsf{A}. \mathsf{k} \hookrightarrow \mathsf{x}$.

$$\begin{array}{llll}
\text{Environments} & \Gamma & ::= & \bullet \mid \Gamma, \mathsf{x} : \mathsf{A}\ \rho \mid \Gamma, \mathsf{k} :_k \mathsf{A} \\
\text{Kinds} & \kappa & ::= & * \mid \Box \\
\text{Types} & \mathsf{A}, \alpha & ::= & \mathsf{Unit} \mid \mathbb{N} \mid \mathsf{L\ e} \mid \Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma \\
\text{Effects} & \rho & ::= & \epsilon \mid [\alpha, \beta] \\
\text{Values} & \mathsf{v} & ::= & \lambda \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{e} \mid \mathsf{rec\ f}_{\Pi \mathsf{k} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e} \\
 & & \mid & () \mid \mathsf{z} \mid \mathsf{suc\ e} \mid \mathsf{nil} \mid\ ::\ \mathsf{e\ e\ e} \\
\text{Terms} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{x} \mid \mathsf{e\ e} \\
 & & \mid & \mathsf{pm\ e\ as\ x\ in}\ \mathbb{N}\ \mathsf{ret\ P\ with\ z} \rightarrow \mathsf{e} \mid \mathsf{suc\ n} \rightarrow \mathsf{e} \\
 & & \mid & \mathsf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \rightarrow \mathsf{e} \mid\ ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e} \\
 & & \mid & \mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e} \mid \mathsf{k} \hookrightarrow \mathsf{e} \mid \langle \mathsf{e} \rangle
\end{array}$$

Figure 6.1: Call-by-name Syntax

As a final remark, we explicitly define the set of values v, which is generally missing in call-by-name languages. The reason is that the elimination rule of `reset` applies only to value-surrounding `reset`. Note that values include *any* inductive data, regardless of whether constructor arguments are values or not.

6.2 Evaluation, Reduction, and Equivalence

6.2.1 Runtime Evaluation

As we did for Dellina-, we give two sets of reduction rules: one for runtime, and the other for type-checking time. In Figures 6.2 – 6.3, we present evaluation contexts and reduction rules. The latter are basically a call-by-name version of Dellina- reduction rules, where the occurrences of values (except for the one in `reset`) are replaced by computations. There is however one major change in the `shift`-reduction rule. Recall that the Dellina reduction rule was defined as follows:

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e}] \rangle \quad \rhd_{\mathcal{S}} \quad \langle \mathsf{e}[\lambda \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}[\mathsf{x}] \rangle / \mathsf{k}] \rangle$$

The rule substitutes a function $\lambda \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}[\mathsf{x}] \rangle$ for k, and in the body e, any occurrence of $\mathsf{k}\ \mathsf{e}'$ reduces to $\langle \mathsf{F}[\mathsf{v}'] \rangle$ assuming that $\mathsf{e}' \ \rhd^{\star}\ \mathsf{v}'$. On the other hand, in Dellina^n, `shift`-reduction is defined in the following way:

$\boxed{\text{Evaluation Contexts } \mathsf{E}, \mathsf{F}}$

$$
\begin{array}{lcl}
\mathsf{E} & ::= & [\,] \mid \mathsf{E}\ \mathsf{e} \\
& & \mid \mathsf{pm}\ \mathsf{E}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e} \\
& & \mid \mathsf{pm}\ \mathsf{E}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e} \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e} \\
& & \mid \langle \mathsf{E} \rangle \\
\mathsf{F} & ::= & [\,] \mid \mathsf{F}\ \mathsf{e} \\
& & \mid \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e} \\
& & \mid \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e} \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}
\end{array}
$$

$\boxed{\text{Plugging Function } \mathsf{plug}\ F\ \mathsf{e} = \mathsf{e}'}$

$$
\begin{array}{rcl}
\mathsf{plug}\ [\,]\ \mathsf{e} & \overset{\text{def}}{\equiv} & \mathsf{e} \\
\mathsf{plug}\ (\mathsf{F}\ \mathsf{e}_1)\ \mathsf{e} & \overset{\text{def}}{\equiv} & \mathsf{plug}\ \mathsf{F}\ (\mathsf{e}\ \mathsf{e}_1) \\
\mathsf{plug}\ (\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} & \overset{\text{def}}{\equiv} & \mathsf{plug}\ \mathsf{F}\ (\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\
\mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2)\ \mathsf{e} & & \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2) \\
\mathsf{plug}\ (\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} & \overset{\text{def}}{\equiv} & \mathsf{plug}\ \mathsf{F}\ (\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with} \\
\mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2)\ \mathsf{e} & & \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2)
\end{array}
$$

Figure 6.2: Call-by-name Evaluation Contexts

$\boxed{\text{Reduction Rules } \mathsf{e} \ \triangleright \ \mathsf{e'}}$

$$
\begin{array}{rcl}
(\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e})\ \mathsf{e}_1 & \triangleright_\beta & \mathsf{e}[\mathsf{e}_1/\mathsf{x}] \\
(\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{k}:\mathsf{A}\ \rho.\,\mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e})\ \mathsf{e}_1 & \triangleright_\mu & \mathsf{e}[\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{k}:\mathsf{A}\ \rho.\,\mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e}/\mathsf{f}, \mathsf{e}_1/\mathsf{x}] \\
\begin{array}{r}\mathsf{pm\ z\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with} \\ \mathsf{z} \rightarrow \mathsf{e}_1 \,|\, \mathsf{suc\ n} \rightarrow \mathsf{e}_2\end{array} & \triangleright_\iota & \mathsf{e}_1 \\
\begin{array}{r}\mathsf{pm\ suc\ e\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with} \\ \mathsf{z} \rightarrow \mathsf{e}_1 \,|\, \mathsf{suc\ n} \rightarrow \mathsf{e}_2\end{array} & \triangleright_\iota & \mathsf{e}_2[\mathsf{e}/\mathsf{n}] \\
\begin{array}{r}\mathsf{pm\ nil\ as\ x\ in\ L\ a\ ret\ P\ with} \\ \mathsf{nil} \rightarrow \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2\end{array} & \triangleright_\iota & \mathsf{e}_1 \\
\begin{array}{r}\mathsf{pm}\ ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{as\ x\ in\ L\ a\ ret\ P\ with} \\ \mathsf{nil} \rightarrow \mathsf{e}'_1 \,|\, ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}'_2\end{array} & \triangleright_\iota & \mathsf{e}'_2[\mathsf{e}_0/\mathsf{m}, \mathsf{e}_1/\mathsf{h}, \mathsf{e}_2/\mathsf{t}] \\
\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e}]\rangle & \triangleright_{\mathcal{S}} & \langle \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]\rangle \\
\langle \mathsf{v} \rangle & \triangleright_{\mathcal{R}} & \mathsf{v}
\end{array}
$$

$\boxed{\text{Single-step Evaluation}}$

$$\frac{\mathsf{e} \ \triangleright \ \mathsf{e'}}{\mathsf{E}[\mathsf{e}] \ \triangleright \ \mathsf{E}[\mathsf{e'}]} \ (\text{R-EVAL})$$

$$\frac{}{\mathsf{e} \ \triangleright^\star \ \mathsf{e}} \ (\text{RS-REFL}) \qquad \frac{\mathsf{e}_0 \ \triangleright \ \mathsf{e}_1 \qquad \mathsf{e}_1 \ \triangleright^\star \ \mathsf{e}_2}{\mathsf{e}_0 \ \triangleright^\star \ \mathsf{e}_2} \ (\text{RS-TRANS})$$

Figure 6.3: Call-by-name Runtime Reduction Rules

$$\begin{aligned}
\mathsf{Unit}[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{Unit} \\
\mathbb{N}[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathbb{N} \\
(\mathsf{L}\ \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{L}\ (\mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]) \\
(\Pi\, \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma)[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \Pi\, \mathsf{x} : \mathsf{A}[\mathsf{k} \Rightarrow \mathsf{F}]\ \rho[\mathsf{k} \Rightarrow \mathsf{F}].\, \mathsf{B}[\mathsf{k} \Rightarrow \mathsf{F}]\ \sigma[\mathsf{k} \Rightarrow \mathsf{F}] \\
\mathsf{x}[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{x} \\
(\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \lambda\, \mathsf{x} : \mathsf{A}[\mathsf{k} \Rightarrow \mathsf{F}].\, \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \\
(\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{k} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\ \mathsf{x}.\, \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{rec}\ \mathsf{f}_{(\Pi \mathsf{k} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma)[\mathsf{k} \Rightarrow \mathsf{F}]}\ \mathsf{x}.\, \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \\
(\mathsf{e}_0\ \mathsf{e}_1)[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{e}_0[\mathsf{k} \Rightarrow \mathsf{F}]\ \mathsf{e}_1[\mathsf{k} \Rightarrow \mathsf{F}] \\
\begin{array}{r}(\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \rightarrow \\ \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e}_2)[\mathsf{k} \Rightarrow \mathsf{F}]\end{array} &\overset{\text{def}}{\equiv} \begin{array}{r}\mathsf{pm}\ \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}[\mathsf{k} \Rightarrow \mathsf{F}]\ \mathsf{with}\ \mathsf{z} \rightarrow \\ \mathsf{e}_1[\mathsf{k} \Rightarrow \mathsf{F}] \mid \mathsf{suc}\ \mathsf{n} \rightarrow \mathsf{e}_2[\mathsf{k} \Rightarrow \mathsf{F}]\end{array} \\
\begin{array}{r}(\mathsf{pm}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \rightarrow \\ \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e}_2)[\mathsf{k} \Rightarrow \mathsf{F}]\end{array} &\overset{\text{def}}{\equiv} \begin{array}{r}\mathsf{pm}\ \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathsf{L}\ \mathsf{a}\ \mathsf{ret}\ \mathsf{P}[\mathsf{k} \Rightarrow \mathsf{F}]\ \mathsf{with}\ \mathsf{nil} \rightarrow \\ \mathsf{e}_1[\mathsf{k} \Rightarrow \mathsf{F}] \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \rightarrow \mathsf{e}_2[\mathsf{k} \Rightarrow \mathsf{F}]\end{array} \\
(\mathcal{S}\mathsf{k}' : \mathsf{A} \rightarrow \alpha.\, \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathcal{S}\mathsf{k}' : \mathsf{A} \rightarrow \alpha[\mathsf{k} \Rightarrow \mathsf{F}].\, \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \quad \text{where } \mathsf{k}' \notin \{\mathsf{k}\} \cup FV(\mathsf{F}) \\
(\mathsf{k} \hookrightarrow \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \langle \mathsf{F}[\mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]] \rangle \\
(\mathsf{k}' \hookrightarrow \mathsf{e})[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \mathsf{k}' \hookrightarrow \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \quad \text{where } \mathsf{k}' \notin \{\mathsf{k}\} \cup FV(\mathsf{F}) \\
\langle \mathsf{e} \rangle[\mathsf{k} \Rightarrow \mathsf{F}] &\overset{\text{def}}{\equiv} \langle \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \rangle
\end{aligned}$$

Figure 6.4: Context Substitution

$$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \rightarrow \alpha.\, \mathsf{e}] \rangle \quad \triangleright_{\mathcal{S}} \quad \langle \mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}] \rangle$$

The rule does not directly substitute a function for k. Instead, it uses a special substitution operation $\mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]$, which is defined in Figure 6.4. Intuitively, $\mathsf{e}[\mathsf{k} \Rightarrow \mathsf{F}]$ turns every occurrences of $\mathsf{k} \hookrightarrow \mathsf{e}'$ into $\langle \mathsf{F}[\mathsf{e}'] \rangle$. The result of the substitution is equivalent to β-reducing $(\mathsf{k}\ \mathsf{e}')[\lambda\, \mathsf{x} : \mathsf{A}.\, \langle \mathsf{F}[\mathsf{x}] \rangle / \mathsf{k}]$, hence the reader might think that we could simply adopt the Dellina reduction rule. However, the rule does *not* apply to a call-by-name language, because we use a distinct syntax form for continuation application; put differently, we do not have terms of the form $\mathsf{k}\ \mathsf{e}$.

6.2.2 Parallel Reduction and Equivalence

In Figures 6.5 - 6.8, we define parallel reduction, which we use to build the notion of equivalence (Figure 6.9). The control-free fragment of the rules is the same as the ones defined for the target language of Dellina- CPS translation. Among the rest of the rules, (P-ResetS) recurses on the evaluation context F, necessiating parallel reduction on evaluation contexts.

Remark As Kameyama and Tanaka [102] point out, η-equivalence is not compatible with the semantics of `reset` in a call-by-name language. Here is a simple example showing this incompatibility:

$$\langle \mathsf{e} \rangle \equiv_\eta \langle \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}\ \mathsf{x} \rangle \triangleright_{\mathcal{R}} \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}\ \mathsf{x} \equiv_\eta \mathsf{e}$$

Since call-by-name η-expansion applies to non-values, we can turn a possibly effectful e into a function $\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}\ \mathsf{x}$. Now, the `reset` is wrapping around a value, hence we may remove it via the $\triangleright_{\mathcal{R}}$-rule. This reduction results in an η-expanded function, which can be converted back into e via η. What this means is that, if we incorporate η-equivalence, `reset` becomes totally useless on higher-order computations.

$$\frac{}{\mathsf{t} \rhd_p \mathsf{t}}\ (\text{P-Refl})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{L}\ \mathsf{e} \rhd_p \mathsf{L}\ \mathsf{e}'}\ (\text{P-List})$$

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \quad \rho \rhd_p \rho' \quad \mathsf{B} \rhd_p \mathsf{B}' \quad \sigma \rhd_p \sigma'}{\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma \rhd_p \Pi \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{B}'\ \sigma'}\ (\text{P-Pi})$$

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \quad \rho \rhd_p \rho' \quad \mathsf{e} \rhd_p \mathsf{e}'}{\lambda \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{e} \rhd_p \lambda \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{e}'}\ (\text{P-Abs})$$

$$\frac{\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma \rhd_p \Pi \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{B}'\ \sigma' \quad \mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e} \rhd_p \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{B}'\ \sigma'}\, \mathsf{x}.\, \mathsf{e}'}\ (\text{P-Rec})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{e}_0\ \mathsf{e}_1 \rhd_p \mathsf{e}_0'\ \mathsf{e}_1'}\ (\text{P-App})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1'}{(\lambda \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{e}_0)\ \mathsf{e}_1 \rhd_p \mathsf{e}_0'[\mathsf{e}_1'/\mathsf{x}]}\ (\text{P-AppBeta})$$

$$\frac{\begin{array}{c}\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma \rhd_p \Pi \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{B}'\ \sigma' \\ \mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1'\end{array}}{(\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e}_0)\ \mathsf{e}_1 \rhd_p \mathsf{e}_0'[\mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}'\ \rho'.\, \mathsf{B}'\ \sigma'}\, \mathsf{x}.\, \mathsf{e}_0'/\mathsf{f}, \mathsf{e}_1'/\mathsf{x}]}\ (\text{P-AppMu})$$

Figure 6.5: Call-by-name Parallel Reduction (Types and λ-terms)

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{suc\ e} \rhd_p \mathsf{suc\ e}'} \ (\text{P-Suc})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 \rhd_p\ ::\ \mathsf{e}_0'\ \mathsf{e}_1'\ \mathsf{e}_2'} \ (\text{P-Cons})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm\ e\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \to \mathsf{e}_1 \,|\, \mathsf{suc\ n} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm\ e}' \mathsf{\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P}' \mathsf{\ with\ z} \to \mathsf{e}_1' \,|\, \mathsf{suc\ n} \to \mathsf{e}_2'\end{array}} \ (\text{P-MatchN})$$

$$\frac{\mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{pm\ z\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \to \mathsf{e}_1 \,|\, \mathsf{suc\ n} \to \mathsf{e}_2 \rhd_p \mathsf{e}_1'} \ (\text{P-MatchZero})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\mathsf{pm\ suc\ e\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \to \mathsf{e}_1 \,|\, \mathsf{suc\ n} \to \mathsf{e}_2 \rhd_p \mathsf{e}_2'[\mathsf{e}'/\mathsf{n}]} \ (\text{P-MatchSuc})$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{l}\mathsf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \to \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm\ e}' \mathsf{\ as\ x\ in\ L\ a\ ret\ P}' \mathsf{\ with\ nil} \to \mathsf{e}_1' \,|\, ::\ \mathsf{m\ h\ t} \to \mathsf{e}_2'\end{array}} \ (\text{P-MatchL})$$

$$\frac{\mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{pm\ nil\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \to \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \to \mathsf{e}_2 \rhd_p \mathsf{e}_1'} \ (\text{P-MatchNil})$$

$$\frac{\mathsf{e}_0 \rhd_p \mathsf{e}_0' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2' \quad \mathsf{e}_3 \rhd_p \mathsf{e}_3'}{\begin{array}{l}\mathsf{pm}\ ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 \mathsf{\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \to \mathsf{e}_1 \,|\, ::\ \mathsf{m\ h\ t} \to \mathsf{e}_2 \rhd_p \\ \mathsf{e}_3'[\mathsf{e}_0'/\mathsf{m}, \mathsf{e}_1'/\mathsf{h}, \mathsf{e}_2'/\mathsf{t}]\end{array}} \ (\text{P-MatchCons})$$

