Radboud Universiteit

Faculty of Science

Bunchless λ-calculus for Bunched Implications

Building resource tracking with bunched implications on top of a simple types

Thesis MSc: Mathematical Foundations of Computer Science

Author:
Thomas Somers

External Supervisor:
Neel Krishnaswami
University of Cambridge

Internal Supervisor:
Robbert Krebbers

Second reader:
Herman Geuvers

November 6, 2024

Abstract

Type systems of modern languages such as Rust provide guarantees of memory safety and memory leak freedom by tracking resources. However, for existing software, porting the entire program to a new language is not always feasible. Instead, it is beneficial to add such resource tracking to existing languages, such as *Linear Haskell* for *Haskell* or *CN* for *C*.

These languages add resource tracking using *linear logic* to ensure that resources are used exactly once. A limitation of *linear logic* is that it is unable to represent functions where the function arguments and captured context share resources. O'Hearn and Pym's *Logic of Bunched Implications* (BI) addresses this by introducing a second type of implication. Prior research on type systems based on BI use a tree-shaped context called *bunches*, which complicates adding bunched implications on top of an existing simply typed language.

We present λ_* a resource-aware language with bunched functions and monadic heap operations. Programs in λ_* *erase* down to a resource-unaware simply typed language $\lambda_{\texttt{loc}}$ with untyped references. Using a denotational semantics of $\lambda_{\texttt{loc}}$, we show that the resource tracking of λ_* provides additional guarantees of *memory safety* and *memory leak freedom* for erased programs. A key component to permitting the erasure from λ_* to simply typed $\lambda_{\texttt{loc}}$ is the use of *assumption labeling* instead of the tree-shaped bunches used in prior type systems with bunched implications.

Contents

Chapter 1

Introduction

In modern concurrent software, it is essential that programs are safe and robust, lacking memory errors such as data races, use-after-free and memory leaks. The interest in languages such as Rust, which provide strong type systems that guarantee memory safety, indicate the importance of these guarantees. Rather than designing entirely new languages, which would require rewriting existing software, it would be beneficial to extend existing languages with such guarantees.

Separation Logic. The first approach would be to prove such guarantees for specific programs, namely, defining a program logic for the language and then using its proof rules to verify programs. One such program logic is separation logic [Reynolds 2002], an extension of Hoare logic that allows reasoning about programs with shared mutable state.

In separation logic, the language of assertions is extended with a separating conjunction $*$, which allows for reasoning about disjoint parts of the heap, and a *pointsto* proposition $l \mapsto v$ which gives exclusive ownership of the value l containing value v. As an example, consider the following program which frees a reference r_1 and then loads the value of reference r_2:

$$\begin{aligned} prog_1\, r_1\, r_2 \stackrel{\text{def}}{=} & \texttt{free}\, r_1; \\ & \texttt{load}\, r_2 \end{aligned}$$

In separation logic, a possible Hoare triple for $prog_1$ would be:

$$\{r_1 \mapsto v_1 * r_2 \mapsto v_2\}\, prog_1\, r_1\, r_2\, \{v.\, v = v_2 * r_2 \mapsto v_2\}$$

This triple states that given exclusive access to two different location r_1 and r_2 containing values v_1 and v_2 respectively, after executing $prog_1$ we get back a return value v equal to v_2 and exclusive access to only r_2 containing v_2. The separating conjunction $*$ is necessary to ensure that r_1 and r_2 are different locations and hence that $prog_1$ does not fail with a use-after-free error.

Separation logic consists of several inference rules which can be used to prove such Hoare-triple specifications of programs. The proof of such a triple also proves that the program is memory safe. For instance, VST-Floyd [Cao et al. 2018] is a separation logic for verifying C programs in the proof assistant Coq. Similarly, OCaml [Mével et al. 2020] and WebAssembly [Rao et al. 2023] have in Iris [Jung et al. 2018] a separation logic framework in Coq, that can be used to verify programs in these languages.

VeriFast [Jacobs et al. 2011] similarly defines a separation logic for C, but integrates the logic by adding the pre- and post-conditions as comment annotations to the code itself, and providing an automated tool to verify these annotations.

Type Refinement. Another approach would be to enrich the types in the existing language with information about ownership. This creates a second layer of types for programs, which is known as refinement typing [Zeilberger 2016]. The key property of refinement typing is that programs that are well-typed in the refined type system are also well-typed in the original type system. This allows for the gradual introduction of ownership tracking into existing programs by refining types of more parts of the program.

For instance, Bernardy et al. [2017] use refinement typing to add linear functions (functions which use their argument exactly once) to Haskell. Using this linearity, the authors simplify the interface of

mutable arrays in Haskell containing a linear function to freeze the same array, which requires linearity to ensure the array is not modified after freezing. Similar type refinements to add resource tracking are RichWasm [Fitzgibbons et al. 2024] for WebAssembly and CN [Pulte et al. 2023] and RefinedC [Sammler et al. 2021] for C. These systems use a concept of linear *capabilities* introduced by Morrisett et al. [2005]. These capabilities are types describing the ability to use a shared resource such as a pointer (similar to a *pointsto* in separation logic), and is separate from the type of the resource itself. The capability itself can only be used linearly, whereas the shared resource itself can be duplicated freely.

Linear Typing. These refinement type systems crucially depend on linear type theory, based on Girard's Linear Logic [Girard 1995], to ensure that resources such as references or capabilities are used exactly once. The linearity ensures both memory safety as each reference can only by one part of the program, and memory leak freedom as each reference must be used exactly once, and therefore freed at some point. As an example, consider the following program with monadic heap operations:

$$
\begin{aligned}
\mathit{prog}_1 : T\,Z \stackrel{\text{def}}{=}\ & \texttt{let!}\, r : \texttt{Ref}\, Z = \texttt{ref}\, 0\, \texttt{in} \\
& \texttt{let}\, f : T\,Z = (\texttt{let!}\, x : Z = \texttt{free}\, r\, \texttt{in}\, \texttt{return}\, x + 1)\, \texttt{in} \\
& \texttt{let}\, g : T\,(Z \times \texttt{Ref}\, \mathit{String}) = (\texttt{let!}\, (y, r') = \texttt{replace}\, r\, \text{"Hello"}\, \texttt{in}\, \texttt{return}\, (y + 1, r'))\, \texttt{in} \\
& \texttt{let}\, h = (\lambda p.\, \texttt{let}\, (x, y) = p\, \texttt{in}\, x)\, \texttt{in} \\
& \texttt{return}\, h\, (f, g)
\end{aligned}
$$

The program first allocates a new location r on the heap, containing the integer value 0 (of type Z). The $\texttt{let!}\, x = e_1\, \texttt{in}\, e_2$ binding is a monadic bind operation, which executes e_1 before continuing with e_2. After allocating r, the program defines two operations f and g, which are not immediately executed. The operation f first frees the reference r which returns the previously stored integer, followed by returning its successor. The operation g instead replaces the integer stored in r with a string "Hello", and returns the successor of the previously stored value and a new reference for the stored string. Finally, the program defines a function h which takes a pair and returns the first element, and applies it to the pair (f, g), resulting in the computation f being returned.

In the example, both f and g use the reference r, so in a linear type system, only one of g and f can be executed. To describe this, linear type systems have two types of conjunction, multiplicative $A * B$, which states that A and B describe different resources – and therefore can both be used independently – and additive $A \wedge B$, which states that A and B describe the same resources, and therefore only one can be used. In the example, the type of $g : T\,(Z * \texttt{Ref}\, \mathit{String})$ uses multiplicative conjunction, as the returned integer and pointer can both be used independently. On the other hand, the type of $(f, g) : T\,Z \wedge T\,(Z * \texttt{Ref}\, \mathit{String})$, and hence the input type of h, uses the additive conjunction as both f and g capture the same reference r, so at most one of them can be executed. In linear type systems, all arguments and captured variables must be used exactly once, which is denoted by the multiplicative function type $A \mathrel{-\!\!*} B$ (also commonly denoted $A \multimap B$). For instance, the type of h would be $(T\,Z \wedge T\,(Z * \texttt{Ref}\, \mathit{String})) \mathrel{-\!\!*} T\,Z$.

