A Fine-Grained Evaluation Strategy for Delimited-Control Operators shift0/dollar

(Drobnoziarnista strategia ewaluacji dla
operatorów ograniczonego sterowania shift0/dollar)

Kamil Listopad

Praca magisterska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

20 września 2022

Abstract

Algebraic effects and handlers are the leading-edge approach to computational effects that are not only booming with theoretical research but also seeing great interest in practice – being implemented by a multitude of effect libraries in languages such as Haskell or natively supported by others. Delimited-control operators are a closely related concept due to their role in the semantics and implementation of effect systems. $\lambda_{c\$}$ calculus is a recently introduced calculus with a novel reduction theory for the delimited-control operators `shift0/dollar` that models the capture of delimited continuations in a fine-grained manner. The control operator `shift0` is of special interest as it has been shown to macro-express the algebraic operations and the so-called deep handlers.

This thesis extends the work on $\lambda_{c\$}$ calculus by defining a missing evaluation strategy designed to be a subset of the existing reduction theory that has the reflection property – the main goal of the paper introducing $\lambda_{c\$}$. The novel fine-grained strategy is proved to be equivalent to the standard `shift0/dollar` evaluation relation. The contribution is formalised in the Coq proof-assistant to achieve machine-verified correctness. The results of this work shed new light on the semantics of control operators `shift0/dollar` and open new research possibilities. Moreover, the novel evaluation strategy that we present can easily be adapted to support the `control0` operator that, thanks to its versatility, has been chosen to be included as a primitive operation in the GHC to support delimited continuations.

Streszczenie

Efekty algebraiczne i handlery są wiodącym podejściem do efektów obliczeniowych, które nie tylko przeżywają rozkwit badań teoretycznych, ale także spotykają się z dużym zainteresowaniem w praktyce - są implementowane przez wiele bibliotek efektów w językach takich jak Haskell lub natywnie obsługiwane przez inne. Operatory ograniczonego sterowania (ang. *delimited-control*) są ściśle powiązanym pojęciem ze względu na ich rolę w semantyce i implementacji systemów z efektami algebraicznymi. Rachunek $\lambda_{c\$}$ jest niedawno wprowadzonym rachunkiem z nowatorską teorią redukcji dla operatorów ograniczonego sterowania `shift0/dollar`, która modeluje przechwytywanie ograniczonych kontynuacji w sposób drobnoziarnisty. Operator sterowania `shift0` jest szczególnie interesujący, ponieważ wykazano, że makrowyraża operacje algebraiczne i tak zwane procedury obsługi głębokiej (ang. *deep handlers*).

Niniejsza rozprawa rozszerza prace nad rachunkiem $\lambda_{c\$}$ poprzez zdefiniowanie brakującej strategii ewaluacji zaprojektowanej jako podzbiór istniejącej teorii redukcji, która posiada własność refleksji - główny cel pracy wprowadzającej $\lambda_{c\$}$. Udowodniliśmy, że nowo wprowadzona strategia drobnoziarnista jest równoważna standardowej relacji ewaluacji `shift0/dollar`. Dowody wraz z procedurami ewaluacji zostały sformalizowane w systemie dowodzenia Coq, aby osiągnąć maszynowo zweryfikowaną poprawność. Wyniki tej pracy rzucają nowe światło na semantykę operatorów ograniczonego sterowania `shift0/dollar` i otwierają nowe możliwości badawcze. Ponadto, nowatorska strategia ewaluacji, którą prezentujemy, może być łatwo zaadaptowana do obsługi operatora `control0`, który dzięki swojej uniwersalności został wybrany do wdrożenia jako prymitywna operacja do GHC w celu wspierania ograniczonych kontynuacji.

Contents

Chapter 1

Introduction

The **continuation** is the *rest of the computation*, represented by the *context* of the current expression being evaluated. For example, in the program

```
(2 + f 3) + 1
```

the continuation of `f 3` is to *add* the number *2*, then the number *1*, to the intermediate result. Such a continuation is usually represented by the context `(2 + _) + 1`, which is a term with one hole `_` that replaced the focused part of the program, namely `f 3`. We can retrieve the original program by *plugging* the term `f 3` into the context `(2 + _) + 1`.

In modern programming it is quite uncommon to encounter continuations in an *explicit* form, but there are some languages that expose them to the user via special *control operators*. In general, control operators are separate language constructs added to support certain manipulations of the remaining computations. For example, exception handling mechanism can be implemented using a pair of control operators, like `catch` and `throw`.

We are going to focus on more sophisticated control operators that give access to the current continuation. The classic one being `call/cc` (*call-with-current-continuation*) [9] that allows the user to capture the current continuation, turn it into an explicit value and let the user dynamically redefine the rest of the program. The operation `call/cc f` binds the current continuation to a value `k` and continues the execution by applying the function `f` to it. Invoking a captured continuation at any later point in time replaces a current continuation with the captured one.

For example, the following programs result in the same value `7`.

```
(2 + call/cc (λ k. 4)) + 1
(2 + call/cc (λ k. k 4)) + 1
(2 + call/cc (λ k. 3 + k 4)) + 1
```

Notice that the captured continuation is the same for every program above and that

it is represented by the context `(2 + _) + 1`. The difference in these programs is that the first one did not use the captured continuation `k`, thus the evaluation continued as if the continuation was never captured. The second program invoked the continuation bound to `k` immediately after capturing it, so the replaced continuation did not change. Whereas the third program at the moment of invoking the captured continuation `k 4` replaced the current continuation equivalent, at that time, to the context `(2 + (3 + _)) + 1` with the captured context `(2 + _) + 1`.

The problem with `call/cc` is that capturing the whole continuation up to the top-level is not ideal. In the same sense that, when programming with exceptions, users usually want to *delimit* part of the code where exceptions can be handled, thus restricting a *part of the continuation* to be discarded when exception is raised without aborting the whole program.

Another limitation is that these continuations are not genuine functions [19], as they do not return, hence cannot be meaningfully composed. Terms such as `k (k 4)` are permitted, but such a composition is futile, since the first continuation activation abandons the rest of the program together with the `k _` context.

1.1 Delimited Control

A **delimited** (or *composable*) continuation is a prefix of the rest of the computation, represented by a delimited part of the whole context. For example, in the program

```
< 2 + f 3 > + 1
```

the continuation of `f 3`, as delimited by the angle brackets, is just to *add* the number *2* to the intermediate result. The rest of the computation outside the delimiter, namely the *addition* of the number *1*, is called a metacontinuation [7].

There are various control operators designed to access delimited continuations. A typical proposal consists of two control operators, the latter one being the delimiter, and the former one capturing the continuation up to an enclosing delimiter, such as Felleisen's `control/prompt` [3] or Danvy and Filinski's `shift/reset` [4].

Activating such a continuation also resumes the computation at the point of capture, similarly to an undelimited continuation captured by `call/cc`, however the current continuation at the point of activation is not abandoned, but resumed once the continuation returns. The difference between these two types of continuations is that the classic undelimited ones model jumps, e.g. tail-calls; whereas composable continuations model non-tail calls.

For the sake of this introduction we shall focus on control operators `shift` (alias `S`) and `reset` (alias `< >`) extending a standard lambda calculus:

Expressions `e ::= ... | S x. e | < e >`

The former operator appears in a binder form `S x. e`, which captures the current continuation, transforms it into a function and binds it to the variable `x` continuing with the body `e`. While the latter operator sets the boundary of a captured continuation, as `shift` operations capture continuations up to an enclosing delimiter.

The following contraction rules describe how these control operators evaluate. Note that the notation `e [x := v]` means a capture-avoiding substitution that replaces free variable occurrences `x` in the term `e` with the value `v`, whereas `K [e]` denotes the plug operation – *plugging* the term `e` into the context `K`.

$$\begin{aligned} \texttt{< v >} &\rightsquigarrow \texttt{v} \\ \texttt{< K [S f. e] >} &\rightsquigarrow \texttt{< e [f := } \lambda \texttt{ x. < K[x] >] >} \end{aligned}$$

When the evaluation under a delimiter proceeds without invoking the continuation-capture operation and results in a value, the delimiter is removed which is shown by the former contraction rule. Otherwise the current delimited continuation `K` gets captured, bound to a variable `f` and *abortively* replaced with the empty context. Notice that the body `e` may opt to invoke the continuation immediately, thus restoring the original context.

There are three variations to the `shift` operator, namely `control` (alias `C`), `shift0` (alias $\texttt{S}_0$) and `control0` (alias $\texttt{C}_0$) that slightly change the last contraction rule:

$$\begin{aligned} \texttt{< K [C f. e] >} &\rightsquigarrow \texttt{< e [f := } \lambda \texttt{ x. K[x]] >} \\ \texttt{< K [S}_0\texttt{ f. e] >} &\rightsquigarrow \texttt{e [f := } \lambda \texttt{ x. < K[x] >]} \\ \texttt{< K [C}_0\texttt{ f. e] >} &\rightsquigarrow \texttt{e [f := } \lambda \texttt{ x. K[x]]} \end{aligned}$$

Notice that the difference between all four variants is the presence or absence of delimiters after the contraction. `control`-like operators remove the delimiter in the captured context `K`, whereas `0`-variants remove the outer delimiter around the body `e`.

This slight distinction between `shift` and `control` causes a significant semantical contrast, as described by Danvy and Filinski in their pioneering work [4] about `shift/reset` that the context abstracted by `shift` is **statically** determined by the syntax, while `control` **dynamically** changes the delimiters enclosing the captured context. However, despite their differences, they are known to be macro-expressible by each other [7].

The dynamic version of `shift`, namely $\texttt{S}_0$, earned great theoretical interest thanks to its ability to inspect the stack of delimited contexts arbitrarily deep [14]. However, the focus of this thesis is the `shift0` variant with the `dollar` operator – a generalised delimiter `< >` described in detail in the next chapter as part of $\lambda_\$$ calculus [14].

One of the reasons for studying delimited continuations is their role in the semantics

and implementation of algebraic effects and handlers [10] – a leading-edge approach to computational effects. The delimited-control operators `shift0/dollar` are shown to macro express algebraic operations and the so-called deep handlers [15] [16]. Whereas the `control0` variant has been accepted [18] to the GHC [13], the most widely used Haskell [11] compiler, as a primitive operation enabling the support for delimited continuations to reduce the performance overhead of using the effect libraries.

1.2 Fine-Grained Reduction

The recent study [17] by Biernacki, Pyzik and Sieczkowski presented an alternative `shift0/dollar` calculus that models the capture of evaluation contexts in a **fine-grained** manner. Meaning, instead of a single contraction rule capturing the context `K` in one step, their calculus provides several rules that capture the context implicitly over a course of several local reduction steps. The fine-grained reduction behaviour is not just interesting in itself as the conventional non-local contraction, capturing the whole context in one step, tends to η-expand terms when the captured context `K` is empty, which is not an issue for fine-grained local reductions. In the study, they established a connection between their reduction theory and existing `shift0/dollar` theories and proved that their CPS translation along with the direct-style translation form a **reflection** [5]. Which means that reductions are preserved by the translations and the direct-style translation is a right inverse of the CPS translation.

However, the scope of their work does not include a deterministic **evaluation** relation (meaning *how programs compute*), but only the reduction theory (*a set of optimisation rules*). Developing the fine-grained evaluation relation sheds new light on the semantics of control operators `shift0/dollar` by providing an alternative way for programs to compute and opens new research possibilities, e.g. relating program optimisation (using the reduction theory) with the fine-grained evaluation by proving an appropriate standardisation [2] theorem. The aim of this work is to extend their calculus by defining an evaluation strategy using just a subset of contractions from their theory in order to preserve the theoretical properties of their calculus and prove a correspondence with the evaluation relation from the existing `shift0/dollar` calculus. Moreover, the results of this work are formalised [20] in the Coq proof assistant including an implementation of evaluation strategies for both fine-grain and coarse-grain calculi and two simulation theorems that state equivalence between both strategies.

1.3 Thesis Outline

Chapter 2 formally defines both `shift0/dollar` calculi: the standard $\lambda_{\$}$, as well as the *fine-grained* $\lambda_{c\$}$. Then the two potential fine-grained evaluation strategies are

described for the latter calculus, before formally defining one of them. A first part of the equivalence proof between the standard coarse-grained and the newly defined evaluation strategy in a form of a simulation theorem is the topic of Chapter 3. Chapter 4 presents the second simulation theorem completing the equivalence proof. Crucial design decisions made during development of the Coq formalisation of the equivalence proof are discussed in Chapter 5. Finally Chapter 6 summarises the thesis and discusses the potential future work.