Figure 6.6: Call-by-name Parallel Reduction (Inductive Data)

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \quad \alpha \rhd_p \alpha' \quad \mathsf{e} \rhd_p \mathsf{e}'}{\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} \rhd_p \mathcal{S}\mathsf{k} : \mathsf{A}' \to \alpha'.\, \mathsf{e}'} \text{ (P-SHIFT)}$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\mathsf{k} \hookrightarrow \mathsf{e} \rhd_p \mathsf{k} \hookrightarrow \mathsf{e}'} \text{ (P-THROW)}$$

$$\frac{\mathsf{e} \rhd_p \mathsf{e}'}{\langle \mathsf{e} \rangle \rhd_p \langle \mathsf{e}' \rangle} \text{ (P-RESET)}$$

$$\frac{\mathsf{A} \rhd_p \mathsf{A}' \quad \mathsf{F} \rhd_p \mathsf{F}' \quad \mathsf{e} \rhd_p \mathsf{e}'}{\langle \mathsf{F}[\mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e}] \rangle \rhd_p \langle \mathsf{e}'[\mathsf{k} \Rightarrow \mathsf{F}'] \rangle} \text{ (P-RESETS)}$$

$$\frac{\mathsf{v} \rhd_p \mathsf{v}'}{\langle \mathsf{v} \rangle \rhd_p \mathsf{v}'} \text{ (P-RESETV)}$$

$$\frac{}{\mathsf{F} \rhd_p \mathsf{F}} \text{ (P-FREFL)}$$

$$\frac{\mathsf{F} \rhd_p \mathsf{F}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1'}{\mathsf{F}\ \mathsf{e}_1 \rhd_p \mathsf{F}'\ \mathsf{e}_1'} \text{ (P-FAPP)}$$

$$\frac{\mathsf{F} \rhd_p \mathsf{F}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{c}\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm}\ \mathsf{F}'\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}'\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1' \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2'\end{array}} \text{ (P-FMATCHN)}$$

$$\frac{\mathsf{F} \rhd_p \mathsf{F}' \quad \mathsf{P} \rhd_p \mathsf{P}' \quad \mathsf{e}_1 \rhd_p \mathsf{e}_1' \quad \mathsf{e}_2 \rhd_p \mathsf{e}_2'}{\begin{array}{c}\mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 \rhd_p \\ \mathsf{pm}\ \mathsf{F}'\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}'\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1' \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2'\end{array}} \text{ (P-FMATCHL)}$$

Figure 6.7: Call-by-name Parallel Reduction (Control Operators and Evaluation Contexts)

$$\frac{}{\mathsf{t} \triangleright_p^\star \mathsf{t}}\ (\text{PS-Refl}) \qquad \frac{\mathsf{t}_0 \triangleright_p \mathsf{t}_1 \quad \mathsf{t}_1 \triangleright_p^\star \mathsf{t}_2}{\mathsf{t}_0 \triangleright_p^\star \mathsf{t}_2}\ (\text{PS-Trans})$$

$$\frac{}{\mathsf{F} \triangleright_p^\star \mathsf{F}}\ (\text{PS-FRefl}) \qquad \frac{\mathsf{F}_0 \triangleright_p \mathsf{F}_1 \quad \mathsf{F}_1 \triangleright_p^\star \mathsf{F}_2}{\mathsf{F}_0 \triangleright_p^\star \mathsf{F}_2}\ (\text{PS-FTrans})$$

Figure 6.8: Call-by-name Parallel Reduction (Reflexivity and Transitivity)

$$\frac{\mathsf{t}_0 \triangleright_p^\star \mathsf{t} \quad \mathsf{t}_1 \triangleright_p^\star \mathsf{t}}{\mathsf{t}_1 \equiv \mathsf{t}_2}\ (\equiv)$$

Figure 6.9: Call-by-name Equivalence

6.3 Typing

Now we have reached the most delicate part of the language: typing rules. The call-by-name type system shares one key principle with the call-by-value one, namely the three restrictions on type dependency. However, since certain subterms are not eagerly evaluated under the call-by-name strategy, not every typing rule is trivially derived from its call-by-value counterpart. This section is devoted to demistify the interaction between dependent types and call-by-name control effects, focusing on the asymmetry in the rules for dependent constructions, as well as the meaning of continuation application.

Environments, Kinds, and Types Let us begin with rules for typing environments (Figure 6.10). As we saw in Section 6.1, extension by an ordinary variable x comes with effect information ρ, therefore we check well-formedness of ρ in rule (G-Ext). The other extension rule, (G-ExtCont), adds a continuation variable k. Notice that the associated type is required to be of the form $\mathsf{A} \to \alpha$. This type communicates the following facts:

1. k receives a pure argument
2. k has a pure body
3. k is a non-dependent function

$\boxed{\text{Well-formed Environments } \vdash \Gamma}$

$$\frac{}{\vdash \bullet}\ (\textsc{G-Empty}) \qquad \frac{\vdash \Gamma \quad \Gamma \vdash A : * \quad \Gamma \vdash \rho}{\vdash \Gamma, x : A\ \rho}\ (\textsc{G-Ext})$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash A \to \alpha : *}{\vdash \Gamma, k :_k A \to \alpha}\ (\textsc{G-ExtCont})$$

$\boxed{\text{Well-formed Types } \Gamma \vdash A : *}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Unit} : *}\ (\textsc{T-Unit}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : *}\ (\textsc{T-Nat}) \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{L}\ e : *}\ (\textsc{T-List})$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \quad \Gamma \vdash \rho \\ \Gamma, x : A\ \rho \vdash B : * \quad \Gamma, x : A\ \rho \vdash \sigma\end{array}}{\Gamma \vdash \Pi x : A\ \rho. B\ \sigma : *}\ (\textsc{T-Pi})$$

Figure 6.10: Call-by-name Environments, Kinds, and Types

The latter two should sound familiar to the reader: continuations have a `reset` operator surrounding their body, and they must have a non-dependent function type to guarantee well-formedness of the initial answer type. The first item, on the other hand, might come as a surprise: since our language is call-by-name, and since continuations are functions, shouldn't k accept an impure argument as well? It turns out that this argument is not quite right in Dellina^n. Evaluation contexts are functions, but they are *not* call-by-name functions. Recall that, in the definition of evaluation contexts E, each occurrence of E tells us which subterm we must reduce to a value during the evaluation of the whole term. That is, the hole of contexts is supposed to be plugged with a pure value, not by a potentially effectful computation. What this suggests is that evaluation contexts are *call-by-value* functions. The value-accepting nature of continuations is explicit in the existing type systems for call-by-name `shift` and `reset` as well: they all assign a pure function type to continuations, even if they do not employ a purity distinction in other typing rules.

The rest of the rules in Figure 6.10 are for kinds and types. There is no major change from Dellina- rules from Section 3.3; the only difference is that (T-Pi) additionaly checks well-formedness of the argument effect annotation ρ.

λ-terms Figure 6.11 shows typing rules for λ-terms. Variables may be concluded as being impure, depending on whether the effect annotation ρ is empty or not. Abstractions and recursive functions are pure values, but they may receive effectful computations. Notice that, in the premise of (E-Rec), the variable f carries no effect annotation, since it represents a function.

The most interesting rule in this figure is (E-DApp). Unlike Dellina-, the call-by-name language has one single rule for application, which accounts for the dependent case. By taking a closer look at the rule, we find that it does not impose a purity restriction on the argument e_1. The relaxation comes from the fact that a call-by-name language allows replacing a variable by an impure computation. In other words, the substitution $\mathsf{B}[\mathsf{e}_1/\mathsf{x}]$ makes sense even if e_1 is not a pure term. This is not the case in a call-by-value language; indeed, we equipped Dellina- with two application rules exactly to avoid substitution of impure computations.

We must however make sure that the post-substitution type is well-formed.

$\boxed{\text{Well-typed Terms } \Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho}$

$$\frac{\vdash \Gamma \qquad \mathsf{x} : \mathsf{A}\ \rho \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{A}\ \rho}\ (\text{E-Var})$$

$$\frac{\Gamma, \mathsf{x} : \mathsf{A}\ \rho \vdash \mathsf{e} : \mathsf{B}\ \sigma}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\ (\text{E-Abs})$$

$$\frac{\begin{array}{c}\Gamma, \mathsf{f} : \Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma, \mathsf{x} : \mathsf{A}\ \rho \vdash \mathsf{e} : \mathsf{B}\ \sigma \\ \Gamma \vdash \Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma : * \qquad \text{guard}(\mathsf{f}, \mathsf{x}, \mathsf{e}, \{\,\})\end{array}}{\Gamma \vdash \mathsf{rec}\ \mathsf{f}_{\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\, \mathsf{x}.\, \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}\ \sigma}\ (\text{E-Rec})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_0 : \Pi \mathsf{x} : \mathsf{A}\ \sigma.\, \mathsf{B}\ \tau\ \rho \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{A}\ \sigma \\ \Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : * \qquad \nu = \text{comp}(\rho, \tau[\mathsf{e}_1/\mathsf{x}]) \qquad \Gamma \vdash \nu\end{array}}{\Gamma \vdash \mathsf{e}_0\ \mathsf{e}_1 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}]\ \nu}\ (\text{E-DApp})$$

Figure 6.11: Call-by-name Typing Rules (λ-terms)

That is, the term it depends on may contain e_1 but is a pure term as a whole. The latter condition, which we check via the well-formedness premise $\Gamma \vdash \mathsf{B}[\mathsf{e}_1/\mathsf{x}] : *$, is satisfied when x is surrounded by a `reset`, or it appears as a subterm that needs not be evaluated, or it does not show up at all. That is, when B is one of the following:

$$\mathsf{L}\ \langle \mathsf{x} \rangle \qquad \mathsf{L}\ (\mathsf{suc}\ \mathsf{x}) \qquad \mathsf{L}\ ((\lambda\, \mathsf{a}.\, \lambda\, \mathsf{b}.\, \mathsf{a})\ \mathsf{z}\ (\lambda\, \mathsf{c}.\, \mathsf{x})) \qquad \mathsf{L}\ ((\lambda\, \mathsf{a}.\, \lambda\, \mathsf{b}.\, \mathsf{a})\ \mathsf{z}\ \mathsf{x}) \qquad \mathsf{L}\ \mathsf{z}$$

the rule admits an impure e_1, since substitution of e_1 for x results in a pure index (note that constructor application to impure terms is pure under the call-by-name semantics). However, when B is one of the following:

$$\mathsf{L}\ \mathsf{x} \qquad \mathsf{L}\ ((\lambda\, \mathsf{a}.\, \mathsf{x})\ \mathsf{z}) \qquad \mathsf{L}\ (\mathsf{pm}\ \mathsf{x}\ \mathsf{as}\ \mathsf{y}\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{z} \,|\, \mathsf{suc}\ \mathsf{n} \to \mathsf{suc}\ \mathsf{z})$$

the rule only admits a pure e_1, since otherwise the list index would be impure.

If we incorporated the rules (E-DApp) and (E-NDApp) from Dellina- (with

$$\frac{\vdash \Gamma}{\Gamma \vdash () : \mathsf{Unit}}\ (\text{E-}\textsc{Unit})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ (\text{E-}\textsc{Zero}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ \mathsf{e} : \mathbb{N}}\ (\text{E-}\textsc{Suc})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nil} : \mathsf{L}\ \mathsf{z}}\ (\text{E-}\textsc{Nil}) \qquad \frac{\Gamma \vdash \mathsf{e}_0 : \mathbb{N} \quad \Gamma \vdash \mathsf{e}_1 : \mathbb{N} \quad \Gamma \vdash \mathsf{e}_2 : \mathsf{L}\ \mathsf{e}_0}{\Gamma \vdash\ ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0)}\ (\text{E-}\textsc{Cons})$$

Figure 6.12: Call-by-name Typing Rules (Inductive Data)

some modifications to effect annotations), and only allowed pure arguments in the dependent case, we would not need the well-formedness presmise of the result type, because the call-by-name variant of the substitution lemma can be stated for computations, not just values. However, this would rule out too many dependent applications that are actually safe, *i.e.*, those involving substitution of an impure argument but resulting in a well-formed type. Since we would like to make our type system as generous as possible, we do not restrict argument effects but keep the well-formedness premise instead.

Putting dependency aside, there is another change in the application rule. Observe that the effect composition operator takes in the annotation of the function (ρ) as well as that of the function's body (τ), but not the annotation of the argument (σ). This reflects the call-by-name evaluation of application: arguments are evaluated after β-reduction, and hence their effect is included in the effect τ of the function's body. As a consequence, an application is classified as pure when the function and its body are pure but the argument is impure.

Inductive Data The rules for inductive data (Figure 6.12) are also worth spending some time on. As stated earlier, a Dellina^n inductive datum is always a value, even if the arguments to the constructor are non-values. This means any constructor application has an empty effect annotation in its typing derivation. Then, how can we make the evaluation of constructor arguments happen? The answer is by destructing data via pattern matching and then using the pattern variables in the branches. In a call-by-name language, pattern variables may represent impure computations, and correspondingly, constructor types may carry effect annotations of their arguments. This broadens the scope of types we can assign to constructors, but it also makes construction of inductive data less flexible. In particular, if a constructor is declared as receiving a pure argument, it can never be applied to an impure argument, and vise versa.

In Dellina^n, we define natural numbers and lists as entirely pure datatypes, that is, the constructors suc and $::$ only accept pure arguments. With this definition, we cannot write things like $\mathsf{suc}\ (\mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{N}.\, \mathsf{z})$. If we wish to make this datum well-typed, we must modify (E-Suc) in the following way:

$$\frac{\Gamma \vdash \mathsf{e} : \mathbb{N}[\mathbb{N}, \mathbb{N}]}{\Gamma \vdash \mathsf{suc}\ \mathsf{e} : \mathbb{N}}\ (\text{E-S}\textsc{uc})$$

But this time, we are unable to write $\mathsf{suc}\ \mathsf{z}$, which is far more natural as an inhabitant of type $\mathbb{N}$.

It is instructive to note that we may define lists as data consisting of impure elements, if we replace (E-Cons) by a different rule. However, in that case, the control effect of all elements must be uniform: *i.e.*, we cannot build a list like the following one:

$$[\mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{N}.\, \mathsf{z};\ \mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{B}.\, ()]$$

Pattern Matching We saw that, in Dellina^n, we have one elimination rule of function types, which derives a dependent application. Then, by analogy, the reader might expect that we have one elimination rule for each datatype, which derives a dependent pattern matching. However, we found that call-by-name pattern matching must be equipped with two separate rules, each accounting for dependent and non-dependent cases.