Now consider a slight modification of the program, where the last two lines are replaced by:

$$
\begin{aligned}
& \texttt{let}\, h' = (\lambda x.\, \lambda y.x)\, \texttt{in} \\
& \texttt{return}\, h'\, f\, g
\end{aligned}
$$

The behavior of this adjusted program is the same as the initial program, where the computation f is returned. However, as arguments of functions in linear type systems must be used exactly once, the function h', and therefore also the program, are no longer typeable in linear type systems, as h' does not use the argument y. Additionally, f and g both capture the same reference r and therefore cannot be passed as two separate arguments to a linear function.

To give a type to h', we move to the logic of Bunched Implications (BI) [O'Hearn and Pym 1999], an extension of linear logic which adds not only the additive conjunction, but also an additive implication $A \Rightarrow B$. The additive implication can be interpreted as functions where the captured variables and arguments share the same resources. In this logic, the function h' has the type $T\,Z \Rightarrow T\,(Z * \texttt{Ref}\, \mathit{String}) \Rightarrow T\,Z$, as the second argument y of h' describes the same resources as the captured variable x. More generally, in the logic of BI the formulas $P \wedge Q \Rightarrow R$ and $P \Rightarrow Q \Rightarrow R$ are equivalent.

There has been limited prior research into creating type systems with bunched implications. First, there is the $\alpha\lambda$ calculus by O'Hearn [1999], introduced as a direct Curry-Howard correspondence of the

logic of Bunched Implications. Like the logic, the typing rules in the $\alpha\lambda$-calculus use a concept called 'bunches', a tree-like rather than list-like context, to represent the sharing and separation of resources. The calculus was extended with polymorphism by Collinson et al. [2008] and dependent types by Ishtiaq and Pym [1999]. Additionally, Berdine and O'Hearn [2006] and Collinson and Pym [2006] both define type systems for BI with heap operations, both using a continuation passing style for heap operations, and limit the heap to storing only integer and unit types. Finally, there is the recent πBI calculus by Frumin et al. [2022], a process calculus where message-passing processes are typed with bunched implications.

Crucially, none of these prior type systems with bunched implications are defined as refinement type systems of languages without resource tracking. In this thesis, we define such a language λ_* with bunched implications and monadic heap operations in the style of Moggi [1989] that is a refinement type system of our simply-typed $\lambda_{\texttt{loc}}$ language. The thesis consists of two main parts.

Part 1: Simply typed language $\lambda_{\texttt{loc}}$. The first is defining the simply-typed language $\lambda_{\texttt{loc}}$, For the language, we define the syntax and typing rules. We give an interpretation to programs by defining a denotational semantics on well-typed programs, in which the monadic type is interpreted as partial functions from heaps to return values and heaps.

We also define an equational theory of $\beta\eta$-equivalences to prove that certain programs are equivalent. The β-equivalences state that abstracting a term with a variable and then applying it to an argument is equivalent to substitution, whereas the η-equivalences state that abstracting over a variable and then immediately applying the variable is the same as not abstracting. In the case of functions, the β- and η-equivalences are:

$$(\lambda x : A.\, e_1)\, e_2 \equiv_\beta [x \mapsto e_2]e_1 \qquad (\lambda x : A.\, e\, x) \equiv_\eta e$$

Here $[x \mapsto e_2]e_1$ corresponds to substituting all occurrences of x in e_1 with e_2. An alternative for defining β-equivalences is to define β reductions, which are transitions between expressions, and can be interpreted as computation steps [Pierce 2002]. For instance, the β-reduction for functions is:

$$(\lambda x : A.\, e_1)\, e_2 \to_\beta [x \mapsto e_2]e_1$$

We opt for equivalences rather than the reduction for $\lambda_{\texttt{loc}}$ as equivalences are generally defined only on well-typed programs, whereas reductions are generally defined on all expressions. As the last step for $\lambda_{\texttt{loc}}$, we prove that equivalent programs have the same denotational semantics.

Part 2: Bunched functions language λ_*. In the second part we extend $\lambda_{\texttt{loc}}$ with bunched implications to create λ_*. The types and expressions of $\lambda_{\texttt{loc}}$ are extended to both additive variants which describe shared resources, and multiplicative variants which describe distinct resources.

The typing rules of λ_* use *labels* to track which expressions and variables contain which resources. Such labeling approaches have previously been used in sequent calculi, such as by Hóu et al. [2015] for Boolean BI – a classical variant of BI – but to the author's knowledge have not previously been used in type systems for BI. Unlike *bunches*, the labeling approach maintains a list-shaped context, meaning that the tracking of resources in λ_* can be erased, resulting in the simply typed $\lambda_{\texttt{loc}}$. Despite removing the tracking when erasing to $\lambda_{\texttt{loc}}$, we can prove using an additional logical predicate defined on types in λ_* that well-typed programs in λ_* have an interpretation in the denotational semantics of $\lambda_{\texttt{loc}}$ that is *safe*. Namely, the interpretation does not encounter memory errors such as use-after-free or memory leaks.

In addition to the β-equivalences for λ_*, we define a β-reduction for the pure (non heap-changing) parts of the language. For this β-reduction we prove the *preservation* property, which states that if e is well-typed in λ_* and $e \to_\beta e'$ then e' is well-typed and additionally $e \equiv_\beta e'$.

1.1 Outline

We start with a brief introduction to linear logic and bunched implications (Chapter 2). After this, we define the simply typed $\lambda_{\texttt{loc}}$ without resource tracking (Chapter 3), as a base language on which to add resource tracking. We interpret this language using a denotational semantics (§ 3.4), where the monadic type is interpreted as a partial function from heaps to heaps, to account for possibly unsafe heap operations. We then define the full resource-tracked λ_* language based on Bunched Implications (Chapter 4) using labeled assumptions to track resources. The typing rules consist not only of logical

rules, corresponding to the form of expressions, but also structural rules to reason about labels. We show that well-typed programs in λ_* erase to well-typed programs in $\lambda_{\texttt{loc}}$ (4.4). By defining a logical predicate on the interpretation of λ_* types to state which values in the erased system are valid for which resources, we prove that well-typed programs are safe, and that well-typed programs that return values with no resources are also memory-leak free (§ 4.4). Finally, we show that the λ_* type system enjoys the *preservation* property for β-reduction in the pure (non heap-changing) parts of the language. Finally, we conclude with discussions on related work (Chapter 6) and future work (Chapter 7).

1.2 Contributions

We present λ_*, a typed language with bunched implications and monadic heap operations that has an explicit erasure to our simply typed language $\lambda_{\texttt{loc}}$ without resource tracking. The complete list of contributions are as follows:

- We define λ_*, a typed language with bunched functions and monadic heap operation. The type system of λ_* uses *labeled assumptions* used in sequent calculi such as by Hóu et al. [2015], rather than *bunched contexts* used in prior work on BI type systems for λ-calculi [Collinson et al. 2008; O'Hearn 1999].
- We present a simply typed language $\lambda_{\texttt{loc}}$ with untyped references and possibly unsafe monadic heap operations with a denotational semantics. We show that λ_* is a refinement type system of $\lambda_{\texttt{loc}}$ by defining an erasure from types, expressions and typing derivations from λ_* to $\lambda_{\texttt{loc}}$. As the latter does not track resources, this erasure removes all worlds and constraints.
- Using a logical predicate, we show that well-typed programs in λ_* erase to programs in $\lambda_{\texttt{loc}}$ whose denotational semantics are safe (*i.e.* heap operations do not fail), and memory leak free (*i.e.* programs that return non-resource carrying values free all allocated references).
- We define β-reduction for the pure (non heap changing) parts of λ_*, for which we prove the *preservation* property. For the proof, we introduce a concept of *context strengthening* to combine multiple structural rules and create a new *layered type system* to restrict where structural rules can be applied.