Chapter 2

Fine-Grained Evaluation

Before discussing viable fine-grained evaluation strategies, we shall first formally introduce the standard (*coarse-grained*) `shift0/dollar` calculus, namely $\lambda_{\$}$ [14], as a baseline for comparison with the fine-grained calculus.

2.1 Coarse-Grained Calculus $\lambda_{\$}$

terms	`e`	::=	`v | p`
values	`v`	::=	`x | λ x. e`
non-values	`p`	::=	`e e' | S0 x. e | e $ e'`

$\lambda_{\$}$ is an extension of the lambda calculus with control operators `shift0` and `dollar`. An expression `e` is either a value `v` or a non-value `p`. Values comprise variables `x` and lambda abstractions `λ x. e`. Whereas, non-values are applications `e e'`, `shift0` abstractions $\mathtt{S}_0$ `x. e` and delimiter expressions `e $ e'`.

The `dollar` operator (`$`) is a generalisation of the control delimiter `< >` that not only delimits context, but also specifies the continuation for the term under the delimiter. Intuitively the expression `e1 $ e2` means *evaluate e1, then run e2 in the delimited context terminated by the result of evaluating e1.* The delimiter `< >` is macro-expressible as a `dollar` with the identity function:
`< e > := (λ x. x) $ e`

An evaluation strategy is best defined in terms of a *reduction semantics*. It is defined as an *eval* function that decomposes a term into an *evaluation context* and a *redex*, *contracts* ($\rightsquigarrow$) the redex and plugs the result into the context.

elementary contexts	`J`	::=	`_ e | v _ | _ $ e`
evaluation contexts	`K`	::=	`_ | J[K]`
trails	`T`	::=	`_ | v $ K[T]`

$$
\begin{array}{lrcl}
(\beta.v) & \texttt{(}\lambda\ \texttt{x. e) v} & \rightsquigarrow & \texttt{e [x := v]} \\
(\$.K) & \texttt{v \$ K [S}_0\ \texttt{f. e]} & \rightsquigarrow & \texttt{e [f :=}\ \lambda\ \texttt{x. v \$ K[x]]} \\
(\$.v) & \texttt{v \$ w} & \rightsquigarrow & \texttt{v w}
\end{array}
$$

$$
\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{K[T[e]]} \longrightarrow \texttt{K[T[e']]}}\ (\mathit{eval})
$$

Evaluation contexts are split between two syntactic categories:

- contexts `K` that do not cross the `dollar` boundary, which means terms are not placed under the delimiter `$` when plugged
- and *trails* `T` that model a stack of delimited contexts `v $ K`

Note that every reduction semantics presented in this thesis is a *deterministic* relation with the *unique-decomposition* [8] property defined for *closed terms* only. These properties are further discussed in Chapter 5.

2.2 Fine-Grained Calculus $\lambda_{c\$}$

As already mentioned in Chapter 1, the fine-grained evaluation must not deviate from the reduction theory, which means the *contraction* rules must come from the reduction theory. Before outlining the fine-grained evaluation strategy, we shall first see the original reduction theory for the $\lambda_{c\$}$ calculus [17] and discuss how a potential evaluation relation can be isolated from the given rules.

terms	`e`	::=	`v | p`
values	`v,w`	::=	`x` $\mid$ λ `x. e` $\mid$ S_0
non-values	`p`	::=	`e e' | let x = e in e' | e $ e'`
elementary contexts	`J`	::=	`_ e | v _ | _ $ e`

The grammar above introduces an extension of lambda calculus with control operators `shift0` and `dollar`, and `let` bindings that helped the authors of the calculus in developing the reflection. An expression `e` is either a value `v` or a non-value `p`. Values comprise variables `x`, lambda abstractions λ `x. e` and the control operator S_0 (`shift0`). Whereas, non-values are applications `e e'`, let-bindings `let x = e in e'`, and delimiter expressions `e $ e'`.

The control operator `shift0` presented here does not have the most common form, that is S_0 `x. e`. However, the binder form is expressible as an application of our combinator to a lambda abstraction by taking S_0 `x. e :=` S_0 `(`λ `x. e)` which is used as syntactic sugar in the remainder of the thesis.

The following reduction rules come from the reduction theory. To properly define the evaluation strategy we must decide which rules are to be included in the *contraction* relation. As to better understand the reduction theory, the reader is advised to treat the following reductions as **optimisation rules** that are used to simplify terms by rewriting parts (*subterms*) of the source terms.

($\beta.v$)	`(λ x. e) v`	$\rightsquigarrow$	`e [x := v]`
($\beta.let$)	`let x = v in e`	$\rightsquigarrow$	`e [x := v]`
($\beta.\$$)	`v $ S`$_0$ `w`	$\rightsquigarrow$	`w v`
($\$.v$)	`v $ w`	$\rightsquigarrow$	`v w`
($\$.let$)	`v $ let x = e in e'`	$\rightsquigarrow$	`(λ x. v $ e') $ e`
($name$)	`J[p]`	$\rightsquigarrow$	`let x = p in J[x]`
($assoc$)	`let x = let y = e`$_1$ `in e`$_2$ `in e`$_3$	$\rightsquigarrow$	`let y = e`$_1$ `in let x = e`$_2$ `in e`$_3$
($\eta.v$)	`(λ x. v x)`	$\rightsquigarrow$	`v`
($\eta.let$)	`let x = e in x`	$\rightsquigarrow$	`e`
($\eta.\$$)	`S`$_0$ `(λ x. x v)`	$\rightsquigarrow$	`v`

Note that the last three rules are η-reductions that are included in this presentation to show the complete reduction theory, however these rules will certainly not be included in the evaluation strategy, which computes only *closed* terms (not evaluating under binders).

The first three rules are β-reductions for functions, let-bindings and control operators. The rule ($\$.v$) eliminates the delimiter when both terms evaluate to a value. It is important to notice that ($\beta.\$$) is a simplified version of the rule ($\$.K$), from the coarse-grained calculus, where the context `K` is always empty. So in order to get the same result, the fine-grained calculus must capture the context `K` using several small steps to transform the term under the delimiter to an appropriate form before using the ($\beta.\$$) rule.

2.3 Context Capture

The *context capture* can be achieved using these two rules: ($\$.let$), ($name$).

($name$) is used to translate parts of the evaluation context (single stack frames `J`) into let expressions, e.g. a term

`v $ J`$_1$`[J`$_2$`[p]]`

can be simplified to

`v $ let x`$_1$ `= let x`$_2$ `= p in J`$_2$`[x`$_2$`] in J`$_1$`[x`$_1$`]`.

($\$.let$) finally captures elementary contexts from let expressions, ultimately creating continuations that correspond to initial evaluation contexts `K` delimited by `v $`. Continuing the previous example, the term would further simplify to

`(λ x1. v $ J1[x1]) $ let x2 = p in J2[x2]`, and then to
`(λ x2. (λ x1. v $ J1[x1]) $ J2[x2]) $ p`.

The **idea** behind these fine-grained reductions is that a delimited context `v $ K` can be translated to an equivalent continuation `w $ _`, thus capturing the context as a single lambda-abstraction `w`. As a result, any `v $ K [p]` term can be simplified to an equivalent `w $ p`, in particular when `p := S0 f. e`, which enables the ($.*let*) rule to be used getting the equivalent result of a single step of ($.$K$) from the coarse-grained calculus.

For example, consider the following term `v $ J1[J2[S0 f. e]]` that in the coarse-grained calculus evaluates in one step to

`e [f := λ x. v $ J1[J2[x]]]`

Whereas, with the reduction theory the term can be reduced to

`e [f := λ x2. (λ x1. v $ J1[x1]) $ J2[x2]]`.

Both captured contexts, that are bound to `f`, have similar structure, though the second one has the context `J1[J2]` intertwined with delimiters. The question is whether both continuations truly behave in the same way.

2.4 Alternative Context Capture

Notice that, in the context-capture strategy described above, let-expressions seem disposable. Their creation is immediately followed by their destruction in the most important use case, which is context capture. Usually let-bindings are expressible as syntactic sugar, in order to simplify the language, but authors of this calculus [17] opted to implement them as a separate category.

We believe that the intuitive strategy described above is possible to implement for $\lambda_{c\$}$ without let-expressions, and that it would still be *equivalent* to the coarse-grained $\lambda_{\$}$ calculus. However that is only a suspicion and not a formalised result, which nevertheless prompted us to investigate a different evaluation strategy, still possible within the reduction theory, which leverages let-expressions to the full extent.

Recall the example term `v $ J1[J2[S0 f. e]]`, as previously described, the (*name*) rule can be used to translate the *evaluation context* `J1[J2]` into let-expressions.

`v $ let x1 = let x2 = S0 f. e in J2[x2] in J1[x1]`.

Notice that instead of eliminating let-expressions, it is possible to use the (*assoc*) rule to *reassociate* the inner `S0 f. e` up the term, so that it appears next to the delimiter `v $`.

$\xrightarrow{(assoc)}$ `v $ let x2 = S0 f. e in let x1 = J2[x2] in J1[x1]`.

Now a single use of ($.*let*) followed by ($\beta$.$) results in

$$\texttt{e [f := } \lambda\ \texttt{x}_2\texttt{. v \$ let x}_1\texttt{ = J}_2\texttt{[x}_2\texttt{] in J}_1\texttt{[x}_1\texttt{]]}.$$

By comparing the result with the single use of $(\$.K)$ from the calculus $\lambda_{\$}$, we can conclude that captured continuations are almost identical. The only difference is the use of let-expressions that built the context $\texttt{J}_1\texttt{[J}_2\texttt{]}$.

2.5 Fine-Grained Evaluation

Following the intuition of the alternative context capture, we shall formally define the fine-grained evaluation strategy. It is sufficient to define evaluation contexts and select a subset of contraction rules from the reduction theory. To differentiate between $\lambda_{\$}$ and $\lambda_{c\$}$ definitions, we are going to use the character prime (') to mark definitions regarding the fine-grained evaluation strategy.

evaluation contexts	K'	::=	_ | let K' in e
trails	T'	::=	_ | v $ K'[T']

Evaluation contexts are built entirely from nested let-expressions, while *trails* have a similar structure to coarse-grained trails T. These contexts indicate that evaluation can go under delimiters and let-expressions when searching for a redex.

$$\begin{array}{lrcl}
(\beta.v) & (\lambda\ \texttt{x. e})\ \texttt{v} & \rightsquigarrow' & \texttt{e [x := v]} \\
(\beta.let) & \texttt{let x = v in e} & \rightsquigarrow' & \texttt{e [x := v]} \\
(\beta.\$) & \texttt{v \$ S}_0\ \texttt{w} & \rightsquigarrow' & \texttt{w v} \\
(\$.v) & \texttt{v \$ w} & \rightsquigarrow' & \texttt{v w} \\
(\$.let) & \texttt{v \$ let x = } \underline{\texttt{S}_0\ \texttt{w}}\texttt{ in e}' & \rightsquigarrow' & (\lambda\ \texttt{x. v \$ e}')\ \texttt{\$}\ \underline{\texttt{S}_0\ \texttt{w}} \\
(name) & \texttt{J[p]} & \rightsquigarrow' & \texttt{let x = p in J[x]} \\
(assoc) & \texttt{let x = let y = } \underline{\texttt{S}_0\ \texttt{w}}\texttt{ in e}_2\texttt{ in e}_3 & \rightsquigarrow' & \texttt{let y = } \underline{\texttt{S}_0\ \texttt{w}}\texttt{ in let x = e}_2\texttt{ in e}_3
\end{array}$$

$$\frac{\texttt{e} \rightsquigarrow' \texttt{e}'}{\texttt{K'[T'[e]]} \longrightarrow' \texttt{K'[T'[e']]}}\ (eval')$$

Every contraction rule (except η-reductions) from the reduction theory is necessary for this strategy. Determinism is accomplished due to the restriction of $(\$.let)$ and $(assoc)$ rules, which is indicated by the underlined term $\texttt{S}_0\ \texttt{w}$ in red. These rules are specialised to work only in the case of context capture.

The rule $(name)$ translates coarse-grained evaluation contexts K into an equivalent fine-grained K' that are nested let-expressions consisting of stack frames J from the context K.