The asymmetry between application and pattern matching originates from our earlier observation that call-by-name inductive data are always pure. Suppose all we have for pattern matching on natural numbers was the following dependent rule:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N} \qquad \Gamma, \mathsf{x} : \mathbb{N} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{x}] \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ n}/\mathsf{x}]\ \rho[\mathsf{suc\ n}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}[\mathsf{e}/\mathsf{x}]\ \rho[\mathsf{e}/\mathsf{x}]} \text{ (E-DMatchN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho \qquad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \qquad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ \_\ in\ } \mathbb{N} \mathsf{\ ret\ P\ with\ z} \rightarrow \mathsf{e}_1 \mid \mathsf{suc\ n} \rightarrow \mathsf{e}_2 : \mathsf{P}\ \tau} \text{ (E-NDMatchN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{L\ n} \qquad \Gamma, \mathsf{a} : \mathbb{N}, \mathsf{x} : \mathsf{L\ a} \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}]\ \rho[\mathsf{z}/\mathsf{a}, \mathsf{nil}/\mathsf{x}] \\ \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L\ m} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ m}/\mathsf{a}, ::\ \mathsf{m\ h\ t}/\mathsf{x}]\ \rho[\mathsf{suc\ m}/\mathsf{a}, ::\ \mathsf{m\ h\ t}/\mathsf{x}]\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in\ L\ a\ ret\ P\ with\ nil} \rightarrow \mathsf{e}_1 \mid ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2 : \mathsf{P}[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]\ \rho[\mathsf{n}/\mathsf{a}, \mathsf{e}/\mathsf{x}]} \text{ (E-DMatchL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : (\mathsf{L\ n})\ \rho \qquad \Gamma \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}\ \sigma \qquad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L\ m} \vdash \mathsf{e}_2 : \mathsf{P}\ \sigma \qquad \tau = \mathrm{comp}(\rho, \sigma)\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ \_\ in\ L\ \_\ ret\ P\ with\ nil} \rightarrow \mathsf{e}_1 \mid ::\ \mathsf{m\ h\ t} \rightarrow \mathsf{e}_2 : \mathsf{P}\ \tau} \text{ (E-NDMatchL)}$$

Figure 6.13: Call-by-name Typing Rules (Pattern Matching)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathbb{N}\ \rho \qquad \Gamma, \mathsf{x} : \mathbb{N}\ \rho \vdash \mathsf{P} : * \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{x}]\ \tau[\mathsf{z}/\mathsf{x}] \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ n}/\mathsf{x}]\ \tau[\mathsf{suc\ n}/\mathsf{x}] \\ \Gamma \vdash \mathsf{P}[\mathsf{e}/\mathsf{x}] : * \qquad \tau = \mathrm{comp}(\rho, \mathsf{P}[\mathsf{e}/\mathsf{x}]) \qquad \Gamma \vdash \tau\end{array}}{\Gamma \vdash \mathsf{pm\ e\ as\ x\ in}\ \mathbb{N}\ \mathsf{ret\ P\ with\ z} \to \mathsf{e}_1 \mid \mathsf{suc\ n} \to \mathsf{e}_2 : \mathsf{P}[\mathsf{e}/\mathsf{x}]\ \tau}\ \text{(E-DMatchN)}$$

We see that the scrutinee e and its placeholder x has the same effect annotation. This perfectly makes sense, because we will eventually replace any reference to x from P with e in the conclusion. We then find that the variable x is replaced by the patterns z and suc m in the two branches. These patterns have a fixed purity, since call-by-name inductive data are uniformly pure. Now we see a problem: when e is an impure term, substitution of patterns is not type-safe. The mismatch seems to suggest that dependent pattern matching is inherently defined on pure data.

With this in mind, let us look at the actual rules for pattern matching (Figure 6.13). In the dependent rule (E-DMatchN), we have a pure scrutinee e, and a pure placeholder x. Compared to the call-by-value variant of the rule, it lacks the well-formed premise of the result type $\mathsf{P}[\mathsf{e}/\mathsf{x}]$, but it is not needed because call-by-name languages admit substitution of computations, and we know e is a pure term. In the non-dependent rule (E-NDMatchN), the scrutinee is impure, therefore we require the predictate P to be closed under environment Γ. The rules for pattern matching on lists can be understood in a similar way.

Control Operators and Conversion Lastly, we look at the rules for control operators and conversion (Figure 6.14). The rules for `shift` and `reset` remain unchanged from the call-by-value language, but we now have additional rules for continuation application. We have two rules for this language construct, differing in the purity of the argument e. When e is pure, we type $\mathsf{k} \hookrightarrow \mathsf{e}$ just like an ordinary, non-dependent application, hence the result type is α. Note that the result type does not depend on e because continuations are non-dependent functions. When e is impure, we see that its initial answer type coincides with the return type α of k, and that the result type of this application is the final answer type β of e. These answer types tell us what a throwing construct $\mathsf{k} \hookrightarrow \mathsf{e}$ does for us: it *evaluates* the

$$\frac{\Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \beta \quad \text{or} \quad \Gamma, \mathsf{k} : \mathsf{A} \to \alpha \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \beta] \qquad \Gamma \vdash \beta : *}{\Gamma \vdash \mathcal{S}\mathsf{k} : \mathsf{A} \to \alpha.\, \mathsf{e} : \mathsf{A}[\alpha, \beta]} \ (\text{E-Shift})$$

$$\frac{\mathsf{k} :_k \mathsf{A} \to \alpha \in \Gamma \qquad \Gamma \vdash \mathsf{e} : \mathsf{A}}{\Gamma \vdash \mathsf{k} \hookrightarrow \mathsf{e} : \alpha} \ (\text{E-ThrowP})$$

$$\frac{\mathsf{k} :_k \mathsf{A} \to \alpha \in \Gamma \qquad \Gamma \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]}{\Gamma \vdash \mathsf{k} \hookrightarrow \mathsf{e} : \beta} \ (\text{E-ThrowI})$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \quad \text{or} \quad \Gamma \vdash \mathsf{e} : \mathsf{B}[\mathsf{B}, \mathsf{A}]}{\Gamma \vdash \langle \mathsf{e} \rangle : \mathsf{A}} \ (\text{E-Reset})$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \qquad \Gamma \vdash \mathsf{B} : * \qquad \Gamma \vdash \sigma \qquad \mathsf{A} \equiv \mathsf{B} \qquad \rho \equiv \sigma}{\Gamma \vdash \mathsf{e} : \mathsf{B}\ \sigma} \ (\text{E-Conv})$$

Figure 6.14: Call-by-name Typing Rules (Control Constructs and Conversion)

argument e in the context k. In terms of CPS, the computation can be described as running $\mathsf{e}^{\div}$ (of type $(\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}$) with a continuation $\mathbf{k}$ (of type $\mathsf{A}^{+} \to \alpha^{+}$). This shows the semantic difference between function and continuation application: the former suspends evaluation of arguments, whereas the latter forces it.

Pure Evaluation Contexts In Dellina^{n}, we have an extra set of typing rules for pure evaluation contexts. The need for these rules originates from the way we eliminate a `shift`-redex. Unlike the call-by-value rule, which replaces the continuation variable k with a function $\lambda\,\mathsf{x} : \mathsf{A}.\,\langle \mathsf{F}[\mathsf{x}] \rangle$, the call-by-name rule replaces k with a pure evaluation context F, which is not a Dellina^{n} term. Now, recall that the preservation property requires a substitution lemma. Since we may replace variables only with objects of the correct type, we need rules for reasoning about the type of pure contexts.

The context typing rules, presented in Figure 6.15, are built on the rules of Biernacka and Biernacki [27], with some refinements for accommodating type dependency and the fine-grained purity distinction. The first rule is simple: an empty context has a pure, non-dependent function type, where the input and output types are the same. The second rule builds a context for evaluating the function part of an application, where the function has a pure body. It would be helpful to recall the following facts:

- Extending F to $\mathsf{F}\ \mathsf{e}$ means evaluating an extra application before going on with F.
- The initial answer type of a term is determined by the final answer type of later computation.

Back to (F-AppP), we see that the return type of the extended context $\mathsf{F}\ \mathsf{e}$ remains unchanged. The reason is that the application has no control effects, that is, evaluating this extra application does not affect the type of the context required by later computation. On the other hand, the hole type has changed from $\mathsf{B}[\mathsf{e}/\mathsf{x}]$ to $\Pi\,\mathsf{x} : \mathsf{A}\ \rho.\,\mathsf{B}$, because the context now accepts a function.

The third rule is yet another typing rule for the application context, but this time the function has an impure body. Observe that the type α in the conclusion's hole type comes from the return type of F, meaning that the hole of F must be

$\boxed{\text{Well-typed Pure Contexts } \Gamma \vdash \mathsf{F} : \mathsf{A} \to \alpha}$

$$\frac{\Gamma \vdash \mathsf{A} : *}{\Gamma \vdash [\,] : \mathsf{A} \to \mathsf{A}} \text{ (F-Hole)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \qquad \Gamma \vdash \mathsf{F} : \mathsf{B}[\mathsf{e}/\mathsf{x}] \to \alpha}{\Gamma \vdash \mathsf{F}\ \mathsf{e} : (\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}) \to \alpha} \text{ (F-AppP)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A}\ \rho \qquad \Gamma \vdash \mathsf{F} : \mathsf{B}[\mathsf{e}/\mathsf{x}] \to \alpha[\mathsf{e}/\mathsf{x}] \qquad \Gamma, \mathsf{x} : \mathsf{A}\ \rho \vdash \beta : *}{\Gamma \vdash \mathsf{F}\ \mathsf{e} : (\Pi \mathsf{x} : \mathsf{A}\ \rho.\, \mathsf{B}[\alpha, \beta]) \to \beta[\mathsf{e}/\mathsf{x}]} \text{ (F-AppI)}$$

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{P} \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \,|\, \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 : \mathbb{N} \to \alpha} \text{ (F-MatchNP)}$$

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\alpha, \beta] \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\alpha, \beta]}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \,|\, \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 : \mathbb{N} \to \beta} \text{ (F-MatchNI)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{n} : \mathbb{N} \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P} \qquad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \,|\, ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{L}\ \mathsf{n} \to \alpha} \text{ (F-MatchLP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{n} : \mathbb{N} \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\alpha, \beta] \qquad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}[\alpha, \beta]\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \,|\, ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{L}\ \mathsf{n} \to \beta} \text{ (F-MatchLI)}$$

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{A} \to \alpha \qquad \Gamma \vdash \mathsf{A}' : * \qquad \Gamma \vdash \alpha' : * \qquad \mathsf{A} \equiv \mathsf{A}' \qquad \alpha \equiv \alpha'}{\Gamma \vdash \mathsf{F} : \mathsf{A}' \to \alpha'} \text{ (F-Conv)}$$

Figure 6.15: Call-by-name Typed Pure Contexts

filled in with a function whose body requires an α-returning context. Notice also that the return type of the new context is updated to $\beta[\mathsf{e}/\mathsf{x}]$, representing what we obtain after the application. This means, if we want to evaluate a term before the impure application to e, it must be something that requires a $\beta[\mathsf{e}/\mathsf{x}]$-returning context.

Analogously to application, pattern matching contexts are given two rules, which differ in the purity of the branches. By taking a closer look at the conclusions, we find that all rules have a dummy symbol _ representing the scrutinee and the index. This means, the rules derive a context that forms a non-dependent pattern matching.

It would be interesting to see what happens if we try to define a dependent variant of (F-MatchNP). The rule builds a context that performs an additional pattern matching on the given number before computing F. Since the return type α depends on the number to be analyzed, the extended context will have type $\Pi\,\mathsf{x} : \mathbb{N}.\,\alpha$. However, in Dellina^n, this is not a valid type for contexts, because we do not allow dependent continuations. Indeed, the dependent rule would require F to be well-formed under an extended environment $\Gamma, \mathsf{x} : \mathbb{N}$, as shown below:

$$\frac{\Gamma, \mathsf{x} : \mathbb{N} \vdash \mathsf{F} : \mathsf{P} \to \alpha \quad \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\mathsf{z}/\mathsf{x}] \quad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\mathsf{suc\ n}/\mathsf{x}]}{\Gamma \vdash \mathsf{pm\ F\ as\ x\ in}\ \mathbb{N}\ \mathsf{ret\ P\ with\ z} \to \mathsf{e}_1 \,|\, \mathsf{suc\ n} \to \mathsf{e}_2 : \Pi\,\mathsf{x} : \mathbb{N}.\,\alpha} \text{(F-DMatchNP)}$$

The last rule is the conversion rule for contexts. As in the conversion rule for terms, we require the new input and output types to be well-formed, and be equivalent to the old ones.

6.4 CPS Translation

In this section, we study how to CPS translate Dellina^n. The translation is essentially a selective version of Kameyama and Tanaka's call-by-name translation [102]. As we will see shortly, the call-by-name translation yields more suspended computations at the level of terms, as well as arrows at the level of types, both of which come from the lazy semantics of the source language. Note that the translation targets the same language as the call-by-value translation; the reader may revisit Section 4.4 for specifications.

$$\frac{}{\vdash \bullet}\ (\textsc{G-Empty}) \overset{+}{\rightsquigarrow} \bullet$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \mathsf{A} : * \quad \Gamma \vdash \rho}{\vdash \Gamma, \mathsf{x} : \mathsf{A}\ \rho}\ (\textsc{G-Ext})$$

$$\overset{+}{\rightsquigarrow} \Gamma^+, \mathbf{x} : \mathsf{A}^+ \text{ if } \rho = \epsilon$$

$$\overset{+}{\rightsquigarrow} \Gamma^+, \mathbf{x} : (\mathsf{A}^+ \to \alpha^+) \to \beta^+ \text{ if } \rho = [\alpha, \beta]$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \mathsf{A} \to \alpha : *}{\vdash \Gamma, \mathsf{k} :_k \mathsf{A} \to \alpha}\ (\textsc{G-ExtCont}) \overset{+}{\rightsquigarrow} \Gamma^+, \mathbf{k} : \mathsf{A}^+ \to \alpha^+$$

Figure 6.16: CPS Translation of Typing Environments

Let us first look at the translation of environments (Figure 6.16). The rule (G-Ext), which extends an environment with an ordinary variable x, has two distinct CPS images depending on the presence of the effect annotation ρ. When ρ is empty, the translation behaves the same as the call-by-value variant, that is, it maps x to a target variable $\mathbf{x}$ of type A^+. When ρ is a pair $[\alpha, \beta]$, the translation gives $\mathbf{x}$ a type of the form $(\mathsf{A}^+ \to \alpha^+) \to \beta^+$. This means, when x represents an effectful computation—which requires a context to be evaluated—in the source, $\mathbf{x}$ represents a suspended computation—which requires a continuation to be evaluated—in the target. In contrast, the rule (G-ExtCont), which extends an environment with a continuation variable k, has one unique CPS image $\mathbf{k}$ of type $\mathsf{A}^+ \to \alpha^+$. The reason we do not defer evaluation of $\mathbf{k}$ is that any source continuation k is a pure function.

Having seen the translation of (G-Ext), it would be fairly easy to figure out what is happening in the translation of kinds and types (Figure 6.17). The only change is in the translation of function types: it has four instead of two different images, since both the domain and co-domain may have an effect annotation.

The translation of variables and functions is also straightforward; the only thing worth noting is that variables have two CPS images: a direct-style variable for pure x, and a suspended computation for impure x.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : * \quad \Gamma \vdash \rho \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : * \quad \Gamma, \mathsf{x} : \mathsf{A} \vdash \sigma\end{array}}{\Gamma \vdash \Pi \mathsf{x} : \mathsf{A}\ \rho. \mathsf{B}\ \sigma : *}\ \text{(T-Pi)}$$

$\overset{+}{\rightsquigarrow}$ $\mathbf{\Pi}\,\mathbf{x} : \mathsf{A}^{+}. \mathsf{B}^{+}$
if $\rho = \sigma = \epsilon$

$\overset{+}{\rightsquigarrow}$ $\mathbf{\Pi}\,\mathbf{x} : \mathsf{A}^{+}. (\mathsf{B}^{+} \to \alpha^{+}) \to \beta^{+}$
if $\rho = \epsilon, \sigma = [\alpha, \beta]$

$\overset{+}{\rightsquigarrow}$ $\mathbf{\Pi}\,\mathbf{x} : (\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}. \mathsf{B}^{+}$
if $\rho = [\alpha, \beta], \sigma = \epsilon$

$\overset{+}{\rightsquigarrow}$ $\mathbf{\Pi}\,\mathbf{x} : (\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}. (\mathsf{B}^{+} \to \gamma^{+}) \to \delta^{+}$
if $\rho = [\alpha, \beta], \sigma = [\gamma, \delta]$

$$\frac{\vdash \Gamma \quad \mathsf{x} : \mathsf{A}\ \rho \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{A}\ \rho}\ \text{(E-Var)}$$

$\overset{+}{\rightsquigarrow}$ $\mathbf{x}$ if $\rho = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : \mathsf{A}^{+} \to \alpha^{+}. \mathbf{x}\ \mathbf{k}$ if $\rho = [\alpha, \beta]$

$$\frac{\Gamma, \mathsf{x} : \mathsf{A}\ \rho \vdash \mathsf{e} : \mathsf{B}\ \sigma}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}\ \rho. \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}\ \rho. \mathsf{B}\ \sigma}\ \text{(E-Abs)}$$

$\overset{+}{\rightsquigarrow}$ $\lambda\,\mathbf{x} : \mathsf{A}^{+}. \mathsf{e}^{+}$ if $\rho = \sigma = \epsilon$

$\overset{+}{\rightsquigarrow}$ $\lambda\,\mathbf{x} : \mathsf{A}^{+}. \mathsf{e}^{\div}$ if $\rho = \epsilon, \sigma = [\alpha, \beta]$

$\overset{+}{\rightsquigarrow}$ $\lambda\,\mathbf{x} : (\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}. \mathsf{e}^{+}$ if $\rho = [\alpha, \beta], \sigma = \epsilon$