Acknowledgments

I would like to thank Neel Krishnaswami and Robbert Krebbers for their supervision of my project, and the University of Cambridge for hosting me during the research phase of my thesis. Further, I would like to thank my second reader Herman Geuvers for their time.

Chapter 2

Background

In this section, we briefly introduce the logics related to describing shared resources in type systems. In § 2.1 we discuss linear logic, a variant of logic in which assumptions must be used exactly once. In § 2.2 we further introduce the logic of bunched implications, an extension of linear logic with resource-sharing implication. In § 2.3 we discuss the propositional variant of separation logic, a program logic to reason about shared data.

2.1 Linear Logic

Whereas intuitionistic logic allows assumptions to be dropped or used multiple times, linear logic introduced by Girard [1995], requires that each assumption is used exactly once. This restriction is useful for reasoning about resources such as references, as it ensures that references are distinct, and that all references have to be freed.

The rules of linear logic with only linear implication are as follows:

$$\frac{}{A \vdash A}\,\textsc{assum} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,\multimap\!\text{I} \qquad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\,\multimap\!\text{E}$$

The assumption rule ASSUM of linear logic is stronger than that of intuitionistic logic, as it requires that the context contains only the conclusion A. The I$\multimap$ rule is the same as that for intuitionistic logic, but the application rule is different. Unlike in intuitionistic logic where both the implication and premise of the implication are derived from the same context, the application rule of linear logic requires that the context is split into two disjoint contexts, one for the implication and one for the argument. This ensures that each assumption is used exactly once. An example typing derivation in linear logic is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \multimap A \vdash A \multimap A}\,\textsc{assum} \qquad \dfrac{}{A \vdash A}\,\textsc{assum}}{A, A \multimap A \vdash A}\,\multimap\!\text{E}}{A \vdash (A \multimap A) \multimap A}\,\multimap\!\text{I}}{\vdash A \multimap (A \multimap A) \multimap A}\,\multimap\!\text{I}$$

In this derivation, the E$\multimap$ rule specifically splits the context $\Sigma, \Delta = A, A \multimap A$ into $\Sigma = A \multimap A$ and $\Delta = A$. As each assumption has to be used exactly once, there is no derivation of $\vdash A \multimap (A \multimap A) \multimap A$ which applies the first premise and ignores the second.

This linear logic can be extended with two types of conjunction describing both shared $\wedge$ and disjoint $\otimes$ propositions. They have the following proof rules:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_i}\,(i = 1, 2) \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma \vdash A \otimes B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C}$$

The first conjunction $A \wedge B$ has the same rules as for intuitionistic logic. However, as the context is linear, only A or B can be used, not both. As such, there is no proof of $A \wedge B \multimap C \vdash A \multimap B \multimap C$,

as the former gives access to either A and B to prove C, whereas the latter gives access to both A and B to prove C. For this reason, there is also the multiplicative conjunction $A \otimes B$, in which both sides are derived from different parts of the context. Indeed, eliminating $A \otimes B$ gives access to both A and B in order to prove C. Hence, there is a correspondence between the multiplicative $\multimap$ and $\otimes$ of the form $A \otimes B \multimap C \dashv\vdash A \multimap B \multimap C$, but no such correspondence exists between the additive conjunction $\wedge$ and another connective.

As assumptions in linear logic must be used exactly once, they lend themselves to type systems with resources such as references, to ensure that for instance a reference cannot be used after it has been freed. Such a type system is employed by the L3 language [Morrisett et al. 2005], which uses linear types to describe the capability to use a reference, ensuring that the type stored at the reference and the type of the capability stay in sync.

2.2 Bunched Implications

The logic of Bunched Implications (BI) [O'Hearn and Pym 1999] extends linear logic with such an extra connective $\Rightarrow$, called the additive implication, which represents an implication where the assumption and conclusion share the same context. The additive conjunction and implication share the same correspondence as the multiplicative conjunction and implication:

$$A \wedge B \Rightarrow C \dashv\vdash A \Rightarrow B \Rightarrow C$$

In the proof rules of bunched implications, contexts are not merely lists of assumptions, but form a tree structure called bunches. To illustrate, the proof rules for the two implications are as follows:

$$\frac{\Gamma; A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \mathrel{-\!\!*} B} \qquad \frac{\Gamma \vdash A \mathrel{-\!\!*} B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

The proof rules for BI have two operators for combining contexts, the first is the additive ';', which can be interpreted as intuitionistic contexts, as they allow assumptions to be ignored (weakening) or used multiple times (contraction). The second operator is the multiplicative ',', and can be interpreted as linear contexts, as they require that both sides are used exactly once. In BI, the multiplicative conjunction and implication are denoted by $*$ and $\mathrel{-\!\!*}$ respectively, instead of the $\otimes$ and $\multimap$ used in linear logic. The products $*$ and $\otimes$ have units 1_{m} and 1_{a} respectively.

As the above rules can be combined, the contexts in BI form not a linear context, but a tree structure called a bunch. These bunches are defined by the following grammar:

$$\Gamma ::= \{\}_{\mathsf{m}} \mid \{\}_{\mathsf{a}} \mid A \mid \Gamma, \Gamma \mid \Gamma; \Gamma$$

As bunches describe both intuitionistic and linear contexts, there are also two kinds of empty contexts, the intuitionistic empty context $\{\}_{\mathsf{a}}$ which appears by weakening a single assumption and is the unit of ';', and $\{\}_{\mathsf{m}}$, the unit of ',', which describes the empty context as in linear logic.

2.3 Propositional Separation Logic

A well known example of a BI-logic is (propositional) separation logic [Calcagno et al. 2005; Reynolds 2002], which extends the assumption and conclusion propositions in Hoare logic to reason about programs with heaps. In propositional separation logic, intuitionistic propositional logic is extended with a pointsto connective $l \mapsto v$ which corresponds to heap location l containing the value v, and a separating conjunction $P * Q$, which corresponds to the propositions P and Q holding on disjoint parts of the heap. As such, formulas in separation logic are defined as:

$$P ::= \top \mid P \wedge Q \mid P \Rightarrow Q \mid \mathtt{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q \mid l \mapsto v$$

Here $\top$ corresponds to the additive unit in BI and $\top$ in intuitionistic logic. It therefore holds in any partial heap. The $P \wedge Q$ connective corresponds to $\wedge$ in intuitionistic logic, and therefore holds for

heaps in which both P and Q hold. The $P \Rightarrow Q$ connective corresponds to the implication connective in intuitionistic logic, and therefore holds for the heaps in which if P holds, then Q holds.

Separation logic also provides multiplicative connectives, notably `emp` which only holds in the empty heap (corresponding to 1_{m} in BI), and $P * Q$ which holds for heaps which can be split into two disjoint heaps in which P and Q hold respectively. The proposition $l \mapsto v$ holds only for the partial heap only containing the location l with value v. Hence, in the separation logic proposition $l \mapsto v * l' \mapsto v'$, the locations l and l' must be distinct.

The separating conjunction $P * Q$ corresponds to the multiplicative conjunction $*$ in BI, and holds for the heaps which can be split into two disjoint heaps in which P and Q hold respectively. Finally, the separating implication $P \mathrel{-\!\!*} Q$ corresponds to the multiplicative implication $-\!\!\circ$ in BI, and holds for the heaps which when combined with a disjoint heap in which P holds, holds for Q.