The key part of context capture is done by a sequence of $(assoc)$ contractions, which is formalised in the following lemma

$$(reassoc)\quad \texttt{K' [let x = S}_0\ \texttt{w in e]} \longrightarrow'^{*} \texttt{let x = S}_0\ \texttt{w in K'[e]}$$

where $\longrightarrow'^*$ is the reflexive transitive closure of the $\longrightarrow'$ relation.

It is worth mentioning that this fine-grained strategy makes it trivial to support the `control0` operator. The extension would only require the addition of three contraction rules: $\mathtt{C}_0$-variants of (β.\$) and (*assoc*), together with another (\$.*let*) rule that would omit the `dollar` occurrence in the captured context:

($\$.let.\mathtt{C}_0$) `v $ let x = C0 w in e'` $\rightsquigarrow'$ `(λ x. v e') $ C0 w`

Conversely, note that it appears impossible to support `control0` for the other context-capturing strategy described in Section 2.3. Building the continuation there means eliminating let-expressions using the (\$.*let*) contractions, which rule out the proper `control0` semantics because of extra delimiters in the captured context. The $\mathtt{C}_0$-variant of (\$.*let*) would not help either, because both rules would be applicable and, since context building happens before encountering either `shift0` or `control0`, we would not know which one to use to capture the continuation.

Chapter 3

Fine-Grained Simulation

The **proof of equivalence** between the novel, *fine-grained* evaluation strategy for the $\lambda_{c\$}$ and the standard, *coarse-grained* evaluation strategy for the $\lambda_{\$}$ consists of proving two simulations that state that both strategies make equivalent computations.

This chapter focuses on the first simulation, which is the following theorem

$$\forall\ \mathtt{e}, \mathtt{v} \in \lambda_{\$}.\quad \forall\ \mathtt{e'} \in \lambda_{c\$}.$$
$$\mathtt{e} \sim \mathtt{e'}\ \ \wedge\ \ \mathtt{e} \longrightarrow^{*} \mathtt{v}\ \ \Longrightarrow\ \ \exists\ \mathtt{v'} \in \lambda_{c\$}.\quad \mathtt{v} \sim \mathtt{v'}\ \ \wedge\ \ \mathtt{e'} \longrightarrow'^{*} \mathtt{v'}$$

It states that: when a closed $\lambda_{\$}$-term `e` evaluates to a value `v`, then any closed $\lambda_{c\$}$-term `e'` that is similar to the term `e` evaluates in the fine-grained strategy to a similar value `v'`. Meaning that computations in $\lambda_{\$}$ can be simulated in $\lambda_{c\$}$.

The theorem above can be specialised to remove the similarity premise $\mathtt{e} \sim \mathtt{e'}$ by providing the *embedding* (ι) of the term `e` in place of a similar term `e'`. The embedding is defined as the following function:

$$\begin{aligned}
\iota : \lambda_{\$} &\to \lambda_{c\$} \\
\iota(\mathtt{x}) &= \mathtt{x} \\
\iota(\lambda\ \mathtt{x.\ e}) &= \lambda\ \mathtt{x}.\iota(\mathtt{e}) \\
\iota(\mathtt{e_1 e_2}) &= \iota(\mathtt{e_1})\iota(\mathtt{e_2}) \\
\iota(\mathtt{S_0\ x.\ e}) &= \mathtt{S_0}(\lambda\ \mathtt{x}.\iota(\mathtt{e})) \\
\iota(\mathtt{e_1 \$ e_2}) &= \iota(\mathtt{e_1})\$\iota(\mathtt{e_2})
\end{aligned}$$

And the *similarity relation* ($\sim$), that is defined below, is designed to admit the following *reflexivity* property of the similarity relation.

$$\forall\ \mathtt{e} \in \lambda_{\$}.\quad \mathtt{e} \sim \iota(\mathtt{e})$$

and with it the simulation theorem specialises to

$$\forall\ \mathtt{e}, \mathtt{v} \in \lambda_{\$}.\quad \mathtt{e} \longrightarrow^{*} \mathtt{v}\ \ \Longrightarrow\ \ \exists\ \mathtt{v'} \in \lambda_{c\$}.\quad \mathtt{v} \sim \mathtt{v'}\ \ \wedge\ \ \iota(\mathtt{e}) \longrightarrow'^{*} \mathtt{v'}$$

3.1 Similarity Relation

To prove the simulation theorem we first need to define the ***similarity*** **relation** $(\sim)$ that relates $\lambda_{\$}$ with $\lambda_{c\$}$ terms, as well as contexts.

$$\frac{\texttt{e} \sim \texttt{e'}}{\lambda\ \texttt{x.}\ \texttt{e} \sim \lambda\ \texttt{x.}\ \texttt{e'}}\ (\sim_{abs}) \qquad \frac{\texttt{e} \sim \texttt{e'}}{\texttt{S}_0\ \texttt{x.}\ \texttt{e} \sim \texttt{S}_0\ (\lambda\ \texttt{x.}\ \texttt{e'})}\ (\sim_{S_0})$$

$$\frac{\texttt{e1} \sim \texttt{e1'} \quad \texttt{e2} \sim \texttt{e2'}}{\texttt{e1 e2} \sim \texttt{e1' e2'}}\ (\sim_{app}) \qquad \frac{\texttt{e1} \sim \texttt{e1'} \quad \texttt{e2} \sim \texttt{e2'}}{\texttt{e1 \$ e2} \sim \texttt{e1' \$ e2'}}\ (\sim_{\$})$$

$$\frac{}{\texttt{x} \sim \texttt{x}}\ (\sim_{var}) \qquad \frac{\texttt{J} \sim \texttt{J'} \quad \texttt{e} \sim \texttt{e'}}{\texttt{J[e]} \sim \texttt{let x = e' in J'[x]}}\ (\sim_{name})$$

$$\frac{\texttt{v} \sim \texttt{v'}}{\lambda\ \texttt{x.}\ \texttt{v \$ x} \sim \texttt{v'}}\ (\sim_{\eta}) \qquad \frac{\texttt{v1} \sim \texttt{v1'} \quad \texttt{v2} \sim \texttt{v2'}}{\texttt{v1 \$ v2} \sim \texttt{v1' v2'}}\ (\sim_{\eta\$})$$

The relation on terms is mutually inductive with the similarity relation on *stack frames* `J`.

$$\frac{\texttt{e} \sim \texttt{e'}}{\_\ \texttt{e} \sim \_\ \texttt{e'}}\ (\sim_{J_{fun}}) \qquad \frac{\texttt{v} \sim \texttt{v'}}{\texttt{v}\ \_ \sim \texttt{v'}\ \_}\ (\sim_{J_{arg}}) \qquad \frac{\texttt{e} \sim \texttt{e'}}{\_\ \texttt{\$ e} \sim \_\ \texttt{\$ e'}}\ (\sim_{J_{\$}})$$

Similarity is also defined for *evaluation contexts* `K` and *trails* `T`.

$$\frac{}{\_ \sim \_}\ (\sim_{K_{nil}}) \qquad \frac{\texttt{J} \sim \texttt{J'} \quad \texttt{K} \sim \texttt{K'}}{\texttt{J[K]} \sim \texttt{let x = K' in J'[x]}}\ (\sim_{K_{cons}})$$

$$\frac{}{\_ \sim \_}\ (\sim_{T_{nil}}) \qquad \frac{\texttt{v} \sim \texttt{v'} \quad \texttt{K} \sim \texttt{K'} \quad \texttt{T} \sim \texttt{T'}}{\texttt{v \$ K[T]} \sim \texttt{v' \$ K'[T']}}\ (\sim_{T_{cons}})$$

The similarity relation extends naturally to contexts with the only exception being *evaluation contexts* that are built from let-expressions in $\lambda_{c\$}$ instead of nested stack frames.

The first five rules of the similarity relation on terms are structural rules that state that *similar terms are built from similar subterms*. The $(\sim_{name})$ rule associates $\lambda_{c\$}$ terms transformed by the contraction rule (*name*). Whereas, the last two rules are there due to technical reasons that we are about to discuss.

Consider the following example term and how it evaluates in both strategies:

$$\begin{array}{lll} \texttt{v \$ S}_0\ \texttt{f. e} & \rightsquigarrow & \texttt{e [f := } \lambda\ \texttt{x. v \$ x]} \\ & \rightsquigarrow'^{*} & \texttt{e [f := v]} \end{array}$$

Recall that the coarse-grained ($\$.K$) rule always plugs the context `K` in the continuation, even when it is empty, and as a result the continuation λ `x. v $ x` is η-expanded with an extra delimiter compared to the fine-grained continuation `v`. Both continuations are equivalent, as both strategies are call-by-value – meaning variables are substituted only for values, thus the only difference is that the expanded one always requires one extra evaluation step to eliminate the delimiter.

The ($\sim_\eta$) rule is needed to state the equivalence between such values. The culprit of this rule is that is makes the inversion on values non-trivial. Now there are two ways to relate the same $\lambda_{c\$}$ value, either structurally with the ($\sim_{abs}$) rule when lambda bodies are similar or with the ($\sim_\eta$) rule when the $\lambda_\$$ value is expanded. As a result proving equivalence of β-reduction is more difficult and requires yet another rule: ($\sim_{\eta\$}$).

3.2 Similarity of β-reduction

The difficulty appears in the proof of the following lemma about similarity of β-reduction.

$$\texttt{(}\lambda\ \texttt{x. e)}\ \texttt{v} \sim \texttt{e}_1 \implies \exists \texttt{e}_2.\ \texttt{e}_1 \longrightarrow'^{*}\ \texttt{e}_2 \wedge \texttt{e [x := v]} \sim \texttt{e}_2$$

It states that a $\lambda_{c\$}$-term similar to a β-redex evaluates in the fine-grained strategy to a term that is similar to the contraction result. The proof goes by inversion on the similarity judgement and complicates when considering cases with the ($\sim_\eta$) rule:

$$\dfrac{\dfrac{\texttt{w} \sim \texttt{w'}}{\lambda\ \texttt{x. w \$ x} \sim \texttt{w'}}\,(\sim_\eta) \qquad \texttt{v} \sim \texttt{v'}}{\texttt{(}\lambda\ \texttt{x. w \$ x) v} \sim \texttt{w' v'}}\,(\sim_{app})$$

Recall the argument that justified equivalence of $\sim_\eta$-similar values. It stated that the η-expanded continuation requires one extra evaluation step, which is exactly the current goal of the proof: `w $ v` $\sim$ `w' v'`. The rule ($\sim_{\eta_\$}$) is there to keep track of these extra evaluation steps.

3.3 Simulation Proof

In order to prove the main simulation theorem, we need an auxiliary theorem first, which is the *simulation step*:

$$\begin{aligned}&\forall\ e_1, e_2 \in \lambda_\$.\quad \forall\ e_1' \in \lambda_{c\$}.\\ &\qquad e_1 \sim e_1' \ \wedge\ e_1 \longrightarrow e_2 \implies \exists\ e_2' \in \lambda_{c\$}.\quad e_2 \sim e_2' \ \wedge\ e_1' \longrightarrow'^{*}\ e_2'\end{aligned}$$

It states that, given a single evaluation step $e_1 \longrightarrow e_2$ in the $\lambda_\$$ calculus, a *similar* ($\sim$) term e_1' from the $\lambda_{c\$}$ can be evaluated to a similar result-term e_2' with a sequence

of $\longrightarrow'$ steps. Meaning that starting with similar terms, each step in $\lambda_{\$}$ can be simulated in $\lambda_{c\$}$, obtaining similar results.

To prove the main simulation theorem we inductively iterate the theorem above and utilise the following property that values are only similar to other values:

$$\forall\, v \in \lambda_{\$},\ e' \in \lambda_{c\$}.\quad v \sim e' \implies \exists\, v' \in \lambda_{c\$}.\quad v' = e' \ \wedge\ v \sim v'$$

All that remains is to prove the *simulation step* theorem.

The proof begins with the inversion on the *eval* judgement $\texttt{e}_1 \longrightarrow \texttt{e}_2$ which implies that the term `e1` decomposes into `K [T [r]]`. It can be shown that when a decomposition is similar to a term:

$(\texttt{K [T [r]]} \sim \texttt{e}'_1)$,

the term evaluates to a decomposition of similar components:

$\texttt{e}'_1 \longrightarrow'^{*}$ `K' [T' [r']]`.