$\overset{+}{\rightsquigarrow}$ $\lambda\,\mathbf{x} : (\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}. \mathsf{e}^{\div}$ if $\rho = [\alpha, \beta], \sigma = [\gamma, \delta]$

Figure 6.17: CPS Translation of Types, Variables, and Abstractions

$$\frac{\Gamma \vdash \mathsf{e_0} : \Pi\,\mathsf{x} : \mathsf{A}\ \sigma.\,\mathsf{B}\ \tau\ \rho \qquad \Gamma \vdash \mathsf{e_1} : \mathsf{A}\ \sigma \qquad \Gamma \vdash \mathsf{B}[\mathsf{e_1}/\mathsf{x}] : * \qquad \nu = \mathrm{comp}(\rho, \tau[\mathsf{e_1}/\mathsf{x}]) \qquad \Gamma \vdash \nu}{\Gamma \vdash \mathsf{e_0}\ \mathsf{e_1} : \mathsf{B}[\mathsf{e_1}/\mathsf{x}]\ \nu}\ \text{(E-DApp)}$$

$\overset{+}{\rightsquigarrow}$ $\mathsf{e_0}^+\ \mathsf{e_1}^+$
if $\rho = \sigma = \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow (\alpha[\mathsf{e_1}/\mathsf{x}])^+.\,\mathsf{e_0}^+\ \mathsf{e_1}^+\ \mathbf{k}$
if $\rho = \sigma = \epsilon, \tau = [\alpha, \beta]$

$\overset{+}{\rightsquigarrow}$ $\mathsf{e_0}^+\ \mathsf{e_1}^\div$
if $\rho = \tau = \epsilon, \sigma = [\alpha, \beta]$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow (\alpha[\mathsf{e_1}/\mathsf{x}])^+.\,\mathsf{e_0}^+\ \mathsf{e_1}^\div\ \mathbf{k}$
if $\rho = \epsilon, \sigma = [\beta[\mathsf{e_1}/\mathsf{x}], \gamma], \tau = [\alpha, \beta]$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow \alpha^+.\,\mathsf{e_0}^\div\ (\lambda\,\mathbf{v_0} : \mathbf{\Pi}\,\mathbf{x} : \mathsf{A}^+.\,\mathsf{B}^+.\,\mathbf{k}\ (\mathbf{v_0}\ \mathsf{e_1}^+))$
if $\rho = [\alpha, \beta], \sigma = \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow (\alpha[\mathsf{e_1}/\mathsf{x}])^+.\,\mathsf{e_0}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \rightarrow (\mathsf{B}^+ \rightarrow \alpha^+) \rightarrow \beta^+.\,\mathbf{v_0}\ \mathsf{e_1}^+\ \mathbf{k})$
if $\rho = [\beta[\mathsf{e_1}/\mathsf{x}], \gamma], \sigma = \epsilon, \tau = [\alpha, \beta]$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow \alpha^+.\,\mathsf{e_0}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \rightarrow \mathsf{B}^+.\,\mathbf{k}\ (\mathbf{v_0}\ \mathbf{e_1}^\div))$
if $\rho = [\beta, \gamma], \sigma = [\alpha, \beta], \tau = \epsilon$

$\overset{\div}{\rightsquigarrow}$ $\lambda\,\mathbf{k} : (\mathsf{B}[\mathsf{e_1}/\mathsf{x}])^+ \rightarrow (\alpha[\mathsf{e_1}/\mathsf{x}])^+.\,\mathsf{e_0}^\div\ (\lambda\,\mathbf{v_0} : \mathsf{A}^+ \rightarrow (\mathsf{B}^+ \rightarrow \alpha^+) \rightarrow \beta^+.\,\mathbf{v_0}\ \mathbf{e_1}^\div\ \mathbf{k})$
if $\rho = [\gamma, \delta], \sigma = [\beta[\mathsf{e_1}/\mathsf{x}], \gamma], \tau = [\alpha, \beta]$

Figure 6.18: CPS Translation of Application

$$\frac{\vdash \Gamma}{\Gamma \vdash () : \mathsf{Unit}} \text{(E-UNIT)} \overset{+}{\rightsquigarrow} ()$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{z} : \mathbb{N}} \text{(E-ZERO)} \overset{+}{\rightsquigarrow} \mathbf{z}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathbb{N}}{\Gamma \vdash \mathsf{suc\ e} : \mathbb{N}} \text{(E-SUC)} \overset{+}{\rightsquigarrow} \mathbf{suc}\ \mathsf{e}^{+}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nil} : \mathsf{L\ z}} \text{(E-NIL)} \overset{+}{\rightsquigarrow} \mathbf{nil}$$

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathbb{N} \quad \Gamma \vdash \mathsf{e}_1 : \mathbb{N} \quad \Gamma \vdash \mathsf{e}_2 : \mathsf{L}\ \mathsf{e}_0}{\Gamma \vdash\ ::\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{L}\ (\mathsf{suc}\ \mathsf{e}_0)} \text{(E-CONS)} \overset{+}{\rightsquigarrow}\ ::\ \mathsf{e}_0{}^{+}\ \mathsf{e}_1{}^{+}\ \mathsf{e}_2{}^{+}$$

$$\frac{\mathsf{k} :_k \mathsf{A} \to \alpha \in \Gamma \quad \Gamma \vdash \mathsf{e} : \mathsf{A}}{\Gamma \vdash \mathsf{k} \hookrightarrow \mathsf{e} : \alpha} \text{(E-THROWP)} \overset{\div}{\rightsquigarrow} \lambda\, \mathbf{k} : \mathsf{A}^{+} \to \alpha^{+}.\, \mathbf{k}\ \mathsf{e}^{+}$$

$$\frac{\mathsf{k} :_k \mathsf{A} \to \alpha \in \Gamma \quad \Gamma \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]}{\Gamma \vdash \mathsf{k} \hookrightarrow \mathsf{e} : \beta} \text{(E-THROWI)} \overset{\div}{\rightsquigarrow} \lambda\, \mathbf{k} : \mathsf{A}^{+} \to \alpha^{+}.\, \mathsf{e}^{\div}\ \mathbf{k}$$

Figure 6.19: CPS Translation of Inductive Data and Throwing Constructs

As in Dellina-, a function application has eight CPS images as before (Figure 6.18). What is different from the call-by-value translation is that, in the cases where the argument e_1 is impure, we have an application of the form $\mathbf{e_0}\ \mathsf{e}_1{}^{\div}$, instead of the continuation-passing pattern $\mathsf{e}_1{}^{\div}\ (\lambda\, \mathbf{v_1}.\, \mathbf{e})$, reflecting the call-by-name semantics of the source language. The application $\mathbf{e_0}\ \mathsf{e}_1{}^{\div}$ is guaranteed to be type-safe, because the function $\mathbf{e_0}$ in these cases must have a type of the form $\mathbf{\Pi\, x} : (\mathsf{A}^{+} \to \alpha^{+}) \to \beta^{+}.\, \mathbf{B}$. Note that we now have *two* cases where the translation yields a direct-style term: when all subterms are pure, and when only e_1 is impure.

Inductive data (Figure 6.19) are uniformly applied the value translation. This is because we have fixed the purity of constructor arguments. Pattern matching is translated exactly the same as in Dellina-, therefore we omit the rules for this construct.

$$\frac{\Gamma \vdash A : *}{\Gamma \vdash [\,] : A \to A}\ (\text{F-Hole}) \overset{+}{\rightsquigarrow} \lambda\, \mathbf{v} : A^{+}.\ \mathbf{v}$$

$$\frac{\Gamma \vdash e : A\ \rho \qquad \Gamma \vdash F : B[e/x] \to \alpha}{\Gamma \vdash F\ e : (\Pi x : A\ \rho.\, B) \to \alpha}\ (\text{F-AppP})$$

$$\overset{+}{\rightsquigarrow} \lambda\, \mathbf{v_0} : \mathbf{\Pi\, x} : A^{+}.\, B^{+}.\ F^{+}\ (\mathbf{v_0}\ e^{+})$$
if $\rho = \epsilon$

$$\overset{+}{\rightsquigarrow} \lambda\, \mathbf{v_0} : \mathbf{\Pi\, x} : (A^{+} \to \gamma^{+}) \to \delta^{+}.\, B^{+}.\ F^{+}\ (\mathbf{v_0}\ e^{\div})$$
if $\rho = [\gamma, \delta]$

$$\frac{\begin{array}{c}\Gamma \vdash e : A\ \rho \qquad \Gamma \vdash F : B[e/x] \to \alpha[e/x] \\ \Gamma, x : A\ \rho \vdash \beta : *\end{array}}{\Gamma \vdash F\ e : (\Pi x : A\ \rho.\, B[\alpha, \beta]) \to \beta[e/x]}\ (\text{F-AppI})$$

$$\overset{+}{\rightsquigarrow} \lambda\, \mathbf{v_0} : \mathbf{\Pi\, x} : A^{+}.\, (B^{+} \to \alpha^{+}) \to \beta^{+}.\ \mathbf{v_0}\ e^{+}\ F^{+}$$
if $\rho = \epsilon$

$$\overset{+}{\rightsquigarrow} \lambda\, \mathbf{v_0} : \mathbf{\Pi\, x} : (A^{+} \to \gamma^{+}) \to \delta^{+}.\, (B^{+} \to \alpha^{+}) \to \beta^{+}.\ \mathbf{v_0}\ e^{\div}\ F^{+}$$
if $\rho = [\gamma, \delta]$

Figure 6.20: CPS Translation of Pure Evaluation Contexts (Hole and Application)

The translation of throwing constructs $k \hookrightarrow e$ clearly shows what they do for us in the source language. When e is pure, $k \hookrightarrow e$ is equivalent to running the rest of the computation with the value of e. If e is impure, $k \hookrightarrow e$ means running the computation e in the surrounding context. Other control constructs are translated the same way as before.

Lastly, we have translation of evaluation contexts (Figures 6.20 – 6.21). An empty context is trivially mapped to an identity function. An application context $F\ e$ has four CPS images, each representing a possible continuation for ${e_0}^{\div}$ in the CPS images of the application $e_0\ e$ (where e_0 is impure). Similarly, the translation of a pattern matching context is exactly what follows $e^{\div}$ in the translation of a

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{P} \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 : \mathbb{N} \to \alpha}\ (\text{F-MatchNP})$$

$$\overset{+}{\rightsquigarrow}\ \lambda\,\mathbf{v} : \mathbb{N}.\, \mathbf{pm}\ \mathbf{v}\ \mathbf{as}\ \_\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with}\ \mathbf{z} \to \mathsf{F}^+\ \mathsf{e}_1{}^+ \mid \mathbf{suc}\ \mathbf{n} \to \mathsf{F}^+\ \mathsf{e}_2{}^+$$

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\alpha, \beta] \qquad \Gamma, \mathsf{n} : \mathbb{N} \vdash \mathsf{e}_2 : \mathsf{P}[\alpha, \beta]}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathbb{N}\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \mid \mathsf{suc}\ \mathsf{n} \to \mathsf{e}_2 : \mathbb{N} \to \beta}\ (\text{F-MatchNI})$$

$$\overset{+}{\rightsquigarrow}\ \lambda\,\mathbf{v} : \mathbb{N}.\, \mathbf{pm}\ \mathbf{v}\ \mathbf{as}\ \_\ \mathbf{in}\ \mathbb{N}\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with}\ \mathbf{z} \to \mathsf{e}_1{}^{\div}\ \mathsf{F}^+ \mid \mathbf{suc}\ \mathbf{n} \to \mathsf{e}_2{}^{\div}\ \mathsf{F}^+$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{n} : \mathbb{N} \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P} \qquad \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{L}\ \mathsf{n} \to \alpha}\ (\text{F-MatchLP})$$

$$\overset{+}{\rightsquigarrow}\ \lambda\,\mathbf{v} : \mathbf{L}\ \mathsf{n}^+.\, \mathbf{pm}\ \mathbf{v}\ \mathbf{as}\ \_\ \mathbf{in}\ \mathbf{L}\ \_\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with}\ \mathbf{nil} \to \mathsf{F}^+\ \mathsf{e}_1{}^+ \mid ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to \mathsf{F}^+\ \mathsf{e}_2{}^+$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : \mathsf{P} \to \alpha \qquad \Gamma \vdash \mathsf{n} : \mathbb{N} \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{P}[\alpha, \beta] \Gamma, \mathsf{m} : \mathbb{N}, \mathsf{h} : \mathbb{N}, \mathsf{t} : \mathsf{L}\ \mathsf{m} \vdash \mathsf{e}_2 : \mathsf{P}[\alpha, \beta]\end{array}}{\Gamma \vdash \mathsf{pm}\ \mathsf{F}\ \mathsf{as}\ \_\ \mathsf{in}\ \mathsf{L}\ \_\ \mathsf{ret}\ \mathsf{P}\ \mathsf{with}\ \mathsf{nil} \to \mathsf{e}_1 \mid ::\ \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 : \mathsf{L}\ \mathsf{n} \to \beta}\ (\text{F-MatchLI})$$

$$\overset{+}{\rightsquigarrow}\ \lambda\,\mathbf{v} : \mathbf{L}\ \mathsf{n}^+.\, \mathbf{pm}\ \mathbf{v}\ \mathbf{as}\ \_\ \mathbf{in}\ \mathbf{L}\ \_\ \mathbf{ret}\ \mathsf{P}^+\ \mathbf{with}\ \mathbf{nil} \to \mathsf{e}_1{}^{\div}\ \mathsf{F}^+ \mid ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to \mathsf{e}_2{}^{\div}\ \mathsf{F}^+$$

Figure 6.21: CPS Translation of Pure Evaluation Contexts (Pattern Matching)

pattern matching construct with an impure scrutinee.

Chapter 7

Logical Understanding of Dellina

In the previous chapters, we have described Dellina mainly from a computational perspective. But programming is not all we can do with dependently typed languages; the real power of such languages comes from their ability to prove theorems. Then, would our language, Dellina, serve as a proof assistant? Also, what does it mean to prove theorems with delimited control, and does its presence affect provability?

We claim that we can build proofs by writing programs in Dellina, but not all Dellina programs are proofs. In particular, we argue that impure terms should not be understood as proofs, because their meaning depends on the surrounding context. The distinction between proofs and non-proofs also gives rise to the conjecture that `shift` and `reset` cannot prove things beyond intuitionistic logic.

7.1 Logical vs. Non-logical Objects

Recall from Section 1.1 that a typing judgment

$$\Gamma \vdash e : A$$

can be understood as: e is a proof of proposition A under the assumptions in environment Γ.

In Dellina, we have two kinds of typing judgments, namely

$$\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A} \quad \text{and} \quad \mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A}[\alpha, \beta]$$

Among these judgments, the first one, which we use for pure terms, consists of the same components as the standard typing judgment. On the other hand, the second one, which we use for impure terms, carries two additional types α and β, describing the control effects involved in the evaluation of e.

Computationally, the pure judgment tells us that, e is a possibly non-value term of type A, and if it terminates, it evaluates to a value of the same type A (since Dellina enjoys the preservation property). Note that evaluation of e can be done in an arbitrary context, because a pure term can never access its surroundings.

On the other hand, the impure judgment says, e is a computation that has type A, and when it is surrounded by a delimited context returning α, it gives us back a value of type β, which serves as the result of the whole `reset` clause surrounding e. This time, evaluation of e requires a particular kind of delimited and meta contexts, and e itself does not have a value.

So the question is: what is the logical interpretation of impure terms? Through the lens of the Curry-Howard correspondence, we could view the impure judgment as saying: when we put e into an A-accepting context that proves α, we obtain a proof of β. The mention to the context is something unique to impure terms, and it is not clear how we may encode this in a standard logic. The same problem arises when interpreting types: we know that a pure function type $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ corresponds to universal quantification $\forall x \in A.\, B$, but it is not obvious how we should interpret an impure function type $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}[\alpha, \beta]$.