The heaps in which a given separation logic proposition hold are specified by the resource semantics. For separation logic, these resource semantics [Pym et al. 2019] are given by a heap satisfiability relation $h \models P$, stating that heap h satisfies separation logic proposition P:

$$
\begin{aligned}
h &\models \top \stackrel{\text{def}}{=} \text{True} \\
h &\models P \wedge Q \stackrel{\text{def}}{=} h \models P \wedge h \models Q \\
h &\models P \Rightarrow Q \stackrel{\text{def}}{=} h \models P \implies h \models Q \\
h &\models \mathtt{emp} \stackrel{\text{def}}{=} \mathtt{dom}\, h = \emptyset \\
h &\models l \mapsto v \stackrel{\text{def}}{=} \mathtt{dom}\, h = \{l\} \wedge h\, l = v \\
h &\models P * Q \stackrel{\text{def}}{=} \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge h_1 \models P \wedge h_2 \models Q \\
h &\models P \mathrel{-\!\!*} Q \stackrel{\text{def}}{=} \forall h_2.\, h \# h_1 \wedge h_1 \models P \implies h \uplus h_2 \models Q
\end{aligned}
$$

The notation $h_1 \# h_2$ states that h_1 and h_2 are disjoint and is defined as $\mathtt{dom}\, h_1 \cap \mathtt{dom}\, h_2 = \emptyset$. The notation $h_1 \uplus h_2$ is the disjoint union of two heaps, and is only defined when $h_1 \# h_2$. We will return to this relation in § 4.4 to describe when values of types are valid for a given heap.

In separation logic, these propositions are used as a Hoare logic, consisting of Hoare triples of the form $\{P\}\, e\, \{Q\}$ [Reynolds 2002], where P and Q are separation logic formulas and e is a program (which may alter the heap). The triple $\{P\}\, e\, \{Q\}$ states that if P holds for a part h of a heap, denoted $h \models P$, then after e executes, the updated part h' satisfies Q, denoted $h' \models Q$, and the part of the heap not specified by P remains unchanged. Generally, this execution is defined as an operational semantics, but the specific details are not relevant to this work.

An important property of Hoare triples in separation logic is that of *framing*, which states that resources not specified in the pre- and post-conditions are unaffected The rule is defined as:

$$
\frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}\ \textsc{frame}
$$

According to the frame rule, any resources R disjoint from P are unaffected by the execution of e, and therefore still hold in the post-condition.

Rather than defining Hoare triples directly, they may instead be specified in terms of the *weakest precondition* of a program e with respect to a post-condition Q, denoted $\mathtt{wp}\, e\, \{Q\}$. The weakest precondition $\mathtt{wp}\, e\, \{Q\}$ is the weakest separation logic formula P such that $\{P\}\, e\, \{Q\}$ holds. Hence the definition of Hoare triples in terms of weakest preconditions is: $\{P\}\, e\, \{Q\} = P \mathrel{-\!\!*} \mathtt{wp}\, e\, \{Q\}$ [1]. Whether a given heap satisfies such a weakest precondition can be informally defined as:

$$
h \models \mathtt{wp}\, e\, \{Q\} \stackrel{\text{def}}{=} \forall h_f \# h.\ \textit{Executing } e \textit{ in heap } h \uplus h_f \textit{ results in a heap } h' \uplus h_f \textit{ s.t. } h' \models Q
$$

The universal quantification of the frame heap h_f ensures that the frame rule holds for both the weakest precondition and corresponding Hoare triple. A formal definition of weakest precondition is generally given in terms of an operational semantics, which is out of scope for this thesis. Even so, the informal description above is useful, as it corresponds to the heap satisfiability of the monad type $T\, A$ in § 4.4.

[1] To ensure that Hoare triples can be used multiple times, the definition in terms of `wp` also requires that no (non-persistent) resources are captured by $\mathrel{-\!\!*}$. In Iris [Jung et al. 2018, §6] this is done using a persistence modality.

Chapter 3

STLC with products and locations

In this section, we introduce $\lambda_{\texttt{loc}}$, a simply typed λ-calculus with locations and monadic heap operations. The calculus does not track any resources, and serves as the base language upon which the resource aware λ_* is built (Chapter 4). In § 3.1 we present the syntax of the language, consisting of the simple types and expressions. In § 3.2 we define the type system for $\lambda_{\texttt{loc}}$ as an extension of those for the Simply Typed Lambda Calculus (STLC). Next, we define β- and η equivalences in the style of Gunter [1993] (§ 3.3) using the simultaneous substitution.

In § 3.4 we give a denotational semantics of $\lambda_{\texttt{loc}}$, in which heap operations are modeled as *partial functions* which are undefined when a heap operation fails. In the style of Gunter [1993], we conclude this section with a proof of soundness of the denotational semantics (Theorem 3.4.13), ensuring that β- and η-equivalent terms have the same interpretation in the denotational semantics. In the next section (Chapter 4), we extend $\lambda_{\texttt{loc}}$ with a more complex type system (λ_*) to track resources, and show that programs with heap operations typeable in λ_* do not get *stuck* (Corollary 4.4.27).

3.1 Syntax

In the base language, we consider the simply typed lambda calculus (STLC) as in for instance Angiuli and Gratzer [2024], and extend it with locations and monadic heap operations.

Definition 3.1.1 (Types)**:**

$$A \in \texttt{Type} ::= 1 \mid A \times A \mid A \to A \mid \texttt{loc} \mid T\,A$$

The function type constructor $A \to B$ corresponds to the function type in STLC. We extend this with the unit type 1 and product types of the form $A \times B$.

To support heaps, we add the location type `loc`, representing a single location on the heap. The `loc` type does not carry any information about the type of the value stored on the heap (similar to a `void*` pointer in C [Free Software Foundation 2024]). We also introduce the monadic type constructor $T\,A$ in the style of Moggi [1989], representing a computation that performs heap operations and returns a value of type A. Such monadic types are common in functional programming languages, such as the `IO` monad in Haskell [Jones 2003]. Consequently, the language supports 2 kinds of functions, those without any heap operations (*i.e.* pure functions) of type $A \to B$, and those that perform heap operations (*i.e.* stateful functions) of type $A \to T\,B$.

Definition 3.1.2 (Expressions)**:** The expression formation rules are defined as follows:

$$\begin{aligned} e \in \texttt{Expr} ::= {} & () \mid (e_1, e_2) \mid \texttt{let}\,(x, y) = e_1\,\texttt{in}\,e_2 \mid \lambda x : A.\, e \mid e_1\,e_2 && (\textit{pure operations}) \\ \mid {} & \texttt{return}\,e \mid \texttt{let!}\,x = e_1\,\texttt{in}\,e_2 && (\textit{monadic operations}) \\ \mid {} & \texttt{ref}_A\,e \mid \texttt{replace}_{A,B}\,e_1\,e_2 \mid \texttt{free}_A\,e && (\textit{heap operation}) \end{aligned}$$

Where $x, y \in \texttt{Var}$ are variables in the countably infinite set of variables `Var` and $A, B \in \texttt{Type}$. Additionally, $\texttt{let}\,x = e_1\,\texttt{in}\,e_2$ is defined as $(\lambda x : A.\, e_2)\,e_1$ (when e_1 has type A).

Pure Operations. The expression former () creates the single element of the unit type 1. Product types $A \times B$ can be introduced with (e_1, e_2). They are eliminated with $\texttt{let}\,(x, y) = e_1\,\texttt{in}\,e_2$, rather than projection functions π_1, π_2. In later systems (Chapter 4) and other substructural type systems, such as linear type theory [Wadler 1993] in which variables have to be used exactly once, the projection functions π_1, π_2 do not exist, as they ignore the other element of the pair. For consistency, we therefore opt to use `let`-style elimination in this system. The $\lambda x : A.\, e$ and $e_1\, e_2$ expression formers correspond to λ-abstraction and application as in STLC respectively.

Monadic Operations. To support the monadic type $T\,A$, the expression formers include $\texttt{return}\,e$ corresponding to the monadic return operation (`return` in Haskell), and $\texttt{let!}\,x : A = e_1\,\texttt{in}\,e_2$ corresponding to the monadic bind operation ($x \leftarrow e_1;\ e_2$ in Haskell `do` notation).

Remark: The `let` bindings *without* `!` corresponds to assigning a variable, whereas the `let!` binding with `!` corresponds to the monadic bind, and first 'unwraps' the value of the monadic type.