The exact lemma is:

$$(\textit{plug inversion})\quad \texttt{K [T [p]]} \sim \texttt{e}' \implies \exists\, \texttt{K}', \texttt{T}', \texttt{p}'.\quad \texttt{e}' \longrightarrow'^{*} \texttt{K}'\ \texttt{[T}'\ \texttt{[p}']\texttt{]} \wedge \texttt{K} \sim \texttt{K}' \wedge \texttt{T} \sim \texttt{T}' \wedge \texttt{p} \sim \texttt{p}'$$

An inquisitive reader might have spotted that the lemma restricted the plug operation to *non-value* terms `p`. This limitation helps to avoid the ambiguity caused by the $(\sim_{\eta\$})$ rule, but is nevertheless irrelevant for the simulation proof.

Plug operation also preserves similarity, which means that plugging similar terms into similar contexts yields similar results. The same property applies for the substitution.

The proof continues by considering all three contraction cases. The β-reduction $(\beta.v)$, as well as the dollar elimination $(\$.v)$ contraction cases follow from these similarity relation properties of the plug operation and the substitution. The remaining contraction case $(\$.K)$ needs to evaluate the term `r'`, which is similar to a redex `r = v $ K`$_r$`[S`$_0$ `f. e]`, to a similar `v' $ K'`$_r$`[S`$_0$ `f. e']`. Now when these evaluation contexts are empty, we proceed with the $(\beta.\$)$ contraction. Otherwise we apply the $(\textit{reassoc})$ lemma to get the $(\$.\textit{let})$ redex and make the contraction. Both cases are finalised with trivial contractions and applications of similarity properties.

Chapter 4

Coarse-Grained Simulation

In the previous chapter we have already discussed how fine-grained evaluation strategy can simulate computations in the standard $\lambda_{\$}$ calculus. To prove that both evaluation strategies are equivalent, we need the second simulation theorem:

$$\forall\, \mathtt{e'}, \mathtt{v'} \in \lambda_{c\$}.\quad \forall\, \mathtt{e} \in \lambda_{\$}.$$
$$\mathtt{e'} \approx \mathtt{e} \;\wedge\; \mathtt{e'} \longrightarrow'^{*}\, \mathtt{v'} \implies \exists\, \mathtt{v} \in \lambda_{\$}.\quad \mathtt{v'} \approx \mathtt{v} \;\wedge\; \mathtt{e} \longrightarrow^{*} \mathtt{v}$$

It is an inverse of the first simulation theorem stating that fine-grained computations can be expressed in coarse-grained steps, resulting in a similar outcome.

The similarity relation ($\sim$) introduced in the previous chapter was not enough to prove the second simulation theorem. The *coarse-grained simulation* not only requires grouping of multiple small $\lambda_{c\$}$-steps that make a single $\lambda_{\$}$-step, but also there are more technical details that derive from the absence of let-expressions in the $\lambda_{\$}$ calculus. The similarity relation ($\approx$) has been defined in this chapter to address these challenges, thus making the end result of the thesis not a standard *bisimilarity* relation, but two separate simulations instead. Notice that the arguments of the similarity relation are flipped compared to the previous chapter in order to emphasise the difference between both relations and the direction of the simulation.

4.1 Embedding (π)

Similarly to the previous chapter, it is possible to specialise the simulation theorem by using an *embedding* $(\pi) : \lambda_{c\$} \to \lambda_{\$}$, provided that the similarity relation agrees with the following reflexivity property: $\mathtt{e'} \approx \pi(\mathtt{e'})$.

Then the theorem specialises to:

$$\forall\, \mathtt{e'}, \mathtt{v'} \in \lambda_{c\$}.\quad \mathtt{e'} \longrightarrow'^{*}\, \mathtt{v'} \implies \exists\, \mathtt{v} \in \lambda_{\$}.\quad \mathtt{v'} \approx \mathtt{v} \;\wedge\; \pi(\mathtt{e}) \longrightarrow^{*} \mathtt{v}$$

The embedding that is defined below must desugar let-expressions as the $\lambda_{\$}$-calculus does not support them. Notice that in the previous chapter all let-expressions were

introduced during evaluation as a result of the (*name*) contraction. It made it clear that every let-expression had a specific form that corresponded to a certain stack frame `J` in the similar term. Now in addition to that, let-expressions can also be arbitrary, coming from the input term itself, and correspond to specific applications that are the result of the embedding.

$$\begin{aligned}
\pi : \lambda_{c\$} &\to \lambda_{\$} \\
\pi(\mathtt{x}) &= \mathtt{x} \\
\pi(\lambda\ \mathtt{x.\ e}) &= \lambda\ \mathtt{x}.\pi(\mathtt{e}) \\
\pi(\mathtt{e_1 e_2}) &= \pi(\mathtt{e_1})\pi(\mathtt{e_2}) \\
\pi(\mathtt{S_0}) &= \lambda\ \mathtt{m.\ S_0\ k.\ m\ k} \\
\pi(\mathtt{e_1\$e_2}) &= \pi(\mathtt{e_1})\$\pi(\mathtt{e_2}) \\
\pi(\mathtt{let\ x\ =\ e_1\ in\ e_2}) &= (\lambda\ \mathtt{x}.\pi(\mathtt{e_2}))\ \pi(\mathtt{e_1})
\end{aligned}$$

The ambiguity of let-expressions greatly influences the similarity relation and complicates proofs as both cases need to be handled when considering an inversion of similarity to let-expressions.

4.2 Proof Sketch

The similarity relation ($\approx$) is a sum of three similarity relations where one of its components is the redefined ($\sim$) relation, analogous to the similarity relation from the previous chapter. Before defining the similarity relation ($\approx$), we shall first discuss how the proof of the simulation theorem is going to proceed in order to understand the need for every rule of the relation.

The proof does not deviate from what was presented in the previous chapter – we need the *simulation step* theorem, which is inductively iterated to prove the main theorem. However, we shall focus on the first part of this theorem that has the structural similarity relation ($\sim$) in the assumption, instead of ($\approx$). The rest of the proof is covered once the ($\approx$) relation is fully defined.

$$\begin{aligned}
&\forall\ e_1', e_2' \in \lambda_{c\$}.\quad \forall\ e_1 \in \lambda_{\$}. \\
&\qquad e_1' \sim e_1 \quad\wedge\quad e_1' \longrightarrow' e_2' \quad\Longrightarrow\quad \exists\ e_2 \in \lambda_{\$}.\quad e_2' \approx e_2 \quad\wedge\quad e_1 \longrightarrow^* e_2
\end{aligned}$$

The proof of the auxiliary theorem goes by inversion on the single evaluation step

$e_1' \longrightarrow' e_2'$

Consequently the similarity judgement becomes

$\mathtt{e_1'\ =\ K'\ [T'\ [r']]}\ \sim\ \mathtt{e_1}$.

We conclude that $\mathtt{e_1}$ must form a similar decomposition $\mathtt{e_1\ =\ K\ [T\ [r]]}$ which is true without extra evaluation steps that were required in the *plug inversion* lemma from the first simulation theorem. Just like in the previous chapter, the plug operation preserves similarity, meaning that plugging similar terms into similar contexts yields similar results. The same property holds for the substitution.

The proof continues by considering all seven contraction cases and using inversion on the similarity judgement. We are going to discuss each of these cases to find out what rules of the similarity relation are necessary to complete the proof.

Case *($\beta.v$):*

Assumptions: $\texttt{(}\lambda\ \texttt{x. e')} \sim \texttt{w} \quad \wedge \quad \texttt{v'} \sim \texttt{v}$
Goal: $\exists \texttt{t.} \quad \texttt{K [T [w v]]} \longrightarrow^{*} \texttt{t} \quad \wedge \quad \texttt{K' [T' [e' [x := v']]]} \approx \texttt{t}$

Recall the lemma from the previous chapter about the *similarity of β-reduction.* The proof here goes quite similarly: we consider every inversion case of the value similarity $\texttt{(}\lambda\ \texttt{x. e')} \sim \texttt{w}$ to determine the body of the lambda `w` and execute a sequence of coarse-grained steps ($\longrightarrow$) to get an equivalent β-reduction result.

Previously, in one of the inversion cases, the ($\sim_{\eta\$}$) rule was required to keep track of the extra evaluation step $\texttt{w' \$ v'} \longrightarrow' \texttt{w' v'}$, but now we can add this step to the sequence and finish the goal with structural rules of the similarity relation ($\sim$) as well as preservation properties. Note that, as we are about to find out later, there are three distinct rules for values in the similarity relation, but all of these cases are easily handled without requiring extra similarity rules.

Case *($\beta.let$):*

Assumptions: $\texttt{let x = v' in e'} \sim \texttt{r}$
Goal: $\exists \texttt{t.} \quad \texttt{K [T [r]]} \longrightarrow^{*} \texttt{t} \quad \wedge \quad \texttt{K' [T' [e' [x := v']]]} \approx \texttt{t}$

As expected the goal is the same as in the previous β-reduction case. The only difficulty is to consider both inversion cases of the similarity judgement. As we have already discussed, terms similar to let-expressions are either desugared applications or decompositions of a stack frame:

Case 1: $\texttt{r = (}\lambda\ \texttt{x. e) v} \quad \wedge \quad \texttt{e'} \sim \texttt{e} \quad \wedge \quad \texttt{v'} \sim \texttt{v}$
Case 2: $\texttt{r = J[v]} \quad \wedge \quad \texttt{e' = J'[x]} \quad \wedge \quad \texttt{J'} \sim \texttt{J} \quad \wedge \quad \texttt{v'} \sim \texttt{v}$

A single β-reduction is needed in the former case, whereas no evaluation is necessary in the latter case. Then both cases trivially follow from the similarity properties.

Case *($\beta.\$$):*

Assumptions: $\texttt{(v' \$ } \mathtt{S}_0 \texttt{ w')} \sim \texttt{r}$
Goal: $\exists \texttt{t.} \quad \texttt{K [T [r]]} \longrightarrow^{*} \texttt{t} \quad \wedge \quad \texttt{K' [T' [w' v']]} \approx \texttt{t}$

From the inversion on the assumption we conclude that

$\texttt{r = v \$ } \mathtt{v}_{\mathtt{S}_0} \texttt{ w} \quad \wedge \quad \texttt{v'} \sim \texttt{v} \quad \wedge \quad \texttt{w'} \sim \texttt{w} \quad \wedge \quad \mathtt{S}_0 \sim \mathtt{v}_{\mathtt{S}_0}$

Assuming that $\mathtt{v}_{\mathtt{S}_0} = \pi(\mathtt{S}_0) = \lambda\ \texttt{m. } \mathtt{S}_0\ \texttt{k. m k}$ the whole term `r` evaluates in the following way:

$\texttt{r = v \$ } \mathtt{v}_{\mathtt{S}_0} \texttt{ w} \longrightarrow \texttt{v \$ } \mathtt{S}_0\ \texttt{k. w k} \longrightarrow \texttt{w (}\lambda\ \texttt{x. v \$ x)}$

To show that both contraction results are similar we need an equivalent ($\sim_\eta$) rule. But admitting this rule complicates the inversion on values, making our previous

assumption that $\mathtt{v_{S_0}} = \lambda\ \mathtt{m.\ S_0\ k.\ m\ k}$ not entirely true as it can be η-expanded with the extra dollar.

To fix our reasoning we need the following lemma:

(S_0 eval) $\mathtt{S_0} \sim \mathtt{v_{S_0}} \implies \mathtt{v_{S_0}\ w} \longrightarrow^* \mathtt{S_0\ k.\ w\ k}$

The proof of the lemma actually requires induction on the derivation of the similarity judgement as it needs to reduce the application for every η-expansion used in the $\mathtt{v_{S_0}}$ term.

Case *($\$.v$):*

Assumptions: $\mathtt{v'} \sim \mathtt{v} \quad \wedge \quad \mathtt{w'} \sim \mathtt{w}$
Goal: $\exists \mathtt{t.} \quad \mathtt{K\ [T\ [v\ \$\ w]]} \longrightarrow^* \mathtt{t} \quad \wedge \quad \mathtt{K'\ [T'\ [v'\ w']]} \approx \mathtt{t}$

This is a trivial case that requires an equivalent evaluation step ($\$.v$) and standard similarity properties.