The above discussion motivates us to treat impure expressions differently from pure ones. That is, we divide types into two classes: propositions and non-propositions, and terms also into two classes: proofs and non-proofs. The classification is defined as follows, using a notion of *completely pure expressions*:

Definition 7.1.1 (Completely Pure Expressions)**.** *An expression* t *is completely pure when it does not contain occurrences of* $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}[\alpha, \beta]$*.*

Definition 7.1.2 (Propositions and Proofs)**.**

- *A type* A *is a proposition when (i)* $\Gamma \vdash \mathsf{A} : \mathsf{Prop}$ *for some* Γ*; and (ii)* A *is completely pure.*

- *A term* e *is a proof when (i)* $\Gamma \vdash \mathsf{e} : \mathsf{A}$*; (ii)* e *is completely pure; and (iii)* A *is a proposition.*

For the reader's enjoyment, below are some examples of proofs/non-proofs and propositions/non-propositions (suppose a and A are completely pure, and $\mathsf{a} : \mathsf{A} : \mathsf{Prop}$):

Propositions	Non-propositions
$\mathsf{A} \to \mathsf{A}$	$\mathsf{A} \to \mathsf{A}[\mathsf{A}, \mathsf{A}]$
$0 = 1$	$0 = \mathcal{S}\mathsf{k} : \mathbb{N} \to \mathbb{N}.\, 1$

Proofs	Non-proofs
$\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{x}$	$\lambda\, \mathsf{x} : \mathsf{A}.\, \mathcal{S}\mathsf{k} : \mathsf{A} \to \mathsf{A}.\, \mathsf{a}$
$\langle \mathcal{S}\mathsf{k} : \mathsf{A} \to \mathsf{A}.\, \mathsf{a} \rangle$	$\lambda\, \mathsf{f} : \mathsf{A} \to \mathsf{A}[\mathsf{A}, \mathsf{A}].\, \langle \mathsf{f}\ \mathsf{a} \rangle$

7.2 Intuitionistic vs. Classical Logic

Having established the notion of propositions/non-propositions and proofs/non-proofs, we next discuss what we can and cannot prove in Dellina. In particular, we would like to answer the question "Is Dellina intuitionistic or classical?"

7.2.1 Intuitionistic and Classical Logic

Let us first make it clear how intuitionistic and classical logics differ from each other. In intuitionistic logic, provability is understood as existence of proofs. That is, to affirm a proposition P, we must actually build a proof of P, and to deny a proposition P, we must build a proof of $\neg P$, by deriving a contradiction from the assumption that P is true. For certain propositions, like the famous halting problem, we cannot build a proof of either P or $\neg P$. In such a case, we say P is neither true nor false.

Intuitionistic logic is also called *constructive logic*, because every proved proposition has an evidence directly stating that the proposition is indeed true. The constructiveness is nicely captured by the following properties:

Proposition 7.2.1 (Disjunction and Existence Properties)**.**

1. *If $A \vee B$, then either A or B holds.*
2. *If $\exists x.\, A(x)$, then there is some t such that $A(t)$ holds.*

The first clause tells us that, if we know that $A \vee B$ is the case, then we must know which of A and B holds. Similarly, the second clause states that, if we know there is some x satisfying $A(x)$, then there must be some t witnessing this fact.

In classical logic, on the other hand, provability is understood as impossibility of refutation. That is, we can prove P is true by deriving a contradiction from the assumption $\neg P$. The proof-by-contradiction approach is possible because classical logic admits *double-negation elimination (DNE)*, namely the proposition $\neg\neg P \to P$. Using DNE, we can also derive the proposition $A \vee \neg A$, which is known as the *law of excluded middle (LEM)*. This means every proposition is either true or false in classical logic. For this reason, classical logic is described as the "god's view" of the world.

Proofs that use DNE are obviously non-constructive, since all they tell us is "it is not the case that P does not hold." Indeed, DNE allows us to prove LEM without showing which of the two disjuncts is actually true. In this sense, classical logic is *weaker* than intuitionistic logic, even if we have more axioms and are able to prove more theorems.

Intuitionistic and classical logics are connected by double negation translations [106], which convert a given classical proof of P into an intuitionistic proof of $\neg\neg P$. The post-translation proposition is classically equivalent to the original one, because we have DNE, but it is intuitionistically weaker, because the positive assersion has been turned into the denial of a denial.

7.2.2 Undelimited Control is Classical

The earliest study on the logical meaning of control operators dates back to 1990, when Griffin [88] incorporated undelimited control into the Curry-Howard picture. Griffin discovered that Felleisen's $\mathcal{C}$ operator [72] can be given a type corresponding to DNE. This means incorporating the $\mathcal{C}$ operator into a calculus turns the underlying logic into a classical one. For instance, we can prove LEM in the following way:

$$\mathcal{C}k : \neg(A \vee \neg A).\, k\ (\mathsf{inj}_2\ (\lambda\, a : A.\, k\ (\mathsf{inj}_1\ a)))$$

The term first asserts that A is not true, but when provided a term a witnessing A, it discards the context and asserts instead that A is true[1]. This somewhat cheating behavior relies on the backtracking ability of $\mathcal{C}$.

Griffin also studied CPS translations that erase the $\mathcal{C}$ operator. He showed that CPS translations that use a fixed answer type play the same role as double negation translations, *i.e.*, they convert classical programs into intuitionistic ones. Later, Murthy [124] claimed that CPS translations are not just double-negation translations, but they must be understood as Friedman's *A-translation* [79]. This refined view allows us to extract more information from classical proofs, such as what value an effectful program aborts with.

7.2.3 Delimited Control is Intuitionistic

Compared to undelimited control, the logical interpretation of delimited control is a rather subtle matter. The complication comes from the presence of answer types: undelimited continuations do not return, hence any continuation is given a negated type $\neg A$, whereas delimited continuations return and compose, therefore their return type can be an arbitrary (and usually non-empty) type. This means the correspondence between continuations and refutations—and equivalently, CPS and double negation—is lost in calculi with delimited control.

We conjecture that `shift` and `reset` do *not* allow proving classical theorems. The conjecture relies on our distinction between logical and non-logical objects. First, it is easy to see that if a type A is inhabited by a term e in the pure λ-calculus, then it must be the case that A is inhabited by e in the `shift`/`reset`-calculus as well, since we can derive $\Gamma \vdash e : A$ using the pure variant of the typing rules. What is challenging is the opposite argument: we must show that, if a proposition A is inhabited by a proof e, where e may use `shift` and `reset`, then A is inhabited by some term e' which does not use `shift` and `reset`. In other words, we want to show that any proposition in Dellina can be proved without using the control operators.

How can we prove the above statement? Our idea is to use the selective CPS translation. Formally, we are going to prove the following proposition:

Proposition 7.2.2. *If* A *is a proposition, then* $\mathsf{A}^{+} \equiv \mathsf{A}$*.*

[1]Wadler [167] has a nice, intuitive story illustrating what is happening in the above proof.

The intuition is as follows. Suppose we have $\mathsf{e} : \mathsf{A}$ in Dellina, where e is a proof and A is a proposition. Since only pure terms can ever be a proof, we know that e must be pure. Therefore, we may apply the value translation to e, and thus erase all the uses of the control operators. By the type preservation theorem (Theorem 4.6.1), we further know that $\mathsf{e}^+ : \mathsf{A}^+$. Now, if A^+ represents the same proposition as A, that means e^+ serves as an intuitionistic proof of A.

Unfortunately, we have not yet been able to actually prove the theorem, but the plan is to prove the theorem by induction on the derivation of A. The interesting cases are when A is a syntactic construct that refers to a term, say $\mathsf{A} = \mathsf{A}'\ \mathsf{e}$ derived by (DApp). By the definition of propositions, A' must be a proposition, and e must be a completely pure term. The induction hypothesis on the former gives us $\mathsf{A}'^+ = \mathsf{A}'$. To take a step further, we need the fact that $\mathsf{e}^+ \equiv \mathsf{e}$. This means, we must simultaneously prove the following proposition:

Proposition 7.2.3. *If* e *is a completely pure term, then* $\mathsf{e}^+ \equiv \mathsf{e}$.

The statement is again proved by induction on the derivation. The challenging case is when $\mathsf{e} = \langle \mathsf{e}' \rangle$ derived by (Reset). If e' is pure, the proof is easy, since $\langle \mathsf{e}' \rangle^+ = \mathsf{e}'^+$. If e' is impure, on the other hand, it is not clear how we can proceed: the proposition only accounts for pure terms, which means we cannot assume anything about e'. As future work, we intend to seek for a way to make this case go through, and show that our conjecture is indeed true.

Conjecture 7.2.1 (Logical Interpretation of Dellina)**.** *Dellina is an intuitionistic calculus.*

7.3 Related Work

Restricted Delimited Control Proves Non-intuitionistic Theorems Ilik [96] studies an extension of predicate logic with a weaker variant of `shift` and `reset`, which do not change answer types[2]. The restricted control effects make Ilik's calculus closer to an ordinary logical system. Specifically, he uses a typing

[2]Strictly speaking, he further restricts answer types to be Σ_1^0-formulae, *i.e.*, types that have no universal quantification. It is known that intuitionistic and classical provability coincide in this fragment, and thus any classical proof in this fragment computes evidence [124].

judgment of the form $\Gamma \vdash_{\alpha} e : A$, where α is an optional annotation that stands for the fixed answer type when present. The absence of answer-type modification also unnecessiates impure function types, which have no logical counterpart. Furthermore, in Ilik's calculus, the typing rule of `shift` looks like double negation elimination, as shown below:

$$\frac{\Gamma, k : A \to \alpha \vdash_{\alpha} e : \alpha}{\Gamma \vdash_{\alpha} \mathcal{S}k.\, e : A} \ (\textsc{Shift})$$

Using this typing rule, Ilik shows that the following term inhabits Double-Negation Shift (DNS), which is not provable in intuitionistic logic:

$$\lambda\, a.\, \lambda\, b.\, \langle b\ (\lambda\, x.\, \mathcal{S}k.\, a\ x\ k)\rangle : (\forall x.\, \neg\neg A(x)) \to \neg\neg\forall x.\, A(x)$$

The proof uses a form of classical reasoning: we first prove $a\ x\ k : \bot$ under the assumption $k : \neg A(x)$, and then derive $A(x)$ using ($\textsc{Shift}$). However, the calculus remains intuitionistic, as it enjoys the disjunction and existence properties:

Proposition 7.3.1.

1. *If* $\bullet \vdash e : A \vee B$, *then either* $\bullet \vdash a : A$ *for some* a, *or* $\bullet \vdash b : B$ *for some* b.
2. *If* $\bullet \vdash e : \Sigma\, x : A.\, B$, *then* $\bullet \vdash p : B(t)$ *for some* p *and* t.

Notice that the subject e in the two clauses is a pure term, as the effect annotation is empty. This means Ilik restricts proof terms to pure terms, although he does not explicitly commit himself to this view.

Given the above proof term, we would naturally wonder if we can prove DNS in Dellina. The answer is no, because Ilik's proof term would be given a non-proposition type. Observe that the term applies a λ-bound variable b to a function whose body is a `shift` construct. Since `shift` is impure, the function has an impure function type $\_ \to \mathsf{A}(\mathsf{x})[\bot, \bot]$. From the use of b, we also know that the domain of b must be convertible with this function type. This implies that the type of the whole term has an impure function type as its subcomponent, that is, it is not a proposition. Therefore, Ilik's proof term is not classified as proof in Dellina.

Full Delimited Control Only Proves Intuitionistic Theorems Shan [146] proves in his manuscript that the simply typed λ-calculus with the full, ATM-allowing `shift` and `reset` corresponds to intuitionistic logic. Similarly to us, Shan builds a `shift`/`reset`-calculus that discriminates pure terms from impure ones, and observes that the pure fragment covers the effect-free λ-calculus. He then defines a selective CPS translation, as well as its inverse translation ($^{-+}$ and $^{-\div}$), and proves the following proposition:

Proposition 7.3.2.

1. *If* $\Gamma^+ \vdash e : A^+$ *is derivable in the pure* λ*-calculus, then* $\Gamma \vdash e^{-+} : A$ *is derivable in the* `shift`/`reset`*-calculus, and* $(e^{-+})^+$ *is* β/η*-equivalent to* e*.*
2. *If* $\Gamma^+ \vdash e : (A^+ \to \alpha^+) \to \beta^+$ *is derivable in the pure* λ*-calculus, then* $\Gamma \vdash e^{-\div} : A[\alpha, \beta]$ *is derivable in the* `shift`/`reset`*-calculus, and* $(e^{-\div})^{\div}$ *is* β/η*-equivalent to* e*.*

The proposition essentially states that any pure λ-term of a CPS type has a possibly effectful counterpart in the `shift`/`reset`-calculus. From this fact, Shan concludes that the CPS translation covers all intuitionistic reasoning, and hence the logical interpretation of `shift` and `reset` is intuitionistic logic.

Our incomplete proof of Conjecture 7.2.1 is inspired by Shan's work, but our use of the CPS translation is different. Whereas Shan uses the translation to show that `shift` and `reset` prove all intuitionistic theorems, we use it to show that they do not prove non-intuitionistic theorems. In our view, Shan's proof lacks the latter argument, and thus is insufficient to conclude that his calculus is intuitionistic.

Intuitionistic Polarity Zeilberger [176] studies the full `shift` and `reset` in terms of *polarity*. Polarity is a notion from linear logic, and induces a perfect symmetry between values and continuations when integrated into a calculus. Traditionally, polarity has been used to understand different embeddings of classical logic into intuitionistic logic. This means polarized logic only accounts for computations in double-negation-based CPS, where control flow is fully explicit and continuations are abortive. Zeilberger relaxes these restrictions by replacing negation with arrows whose conclusion is a *positive* type, which are inhabited by observable data. Thus, he obtains an intuitionistic notion of polarity, or equivalently, a double-negation interpretation of intuitionistic logic. An interesting fact is that, when dealing with `shift` and `reset` in polarized logic, impure function types can be represented using ordinary logical connectives. This is because polarized logic already has facilities for expressing answer types and their modification.

Expressiveness of Classical Control Operators We saw in Section 7.2 that Felleisen's $\mathcal{C}$ operator has a type representing double-negation elimination. It is also well-known that `call/cc` can be assigned a type that corresponds to Pierce's law $((A \to B) \to A) \to A$. These laws are commonly used to turn an intuitionistic system into a classical one, but they are not equivalent: DNE proves Pierce's law, but not the other way around [7]. Interestingly, the relative logical expressiveness of these laws coincides with the relative computational expressiveness of their inhabitants: $\mathcal{C}$ can express `call/cc`, but not vice versa [73]. Based on this observation, Ariola and Herbelin [7] investigate a variety of classical logics and their corresponding calculi. For instance, the computational content of minimal classical logic, which admits Pierce's law but not EFQ, is the $\lambda\mu$-calculus of Parigot [131].

Chapter 8

Conclusion and Perspectives

"Stories never really end. They can go on and on and on. It's just that sometimes, at a certain point, one stops telling them."

Mary Norton, *The Borrowers*

This thesis introduced Dellina, a dependently typed language with the `shift` and `reset` operators. Our motivation was to concisely implement sophisticated behaviors while maintaining safety guarantees. Past work has shown that the key to combining control operators and dependent types is to disallow types dependent on effectful terms. We observed that unconstrained use of `shift` and `reset` further gives rise to dependent continuations and dependently used continuations, which bring open answer types into typing judgments.

By restricting the three kinds of problematic dependency, we obtained our first language Dellina-, which has the indexed list type as the sole source of dependency. We showed that Dellina- enjoys type soundness, and has a type-preserving CPS translation. In particular, the latter result relies heavily on the purity restriction on type dependency, as well as the selective nature of the translation.

We then showed that our design principle scales to a richer language, by extending Dellina- with polymorphism, universes, and inductive datatypes. To show the usefulness of the extended language Dellina, we presented a type-safe evaluator supporting efficient exception handling, where both control operators and dependent types are used in a non-trivial manner.

As our result suggests the possibility of extending proof assistants with `shift` and `reset`, we also addressed the question of what it means to use these operators to prove theorems. Although there seems to be no single answer to the question, we proposed to view pure types/terms as propositions/proofs, and impure ones as non-logical objects. The restriction turned out to limit the use of delimited control in proofs, leading to the conjecture that `shift` and `reset` only prove intuitionistic theorems.

Our journey is however just started. To make Dellina ready for real-world programming, we must further extend the language both effect-wise and type-wise. In the remainder of this chapter, we list some of the to-do's we have in mind.

8.1 Multiple Effects

In Dellina, we can simulate a wide range of effects using `shift` and `reset`, but only one single effect at a time. The limitation is due to the inflexibility in the way `shift` and `reset` pair up. Consider the following program:

$$\mathsf{run\text{-}state}\ (\lambda\,()\,:\,\mathsf{Unit}.\ \mathsf{run\text{-}choose}\ (\lambda\,()\,:\,\mathsf{Unit}.\ 1 + \mathsf{choose}\ (\mathsf{get}\ ())\ (\mathsf{inc}\ ();\ \mathsf{get}\ ())))$$

The program involves two kinds of effects: state and non-determinism. To remind the reader, run-state initializes the value of the state to 0, and choose wraps its arguments with the surrounding context. With these behaviors in mind, we would expect that the program reduces to [1; 2]. However, it turns out that the program does not run in this way. Since Dellina is call-by-value, we first evaluate the state-accessing computation get (). The `shift` operator in get is supposed to be paired with the `reset` operator in run-state, but this association is blocked by the `reset` in run-choose, leading to a type mismatch. Swapping run-state and run-choose does not help us either: it gives us the right association between get and run-state, but this time, the `shift` operator in choose will be unexpectedly paired with run-state. To make the above example runnable, we need to somehow let the second `shift` to skip the intervening `reset`. There are several ways to support this skipping, as we describe below.