Heap Operations. The `ref`, `replace` and `free` operations modify the heap. The $\texttt{ref}_A\,e$ operation allocates a new location on the heap containing the value e of type A. The $\texttt{replace}_{A,B}\,e_1\,e_2$ operation replaces the value of type A at location e_1 with the expression e_2 of type B, returning the old value. The $\texttt{free}_A\,e$ operation deallocates the location e, returning the value of type A previously stored at that location. We opt for a `replace` operation (a type-changing variant of `replace` in Rust [*replace in std::mem - Rust* 2024]), rather than separate load and store operations to match the substructural type systems later in this thesis, similar to eliminating pairs with `let`. As the `replace` operation can change the type of a location, and variables in the language can be used multiple times, we use the `loc` type, rather than a reference type like $\texttt{Ref}\,A$.

Remark: The `ref`, `replace` and `free` are all indexed by the type of the value stored at the location. When the type of the value in the heap is not the same as the type specified by the operation, the operation gets *stuck*. This ensures that for any program that does not get *stuck*, the values stored at locations were consistent with the types specified by the heap operations.

Remark: We assume the Barendregt Convention [Barendregt 1985] for α-equivalence, meaning that all variables that occur in any definition or proof are considered to be unique. This avoids having to rename variables in definitions and proofs, improving the presentation of the system.

Definition 3.1.3: In a lambda abstraction of the form $\lambda x : A.\, e$ or `let`-binding such as $\texttt{let}\,x = e'\,\texttt{in}\,e$, we say that the variable x is bound in e. The variables that occur in an expression e that are not bound by such a λ-abstraction or `let` or `let!` binding are called free in e. The set of free variables in an expression e is denoted $\texttt{fv}\,e$.

The two common styles of λ terms are Church style and Curry style [Hindley and Seldin 2008], where the former fixes the type of the argument ($\lambda x : A.\, e$), whereas the latter does not ($\lambda x.\, e$). We apply the Church style to our system, as it results in unique typing derivations (Theorem 3.2.5) which simplifies the proof of soundness of the denotational semantics with respect to η-equivalences (§ 3.3). Though we use Curry style, we omit type annotations in this thesis for brevity when they can be inferred from the context, or explicitly add type annotations not specified in the grammar of expressions for clarity.

As an example program in $\lambda_{\texttt{loc}}$, consider the following which swaps the two elements of a pair:

Example 3.1.4 (Swap Pair)**:**

$$\texttt{swap\_pair} \stackrel{\texttt{def}}{=} \lambda p : A \times B.\, \texttt{let}\,(x : A, y : B) = p\,\texttt{in}\,(y, x)$$

This program consists of a λ abstraction binding a variable p of type $A \times B$, followed by deconstructing the pair into x of type A and y of type B, and finally constructing a pair with the elements swapped. For clarity, we adjust the notation of such programs slightly in the remainder of this thesis, by moving the λ-abstractions to the left of the definition, adding an output type annotation, and removing type annotations which are clear from context. In this notation, the `swap_pair` program is written as:

$$\texttt{swap\_pair}\,(p : A \times B) : B \times A \stackrel{\texttt{def}}{=} \texttt{let}\,(x, y) = p\,\texttt{in}\,(y, x)$$

As $\lambda_{\texttt{loc}}$ supports weakening (leaving variables unused), the projection functions π_1, π_2 can be defined, and the `swap_pair` function can also be written as:

Example 3.1.5 (Swap Pair Projections)**:**

$$\pi_1\,(z : A \times B) : A \stackrel{\text{def}}{=} \texttt{let}\,(x, y) = z\,\texttt{in}\,x$$
$$\pi_2\,(z : A \times B) : B \stackrel{\text{def}}{=} \texttt{let}\,(x, y) = z\,\texttt{in}\,y$$
$$\texttt{swap\_pair\_2}\,(z : A \times B) : B \times A \stackrel{\text{def}}{=} (\pi_2\,z, \pi_1\,z)$$

Here the π_1 and π_2 functions project the first and second elements of the pair respectively, and the `swap_pair_2` function takes a pair and constructs a new pair with the elements swapped using the projection functions.

Heap Operations. The heap operations are monadic, returning values of type $T\,A$, and can be sequenced using the bind operation. For instance, the following program swaps the values at two locations:

Example 3.1.6 (Swap Location)**:**

$$\begin{aligned}\texttt{swap\_loc\_AxB}\,(p : \texttt{loc} \times \texttt{loc}) : T\,1 \stackrel{\text{def}}{=}\ & \texttt{let}\,(l_1, l_2) = p\,\texttt{in}\\ & \texttt{let!}\,x = \texttt{replace}_{A,1}\,l_1\,()\,\texttt{in}\\ & \texttt{let!}\,y = \texttt{replace}_{B,A}\,l_2\,x\,\texttt{in}\\ & \texttt{replace}_{1,B}\,l_1\,y\end{aligned}$$

In this program, the input p describes a pair of locations which is deconstructed into locations l_1 and l_2. The program then performs 3 `replace` operations: 1) it replaces the value of type A stored at l_1 with a temporary unit value, 2) it replaces the value of type B stored at l_2 with the value previously stored at l_1, and 3) it replaces the unit value currently stored at l_1 with the value previously stored at l_2. Due to the added monadic type constructor $T\,A$, the heap operations describe heap computations, which have to be sequenced (the reader may interpret this as executed) by the `let!` (bind) operation. A heap computation $T\,A$ can be used multiple times, similar to Haskell's `IO` monad.

Modeling Recursion. Despite the lack of a recursion operation in the base language, recursion can still be modeled in the untyped expression language using *Landin's Knot* [Landin 1964].

Example 3.1.7 (Landin's Knot)**:**

$$\begin{aligned}&\texttt{landin}\,(f : (A \to T\,B) \to A \to T\,B) : A \to T\,B \stackrel{\text{def}}{=} \lambda x : A.\\ &\quad \texttt{let!}\,l : \texttt{loc} = \texttt{ref}_1\,()\,\texttt{in}\\ &\quad \texttt{let}\,rec : A \to T\,B = f\,(\lambda x : A.\,\texttt{let!}\,rec = \texttt{load}\,l\,\texttt{in}\,rec\,x)\,\texttt{in}\\ &\quad \texttt{let!}\,v : 1 = \texttt{replace}_{1,A\to T\,B}\,l\,rec\,\texttt{in}\\ &\quad rec\,x\\ &\texttt{load}(l : \texttt{loc}) : T\,(A \to T\,B) \stackrel{\text{def}}{=}\\ &\quad \texttt{let!}\,f : A \to T\,B = \texttt{replace}_{A\to T\,B,1}\,l\,()\,\texttt{in}\\ &\quad \texttt{let!}\,v = \texttt{replace}_{1,A\to T\,B}\,l\,f\,\texttt{in}\\ &\quad \texttt{return}\,f\end{aligned}$$

In this example, the `load` program takes a location, replaces the value of type $A \to T\,B$ stored at the location l with a unit assigning f to be the old value, followed by replacing the unit with the originally stored value f, and then returning f.

The `landin` program takes a function $f : (A \to T\,B) \to (A \to T\,B)$ and supplies as argument a function which recursively calls f applied to itself. To do this, it first allocates a location l in which to store the recursive function as a location containing a unit value. In `ref`, it applies f to a function that attempts to load a value of type $A \to T\,B$ from l, and applies that function to the argument. The rec function itself is then stored at the location l, meaning that calls to the argument of f recursively call the function f. As the program requires monadic heap operations to set up the recursive function rec, the program first takes an input x (of type A) and executes rec on that specific input at the end.

To simplify the denotational model of the system (§ 3.4) we do not consider recursive functions. As such, we limit the values that can be stored on the heap through the type system, ensuring that programs such as `landin` are not typeable. Specifically, we do not allow values of types containing the monadic type constructor $T\,A$ to be placed on the heap. In the case of `landin`, this means that the function $rec : A \to T\,A$ cannot be placed on the heap. A function of type $A \to B$ would not be sufficient to create Landin's knot, as it could not use heap operations to load itself from the heap.