Case *($\$.let$):*

Assumptions: $\mathtt{v'} \sim \mathtt{v} \quad \wedge \quad \mathtt{(let\ x = S_0\ w'\ in\ e')} \sim \mathtt{e}_{let}$
Goal: $\exists \mathtt{t.} \quad \mathtt{K\ [T\ [v\ \$\ e}_{let}\mathtt{]]} \longrightarrow^* \mathtt{t}$
$\wedge \quad \mathtt{K'\ [T'\ [(\lambda\ x.\ v'\ \$\ e')\ \$\ S_0\ w']]} \approx \mathtt{t}$

This is yet another case with let-expression similarity that requires us to consider each case separately. However both cases follow from identical methods, so with no loss of generality we can focus on the stack frame case. And as we perform the evaluation we encounter a technical problem. Consider the following evaluation steps:

$$\mathtt{v\ \$\ J[v_{S_0}\ w]} \xrightarrow{(S_0\ eval)}{}^{*} \mathtt{v\ \$\ J[S_0\ k.\ w\ k]} \xrightarrow{(\$.K)} \mathtt{w\ (\lambda\ x.\ v\ \$\ J[x])}$$

Notice that the goal is actually one step behind the above evaluation sequence and that we could easily prove similarity if we were allowed to reduce the goal as follows:

$$\mathtt{(\lambda\ x.\ v'\ \$\ J'[x])\ \$\ S_0\ w'} \xrightarrow{(\beta.\$)} \mathtt{w'\ (\lambda\ x.\ v'\ \$\ J'[x])}$$

Unfortunately as we had to introduce the ($\sim_{\eta\$}$) rule in the previous chapter to keep track of an extra evaluation step we need a similar rule so that we can postpone the ($\$.K$) step.

However this extra rule will not be added to the similarity relation ($\sim$), but to a separate relation component of ($\approx$) that is introduced later where it is also explained how the proof is completed.

Case *(name):*

Assumptions: $\mathtt{p'} \sim \mathtt{p} \quad \wedge \quad \mathtt{J'} \sim \mathtt{J}$
Goal: $\mathtt{K'\ [T'\ [let\ x = p'\ in\ J'[x]]]} \approx \mathtt{K\ [T\ [J[p]]]}$

This is the case where let-expressions are created in the *fine-grained* calculus. There is no equivalent *coarse-grained* step, so we need a similarity rule like ($\sim_{name}$) that

associates such let-expressions with stack frame decompositions.

Case (*assoc*)*:*

Assumptions: $\texttt{let x = let y = S}_0\ \texttt{w}'\ \texttt{in e}'_2\ \texttt{in e}'_3 \sim \texttt{r}$
Goal: $\exists \texttt{t}.\ \texttt{K [T [r]]} \longrightarrow^* \texttt{t}$
$\wedge\ \texttt{K}'\ \texttt{[T}'\ \texttt{[let y = S}_0\ \texttt{w}'\ \texttt{in let x = e}'_2\ \texttt{in e}'_3\texttt{]]} \approx \texttt{t}$

(*assoc*) contractions indicate the context building process which is not performed in the *coarse-grained* calculus, so there is no equivalent evaluation step to perform in this case (except from administrative reductions from the *(S_0 eval)* lemma).

We must, however, record progress, if not in the evaluation itself, then in the similarity judgement. But adding new rules to the currently considered similarity relation ($\sim$) is hardly ideal. Remember that each ($\$.K$) contraction (with non-empty context K) corresponds to a sequence of (*assoc*) *fine-grained* steps followed by the final contraction ($\$.let$), so the similarity relation ($\approx$) needs to group these (*assoc*) steps even in the middle of the whole context building process – captured in the (*reassoc*) lemma.

The idea for the rule is to associate the following terms (assuming ($\sim$)-similarity of the corresponding elements):

$$\texttt{let x = S}_0\ \texttt{w' in K'[J'[x]]} \quad \sim_a \quad \texttt{K [J [S}_0\ \texttt{k. w k]]}$$

Notice that the rule allows us to track the progress of reassociation (context building) but we want to keep this rule separate from our usual structural similarity rules as the context building process appears only as a redex in the evaluation, so there is no need to associate such terms nested as arbitrary subterms.

It is later explained how the proof is completed in the presence of these separate similarity rules. Having discussed every proof case of the simulation step theorem we can finally define the similarity relation.

4.3 Similarity Relation

The similarity relation ($\approx$) needed for the second simulation theorem is the sum of the following three similarity relations:

($\sim$) — which is a redefined similarity relation from the previous chapter.

($\sim_a$) — which associates $\lambda_{c\$}$-terms that are in the process of context building (*reassociation* with a sequence of (*assoc*) contractions). Intuitively this syntactic judgement $\texttt{e'} \sim_a \texttt{e}$ is equivalent to the evaluation using only the (*assoc*) rule: $\texttt{e'} \ {}^* \xleftarrow{(assoc)} \pi(\texttt{e})$.

($\sim_{\rightarrow}$) — which keeps track of the extra evaluation step needed after the ($\$.let$) contraction that always produces a redex.

What remains to be covered in this chapter is to first formally define these similarity

relations and then discuss how the proof is completed.

4.3.1 Similarity Relation $(\sim)$

We start by defining the main component of the similarity relation, namely $(\sim)$:

$$\frac{\texttt{e' } \sim \texttt{ e}}{\lambda\ \texttt{x. e' } \sim \lambda\ \texttt{x. e}}\ (\sim_{abs}) \qquad \frac{}{\texttt{S}_0 \sim \lambda\ \texttt{m. S}_0\ \texttt{k. m k}}\ (\sim_{S_0})$$

$$\frac{\texttt{e1' } \sim \texttt{ e1 e2' } \sim \texttt{ e2}}{\texttt{e1' e2' } \sim \texttt{ e1 e2}}\ (\sim_{app}) \qquad \frac{\texttt{e1' } \sim \texttt{ e1 e2' } \sim \texttt{ e2}}{\texttt{e1' \$ e2' } \sim \texttt{ e1 \$ e2}}\ (\sim_{\$})$$

$$\frac{}{\texttt{x } \sim \texttt{ x}}\ (\sim_{var}) \qquad \frac{\texttt{J' } \sim \texttt{ J} \quad \texttt{e' } \sim \texttt{ e}}{\texttt{let x = e' in J'[x] } \sim \texttt{ J[e]}}\ (\sim_{name})$$

$$\frac{\texttt{v' } \sim \texttt{ v}}{\texttt{v' } \sim \lambda\ \texttt{x. v \$ x}}\ (\sim_{\eta}) \qquad \frac{\texttt{e1' } \sim \texttt{ e1} \quad \texttt{e2' } \sim \texttt{ e2}}{\texttt{let x = e1' in e2' } \sim \texttt{ (}\lambda\ \texttt{x. e2) e1}}\ (\sim_{let})$$

$$\frac{\texttt{v' } \sim \texttt{ v} \quad \texttt{K' } \sim \texttt{ K} \quad \texttt{e' } \sim \texttt{ e}}{\lambda\ \texttt{x. v' \$ K'[e'] } \sim \lambda\ \texttt{x. v \$ K[(}\lambda\ \texttt{x. e) x]}}\ (\sim_{exp})$$

The $(\sim)$ relation on terms is mutually inductive with the similarity relation on *stack frames* `J`, as well as *evaluation contexts* `K` which is required by the last rule.

$$\frac{\texttt{e' } \sim \texttt{ e}}{\texttt{\_ e' } \sim \texttt{ \_ e}}\ (\sim_{J_{fun}}) \qquad \frac{\texttt{v' } \sim \texttt{ v}}{\texttt{v' \_ } \sim \texttt{ v \_}}\ (\sim_{J_{arg}}) \qquad \frac{\texttt{e' } \sim \texttt{ e}}{\texttt{\_ \$ e' } \sim \texttt{ \_ \$ e}}\ (\sim_{J_{\$}})$$

$$\frac{}{\texttt{\_ } \sim \texttt{ \_}}\ (\sim_{K_{nil}}) \qquad \frac{\texttt{J' } \sim \texttt{ J'} \quad \texttt{K' } \sim \texttt{ K}}{\texttt{let x = K' in J'[x] } \sim \texttt{ J[K]}}\ (\sim_{K_{cons}})$$

$$\frac{\texttt{e' } \sim \texttt{ e'} \quad \texttt{K' } \sim \texttt{ K}}{\texttt{let x = K' in e' } \sim \texttt{ (}\lambda\ \texttt{x. e) K}}\ (\sim_{K_{let}})$$

Similarity is also defined for *trails* `T`.

$$\frac{}{\texttt{\_ } \sim \texttt{ \_}}\ (\sim_{T_{nil}}) \qquad \frac{\texttt{v' } \sim \texttt{ v} \quad \texttt{K' } \sim \texttt{ K} \quad \texttt{T' } \sim \texttt{ T}}{\texttt{v' \$ K'[T'] } \sim \texttt{ v \$ K[T]}}\ (\sim_{T_{cons}})$$

Most of the rules presented above should be familiar as they come from the $(\sim)$ relation from the previous chapter. These rules include simple structural rules, like

($\sim_{abs}$) and ($\sim_{app}$), and more special rules: ($\sim_{\eta}$) and ($\sim_{name}$) that we have already discussed.

Notice the difference in the ($\sim_{S_0}$) compared to the previous chapter. The syntax of $\lambda_{\$}$ does not support `shift0` as a standalone combinator, but as an abstraction instead. Thus similar $\lambda_{c\$}$-terms were restricted to the binder-like form as well. However this is not the case here as the syntax of $\lambda_{c\$}$ dictates rules now and the case of the standalone combinator must be covered in order to satisfy the reflexivity property of the (π) embedding.

The difference in syntax together with the (π) embedding dictated the need for the ($\sim_{let}$) rule as well. It associates let-expressions with their desugarred forms introduced by the embedding.

It is interesting to note that by not allowing the use of let-expressions in the input terms to the evaluation, thus not needing to desugar let-expressions in the embedding, the let-expression ambiguity could be avoided. It would make the inversion on let-expressions trivial as in the previous chapter, thus greatly simplifying the proof and making some of the similarity rules redundant. It is important to mention the complications that arise from these differences in syntax.

The last rule ($\sim_{exp}$) is yet another remedy for these complications. As it turns out after the context building process the continuation in the result of the ($\$.K$) step can have the following term `(λ x. e) x` deeply nested in the evaluation context in the body of the continuation. The problematic term is trivially equivalent to just `e` and it would reduce to it too, if it was not under a binder. Since it cannot reduce, we need a similarity rule that would associate `e'` with `(λ x. e) x` given `e' ∼ e`. However, introducing such a rule would be devastating to the inversion property and wasteful of our efforts to handle all inversion cases. The ($\sim_{exp}$) rule is made to be as specific as possible, thus it includes the whole continuation together with the `dollar` and evaluation contexts, to reduce the number of cases in the proof where we need to handle this complication. This situation happens when the inner-most part of the evaluation context is a desugarred let-expression. We are going to discuss this case later when it comes up during the completion of the proof.

Context similarity rules are almost identical to the ones from the previous chapter. There is yet one major difference, the extra ($\sim_{K_{let}}$) rule for evaluation contexts. The existence of the rule is, again, due to the let-expression ambiguity which means that evaluation context frames are made of stack frames `J` or desugarred let-expressions.

Notice that the similarity relation ($\sim$) associates ($\sim$)-similar terms in arbitrary ($\sim$)-similar contexts, even under binders. This is not the case for the remaining similarity relations ($\sim_a$) and ($\sim_{\rightarrow}$) as they are designed to associate only the specific terms that appear during the evaluation process.

4.3.2 Similarity Relation $(\sim_a)$

The following rules of the $(\sim_a)$ relation associate terms that appear during the *reassociation* (context building) process in the evaluation. Because of the let-expression ambiguity we technically need two, almost identical, rules. The differences between both rules are highlighted in green.

$$\frac{\mathtt{K}_0' \sim \mathtt{K}_0 \qquad \mathtt{T}_0' \sim \mathtt{T}_0 \qquad \mathtt{v}' \sim \mathtt{v} \qquad \mathtt{K}' \sim \mathtt{K} \qquad \mathtt{J}' \sim \mathtt{J}}{\mathtt{K}_0'\ [\mathtt{T}_0'\ [\mathtt{let\ x = S}_0\ \mathtt{v}'\ \mathtt{in\ K}'[\mathtt{J}'[\mathtt{x}]]]] \sim_a \mathtt{K}_0\ [\mathtt{T}_0\ [\mathtt{K}\ [\mathtt{J}\ [\mathtt{S}_0\ \mathtt{k.\ v\ k}]]]]}\ (\sim_{a_J})$$

$$\frac{\mathtt{K}_0' \sim \mathtt{K}_0 \qquad \mathtt{T}_0' \sim \mathtt{T}_0 \qquad \mathtt{v}' \sim \mathtt{v} \qquad \mathtt{K}' \sim \mathtt{K} \qquad \mathtt{e}' \sim \mathtt{e}}{\mathtt{K}_0'\ [\mathtt{T}_0'\ [\mathtt{let\ x = S}_0\ \mathtt{v}'\ \mathtt{in\ K}'[\mathtt{e}']]] \sim_a \mathtt{K}_0\ [\mathtt{T}_0\ [\mathtt{K}\ [(\lambda\ \mathtt{x.\ e})(\mathtt{S}_0\ \mathtt{k.\ v\ k})]]]}\ (\sim_{a_{let}})$$

Notice the use of top-level contexts with the $_0$ subscript. These are designed to correspond to the evaluation contexts and trails from the *eval* steps.