CPS Hierarchy The most lightweight approach is to incorporate the *CPS hierarchy*. Recall that we have augmented Dellina with a hierarchy of universes, extending the "type-of" relation to an infinite one. There is an analogous notion for continuations as well, which gives rise to a tower of the "context-of" relations. That is, we extend pure evaluation contexts F into layered ones $\mathsf{F_i}$, which are captured by a family of operators $\texttt{shift}_i$ and $\texttt{reset}_i$ [56]. The following examples show how indexed `shift` and `reset` work:

(11) $\langle 1 + \langle 2 + \mathcal{S}_1 k.\, 3\rangle_1\rangle_2 = \langle 1 + 3\rangle_2 = 4$

(12) $\langle 1 + \langle 2 + \mathcal{S}_2 k.\, 3\rangle_1\rangle_2 = \langle 3\rangle_2 = 3$

In program (11), $\texttt{shift}_1$ only discards the context within $\texttt{reset}_1$, namely addition of 2. On the other hand, in program (12), $\texttt{shift}_2$ discards the whole thing within $\texttt{reset}_2$, including addition of 1, hence the program reduces to a different value. With these operators, we can specify the matching `reset` operator for each `shift`, avoiding undesired interference.

The layered control operators can be simulated by iterative CPS translations [56]. That is, if we wish to deal with n levels of contexts, then we CPS translate the source program n times. The target program will require n continuation arguments, where the i'th argument corresponds to the continuation captued by the level-i `shift`.

Dynamic Shift and Reset An alternative approach would be supporting a dynamic variant of `shift` and `reset` known as $\texttt{shift}_0$ and $\texttt{reset}_0$. The dynamic nature of these operators comes from the following reduction rule:

$$\langle F[\mathcal{S}_0 k.\, e]\rangle_0 \quad \triangleright \quad e[\lambda\, x.\, \langle F[x]\rangle_0 / k]$$

Notice that elimination of $\texttt{shift}_0$ removes the enclosing $\texttt{reset}_0$. This means, if the body e has occurrences of $\texttt{shift}_0$, they will be delimited by outer $\texttt{reset}_0$ operators. As Materzok and Biernacki [118] show, the above reduction rule makes $\texttt{shift}_0$ and $\texttt{reset}_0$ fully express, and even go beyond, the CPS hierarchy.

The removal of $\texttt{reset}_0$ makes more contexts relevant to evaluation, that is, we must take care of the innermost delimited context, *and* any other contexts within outer `reset`$_0$'s. For this reason, a type system for `shift`$_0$ and `reset`$_0$ has to maintain a list of contexts [118], often called *trails* in the literature. Correpondingly, the CPS translation of these operators should handle multiple continuations, either by passing around a continuation list [148], or by recursively abstracting over continuations via nested λ's [118]. In a typed setting, it seems that the curried translation would be easier to implement, since continuations generally have different types and hence cannot be put in a single homogeneous list.

Multi-Prompt Shift and Reset A more powerful but non-trivial extension is to adopt *multi-prompt* `shift` and `reset` [66]. The variant of `shift` and `reset` is a generalization of `shift`$_i$ and `reset`$_i$: they carry a *prompt tag* representing the intended association between the two operators, but the tags do not have orderings.

As Kiselyov and Sivaramakrishnan [105] show, multi-prompt `shift` and `reset` can be used to support *algebraic effects and handlers* [136], a promising approach to formalizing user-defined effects. Roughly speaking, an algebraic effect is something like a datatype, where constructors represent the operations on that effect (such as `get` and `inc` in the case of mutable state). Operations are performed by a handler, which is similar to a pattern matching construct. The idea of Kiselyov and Sivaramakrishnan is to implement operations using `shift`, and handlers using `reset`, together with an invariant that they use a common prompt tag defined for the effect they constitute.

The expressiveness of multi-prompt `shift` and `reset` comes at the cost of losing a succinct CPS semantics. Indeed, these operators are able to express dynamic binding [104], and for this reason, if we wish to simulate their behavior in a pure language, we need a CPS translation and an *environment-passing style* translation that erases dynamic variables [65]. This gives rise to a concern: dynamic binding has been considered only in non-dependent languages, and intuitively, it would not mix well with dependent types, due to its dynamic nature—we can at least imagine that dependency on dynamic variables would make static type checking impossible. The interaction between dynamic binding and dependent types is however worth studying on its own, from both theoretical and practical perspectives.

8.2 Control Effects at Higher Levels

Another interesting extension would be allowing control effects in higher universes. The simplest case is illustrated by the following program:

(13) $\langle (\lambda\, \alpha : \mathsf{Set}.\, \mathbb{N})\ (\mathcal{S}\mathsf{k} : \mathsf{Set} \to \mathsf{Set}.\, \mathbb{B})\rangle$

The `shift` operator is a type-level expression, which captures a type-returning continuation and returns a type to the delimited context. That is, the type and the two answer types of the `shift` construct are all Set.

We can make things more interesting by allowing the three types to reside in different universes. For instance, the following examples involve continuations that go across universe levels:

(14) $\langle (\lambda\, \mathsf{n} : \mathbb{N}.\, \mathbb{N})\ (\mathcal{S}\mathsf{k} : \mathbb{N} \to \mathsf{Set}.\, \mathbb{B})\rangle$

(15) $\langle 1 + ((\lambda\, \alpha : \mathsf{Set}.\, 2)\ (\mathcal{S}\mathsf{k} : \mathsf{Set} \to \mathbb{N}.\, 3))\rangle$

In program (14), the captured continuation "goes up", in that it receives a term and returns a type. By contrast, program (15) has a continuation that "goes down", as it receives a type and returns a term.

Higher control effects also give rise to *answer-universe modification*. Suppose a inhabits A, which is a Prop-type:

(16) $\langle (\lambda\, \mathsf{x} : \top.\, \top)\ (\mathcal{S}\mathsf{k} : \top \to \mathsf{Prop}.\, \mathsf{tt})\rangle$

The `reset` operator initially encloses a Prop-returning context, but the context is replaced by tt in the course of evaluation, hence the final answer type is one level lower than the initial answer type. Note that we are using a Prop type to ensure type preservation of the CPS translation. If we replace tt and $\top$ with 1 and $\mathbb{N}$, the CPS counterpart of the `shift` clause will have type $(\mathbb{N} \to \mathbf{Set}) \to \mathbb{N}$, which is ill-formed because $\mathbb{N} \to \mathbf{Set} : \mathbf{Type_1}$.

Higher-level control effects have been studied by Boutillier and Herbelin [32]. As we mentioned in Section 5.2, they incorporate `shift` and `reset` into Pure Type Systems, but leaving the discussion of language properties—including type preservation of the CPS translation—for future work. In a different context, Barthe et al. [22] give a CPS translation of non-dependent PTSs. They call their translation *pervasive*, as it internalizes both term-level and type-level contexts. The translation is however not designed for implementing control effects, but for proving the so-called *Barendregt-Geuvers-Klop conjecture* [82]. This conjecture states that any weakly normalizing PTS is also strongly normalizing. A popular approach to derive strong normalization from weak normalization is to define a mapping from the source calculus to Barendregt's λI-calculus [15], where every λ-bound variable is used at least once [152, 171]. The CPS translation of Barthe et al. is essentially such a mapping; however, they have not yet been able to scale the result to PTSs featuring dependent types, and the conjecture is still left open.

It is unclear how control effects at higher levels would be useful in dependently typed programming, but it clearly breaks the traditional view of types as being static objects. That is, when types have effects, we cannot statically determine what property or proposition they represent, just like when types depend on impure terms. Also, erasing effectful types would cause unexpected runtime behavior, which means the traditional compilation and program extraction no longer work.

On the other hand, type-level effects, as well as answer-universe modification, seem to have certain applications in linguistics, more specifically, in natural language semantics. We do not intend to discuss this in detail, but the intution is as follows. Natural language semantics is a study concerned with how to compute the meaning of natural language sentences. In the past decades, researchers found that a number of tools from programming languages, including control operators and dependent types, allow us to account for challenging linguistic phenomena in an elegant manner [17, 62, 18, 147, 25, 44, 46, 139]. Recently, Cong and Bowman [47] integrated a continuation-based treatment of *focus* [141] into Dependent Type Semantics of Bekki and Mineshima [26], showing that control operators must handle type-level contexts when used for natural language purposes. Although the work is still ongoing, it is an interesting observation that natural languages require more effects than what Dellina provides.

8.3 Propositional Equality

We have discussed two extensions for making more effects expressible. We next look at features for making more programs typable, or, equivalently, for proving more theorems. In Dellina, we say two types are equivalent when they are *definitionally* equal, *i.e.*, when they reduce to a common type. In the mainstream dependent languages, types may also judged *propositionally* equal, which is to say, two types are not convertible in terms of reduction, but they can be proven equal. The need for propositional equality arises when *e.g.* we prove associativity of list append:

$$\begin{array}{l}\Pi\, \mathsf{n_1\ n_2\ n_3} : \mathbb{N}.\, \Pi\, (\mathsf{l_1 : L\ n_1, l_2 : L\ n_2, l_3 : L\ n_3}).\\ \quad \mathsf{append}\ (\mathsf{n_1} + \mathsf{n_2})\ \mathsf{n_3}\ (\mathsf{append\ n_1\ n_2\ l_1\ l_2})\ \mathsf{l_3} =_{\mathsf{L}\ ((\mathsf{n_1} + \mathsf{n_2}) + \mathsf{n_3})}\\ \quad \mathsf{append\ n_1}\ (\mathsf{n_2} + \mathsf{n_3})\ \mathsf{l_1}\ (\mathsf{append\ n_2\ n_3\ l_2\ l_3})\end{array}$$

where $=$ is a type constant with a single constructor representing reflexivity (note that the left- and right-hand sides must be a pure term):

$$\frac{\Gamma \vdash \mathsf{e_1 : A} \qquad \Gamma \vdash \mathsf{e_2 : A}}{\Gamma \vdash \mathsf{e_1 =_A e_2 : Prop}}\ (\textsc{Equal}) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{e_1 =_A e_2 : Prop}\\ \mathsf{e_1} \rhd_p^\star \mathsf{e} \qquad \mathsf{e_2} \rhd_p^\star \mathsf{e}\end{array}}{\Gamma \vdash \mathsf{refl : e =_A e}}\ (\textsc{Refl})$$

The reader may be interested in how we prove this theorem, but it turns out that the theorem itself is ill-formed. The problem is in the equality type: the equality constructor $=$ requires two terms of type $\mathsf{L}\ ((\mathsf{n_1} + \mathsf{n_2}) + \mathsf{n_3})$, while the second list has type $\mathsf{L}\ (\mathsf{n_1} + (\mathsf{n_2} + \mathsf{n_3}))$. This means, we have to separately prove associativity of addition, and then rewrite the type of the second list using the subst operation:

$$\frac{\begin{array}{c}\Gamma, \mathsf{x : A} \vdash \mathsf{B : s} \qquad \mathsf{s} \in \mathsf{S}\\ \Gamma \vdash \mathsf{e : B[e_1/x]}\ \rho[\mathsf{e_1/x}] \qquad \Gamma \vdash \mathsf{p : e_1 =_A e_2}\\ \Gamma \vdash \mathsf{B[e_2/x] : s} \qquad \Gamma \vdash \rho[\mathsf{e_2/x}] : (\mathsf{s, s'})\end{array}}{\Gamma \vdash \mathsf{subst\ p\ e : B[e_2/x]}\ \rho[\mathsf{e_2/x}]}\ (\textsc{Subst}) \qquad \mathsf{subst\ refl\ e} \rhd \mathsf{e}$$

An alternative approach is to incorporate *heterogeneous equality*, where the left- and right-hand sides of the equality constructor may have different types. That is to say, we adopt the following equality formation rule:

$$\frac{\Gamma \vdash \mathsf{e_1 : A} \qquad \Gamma \vdash \mathsf{e_2 : B}}{\Gamma \vdash \mathsf{e_1 = e_2 : Prop}}\ (\textsc{HEqual})$$

The relaxed notion of equality reduces tedious work that indexed types give rise to, making proofs more concise and readable. On the other hand, heterogeneous equality considerably complicates the theory of the language. For instance, we must refine the (Subst) rule to an n-ary one, allowing multiple equalities to be eliminated simultaneously [151]. This is necessary for avoiding generation of ill-typed terms in the intermediate equalities.

Aside from the homogeneous vs. heterogeneous discussion, we would like to adress a bonus advantage of having propositional equality in Dellina and the target of the CPS translation. Recall from Section 4.4.3 that we use equivalence assumptions to make CPS-translated pattern matching well-typed, which may bring non-terminating behavior to programs. This issue can be solved if we use equality assumptions and convert types using subst. The reason is that reduction of subst does not happen unless an equality proof is actually given; in other words, if the equality really holds.

Propositional equality is used for compilation of dependent pattern matching in general [86, 153]. However, as existing studies suggest, the equality often requires a stronger elimination rule, known as the *axiom K*:

$$K : \Pi\, A : \mathsf{Type}.\, \Pi\, x : A.\, \Pi\, P : x = x \to \mathsf{Prop}.\, P(\mathtt{refl}) \to \forall h x = x P(h)$$

The type of K essentially states that all identity proofs are the same as `refl`. It is known that the axiom is incompatible with the *univalence axiom* from homotopy type theory: *e.g.*, Cockx et al. [42] shows how the combination allows us to prove $\mathsf{true} = \mathsf{false}$. As future work, we intend to investigate if our CPS translation demands the axiom K, or it suffices to equip the target language with the [Subst] rule.

8.4 Specialization by Unification

Another facility we would like to incorporate is Agda-style pattern matching, which automatically skips impossible branches. As a simple example, recall the head function from Section 1.1 (with a minor tweak to the list index):

$$\mathsf{head} : \Pi\, n : \mathbb{N}.\, \mathrm{L}\ (\mathsf{suc}\ n) \to \mathbb{N}$$

By declaring `head` in this way, we commit ourselves to run the function only with a non-empty list. Therefore, in the definition of `head`, we want to ignore the empty case and only account for the cons branch. This is exactly what Agda allows us to do: in Agda, the following program serves as the complete definition of the `head` function[1]:

```
head : ∀ (n : Nat) -> (L (suc n)) -> Nat
head .m (cons m h t) = h
```

When type checking the program, Agda will not complain that the pattern matching is non-exhaustive, because it knows that the empty case is impossible: the list argument must be indexed by a successor of some number, whereas an empty list is indexed by zero, which has a different head constructor. The skipping technique is called *specialization by unification* [86], since it specializes a pattern matching construct based on unification over type indices.

Smart pattern matching has been studied in order to make dependently typed programming easier [42, 43, 41]. We draw attention to this feature because, in Dellina, it would further serve as an appraoch to avoiding uses of inconsistent assumptions. Recall that inconsistency arises only in impossible branches. If we do not generate those branches in the first place, we will never need inconsistent assumptions. Of course, to incorporate the skipping mechanism, we have to establish a lot of properties; in particular, we must make sure that the pre- and post-CPS versions of a program have the same branches. We believe that the proofs would not be complicated too much by the presence of control effects, because type indices are all pure terms.

[1]The dot on the first occurrence of `m` means that, for the whole program to be well-typed, it must coincide with the argument `m` of the `cons` constructor.

Appendix A

Control Operators and Dependent Types in Natural Languages

In this appendix, we study an application of control operators and dependent types in a non-programming context, namely in natural language sentences. This might be surprising to the reader, but linguists do "functional programming" as well, and more excitingly, it has proven that control operators and dependent types are powerful tools for solving a wide range of linguistic challenges. Below, we give a gentle introduction to natural language semantics (Section A.1), present two long-standing puzzles as well as their PL-based solutions (Section A.2), and discuss the possibility of using Dellina for linguistic purposes (Section A.3).