Stuck Programs. Programs accessing the heap can get *stuck* when the shape of the heap does not match with the next operation, for instance when the type of the value currently stored at a location does not match the type specified by a heap operation, or when a location is used after it has been freed. As a slightly more complex example, consider the following two programs `safe` and `unsafe`, which both free a pair of locations:

Example 3.1.8 (Double Free)**:**

$$\texttt{duplicate}\,(l : \texttt{loc}) : \texttt{loc} \times \texttt{loc} \stackrel{\text{def}}{=} (l, l) \qquad \texttt{free\_pair}\,(p : \texttt{loc} \times \texttt{loc}) : T\,(1 \times 1) \stackrel{\text{def}}{=} \begin{array}{l}\texttt{let}\,(l_1, l_2) = p\,\texttt{in}\\ \texttt{let!}\,x = \texttt{free}_1\,l_1\,\texttt{in}\\ \texttt{let!}\,y = \texttt{free}_1\,l_2\,\texttt{in}\\ \texttt{return}\,(x, y)\end{array}$$

$$\texttt{safe}\,() \stackrel{\text{def}}{=} \begin{array}{l}\texttt{let!}\,l_1 = \texttt{ref}_1\,()\,\texttt{in}\\ \texttt{let!}\,l_2 = \texttt{ref}_1\,()\,\texttt{in}\\ \texttt{let!}\,r = \texttt{free\_pair}\,(l_1, l_2)\,\texttt{in}\\ \texttt{return}\,()\end{array} \qquad \texttt{unsafe}\,() \stackrel{\text{def}}{=} \begin{array}{l}\texttt{let!}\,l = \texttt{ref}_1\,()\,\texttt{in}\\ \texttt{let}\,l_{pair} = \texttt{duplicate}\,l\,\texttt{in}\\ \texttt{let!}\,r = \texttt{free\_pair}\,l_{pair}\,\texttt{in}\\ \texttt{return}\,()\end{array}$$

In this example, the `safe` operation allocates two different new locations l_1 and l_2, followed by using `free_pair` to free both locations. On the other hand, the `unsafe` operation allocates a single new location l, which it duplicates into a pair using `duplicate`, and then frees both locations using `free_pair`. As the `unsafe` operation attempts to free the same location twice, it gets stuck on the second free operation, as the location is no longer allocated. The `safe` operation on the other hand does not get stuck, as it frees two different locations.

In the example, we extract the `duplicate` and `free_pair` functions from the `safe` and `unsafe` programs to highlight that types in this system do not prevent programs from getting stuck. Specifically, the `duplicate` function uses the location variable twice (corresponding to contraction), and the type of `free_pair` does not enforce that both locations are distinct. The types will be compared to those of the resource tracking λ_* in Chapter 4.

3.2 Type System

To ensure that programs are not recursive, the type system creates a distinction between the types that do not contain heap operations (pure types), and those that do (stateful types). By ensuring only pure types can be stored on the heap, we prevent recursion through Landin's Knot. This distinction is captured by a *kind* rule of types:

Definition 3.2.1 (Kinds and well-kinded types)**:** For the kinds $k ::= \texttt{Pure} \mid \texttt{Stateful}$, the well-kinded types are defined as follows:

$$\boxed{\vdash A : k}$$

$$\vdash 1 : k \qquad \vdash \texttt{loc} : k \qquad \frac{\vdash A : k \qquad \vdash B : k}{\vdash A \times B : k} \qquad \frac{\vdash A : k \qquad \vdash B : k}{\vdash A \to B : k} \qquad \frac{\vdash A : k}{\vdash T\,A : \texttt{Stateful}}$$

In this definition, the types $A \in \texttt{Type}$ for which $\vdash A : \texttt{Pure}$ is derivable are the pure types, which do not contain the heap constructor $T\,A$ and therefore can be placed on the heap. The location type `loc` describes only the location on a heap (similar to how pointers in assembly are integers), they can be

placed on the heap. Furthermore, as the only rule fixing a kind is for $T\,A$, all pure types can also be used as stateful types.

Lemma 3.2.2: For type $A \in \texttt{Type}$, if $\vdash A : \texttt{Pure}$ then $\vdash A : \texttt{Stateful}$.

Proof. A straightforward induction on the derivation of $\vdash A : \texttt{Pure}$. □

The typing contexts and typing rules of $\lambda_{\texttt{loc}}$ are defined as follows:

Definition 3.2.3 (Context)**:**

$$\Gamma \in \texttt{Ctx} ::= \cdot \mid \Gamma, x : A$$

Where $x \in \texttt{Var}$ and $A \in \texttt{Type}$.

Definition 3.2.4 (Typing Rules)**:** The typing judgment $\Gamma \vdash e : A$ denotes that an expression $e \in \texttt{Expr}$ has type $A \in \texttt{Type}$ in typing context $\Gamma \in \texttt{Ctx}$. The judgment is derived by the following rules:

$$\text{STLC-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{STLC-FUN-I}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \to B} \qquad \text{STLC-FUN-E}\ \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \qquad \text{STLC-UNIT-I}\ \frac{}{\Gamma \vdash () : 1}$$

$$\text{STLC-PAIR-I}\ \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \qquad \text{STLC-PAIR-E}\ \frac{\Gamma \vdash e_1 : A \times B \qquad \Gamma, x : A, y : B \vdash e_2 : C}{\Gamma \vdash \texttt{let}\,(x, y) = e_1\,\texttt{in}\,e_2 : C} \qquad \text{STLC-RET}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \texttt{return}\,e : T\,A}$$

$$\text{STLC-BIND}\ \frac{\Gamma \vdash e_1 : T\,A \qquad \Gamma, x : A \vdash e_2 : T\,B}{\Gamma \vdash \texttt{let!}\,x : A = e_1\,\texttt{in}\,e_2 : T\,B} \qquad \text{STLC-REF}\ \frac{\Gamma \vdash e : A \qquad \vdash A : \texttt{Pure}}{\Gamma \vdash \texttt{ref}_A\,e : T\,\texttt{loc}} \qquad \text{STLC-FREE}\ \frac{\Gamma \vdash e : \texttt{loc} \qquad \vdash A : \texttt{Pure}}{\Gamma \vdash \texttt{free}_A\,e : T\,A}$$

$$\text{STLC-REPLACE}\ \frac{\Gamma \vdash e_1 : \texttt{loc} \qquad \Gamma \vdash e_2 : B \qquad \vdash A : \texttt{Pure} \qquad \vdash B : \texttt{Pure}}{\Gamma \vdash \texttt{replace}_{A,B}\,e_1\,e_2 : T\,A}$$

The STLC-VAR, STLC-FUN-I and STLC-FUN-E rules correspond to the variable rule, abstraction rule and application rule of STLC [Pierce 2002].

The STLC-UNIT-I rule is the introduction rule for the unit type and corresponds to the unit type extension of STLC. The unit type does not have an elimination rule.

The product type $A \times B$ is introduced by the STLC-PAIR-I rule, but eliminated by a `let` binding rather than projection functions. This corresponds to the elimination rule of product types in linear type systems [Wadler 1990].

The STLC-RET and STLC-BIND rules correspond to the return and bind operations of monads. As such, return lifts an expression $e \in \texttt{Expr}$ of type $A \in \texttt{Type}$ to an expression $\texttt{return}\,e$ of type $T\,A$. Similarly, STLC-BIND consists of first executing an expression e_1 of type $T\,A$, then sequencing the resulting x of type A in the second computation e_2 of type $T\,B$. The typing rules correspond to the $[\_]_T$ and `let` rules of the Simple Metalanguage by Moggi [1991].