Despite the difference in both rules, the idea behind them is identical. Each rule indicates how many (*assoc*) contractions were executed resulting with the $\lambda_{c\$}$-term that is on the left-hand side of the $(\sim_a)$ judgement. The number of contractions is encoded in the evaluation context `K'` meaning that the following sequence of steps was executed:

$$\mathtt{K'\ [let\ x = S}_0\ \mathtt{v'\ in\ J'[x]]} \xrightarrow{(assoc)}{}^{*}\ \mathtt{let\ x = S}_0\ \mathtt{v'\ in\ K'[J'[x]]} \quad \text{or}$$
$$\mathtt{K'\ [let\ x = S}_0\ \mathtt{v'\ in\ e'\ \ ]} \xrightarrow{(assoc)}{}^{*}\ \mathtt{let\ x = S}_0\ \mathtt{v'\ in\ K'[e'\ \ ]}$$

Therefore, we can group a sequence of (*assoc*) steps in a single $(\sim_a)$ rule, so that when no more (*assoc*) contractions can be executed, we can use the premises of either rule to prove that a grouped sequence of (*assoc*) steps together with the ($\$.let$) contraction and finished by the ($\beta.\$$) step is $(\sim)$-similar to a single ($\$.K$) contraction.

Note that choosing the correct rule of the two depends on the inner-most frame of the evaluation context that is enclosing the `shift0` operation.

4.3.3 Similarity Relation $(\sim_\rightarrow)$

The $(\sim_\rightarrow)$ is actually not a syntactic relation. Its premises simply require a single evaluation step and a proof of $(\sim)$-similarity of the result. It is used in the proof of the *simulation step* lemma to keep track of the extra evaluation step needed after the ($\$.let$) contraction that always produces a redex. Since we are not allowed to perform any evaluation steps on the $\lambda_{c\$}$-term, we postpone the $(\sim)$-similarity conclusion until the next step is considered.

$$\frac{\mathsf{e}_r' \longrightarrow' \mathsf{e}' \qquad \mathsf{e}' \sim \mathsf{e}}{\mathsf{e}_r' \sim_\rightarrow \mathsf{e}}\ (\sim_\rightarrow)$$

4.4 Proof Continuation

Having defined the final similarity relation, we can complete the proof by considering all the missing cases. Recall that to prove the main simulation theorem, we need the auxiliary *simulation step* theorem:

$$\forall\, e_1', e_2' \in \lambda_{c\$}.\quad \forall\, e_1 \in \lambda_{\$}.\\ \quad e_1' \approx e_1 \;\wedge\; e_1' \longrightarrow' e_2' \implies \exists\, e_2 \in \lambda_{\$}.\quad e_2' \approx e_2 \;\wedge\; e_1 \longrightarrow^* e_2$$

We start with an inversion on the similarity judgement $\mathtt{e}_1' \approx \mathtt{e}_1$ first. There are three cases to consider, but notice that the first one was already handled in the sketch of the proof, which appears in the formalisation as a separate lemma:

$$\forall\, e_1', e_2' \in \lambda_{c\$}.\quad \forall\, e_1 \in \lambda_{\$}.\\ \quad e_1' \sim e_1 \;\wedge\; e_1' \longrightarrow' e_2' \implies \exists\, e_2 \in \lambda_{\$}.\quad e_2' \approx e_2 \;\wedge\; e_1 \longrightarrow^* e_2$$

The only difference to the *simulation step* lemma is the similarity premise highlighted in green. The plan for the rest of the chapter is to handle the remaining two cases of the auxiliary lemma, thus completing the proof of the main simulation lemma.

Case $(\sim_{\rightarrow})$*:*

Assumptions: $\mathtt{e}_1' \sim_{\rightarrow} \mathtt{e}_1 \;\wedge\; \mathtt{e}_1' \longrightarrow' \mathtt{e}_2'$

The latter case is trivial as all the reasoning is already done and contained within the similarity judgement $\mathtt{e}_1' \sim_{\rightarrow} \mathtt{e}_1$ which, by inversion, implies:

New Assumptions: $\mathtt{e}_1' \longrightarrow' \mathtt{e}' \;\wedge\; \mathtt{e}' \sim \mathtt{e}_1$

Notice that there are now two distinct $(\longrightarrow')$ steps from the same term $\mathtt{e}_1'$. Since the evaluation is deterministic the conclusion is that these steps are equal, so their results are equal as well: $\mathtt{e}_2' = \mathtt{e}'$. Which means that no $(\longrightarrow)$ step is needed and we can let $\mathtt{e}_2 := \mathtt{e}_1$ and finish the goal with the assumption $\mathtt{e}' \sim \mathtt{e}_1$.

Case $(\sim_a)$*:*

Assumptions: $\mathtt{e}_1' \sim_a \mathtt{e}_1 \;\wedge\; \mathtt{e}_1' \longrightarrow' \mathtt{e}_2'$

Inversion of the $(\sim_a)$-similarity judgement is needed to proceed with the proof. There are two cases to handle, however, as both require the same reasoning, only the former case is considered.

Assumptions: $\mathtt{K}_0'\ \mathtt{[T}_0'\ \mathtt{[let\ x\ =\ S}_0\ \mathtt{v'\ in\ K'[J'[x]]]]} \longrightarrow' \mathtt{e}_2'$
Goal: $\exists \mathtt{e}_2.\quad \mathtt{e}_2' \approx \mathtt{e}_2 \;\wedge\; \mathtt{K}_0\ \mathtt{[T}_0\ \mathtt{[K\ [J\ [S}_0\ \mathtt{k.\ v\ k]]]]} \longrightarrow^* \mathtt{e}_2$

Recall the intuition behind $(\sim_a)$ rules – the evaluation is in the middle of the *reassociation* process. The task is to determine whether the context building process is already done, and finished by the ($\$.let$) contraction, or it requires another ($assoc$) step. Notice that it depends on the enclosing trail context $\mathtt{T}_0'$.

Case 1: $\mathtt{T}_0' = \_$

When it is empty we can conclude that $\mathtt{K}'_0$ = $\mathtt{K}'_1$ [let y = _ in $\mathtt{e}'_y$] because $\mathtt{K}'_0$ cannot be empty as it would contradict the evaluation step assumption. Determinism of the $(\longrightarrow')$ relation proves the following:

$\mathtt{K}'_1$ [let y = let x = $\mathtt{S}_0$ v′ in K′[J′[x]] in $\mathtt{e}'_y$] $\longrightarrow'$
$\mathtt{K}'_1$ [let x = $\mathtt{S}_0$ v′ in let y = K′[J′[x]] in $\mathtt{e}'_y$] = $\mathtt{e}'_2$

The goal is finished by using the $(\sim_{a_J})$ rule. Notice that the proof in this case simply reconstructed the $(\sim_{a_J})$ rule by taking the inner-most frame from the $\mathtt{K}'_0$ context and putting it on top of the evaluation context K′.

Case 2: $\mathtt{T}'_0$ = $\mathtt{T}'_1$ [$\mathtt{w}'_1$ $ $\mathtt{K}'_1$]

The latter case, when the trail is not empty, further branches out depending on the inner-most evaluation context $\mathtt{K}'_1$ in the trail.

Case 2.1: $\mathtt{K}'_1$ = _

When it is empty there is only one possible evaluation step and that is the ($\$.let$) contraction:

$\mathtt{K}'_0$ [$\mathtt{T}'_1$ [$\mathtt{w}'_1$ $ [let x = $\mathtt{S}_0$ v′ in K′[J′[x]]]]] $\longrightarrow'$
$\mathtt{K}'_0$ [$\mathtt{T}'_1$ [(λ x. $\mathtt{w}'_1$ $ K′[J′[x]]) $ $\mathtt{S}_0$ v′]] = $\mathtt{e}'_2$

Since the $\lambda_{c\$}$ term finished the *reassociation* process and executed the ($\$.let$) contraction, its similar $\lambda_{\$}$ counterpart must evaluate equivalently using the ($\$.K$) contraction.

$\mathtt{K}_0$ [$\mathtt{T}_1$ [$\mathtt{w}_1$ $ K [J [$\mathtt{S}_0$ k. v k]]]] $\longrightarrow$
$\mathtt{K}_0$ [$\mathtt{T}_1$ [v (λ x. $\mathtt{w}_1$ $ K [J [x]])]] =: $\mathtt{e}_2$

The similarity of both results is proven with the $(\sim_{\rightarrow})$ rule where the extra ($\beta.\$$) contraction is used to make the following step:

$\mathtt{K}'_0$ [$\mathtt{T}'_1$ [(λ x. $\mathtt{w}'_1$ $ K′[J′[x]]) $ $\mathtt{S}_0$ v′]] $\xrightarrow{(\beta.\$)}{}'$
$\mathtt{K}'_0$ [$\mathtt{T}'_1$ [v′ (λ x. $\mathtt{w}'_1$ $ K′[J′[x]])]]

Which makes proving the $(\sim)$-similarity of both results trivial as their structure is identical and the corresponding elements are $(\sim)$-similar.

Case 2.2: $\mathtt{K}'_1 \neq$ _

This case is almost identical to the *Case 1* where the context building continues, so the same reasoning applies here. The only difference is that it happens in a larger context now, however this is not a problem since both similarity and evaluation are preserved by the plug operation.

Chapter 5

The Formalisation in Coq

The simulation theorems that were covered in previous chapters are formalised in Coq, together with the implementation of both *coarse-grained* $\lambda_{\$}$ and *fine-grained* $\lambda_{c\$}$ calculi with their evaluation relations and necessary lemmas about them [20]. The formalisation guarantees the correctness of the result, however the proofs become much more difficult as the machine diligently checks that every case is handled with utmost care and formal precision. Every step in the proof, however trivial, must be justified by a formal proof. Nevertheless, some level of automation is possible with the use of *tactics* that reduce the boilerplate code that constructs the proofs.

The previous chapters present the general idea behind the simulation theorem proofs and deliberately avoid some details in order to clearly convey the idea behind the simulation process between both calculi and focus on the intuition that justifies their correctness in terms of similarity. The purpose of this chapter is not to clarify every detail, as for this information it is best to refer to the formalisation code itself, but to discuss key design decisions made to model both calculi with their evaluation relations and how they affected the implementation.

The plan is to focus on the term representation and the variable encoding first as this choice greatly affects the rest of the formalisation. Secondly, both evaluation relations are going to be covered next, explaining their implementation together with their properties used throughout the majority of proofs. And finally the last topic is the preservation property of the similarity relation by the substitution and the plug operation.

5.1 The Term Representation

To represent terms in Coq one must pick a method of encoding variables. The simplest solution would be to represent variables as strings, which is the most readable format. Unfortunately this approach makes α-equivalent terms distinct which forces us to use equivalence relation, instead of Coq equality, making the whole process

more complicated. Moreover it lacks any form of controlling free variables, which is problematic when working with closed terms, and working with the evaluation means dealing with closed terms almost exclusively.

The alternative is to use the de Bruijn indices [1]. This format represents occurrences of variables as natural numbers that indicate the number of binders that are in scope between that occurrence and its corresponding binder. It resolves the α-equivalence issue as variables are identified by their occurrence and not their name, but it still gives no way of controlling free variables.

A solution to both issues is the format called *nested datatypes* [12], which asserts that in a term of type `tm A` the only free variables are of type `A`. As a result the constructor of a variable occurrence needs to have a type `A → tm A`, whereas the application has type `tm A → tm A → tm A` as it takes two terms with the same set of free variables to create an application term of both having the same set of free variables. The question is how a lambda abstraction, or any binder, is represented in this format?