A.1 Introduction to Natural Language Semantics

Natural language semantics is concerned with two questions: (i) how to represent the meaning of sentences in a *formal* way; and (ii) how to compute the meaning in a *natural* manner. Let us analyze the following sentence as a natural language semanticist would do:

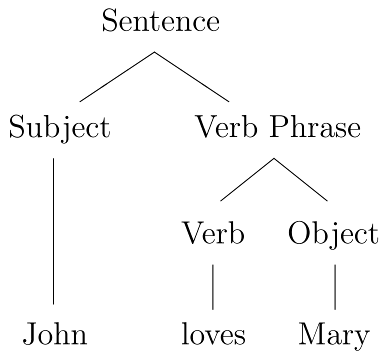
(17) John loves Mary.

Our first task is to find a *semantic representation*, which foramlly represents the meaning communicated by the sentence. Intuitively, we can represent sentence (17) as the following logical formula:

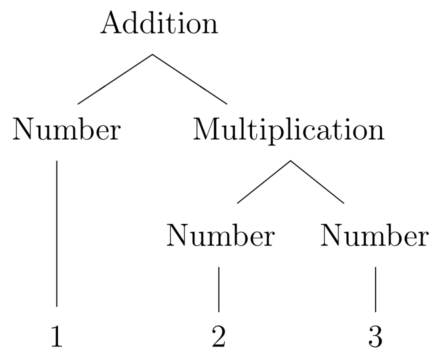
(18) $\mathsf{Love}\,(\mathsf{j},\ \mathsf{m})$

where Love is a two-place predicate, and j and m are constants. There are at least two ways to link between sentence (17) and formula (18). One popular approach is to say: "John loves Mary" is true in a given *model* M iff the interpretation of $\mathsf{Love}\,(\mathsf{j},\ \mathsf{m})$ with regard to M is true. This is the idea underlying *model-theoretic* frameworks, which is a popular paradigm of natural language semantics. An alternative approach is to view $\mathsf{Love}\,(\mathsf{j},\ \mathsf{m})$ as a type, and say that the sentence is true iff $\mathsf{Love}\,(\mathsf{j},\ \mathsf{m})$ is inhabited. This idea constitutes *proof-theoretic* semantics, which has been studied by researchers in the intersection of linguistics and type theory. We do not commit ourselves to one particular paradigm until next section; for now, it suffices to keep in mind that we represent meanings as logical formulas.

Having found a semantic representation, our next task is to map sentence (17) to formula (18) in the "right way", *i.e.*, as humans interpret the sentence. Intuitively, given the sentence "John loves Mary", we implicitly convert it into the following tree-like structure:



That is, we recognize that the verb "loves" describes a loving relation, where the subject is John and the object is Mary. The reader may find the diagram similar to an abstract syntax tree, *e.g.*:



By converting sentences and programs into trees, we can clearly see the individual components and the way they are connected. Then, we can compute the meaning of the whole in a *compositional* way, *i.e.*, from the meaning of its components and their relationships. The principle of compositionality goes all the way back to Frege, and is adopted in a majority of natural language frameworks—if we did not compute the meaning of a sentence from its components, how could we interpret sentences that we have never seen before? Compositionality is also preferred in programming languages: for instance, when defining a CPS translation, it is easier to prove its correctness if the translation is compositional [59].

So, how do we compositionally convert sentence (17) into formula (18)? The answer is to view verbs as functions, individuals as constants, and nodes as function application. That is, we define each word in the following way[1]:

$$\begin{aligned} \text{John} &\overset{\text{def}}{\equiv} \mathsf{j} \\ \text{Mary} &\overset{\text{def}}{\equiv} \mathsf{m} \\ \text{loves} &\overset{\text{def}}{\equiv} \lambda\, y.\, \lambda\, x.\, \mathsf{Love}\,(x,\, y) \end{aligned}$$

and then combine them using left and right application ($\cdot_l$ and $\cdot_r$):

$$\text{John loves Mary} = \mathsf{j}\ \cdot_r\ ((\lambda\, y.\, \lambda\, x.\, \mathsf{Love}\,(x,\, y))\ \cdot_l\ \mathsf{m}) = \mathsf{Love}\,(\mathsf{j},\, \mathsf{m})$$

Thus, the λ-calculus is not just a foundation of programming languages; it also serves as a framework for analyzing natural languages.

[1]Notice that "loves" first receives an object and then a subject. This is because "loves Mary" syntactically makes sense, whereas "John loves" does not.

A.2 Solving NL Challenges using PL Techniques

So far, everything looks trivial. But of course, there are cases where it is not obvious how to represent or compositionally compute meanings. In this section, we discuss two challenging phenomena in natural languages, and show how programming language techniques help us solve these challenges in an elegant way.

A.2.1 Quantifiers

The first challenge we would like to discuss is *quatifiers*. Let us consider the following sentences:

(19) Everyone loves John.

(20) John loves everyone.

It is easy to see that the word "everyone" gives rise to a universally quantified interpretation. Therefore, the two sentences should have the following semantic representations:

(21) $\forall x.\, \mathsf{Love}\,(x,\, j)$

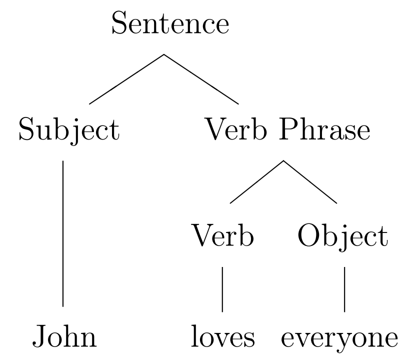
(22) $\forall x.\, \mathsf{Love}\,(j,\, x)$

Interestingly, while these representations are structurally similar, obtaining the latter is harder than obtaining the former. Let us start by the easy case. To convert sentence (19) into formula (21), we simply define "everyone" as the following higher-order function:

$$\text{everyone}_{sbj} \;\overset{\text{def}}{\equiv}\; \lambda\, P.\, \forall x.\, P\ x$$

A subject-position "everyone" takes a verb phrase P, and returns a universally quantified statement saying that every x satisfies P. In our running example, "everyone" will be applied to the function $\lambda\, x.\, \mathsf{Love}\,(x,\, j)$, yielding $\forall x.\, \mathsf{Love}\,(x,\, j)$.

We next convert sentence (20) into formula (22). Observe that, in the original sentence, "everyone" appears as an object, which is one level deeper than the subject in the tree representation:



The tree structure tells us that we must form a verb phrase by applying “loves” to “everyone”. This means “everyone” should be of the entity type. However, if “everyone” was a mere entity, like “John” and “Mary”, how could behave like a universal quantifier?

To solve this puzzle, let us focus our attention to the body part of formula (22), namely $\mathsf{Love}\,(\mathsf{j},\, x)$. By the β-rule, the representation is equivalent to $(\lambda\, x'.\, \mathsf{Love}\,(\mathsf{j},\, x')) \cdot_l x$. Interestingly, the function $\lambda\, x'.\, \mathsf{Love}\,(\mathsf{j},\, x')$ corresponds exactly to the continuation of “everyone”. If we can expose this continuation, we can obtain formula (22) by wrapping $(\lambda\, x'.\, \mathsf{Love}\,(\mathsf{j},\, x')) \cdot_l x$ around a universal quantifier. This leads to the following semantic representation of object-position “everyone”:

$$\text{everyone}_{obj} \;\overset{\text{def}}{\equiv}\; \mathcal{S}k.\, \forall x.\, k\ x$$

An object-position “everyone” captures its surrounding context k, which is a sentence with a hole, and builds a universally quantified statement saying that every x satisfies k. From a computational point of view, the use of `shift` in the above semantic representation is analogous to the one in the `reverse` example from Section 1.2, in that we obtain the correct scoping by calling the continuation in a non-tail position.

With this definition, together with a top-level `reset` surrounding the whole sentence (which is reasonable to assume), we can compositionally convert sentence (20) into formula (22):

$$\text{John loves everyone} = \mathsf{j} \cdot_r ((\lambda\, y.\, \lambda\, x.\, \mathsf{Love}\,(x,\, y)) \cdot_l (\mathcal{S}k.\, \forall x.\, k\ x)) = \forall x.\, \mathsf{Love}\,(\mathsf{j},\, x)$$

Continuations found their way into natural language semantics about twenty years ago [62, 17]. Interestingly, the notion of "expressions with a hole" arise not only in sentences involving quantifiers, but in many other linguistic phenomena as well [17, 145, 144, 18, 147, 103, 25, 44, 39, 46]. Furthermore, it has proven that continuations in natural languages generally require some delimitation [18], and sometimes need to be layered [19].

A.2.2 Anaphora

The second challenge we would like to address is anaphoric expressions. Consider the mini discourse below:

(23) Someone bought TAPL. He then bought ATAPL.

The discourse consists of two sentences. We can easily find an appropriate semantic representation for the first sentence: since the subject "someone" has an existential interpretation, we existentially quantify over the entity who bought TAPL, as in $\exists x.\, \mathsf{Buy}\,(x,\, \mathsf{t})$. But what about the second sentence? Intuitively, its semantic representation must be something like $\mathsf{Buy}\,(??,\, \mathsf{a})$, and we, as a human, know that ?? must be convertible with the entity that "someone" refers to. The problem is that, if we naïvely put the variable x, we would end up with an open formula $\mathsf{Buy}\,(x,\, \mathsf{a})$, because the existential quantification is only effective in the first sentence.

Discourse (23) suggests that the two successive sentences cannot be interpreted individually; in particular, the meaning of the second sentence *depends* on the first one. Therefore, we must somehow make the information carried by the first sentence available when interpreting the second sentence. This requires a means to "pack" the meaning of sentences, so that we can pass it between sentences, and a means to "unpack" the meaning, so that we can extract the context when necessary. Now, recall from Section 5.3 that there is a kind of type equipped with a pair of pack and unpack rules: Σ types! This observation was first made by Ranta [139], and was later incorporated into Bekki and Mineshima's Dependent Types System (DTS) [26]. DTS is a proof-theoretic framework of natural languages, where meanings are represented as types. For instance, "Someone bought TAPL" is represented as a Σ type in DTS:

(24) $\Sigma\, x : \mathsf{Entity}.\, \mathsf{Buy}\,(x,\, \mathsf{t})$

As in other proof-theoretic frameworks, we say "Someone bought TAPL" is true when the Σ type is inhabited, put differently, if we have a proof of $\Sigma\, x : \mathsf{Entity}.\, \mathsf{Buy}\,(x,\ \mathsf{t})$. As an example, the dependent pair (j, p), where p is a proof of "John bought TAPL", serves as a proof of the sentence.

Going back to discourse (23), our challenge is to get access to the entity representing "someone" from the second sentence. In DTS, we have the following *dynamic conjunction rule*, which lets us interpret a sentence with a proof of the preceding part of a discourse:

$$M; N \ \overset{\text{def}}{\equiv}\ \lambda\, c.\, \Sigma\, u : M\ c.\, N\ (c,\ u)$$

The rule essentialy tells us that, when we have a discourse $S_1; S_2; ...; S_n$, the meaning of S_i depends on the information carried by all S_j such that $j < i$. This proof-passing works if we turn every sentence, as well as the whole discourse, into a proof-awaiting function $\lambda\, c.\, e$. In the actual definition, M stands for the sequence of sentences, and N stands for a sentence following it. The meaning of their conjunction is a function that receives a *local context*, which is a proof we may use to solve anaphoric references in M, and returns a Σ type, which makes the proof u of M available during interpretation of N.

Using the conjunction rule, we can represent the meaning of discourse 23 as:

(25) $\lambda\, c.\, \Sigma\, u : (\Sigma\, x : \mathsf{Entity}.\, \mathsf{Buy}\,(x,\ \mathsf{t})).\, \mathsf{Buy}\,(\mathsf{fst}\ u,\ \mathsf{a})$

The semantic representation correctly captures the meaning of the sentence, and has no free variable. However, there is still one thing left to do: we must figure out what the general representation of pronouns would be. In discourse (23), the only candidate for "he" is the entity corresponding to "someone", but in the following example, we have multiple choices when resolving anaphora:

(26) A dog barked at a kitten. Then a boy came and took it.

The second sentence has two possible readings: (i) the boy took the dog to save the kitten; and (ii) the boy took the kitten for the same purpose. What this suggests is that we must allow some ambiguity when giving a meaning to pronouns. Based on this idea, Bekki and Mineshima [26] define pronouns as an *underspecified* term:

$$\text{he, she, it} \overset{\text{def}}{\equiv} @_i$$

Here, @ is a symbol representing some unknown entity, and i is a unique index given to each pronoun. Finding a concrete entity for $@_i$ now boils down to finding a term that makes the type referring to $@_i$ inhabited. In other words, anaphora resolution has reduced to *proof search*.

As Bekki and Mineshima [26] show, the idea presented here not only solves the issue with pronouns, but also scales to other linguistic phenomena that requires some form of non-local reasoning. Moreover, the type-theoretical aspect of DTS also makes it well-suited for textual inference tasks: *e.g.*, if we have a proof of "Someone bought TAPL. He then bought ATAPL", we can easily obtain a proof of "Someone bought ATAPL" via the second projection.

A.3 Dellina for Natural Language Semantics

Having seen the linguistic uses of control operators and dependent types, we would wonder if our language, Dellina, serves as a uniform framework for analyzing challenging phenomena in natural languages. For instance, with `shift`, `reset`, and Σ types at hand, we could explain discourses that have both pronouns and object-position quantifiers:

(27) Someone bought TAPL. He then bought every PL book.

(28) $\lambda\, c.\, \Sigma\, u : (\Sigma\, x : \mathsf{Entity}.\, x \cdot_r (\mathsf{buy} \cdot_l \mathsf{t})).\, \langle (\mathsf{fst}\, u) \cdot_r (\mathsf{buy} \cdot_l (\mathcal{S}k.\, \Pi\, y : \mathsf{PLBook}.\, k\, y)) \rangle$
$= \lambda\, c.\, \Sigma\, u : (\Sigma\, x : \mathsf{Entity}.\, \mathsf{Buy}\, (x,\, \mathsf{t})).\, \forall y.\, \mathsf{PLBookBuy}\, (\mathsf{fst}\ u,\, y)$

Is formula (28) a well-kinded Dellina type? The answer is no, because Dellina does not allow type dependency on impure terms. This limitation rules out the application of the type-level function `buy` to the impure `shift` construct representing "everyone". In fact, the problem is already present in sentences that have quantifiers but not anaphoras, such as our earlier example "John loves everyone". Therefore, although Dellina has both control operators and dependent types, it cannot be used for natural language purposes without relaxing the restriction on type dependency

A closer look at the semantic representation (28) reveals another interesting fact: the captured continuation goes across the term-type boundary. Observe that the `shift` operator is surrounded by a context that returns a type, which represents a buying proposition. The `shift` operator itself is however a term of type Entity. This means the captured continuation has type Entity $\rightarrow$ Prop, which Dellina does not support.

It seems that type-returning continuations are prevalent in natural languages, because we assume a default `reset` surrounding every sentence. That means, if there is no intervening expression that behaves like `reset`[2], an effectful expression will capture a Prop-returning continuation. There are also cases where we capture a continuation from a type-level expression, such as verbs and adjectives (see sentence (22a) of Barker [18]). To account for these cases, we would need the `shift` and `reset` operators in the type language.

In summary, if we want to combine continuation-based treatments of quantifiers and type-theoretical analysis of pronouns, we must allow (at least) (i) types dependent on impure terms; (ii) continuations from terms to types; and (iii) continuations from types to types. From a programming perspective, the resulting language would be harder to reason about, because the meaning of types becomes less clear in the presence of control effects. Also, there cannot exist a type-preserving CPS of such a language, be it selective or unselective. However, it is definitely an interesting observation that natural languages require more than what Dellina provides.

[2]As an example of `reset`-like expressions, adverbs like "only" and "also" delimit relevant contexts to verb phrases [18], which are type-level functions.

Bibliography

[1] D. Ahman. *Fibred Computational Effects*. PhD thesis, University of Edinburgh, 2017.

[2] D. Ahman. Handling fibred algebraic effects. *Proc. ACM Program. Lang.*, 2(POPL):7:1–7:29, Dec. 2017.

[3] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444, Sept. 2011.

[4] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, CSL '99, pages 453–468, London, UK, 1999. Springer-Verlag.

[5] A. Anand, A. W. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Bélanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[6] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[7] Z. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP: Annual International Colloquium on Automata, Languages and Programming, volume 2719 of LNCS*, pages 871–885. Springer-Verlag, 2003.

[8] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher Order and Symbolic Computation*, 22(3):233–273, Sept. 2009. online from 2007.

[9] K. Asai. Offline partial evaluation for shift and reset. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 3–14, New York, NY, USA, 2004. ACM.