The STLC-REF rule allocates a new location on the heap containing the value $e \in \texttt{Expr}$ of type $A \in \texttt{Type}$ and therefore results in a monad computation containing the allocated location $T\,\texttt{loc}$. As only pure values can be stored on the heap, the typing rule also requires that A is a `Pure` type. Similarly, the STLC-FREE rule deallocates a location $e \in \texttt{Expr}$ of type `loc`, returning the previously stored value of `Pure` type A. The STLC-REPLACE rule replaces the value at a location $e_1 : \texttt{loc}$ of pure type $A \in \texttt{Type}$ with the value of expression $e_2 : B$ of pure type $B \in \texttt{Type}$, returning the old value as a heap computation $T\,A$.

Remark: The typing rules STLC-REPLACE and STLC-FREE do not guarantee that the operations do not get stuck, as the location type `loc` neither guarantees that a location is still allocated nor that it contains a value of a certain type.

Due to annotating the expressions – including types of heap operations – in Church style, the typing derivations $\Gamma \vdash e : A$ are unique:

Theorem 3.2.5 (Uniqueness of Derivations)**:** For any context Γ, expression $e \in \texttt{Expr}$ and types $A, B \in \texttt{Type}$. If there are derivations d_a of $\Gamma \vdash e : A$ and d_b of $\Gamma \vdash e : B$ then $A = B$ and d_a and d_b are the same derivation.

Proof. By induction on e, using the property that each expression constructor corresponds to exactly one typing rule. □

Using the typing system, we can derive typing judgment of the example programs, such as $\cdot \vdash \texttt{swap\_pair} : A \times B \to B \times A$ and $\cdot \vdash \texttt{free\_pair} : \texttt{loc} \times \texttt{loc} \to T\,(1 \times 1)$. We say that an expression e has a type A when $\cdot \vdash e : A$ is derivable in the system. In particular, both `safe` and `unsafe` are typable with type $1 \to T\,1$, despite the latter getting stuck. This will be resolved in the type system in Chapter 4.

An important property of type systems is that of *thinning* (a multiple variable weakening), which states that if an expression e has type A in a given context Γ, then e also has type A in any larger context $\Gamma' \supseteq \Gamma$. Formally, thinnings and the thinning theorem are defined as:

Definition 3.2.6 (Thinning)**:**

$$\text{STLC-THINNING-EMP}\ \ \cdot \supseteq \cdot \qquad \text{STLC-THINNING-TAKE}\ \ \frac{\Gamma \supseteq \Gamma'}{\Gamma, x : A \supseteq \Gamma', x : A} \qquad \text{STLC-THINNING-SKIP}\ \ \frac{\Gamma \supseteq \Gamma'}{\Gamma, x : A \supseteq \Gamma'}$$

Theorem 3.2.7 (Thinning)**:** Given two contexts Γ', Γ, an expression $e \in \texttt{Expr}$ and a type $A \in \texttt{Type}$, if $\Gamma' \supseteq \Gamma$ and $\Gamma \vdash e : A$ then $\Gamma' \vdash e : A$.

Though derivations are unique by Theorem 3.2.5, the specific derivation from this theorem is denoted as $\Gamma' \supseteq \Gamma; \Gamma \vdash e : A$ to be consistent with λ_* (Chapter 4).

By the thinning lemma, we can reuse typing derivation of expressions in larger contexts. For instance, we can reuse the typing derivation for $\Gamma \vdash \texttt{swap\_pair} : 1 \times 1 \to 1 \times 1$ (taking $A, B = 1$) to find a typing derivation of $\cdot \vdash \lambda x : 1.\ \texttt{swap\_pair}\ (x, x)$. The use of the thinning lemma is marked in blue in the derivation:

$$\dfrac{\dfrac{\dfrac{\cdot \vdash \texttt{swap\_pair} : 1 \times 1 \to 1 \times 1}{x : 1 \vdash \texttt{swap\_pair} : 1 \times 1 \to 1 \times 1}\,\text{Theorem 3.2.7} \qquad \dfrac{\dfrac{x : 1 \in x : 1}{x : 1 \vdash x : 1}\,\text{STLC-VAR} \quad \ldots}{x : 1 \vdash (x, x) : 1 \times 1}\,\text{STLC-PAIR-I}}{x : 1 \vdash \texttt{swap\_pair}\ (x, x) : 1 \times 1}\,\text{STLC-FUN-E}}{\cdot \vdash \lambda x : 1.\ \texttt{swap\_pair}\ (x, x) : 1 \to 1}\,\text{STLC-FUN-I}$$

3.3 Substitution

In an operational semantics of STLC, the execution of terms is modeled using β-reductions of the form $(\lambda x : A.\, e_1)e_2 \to_\beta [x \mapsto e_2]e_1$, where $[x \mapsto e_2]e_1$ corresponds to replacing occurrences of x in e_1 with e_2. In such a reduction, the combination of introduction rule $\lambda x : A.\, e_1$ and elimination rule e_2 is called a β-redex. Terms which reduce to the same term through 0 or more β-reductions are considered β-equivalent.

Though we opt for a denotational semantics in this work (*i.e.* giving a mathematical model of typed expressions), rather than an operational semantics, we still want the property that β (and η) equivalent programs have the same interpretation. To reason about these equalities, we first consider simultaneous substitutions of multiple variables, followed by defining well typed substitutions, which only substitute expressions of the correct type, and therefore do not for instance substitute a unit () for a variable of a function type $1 \to 1$.

Definition 3.3.1 (Syntactic Substitution)**:** A syntactic substitution $\delta : \texttt{Var} \xrightarrow{\text{fin}} \texttt{Expr}$ is a map from a finite subset of variables to expressions.

The domain $\texttt{dom}\,\delta$ of a substitution is the finite set of variables $V \subseteq \texttt{Var}$ on which δ is defined. The following are the main notations for defining and updating substitutions:

Notation:

1. The empty substitution $\emptyset$ is the substitution which is undefined for all variables:
$$\emptyset\, x = \bot$$
2. The identity substitution I_V is the substitution which maps each variable in a finite subset $V \subseteq \texttt{Var}$ to itself, and is undefined for all other variables:
$$I_V\, x \stackrel{\text{def}}{=} \begin{cases} x & \textit{if } x \in V \\ \bot & \textit{otherwise} \end{cases}$$
The specific subset V of the identity substitution I is generally left implicit.
3. The update substitution $\delta\langle x \mapsto e\rangle$ is the substitution which maps all variables according to δ, but additionally maps x to e. By the Barendregt convention we assume $x \notin \texttt{dom}\,\delta$:
$$\delta\langle x \mapsto e\rangle\ y \stackrel{\text{def}}{=} \begin{cases} e & \textit{if } y = x \\ \delta\, y & \textit{otherwise} \end{cases}$$

Syntactic substitutions can be applied to expressions to substitute the free variables of e with those defined by the substitution:

Definition 3.3.2 (Substitution on Expressions)**:** For an expression $e \in \texttt{Expr}$ and substitution δ such that $\texttt{fv}\, e \subseteq \texttt{dom}\,\delta$, the substitution of e by δ, denoted $[\delta]e$, is defined as:

$$[\delta]x \stackrel{\text{def}}{=} \delta\, x \qquad [\delta]() \stackrel{\text{def}}{=} ()$$
$$[\delta](e_1, e_2) \stackrel{\text{def}}{=} ([\delta]e_1, [\delta]e_2) \qquad [\delta](\lambda x : A.\, e) \stackrel{\text{def}}{=} \lambda x : A.\, [\delta\, \langle x \mapsto x\rangle]e$$
$$[\delta](\texttt{let}\,(x, y) = e_1\,\texttt{in}\,e_2) \stackrel{\text{def}}{=} \texttt{let}\,(x, y) = [\delta]e_1\,\texttt{in}\,[\delta\, \langle x \mapsto x, y \mapsto y\rangle]e_2$$
$$[\delta](e_1\, e_2) \stackrel{\text{def}}{=} ([\delta]e_1)\,[\delta]e_2 \qquad [\delta](\texttt{return}\, e) \stackrel{\text{def}}{=} \texttt{return}\,[\delta]e$$
$$[\delta](\texttt{let!}\, x = e_1\,\texttt{in}\,e_2) \stackrel{\text{def}}{=} \texttt{let!}\, x = [\delta]e_1\,\texttt{in}\,[\delta\, \langle x \mapsto x\rangle]e_2 \qquad [\delta](\texttt{ref}_A\, e) \stackrel{\text{def}}{=} \texttt{ref}_A\,[\delta]e$$
$$[\delta](\texttt{replace}_{A,B}\, e_1\, e_2) \stackrel{\text{def}}{=} \texttt{replace}_{A,B}\,[\delta]e_1\,[\delta]e_2 \qquad [\delta](\texttt{free}_A\, e) \stackrel{\text{def}}{=} \texttt{free}_A\,[\delta]e$$

A key difference between our syntactic substitution and single variable substitution is the condition that all free variables in the expression must be in the domain of the substitution. As such, the bound variables in lambda abstractions and let-bindings are added to the substitution when substituting their subexpressions. By the Barendregt convention, we assume that the newly bound variables are not in the domain of the substitution.