The answer takes the inspiration from de Bruijn indices. The abstraction has type `tm (option A) → tm A` which means that the occurrences of variables in the body are either free and equal to `Some a` where `a : A`, or bound by this binder and equal to `None`. By defining n `:= Some`^{n} `None` we can translate any closed term with de Bruijn indices to the *nested datatypes* format. The `Some` constructor works just like a successor for the indices, whereas `None` acts as the `0` index that is bound by the enclosing lambda.

With this format it is trivial to define closed terms by simply taking an uninhabited type for the type parameter `A`. For example the following $\emptyset$ type is uninhabited as it does not have any constructors.

```
Inductive ∅ : Type := .
```

Using the *nested datatypes* format we can define terms for the $\lambda_{\$}$ calculus as the following `tm A` type:

```
Inductive tm A :=
| tm_var : A → tm A             (*  x        *)
| tm_abs : tm ^A → tm A         (*  λ  x. e   *)
| tm_s_0 : tm ^A → tm A         (*  S0 f. e   *)
| tm_app : tm A → tm A → tm A   (*  e   e     *)
| tm_dol : tm A → tm A → tm A   (*  e $ e     *)
.
```

To simplify the format, the notation `^A` is used instead of `option A`. The `tm A` has five constructors, the first one creates variable occurrences, the next two stand for

lambda and `shift0` abstractions, whereas the last two are the application constructor and the `dollar` delimiter expression. The comment next to each constructor describes the equivalent standard format with named variables.

Whereas the definition of $\lambda_{c\$}$ terms is:

```
Inductive tm' A :=
| tm_var' : A → tm' A                (* x                *)
| tm_s_0' : tm' A                    (* S_0              *)
| tm_abs' : tm' ^A → tm' A           (* λ x. e           *)
| tm_app' : tm' A → tm' A → tm' A    (* e e              *)
| tm_dol' : tm' A → tm' A → tm' A    (* e $ e            *)
| tm_let' : tm' A → tm' ^A → tm' A   (* let x = e1 in e2 *)
.
```

Notice that the let-expression constructor is also a binder for its second argument as indicated by its type of free variables. Both term definitions are quite similar, recall the convention to use ' to indicate the definitions of the fine-grained calculus. Since all the definitions described in this chapter have their ' version, we will not mention them from now on.

Notice that the structure of both these types is flat, there is no distinction between *values* and *non-values*. These syntactic categories are defined as separate datatypes (`val A` and `non A` respectively) in order to avoid their natural mutual recursion between these three categories, thus making the induction principle for these types easier to work with. However the alternative would also be viable.

5.2 Evaluation

The definition of the evaluation relation matches the form of the (*eval*) definition. It is defined as the following `step` relation that uses the contraction relation `contr` and the plug operation, which is hidden behind the notation.

```
Inductive contr : tm ∅ → tm ∅ → Prop :=
| contr_tm : ∀ (r : redex ∅), r ~> contract r
where "e1 ~> e2" := (contr e1 e2).

Inductive step : tm ∅ → tm ∅ → Prop :=
| step_tm : ∀ (k : K ∅) (t : T ∅) (e1 e2 : tm ∅),
    e1 ~> e2 →
    <{ k[t[e1]] }> --> <{ k[t[e2]] }>
where "e1 --> e2" := (step e1 e2).
```

The `contr` relation associates any closed-term redex with its result after the contraction, which is done by the function `contract : redex ∅ → tm ∅` designed according to the $(\rightsquigarrow)$ relation.

The `plug` operation is overloaded by being defined in the type class `Plug` so that a single notation is applicable for every context `J`, `K` and `T`.

```
Class Plug (C : Type → Type) := plug : ∀ {A}, C A → tm A → tm A.

Notation "C [ e ]" := (plug C e)
  (* ... *).
```

There is also a set of notations applicable inside `<{ ... }>` brackets used to build terms in a natural style without referring to the explicit constructors. The equivalent notations for the $\lambda_{c\$}$-terms are applied inside `<| ... |>` brackets.

The definition of the `step` relation makes it easy to prove the following `step_kt` lemma that together with the `multi_kt` variant are one of the most frequently used lemmas in both simulation proofs.

```
Lemma step_kt : ∀ {e1 e2} {k : K ∅} {t : T ∅},
         e1      -->       e2 →
  <{ k[t[e1]] }> --> <{ k[t[e2]] }>.
```

It means that an evaluation step can be executed in an arbitrary `K[T]` context and this property naturally extends to a multi-step relation `-->*` that is defined as a reflexive transitive closure of the `-->` step relation.

```
Lemma multi_kt : ∀ {e1 e2} {k : K ∅} {t : T ∅},
         e1      -->*       e2 →
  <{ k[t[e1]] }> -->* <{ k[t[e2]] }>.
```

5.3 Unique Decomposition

The `step` relation is defined in a declarative way as a one-step evaluation relation. However, it is not immediately obvious how an evaluation procedure, that computes terms until the result is a value or a stuck term, can be derived from it.

An evaluation procedure given a closed term `e` must decompose it into the context `k[t]` and a redex `r` such that `e = <{ k[t[r]] }>`. Notice that once the decomposition is done, it is easy to proceed with the evaluation by contracting the redex using the `contract` function and plugging the result back into the context, thus completing a single evaluation step that can easily be iterated.

It is only reasonable to expect this procedure to be deterministic, meaning that there are no two distinct steps from the same term, which is easily proven given that the *unique decomposition* [8] property holds. Intuitively, it means that the decomposition and the plug operation are both inverses of each other.

The `decompose` function is provided to prove the uniqueness property thus ensuring the correctness of the evaluation relation `step`. Since, in general, evaluation may be non-terminating or stuck, the `decompose` function has the following return type:

```
Inductive dec :=
| dec_value : val ∅ → dec                  (*  value        *)
| dec_stuck : K ∅ → tm ^∅ → dec            (*  K[S₀ f. e]    *)
| dec_redex : K ∅ → T ∅ → redex ∅ → dec    (*  K[T[Redex]]  *)
.

Fixpoint decompose : tm ∅ -> dec.
(* ... *)
```

The result of the decomposition can be a value – when the term is fully evaluated, or a stuck term – when there is no matching `dollar` delimiter for a `shift0` operation (which can appear in an arbitrary evaluation context `K`), or a decomposition into an evaluation context with a trail and a redex.

The provided implementation pattern-matches on the input term, calling itself recursively in case of an application or a delimiter expression and constructs the appropriate `dec` result.

Once the `decompose` function is implemented, the unique decomposition property can be stated and proven with the following set of lemmas.

```
(* plug ∘ decompose = id *)
Lemma decompose_value_inversion : ∀ e (v : val ∅),
  decompose e = dec_value v → e = v.
Lemma decompose_stuck_inversion : ∀ e k e',
  decompose e = dec_stuck k e' → e = <{ k[S₀ e'] }>.
Lemma decompose_redex_inversion : ∀ e t k r,
  decompose e = dec_redex k t r → e = <{ k[t[r]] }>.

(* decompose ∘ plug = id *)
Lemma decompose_plug_value : ∀ (v : val ∅),
  decompose v = dec_value v.
Lemma decompose_plug_stuck : ∀ k e,
  decompose <{ k[S₀ e] }> = dec_stuck k e.
Lemma decompose_plug_redex : ∀ k t (r : redex ∅),
  decompose <{ k[t[r]] }> = dec_redex k t r.
```

The comment above each group of lemmas describe their intend. The former group proves that plugging the decomposition result is an identity function on closed terms, whereas the latter proves the reverse property. Since there are three distinct cases of the decomposition, there is a separate lemma for each case in each group.

The last lemma is the main building block in the proof of the determinacy of the `step` relation:

```
Lemma deterministic_step : ∀ e e1 e2,
  e --> e1 →
  e --> e2 →
  e1 = e2.
```

5.4 Contraction

So far we have omitted the implementation details behind the `contract` function. While the intent of this function is trivial as it has to perform term rewriting according to the rules from the ($\rightsquigarrow$) relation, the actual implementation requires both the *substitution* as well as the *lift* operation that we are about to describe. Consider the following implementation.

```
Definition contract (r : redex ∅) : tm ∅ :=
  match r with
  (*           (λ x.    e) v  ~>     e [x := v] *)
  | redex_beta (val_abs e) v  =>  <{ e [0 := v] }>

  (*             v1 $ v2  ~>     v1 v2 *)
  | redex_dollar v1   v2  =>  <{ v1 v2 }>

  (*            v $ K[S₀ f. e]  ~>     e [f := λ x. v $  K[x]] *)
  | redex_shift v   k       e   =>  <{ e [0 := λ   ↑v $ ↑k[0]] }>
  end.
```

The aligned comments above each pattern-match case describe the corresponding ($\rightsquigarrow$) rule. The first rule performs the substitution, which in the nested-datatypes format means that the `None` occurrences are replaced with the substituted value. Instead of using the `None` constructor explicitly, we use the number `0` as a notation for it.

The substitution has the following type `tm ^A → val A → tm A`. Notice that once all `None` occurrences are replaced, every other variable is equal to `Some v` and must drop the `Some` constructor in order to match the required result type `tm A`.

The above substitution works only for replacing the variables `0`. While such an operation is sufficient for our needs, thus there is only one notation for the substitution and that is `e [0 := v]` with the fixed `0`, its implementation is more sophisticated and requires an extremely general approach that is covered in the next section.

The last rule in the `contract` function uses yet another new operation that is called `lift`, hidden behind the notation `↑`. This is yet another overloaded operation that can be implemented for any definition that contains terms, e.g. evaluation contexts and trails. *Lifting* is a process known from the format of de Bruijn indicies and its purpose is identical here. Like in any type-safe format the implementation of the operation can be at least partly deduced from its type and the type of the `lift` operation for terms is `tm A → tm ^A`. It wraps each **free** variable occurrence with the `Some` constructor. The desirable outcome of this operation is that the substitution declared above behaves almost like an identity function on lifted terms. For example the following lemma holds.

```
Lemma subst_lift : ∀ {A} (e : tm A) v,
  <{ (↑e) [0 := v] }> = e.
```

Notice that in the third rule both the value `v` and the evaluation context `K` were placed inside a lambda, which in general requires lifting to avoid the accidental *name capture* (or *variable capture* [6]). Another advantage to using the nested-datatypes is its type-safety that lead to fewer programming mistakes. For example, forgetting any `↑` operation in the last rule would result in a type error.

The implementation of both the lift operation as well as substitution should not be implemented naively as their direct implementation leads to laborious proofs with dependent types [21].

5.5 Lift

The proper way of providing these specific operations is to define them as instances of the more general operations. Notice that lifting is a specific variable renaming, which can be implemented as:

```
Fixpoint map {A B : Type} (f : A → B) (e : tm A) : tm B :=
  match e with
  | <{ var a }> => <{ var {f a} }>
  | <{ e1   e2 }> => <{ {map f e1}   {map f e2} }>
  | <{ e1 $ e2 }> => <{ {map f e1} $ {map f e2} }>
  | <{ λ  e' }> => <{ λ  {map (option_map f) e'} }>
  | <{ S₀ e' }> => <{ S₀ {map (option_map f) e'} }>
  end.
```

The `map` operation changes free variable occurrences using the function `f`. Notice the use of the `option_map` function in the last two cases, which prevents altering the bound occurrences under the binders. The `tm` type has the functorial structure which is proved with the following lemmas about the Functor laws.

```
Lemma map_id_law : ∀ A (e : tm A),
  map id e = e.

Lemma map_map_law : ∀ A e B C (f : A → B) (g : B → C),
  map g (map f e) = map (g ∘ f) e.
```

And now the lift operation on terms is simply `map Some : tm A → tm ^A`. The Functor implementation has to be carried out for every relevant syntactic category that requires the lift operation.

5.6 Substitution

Similarly to the lift operation we need to define a substitution more generally. The `map` operation would not be enough as the substitution should replace variables with other terms. If we were to use the map operation regardless, we would get a result of type `tm (tm A)` that would require further flattening to just `tm A`. A Haskell programmer might recognise this pattern and correctly identify that the desired implementation requires the `bind` method and that the type constructor `tm` is also a monad.

```
Fixpoint bind {A B : Type} (f : A → tm B) (e : tm A) : tm B :=
  match e with
  | <{ var a }> => f a
  | <{ e1   e2 }> => <{ {bind f e1}   {bind f e2} }>
  | <{ e1 $ e2 }> => <{ {bind f e1} $ {bind f e2} }>
  | <{ λ  e' }> => tm_abs (bind (fun a' =>
      match a' with
      | None   => tm_var None
      | Some a => map Some (f a)
      end) e')
  | <{ S_0 e' }> => tm_s_0 (bind (fun a' =>
      match a' with
      | None   => tm_var None
      | Some a => map Some (f a)
      end) e')
  end.
```

When interpreting a term `tm A` as a tree datatype with leaves of type `A`, the `map` operation changes the leaves without altering the structure of the whole tree, whereas the `bind` method replaces leaves with new sub-trees. This behaviour is ideal for the substitution as we want to replace certain free variable occurrences with a value-term and attach it as a sub-tree, thus changing the structure of the whole tree.