[10] K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS '07, pages 239–254, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] K. Asai and O. Kiselyov. Introduction to programming with shift and reset. In *ACM SIGPLAN Continuation Workshop*, Sept. 2011.

[12] K. Asai and C. Uehara. Selective CPS transformation for shift and reset. In *Proceedings of the 2018 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '18, pages 40–52, 2018.

[13] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, July 2009.

[14] L. Augustsson. Cayenne&mdash;a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM.

[15] H. P. Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[16] H. P. Barendregt. Lambda calculi with types. 1992.

[17] C. Barker. Introducing continuations. In *Semantics and Linguistic Theory*, volume 11, pages 20–35, 2001.

[18] C. Barker. Continuations in natural language. *ACM SIGPLAN Continuation Workshop*, 4:1–11, 2004.

[19] C. Barker and C.-c. Shan. *Continuations and natural language*, volume 53. Oxford studies in theoretical linguistics, 2014.

[20] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 90–101, New York, NY, USA, 2009. ACM.

[21] G. Barthe, J. Hatcliff, and M. H. Sørensen. A notion of classical pure type system. *Electronic Notes in Theoretical Computer Science*, 6:4–59, 1997.

[22] G. Barthe, J. Hatcliff, and M. H. Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2):317–361, Nov. 2001.

[23] G. Barthe, J. Hatcliff, and M. H. B. Sørensen. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, Sept. 1999.

[24] G. Barthe and T. Uustalu. CPS translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 131–142, New York, NY, USA, 2002. ACM.

[25] D. Bekki and A. Kenichi. Representing covert movements by delimited continuations. *New Frontiers in Artifical Intelligence*, pages 161–180, 2009.

[26] D. Bekki and K. Mineshima. Context-passing and underspecification in dependent type semantics. pages 11–41, 2017.

[27] M. Biernacka and D. Biernacki. Context-based proofs of termination for typed delimited-control operators. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 289–300. ACM, 2009.

[28] M. Biernacka, D. Biernacki, S. Lenglet, and M. Materzok. Proving termination of evaluation for system F with control operators. In *Proceedings of the 1st Workshop on Control Operators and their Semantics*, COS 2013, 2013.

[29] D. Biernacki and P. Polesiuk. Logical relations for coherence of effect subtyping. *Logical Methods in Computer Science*, 14, 2018.

[30] S. Boulier, P.-M. Pédrot, and N. Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs - CPP 2017*, pages 182–194, Paris, France, Jan. 2017. ACM Press.

[31] P. Boutillier. A relaxation of Coq's guard condition. In *JFLA - Journées Francophones des langages applicatifs - 2012*, pages 1 – 14, Carnac, France, Feb. 2012.

[32] P. Boutillier and H. Herbelin. Delimited control and continuation passing style in pure type systems. Sept. 2011.

[33] W. J. Bowman. *Compiling with Dependent Types*. PhD thesis, Northeastern University, 2019.

[34] W. J. Bowman, Y. Cong, N. Rioux, and A. Ahmed. Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.*, 2(POPL):22:1–22:33, Dec. 2017.

[35] L. Cardelli. Phase distinctions in type theory. Technical report, 1988.

[36] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[37] C. Casinghino. *Combining Proofs and Programs*. PhD thesis, University of Pennsylvania, 2014.

[38] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 33–45, New York, NY, USA, 2014. ACM.

[39] S. Charlow. *On the semantics of exceptional scope*. PhD thesis, New York University, 2014.

[40] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[41] J. Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, KU Leuven, 2017.

[42] J. Cockx, D. Devriese, and F. Piessens. Pattern matching without k. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 257–268, New York, NY, USA, 2014. ACM.

[43] J. Cockx, D. Devriese, and F. Piessens. Unifiers as equivalences: Proof-relevant unification of dependently typed data. In *Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, pages 270–283. ACM, 2016.

[44] Y. Cong. Analysis and implementation of focus and inverse scope by delimited continuation. In *Proceedings of ESSLLI 2014 Student Session*, pages 177–189, 2014.

[45] Y. Cong and K. Asai. Handling delimited continuations with dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):69:1–69:31, Sept. 2018.

[46] Y. Cong, K. Asai, and D. Bekki. Focus, inverse scope, and delimited control. In *Proceedings of the 12th International Workshop on Logic and Engineering in Natural Language Semantics (LENLS 12)*, pages 137–148, 2015.

[47] Y. Cong and J. W. Bowman. Only control effects and dependent types. Presented at the 6th ACM SIGPLAN Workshop on Higher-Order Programming with Effects (HOPE 2017), 2017.

[48] T. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science (LICS)*, pages 227–236, 1986.

[49] T. Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, 1992.

[50] T. Coquand and G. Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[51] T. Coquand and C. Paulin. Inductively defined types. In *Proceedings of the International Conference on Computer Logic*, COLOG '88, pages 50–66, London, UK, 1990. Springer-Verlag.

[52] P.-L. Curien and H. Herbelin. The duality of computation. In *ACM sigplan notices*, volume 35, pages 233–243. ACM, 2000.

[53] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[54] O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 242–257, New York, NY, USA, 1996. ACM.

[55] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical report, University of Copenhagen, 1989.

[56] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

[57] O. Danvy and A. Filinski. Representing control: a study of the cps transformation, 1992.

[58] O. Danvy, C. Keller, and M. Puech. Typeful normalization by evaluation. In *20th International Conference on Types for Proofs and Programs, TYPES 2014*, 2014.

[59] O. Danvy and L. R. Nielsen. A first-order one-pass cps transformation. *Theoretical Computer Science*, 308(1-3):239–257, 2003.

[60] O. Danvy and L. R. Nielsen. Cps transformation of beta-redexes. *Information Processing Letters*, 94(5):217–224, 2005.

[61] M. Davis, W. Meehan, and O. Shivers. No-brainer CPS conversion (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP):23:1–23:25, Aug. 2017.

[62] P. De Groote. Type raising, continuations, and classical logic. In *Proceedings of the thirteenth Amsterdam Colloquium*, pages 97–101, 2001.

[63] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

[64] G. Dowek. The undecidability of typability in the lambda-pi-calculus. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, pages 139–145, London, UK, UK, 1993. Springer-Verlag.

[65] P. Downen and Z. M. Ariola. A systematic approach to delimited control with multiple prompts. In *European Symposium on Programming*, ESOP '12, pages 234–253. Springer, 2012.

[66] P. Downen and Z. M. Ariola. Delimited control and computational effects. *Journal of functional programming*, 24(1):1–55, 2014.

[67] B. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ml. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 163–173, New York, NY, USA, 1991. ACM.

[68] P. Dybjer. Logical frameworks. chapter Inductive Sets and Families in Martin-Lo&Uml;F's Type Theory and Their Set-theoretic Semantics, pages 280–306. Cambridge University Press, New York, NY, USA, 1991.

[69] R. K. Dyvbig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

[70] J. Egger, R. E. Møgelberg, and A. Simpson. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation*, 24(3):615–654, 2012.

[71] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM.

[72] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190. ACM, 1988.

[73] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35 – 75, 1991.

[74] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52(3):205–237, June 1987.

[75] M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. F. Duba. Reasoning with continuations. In *Logic in Computer Science*, LICS '86, 1986.

[76] A. Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM.

[77] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[78] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 165–176, New York, NY, USA, 2007. ACM.

[79] H. Friedman. Classically and intuitionistically provably recursive functions. In *Higher set theory*, pages 21–27. Springer, 1978.

[80] H. Geuvers. The church-rosser property for $\beta\eta$-reduction in typed lambda-calculi. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, LICS '92, pages 453–460. IEEE, 1992.

[81] H. Geuvers and B. Werner. On the church-rosser property for expressive type systems and its consequences for their metatheoretic study. In *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*, pages 320–329. IEEE, 1994.

[82] J. H. Geuvers. *Logics and type systems.* PhD thesis, University of Nijmegen, 1993.

[83] E. Giménez. Codifying guarded definitions with recursive schemes. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '94, pages 39–59, London, UK, 1995. Springer-Verlag.

[84] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique.* PhD thesis, Université Paris VII, 1972.

[85] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge University Press, 1989.

[86] H. Goguen, C. McBride, and J. McKinna. *Eliminating Dependent Pattern Matching*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[87] C. S. Gordon. A generic approach to flow-sensitive polymorphic effects. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[88] T. G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[89] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association.

[90] P. R. Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, New York, NY, USA, 2012.

[91] R. Harper and M. Lillibridge. Polymorphic type assignment and cps conversion. *Lisp Symb. Comput.*, 6(3-4):361–380, Nov. 1993.

[92] R. Harper and M. Lillibridge. Operational interpretations of an extension of f-omega with control operators. *Journal of Functional Programming*, 6, 1996.

[93] H. Herbelin. On the degeneracy of Σ-types in presence of computational classical logic. In *International Conference on Typed Lambda Calculi and Applications*, TLCA '05, pages 209–220. Springer, 2005.

[94] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 365–374. IEEE Computer Society, 2012.

[95] W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[96] D. Ilik. Delimited control operators prove double-negation shift. *Annals of Pure and Applied logic*, 163(11):1549–1559, 2012.

[97] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau. The definitional side of the forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 367–376, New York, NY, USA, 2016. ACM.

[98] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 27–38, New York, NY, USA, 2008. ACM.

[99] L. Jia, J. Zhao, V. Sjöberg, and S. Weirich. Dependent types and program equivalence. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 275–286, New York, NY, USA, 2010. ACM.

[100] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programmin*, ICFP '03, pages 177–188, New York, NY, USA, 2003. ACM.

[101] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 111–120, New York, NY, USA, 2009. ACM.

[102] Y. Kameyama and A. Tanaka. Equational axiomatization of call-by-name delimited control. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 77–86, New York, NY, USA, 2010. ACM.

[103] O. Kiselyov. Call-by-name linguistic side effects. In *ESSLLI 2008 Workshop on Symmetric calculi and Ludics for the semantic interpretation*, 2008.

[104] O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. In *The 33th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 26–37. ACM, 2006.

[105] O. Kiselyov and K. Sivaramakrishnan. Eff directly in ocaml. In *ML Workshop*, 2016.

[106] A. Kolmogohov. On the principle of the excluded middle. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 414–437. Cambridge, Massachusetts: Harvard University Press, 1967.

[107] J. Koppel, G. Scherer, and A. Solar-Lezama. Capturing the future by replaying the past (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP):76:1–76:29, July 2018.

[108] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme web server. *Higher Order Symbolic Computation*, 20(4):431–460, Dec. 2007.

[109] Y. Lafont, B. Reus, and T. Streicher. *Continuation semantics or expressing implication by negation.* Univ. München, Inst. für Informatik, 1993.

[110] R. Lepigre. A classical realizability model for a semantical value restriction. In *European Symposium on Programming Languages and Systems*, ESOP '16, pages 476–502. Springer, 2016.

[111] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, 2006.

[112] P. Letouzey. A new extraction for coq. In *Proceedings of the 2002 International Conference on Types for Proofs and Programs*, TYPES'02, pages 200–219, Berlin, Heidelberg, 2003. Springer-Verlag.

[113] P. B. Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer Science & Business Media, 2012.

[114] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[115] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[116] Z. Luo. Notes on universes in type theory, 2012.

[117] M. Masuko and K. Asai. Caml light+ shift/reset= caml shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011.

[118] M. Materzok and D. Biernacki. A dynamic interpretation of the CPS hierarchy. In *Asian Symposium on Programming Languages and Systems*, APLAS '12, pages 296–311. Springer, 2012.

[119] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In *Proceedings of the Conference on Logic of Programs*, pages 219–224, London, UK, 1985. Springer-Verlag.

[120] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[121] É. Miquey. A classical sequent calculus with dependent types. In *European Symposium on Programming*, pages 777–803. Springer, 2017.

[122] E. Miquey. A sequent calculus with dependent types for classical arithmetic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 720–729, New York, NY, USA, 2018. ACM.

[123] S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*, PLPV '10, pages 1–8, New York, NY, USA, 2010. ACM.

[124] C. R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, LICS '91, pages 96–107. IEEE, 1991.

[125] R. P. Nederpelt. *Strong normalization in a typed lambda calculus with lambda structured types.* PhD thesis, Technische Hogeschool, Eindhoven, the Netherlands, 1973.

[126] L. R. Nielsen. A selective cps transformation. *Electronic Notes in Theoretical Computer Science*, 45:311–331, 2001.

[127] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[128] U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[129] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, 2004.

[130] E. Palmgren. On universes on type theory, 1998.

[131] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 190–201, London, UK, 1992. Springer-Verlag.

[132] P.-M. Pédrot. A parametric cps to sprinkle cic with classical reasoning. In *Syntax and Semantics of Low-Level Languages*, LOLA '17, 2017.

[133] M. Petrolo. Negative translations and duality: toward a unified approach. In *Ludics, dialogue and interaction*, pages 188–204. Springer, 2011.

[134] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[135] G. Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.

[136] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, ESOP '09, pages 80–94. Springer, 2009.

[137] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[138] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN Notices*, volume 35, pages 23–33. ACM, 2000.

[139] A. Ranta. *Type-theoretical grammar*. Oxford: Clarendon Press, 1994.

[140] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328. ACM, 2009.

[141] M. Rooth. A theory of focus interpretation. *Natural Language Semantics*, 1(1):75–116, 1992.

[142] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Softw. Eng.*, 24(9):709–720, Sept. 1998.

[143] P. Severi and E. Poll. Pure type systems with definitions. In *International Symposium on Logical Foundations of Computer Science*, pages 316–328. Springer, 1994.

[144] C.-c. Shan. A continuation semantics of interrogatives that accounts for baker's ambiguity. In *Semantics and Linguistic Theory*, volume 12, pages 246–265, 2002.

[145] C.-c. Shan. Monads for natural language semantics. In *Proceedings of the ESSLLI 2001 Student Session*, pages 285–298, 2002.

[146] C.-C. Shan. From shift and reset to polarized linear logic. Unpublished, 2003.

[147] C.-c. Shan. *Linguistic side effects.* PhD thesis, Harvard University, 2005.

[148] C.-c. Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

[149] M. A. Sheldon and D. K. Gifford. Static dependent types for first class modules. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 20–29, New York, NY, USA, 1990. ACM.

[150] V. Sjöberg. *A Dependently Typed Language with Nontermination.* PhD thesis, University of Pennsylvania, 2015.

[151] V. Sjöberg, C. Casinghino, N. Collins, Y. K. Ahn, T. Sheard, H. D. Eades III, P. Fu, G. Kimmell, A. Stump, and S. Weirich. Irrelevance, heterogeneous equity, and call-by-value dependent type systems. In *Fourth Workshop on Mathematically Structured Functional Programming*, MSEP '12, 2012.

[152] M. H. Sørensen. Strong normalization from weak normalization in typed λ-calculi. *Information and Computation*, 133(1):35–71, 1997.

[153] M. Sozeau. *Un environnement pour la programmation avec types dépendants.* PhD thesis, Université de Paris-Sud. Faculté des Sciences d'Orsay (Essonne), 2008.

[154] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming*, pages 104–128. Springer, 2008.

[155] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[156] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, New York, NY, USA, 2016. ACM.

[157] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, New York, NY, USA, 2013. ACM.

[158] M. Takahashi. Parallel reductions in λ-calculus. *Information and computation*, 118(1):120–127, 1995.

[159] A. Tanaka and Y. Kameyama. A call-by-name cps hierarchy. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming*, pages 260–274, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[160] The Coq Development Team. The Coq proof assistant reference manual. https://coq.inria.fr/refman/, Mar. 2018.

[161] The DeepSpec Project. https://deepspec.org.

[162] H. Thielecke. From control effects to typed continuation passing. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages"*, POPL '03, 2003.

[163] M. Vákár. *In Search of Effectful Dependent Types.* PhD thesis, Oxford University, 2017.

[164] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, Aug. 2014.

[165] P. Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM, ACM.

[166] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[167] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 189–201, New York, NY, USA, 2003. ACM.

[168] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris-Diderot - Paris VII, May 1994.

[169] A. K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, Dec. 1995.

[170] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[171] H. Xi. Weak and strong beta normalisations in typed λ-calculi. In *International Conference on Typed Lambda Calculi and Applications*, TLCA '97, pages 390–404. Springer, 1997.

[172] H. Xi. Dependent ml: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, Mar. 2007.

[173] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM.

[174] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[175] H. Xi and C. Schürmann. Cps transform for dependent ML. In *Meeting Report of the 8th Workshop on Logic, Language, Information and Computation (WoLLIC '01)*, pages 739–754, 2001.

[176] N. Zeilberger. Polarity and the logic of delimited continuations. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*, pages 219–227. IEEE, 2010.