Notice that in the implementation the last two cases bind the body `e'` with a carefully constructed function that does not alter the bound occurrences of `None` and that the result of applying the function `f` is lifted to prevent the variable capture as well as ensuring that the body of the binder has the correct type.

Once the `bind` is implemented the substitution becomes:

```
Definition var_subst {A} (v : val A) (o : ^A) : tm A :=
  match o with
  | None => v
  | Some a => <{ var a }>
  end.

Notation "e [ 0 := v ]" := (bind (var_subst v) e)
  (* ... *).
```

5.7 Laws

The `tm` type constructor is a monad because the following laws hold:

```
Lemma bind_bind_law : ∀ {A B C} (g : B → tm C) (f : A → tm B) (e : tm A),
  bind g (bind f e) = bind (λ a, bind g (f a)) e.

Lemma bind_pure : ∀ {A} (e : tm A),
  bind (λ a, <{ var a }>) e = e.

Lemma bind_trivial_identity : ∀ {A B} (f : A → tm B) (a : A),
  bind f <{ var a }> = f a.
```

Proving the Functor and Monad laws seem unnecessary, however they are crucial in simplifying expressions that concern both lifting and the substitution. For example these laws construct the proof of the `subst_lift` lemma, which is frequently used to simplify terms in both simulation theorem proofs.

Another pivotal rewriting rule is the following `bind_var_subst_lift_k` lemma that is necessary to prove the latter simulation. It is a generalised version of the `subst_lift` lemma that describes the interaction of the substitution on a term plugged into a lifted context (which is done by lifting every term in the context).

```
Lemma bind_var_subst_lift_k : ∀ {A} (k : K A) e v,
  <{ ↑k[e][0 := v] }> = <{ k [e [0 := v]] }>.
```

All necessary rewriting rules are a part of the `laws` tactic that automatically simplify all terms in the current proof context.

5.8 Similarity Preservation

Both similarity relations are directly implemented as described in previous chapters. These are mutually recursive definitions for terms as well as contexts.

The purpose of this section is to describe the interaction between the term operations defined in this chapter and the similarity relations. In general, we can describe these properties as the similarity preservation that we have assumed to be true in previous chapters. Note that the properties in this chapter refer to the first simulation theorem, however the methods described in this section are applicable to definitions used in the second simulation theorem as well.

The operations that should preserve the similarity are: the plug operation, lifting and the substitution. The following lemmas are abundantly used in both simulation theorem proofs:

```
Lemma sim_plug_kt : ∀ {A} (k :  K A) (k' :  K' A)
                          (t :  T A) (t' :  T' A)
                          (e : tm A) (e' : tm' A),
  k ~ k' →
  t ~ t' →
  e ~ e' →
  <{ k[t[e]] }> ~ <| k'[t'[e']] |>.

Lemma sim_lift : ∀ {A} {e : tm A} {e'},
   e ~  e' →
  ↑e ~ ↑e'.

Lemma sim_lift_k : ∀ {A} {k : K A} {k'},
   k ~  k' →
  ↑k ~ ↑k'.

Lemma sim_subst_lemma : ∀ e e' v (v' : val' ∅),
  e ~ e' →
  v ~ v' →
  <{ e [0 := v] }> ~ <| e' [0 := v'] |>.
```

The first property is easily derived from intermediary lemmas that are proven with a simple structural induction over contexts. However, proving the rest of the lemmas is more complicated as both lifting and the substitution are implemented using the `map` and `bind`. Instead of proving these properties directly, we need to state general preservation properties by these operations first and then use them to derive the specific properties that we need.

The first general similarity preservation theorem is the following:

```
Lemma sim_map : ∀ {A} {e e' B} {f : A → B},
  e ~ e' →
  map f e ~ map' f e'.
```

It states that regardless of the variable renaming method the similarity is still preserved. The proof goes by induction over the similarity judgement which, since the similarity is mutually-recursive, requires the mutual induction. The only complication over a simple induction is that we need to provide the inductive hypothesis for every other syntactic category used in the mutually recursive similarity judgement. Refer to the formalisation code [20] for further details on this subject.

Several interesting, yet obvious, properties are required to complete the proof and these are the following:

```
Lemma lift_map : ∀ {A B} (e : tm A) (f : A → B),
  ↑(map f e) = map (option_map f) (↑e).

Lemma map_plug_k_is_plug_of_maps : ∀ {A B} (k : K A) e (f : A → B),
  map f <{ k[e] }> = <{ (map f k) [(map f e)] }>.
```

The former property is a corollary of the functor composition law. It states that mapping a term and then lifting the result is the same as lifting the input term first and mapping the result with a function *lifted* over the `option` type (which is done using the `option_map`). The letter lemma is a distributive law of `map` over the plug operation.

The second general similarity preservation theorem is:

```
Lemma sim_bind : ∀ {A} {e e' B} {f : A → tm B} {f' : A → tm' B},
  e ~ e' →
  (∀ a, f a ~ f' a) →
  bind f e ~ bind' f' e'.
```

Is states that the similarity of `bind` is preserved as long the input terms are similar and that the binding functions preserve the similarity too. The proof is carried out

analogously to the previous one and it also requires additional properties about the `bind` operation to complete.

```
Lemma bind_lift : ∀ {A B} e (f : ^A → tm B),
  bind f (↑e) = bind (f ∘ Some) e.

Lemma lift_bind : ∀ {A B} e (f : A → tm B),
  ↑(bind f e) = bind (lift ∘ f) e.

Lemma bind_plug_k_is_plug_of_binds : ∀ {A B} (k : K A) e (f : A → tm B),
  bind f <{ k[e] }> = <{ (bind f k) [(bind f e)] }>.
```

The first two lemmas describe the interaction between lifting and binding, whereas the last lemma is a distributivity law, but for the `bind` operation.

Once the properties described in this chapter are available the actual proofs of both simulation theorems become easy to carry out according to the proving strategy described in previous chapters.

Chapter 6

Conclusion and Future Work

The result of this thesis is a novel evaluation strategy for the $\lambda_{c\$}$ calculus that is defined as a subset of the $\lambda_{c\$}$ reduction theory in order to preserve the theoretical properties proven in the original work. The fine-grained evaluation is proven equivalent, via a pair of simulation theorems formalised in Coq, to the standard $\lambda_{\$}$ evaluation that explicitly manipulates the context in the contraction rules.

The technical problems induced by the differences in syntax between both calculi proved too difficult to provide the equivalence proof as a standard bisimulation. It would be interesting to determine how the whole formalisation simplifies by providing a common calculus for both evaluation strategies, thus focusing primarily on the difference between the coarse-grained and fine-grained evaluation.

Another possible extension of this thesis would be to carry out the formalisation of the alternative fine-grained evaluation, described in the second chapter, that does not use the (*assoc*) contraction which usually is not required in an evaluation relation. This strategy behaves two-fold depending on the context, changing its behaviour when evaluating under a delimiter. As a result the `step_kt` lemma would hold only for empty trails or for a common subset of steps that are applicable under delimiters as well.

An interesting project topic is to combine both ideas together and to formalise a proof of bisimilarity for the alternative evaluation where both fine-grained and coarse-grained strategies are defined for the same calculus – reducing the unnecessary differences as much as possible.

A separate future work would explore the possibility of the similarity relation implementation restricted to closed terms only. This change would require the use of *closures* and *environments* instead of a direct substitution. This approach would utilise the fact that, since the evaluation operates on closed terms only, there is no need to implement the similarity relation for arbitrary terms, which might simplify the whole formalisation.

The fine-grained evaluation strategy that uses the dreadful ($assoc$) contraction might actually prove advantageous if we were to extend both calculi with the `control0` operator as it is easy to extend this fine-grained evaluation strategy to support it. Unlike the alternative strategy where it seems to be impossible as `dollar` delimiters are eagerly inserted during the context capture. The future work would be to add the `control0` operator and carry out the simulation theorems.

Bibliography

[1] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". W: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), s. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[2] G.D. Plotkin. "Call-by-name, call-by-value and the λ-calculus". W: *Theoretical Computer Science* 1.2 (1975), s. 125–159. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90017-1.

[3] Matthias Felleisen i in. "Abstract Continuations: A Mathematical Semantics for Handling Full Jumps." W: sty. 1988, s. 52–62. DOI: 10.1145/62678.62684.

[4] Olivier Danvy i Andrzej Filinski. "Abstracting Control". W: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: Association for Computing Machinery, 1990, s. 151–160. ISBN: 089791368X. DOI: 10.1145/91556.91622. URL: https://doi.org/10.1145/91556.91622.

[5] Amr Sabry i Philip Wadler. "A Reflection on Call-by-Value". W: *ACM Trans. Program. Lang. Syst.* 19.6 (list. 1997), s. 916–941. ISSN: 0164-0925. DOI: 10.1145/267959.269968. URL: https://doi.org/10.1145/267959.269968.

[6] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[7] Chung-chieh Shan. "Shift to control". W: *Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA*. (2004).

[8] Malgorzata Biernacka i D. Biernacki. "Formalizing constructions of abstract machines for functional languages in Coq". W: *Electronic Notes in Theoretical Computer Science* (sty. 2007), s. 84–99.

[9] "Revised^6 Report on the Algorithmic Language Scheme". W: *Journal of Functional Programming* 19 (2007), s. 1–301.

[10] Gordon Plotkin i Matija Pretnar. "Handlers of Algebraic Effects". W: *Programming Languages and Systems*. Red. Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 80–94. ISBN: 978-3-642-00590-9.

[11] Simon Marlow. "Haskell 2010 Language Report". W: (lip. 2010).

[12] André Hirschowitz i Marco Maggesi. “Nested Abstract Syntax in Coq”. W: *Journal of Automated Reasoning - JAR* 49 (paź. 2012), s. 1–18. DOI: 10.1007/s10817-010-9207-9.

[13] Simon Marlow i Simon Peyton Jones. “The Glasgow Haskell Compiler”. W: *The Architecture of Open Source Applications, Volume 2*. The Architecture of Open Source Applications, Volume 2. Lulu, sty. 2012. URL: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/.

[14] Marek Materzok i Dariusz Biernacki. “A Dynamic Interpretation of the CPS Hierarchy”. W: *Programming Languages and Systems*. Red. Ranjit Jhala i Atsushi Igarashi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 296–311. ISBN: 978-3-642-35182-2.

[15] Yannick Forster i in. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control”. W: *CoRR* abs/1610.09161 (2016). arXiv: 1610.09161. URL: http://arxiv.org/abs/1610.09161.

[16] Maciej Piróg, Piotr Polesiuk i Filip Sieczkowski. “Typed Equivalence of Effect Handlers and Delimited Control”. W: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Red. Herman Geuvers. T. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 30:1–30:16. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.30. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10537.

[17] Dariusz Biernacki, Mateusz Pyzik i Filip Sieczkowski. “Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory”. W: *23rd International Symposium on Principles and Practice of Declarative Programming*. PPDP 2021. Tallinn, Estonia: Association for Computing Machinery, 2021. ISBN: 9781450386890. DOI: 10.1145/3479394.3479399. URL: https://doi.org/10.1145/3479394.3479399.

[18] Alexis King. *Delimited continuation primops*. URL: https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst.

[19] Oleg Kiselyov. *Undelimited continuations are not functions*. URL: https://okmij.org/ftp/continuations/undelimited.html.

[20] Kamil Listopad. *The Coq Formalisation of the Delimited-Control Operators shift0/dollar*. URL: https://github.com/Kamirus/fine-grained-shift0-dollar.

[21] Kamil Listopad. *The Coq Formalisation of the Simply Typed Lambda Calculus with Type-Level de Bruijn indices*. URL: https://github.com/Kamirus/lambda-formalizations/blob/2175eebe38a89db7141cb11e7da77c3d113674ce/stlc.v.