The definition of syntactic substitution merely ensures that all free variables in the expression are substituted, but does not ensure that the resulting expression is well-typed. For instance, $e = x\, y$ can be typed as $x : 1 \mapsto 1, y : 1 \vdash e : 1$, and $\emptyset\, \langle x \mapsto (), y \mapsto ()\rangle$ substitutes all free variables in e, but the resulting expression $[\delta]e = ()\,()$ cannot be typed.

As substitutions replace all free variables in the expression, they can be used to change the context in which an expression is typed as a general case of thinning. Substitutions can be seen as an n-valued tuple containing an expression e_i for each variable x_i of type A_i. As such, type preserving substitutions are defined as:

Definition 3.3.3 (Type Preserving Substitution)**:** A type preserving substitution $\Gamma' \vdash \delta : \Gamma$ is a syntactic substitution δ defined on the variables defined in Γ, such that each variable $x_i : A_i \in \Gamma$ is mapped to an expression e_i of the same type in Γ', denoted by the judgment $\Gamma' \vdash e_i : A_i$. Inductively, type preserving substitutions are defined by the following rules:

$$\boxed{\Gamma' \vdash \delta : \Gamma}$$

STLC-SUBST-NIL
$$\Gamma' \vdash \emptyset : \cdot$$

STLC-SUBST-CONS
$$\frac{\Gamma' \vdash \delta : \Gamma \qquad \Gamma' \vdash e : A}{\Gamma' \vdash \delta\, \langle x \mapsto e\rangle : \Gamma, x : A}$$

Remark: At first glance, the order of Γ and Γ' in the notation $\Gamma' \vdash \delta : \Gamma$ in this definition seems backwards, as the substitution sends variables and expressions in Γ to expressions in Γ'. We adopt this notation by Angiuli and Gratzer [2024] for reasons discussed in § 3.4.

As indicated by the name, type preserving substitutions preserve typeability of terms under the substitution. This is captured by the following conjecture:

Theorem 3.3.4 (Substitution of Derivation)**:** If $\Gamma' \vdash \delta : \Gamma$ and $\Gamma \vdash e : A$ then $\Gamma' \vdash [\delta]e : A$.
The specific derivation is denoted $\Gamma' \vdash \delta : \Gamma; \Gamma \vdash e : A$.

We will prove this theorem by induction on the typing derivation $\Gamma \vdash e : A$. As the λ, `let` and `let!` cases extend the substitution, the proof requires extending the substitution with new variables mapped to themselves. This is captured by the following lemmas, which thin the substitution with the new variable, followed by adding the new variable respectively.

Lemma 3.3.5 (Thinning Substitution)**:** If $\Gamma_1 \supseteq \Gamma'$ and $\Gamma' \vdash \delta : \Gamma$ then $\Gamma_1 \vdash \delta : \Gamma$.

Proof. By induction on the derivation of $\Gamma' \vdash \delta : \Gamma$ and applying Theorem 3.2.7. □

Corollary 3.3.6 (Substitution Abstraction)**:** If $\Gamma_1 \vdash \delta : \Gamma_2$ then $\Gamma_1, x : A \vdash \delta \langle x \mapsto x \rangle : \Gamma_2, x : A$.

Proof. By Lemma 3.3.5 and Theorem 3.2.7. □

Proof of Theorem 3.3.4. By induction on the typing derivation $\Gamma \vdash e : A$. In the STLC-FUN-I, STLC-PAIR-E and STLC-BIND cases, we apply Corollary 3.3.6 to extend the substitution with newly bound variables. □

Equivalences

In this system, the heap operations $\texttt{replace}_{A,B}$ and $\texttt{free}_A$ are annotated with types, rather than using either typed references [Pierce 2002], or capabilities as in L^3 [Morrisett et al. 2005]. As such, typeable programs such as `unsafe` still get stuck. We give a denotational semantics in § 3.4, which utilizes partiality to encode getting stuck. This partiality is then removed by refinement using the labeled type system in Chapter 4. We want to ensure that the interpretations of β- and η-equivalent programs in the pure part of the language are equal. Hence, we define a typed equivalence judgment:

Definition 3.3.7 (Equivalences)**:** The equivalence judgment $\Gamma \vdash e_1 \equiv e_2 : A$ states that e_1 and e_2 are equivalent typeable expressions of type A in a context Γ.

$$\boxed{\Gamma \vdash e_1 \equiv e_2 : A}$$

STLC-EQ-REFL
$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \equiv e : A}$$

STLC-EQ-SYM
$$\frac{\Gamma \vdash e_1 \equiv e_2 : A}{\Gamma \vdash e_2 \equiv e_1 : A}$$

STLC-EQ-TRANS
$$\frac{\Gamma \vdash e_1 \equiv e_2 : A \qquad \Gamma \vdash e_2 \equiv e_3 : A}{\Gamma \vdash e_1 \equiv e_3 : A}$$

STLC-EQ-FUN-BETA
$$\frac{\Gamma, x : A \vdash e_1 : B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash (\lambda x : A.\, e_1)\, e_2 \equiv [I \langle x \mapsto e_2 \rangle] e_1 : B}$$

STLC-EQ-PAIR-BETA
$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B \qquad \Gamma, x : A, y : B \vdash e_3 : C}{\Gamma \vdash (\texttt{let}\,(x, y) = (e_1, e_2)\,\texttt{in}\,e_3) \equiv [I \langle x \mapsto e_1, y \mapsto e_2 \rangle] e_3 : C}$$

STLC-EQ-FUN-ETA
$$\frac{\Gamma \vdash e : A \to B \qquad x \notin \texttt{fv}\, e}{\Gamma \vdash (\lambda x : A.\, e\, x) \equiv e : A \to B}$$

STLC-EQ-PAIR-ETA
$$\frac{\Gamma \vdash e : A \times B \qquad x, y \notin \texttt{fv}\, e}{\Gamma \vdash (\texttt{let}\,(x, y) = e\,\texttt{in}\,(x, y)) \equiv e : A \times B}$$

STLC-EQ-BIND-BETA
$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : T\, B}{\Gamma \vdash (\texttt{let!}\, x = (\texttt{return}\, e_1)\,\texttt{in}\, e_2) \equiv [I \langle x \mapsto e_1 \rangle] e_2 : T\, B}$$

STLC-EQ-BIND-ETA
$$\frac{\Gamma \vdash e_1 : T\, A}{\Gamma \vdash (\texttt{let!}\, x = e_1\,\texttt{in}\,\texttt{return}\, x) \equiv e_1 : T\, A}$$

$$\frac{\Gamma \vdash e_1 : T\,A \qquad \Gamma, x : A \vdash e_2 : T\,B \qquad \Gamma, y : B \vdash e_3 : T\,C}{\Gamma \vdash (\texttt{let!}\; y = (\texttt{let!}\; x = e_1 \;\texttt{in}\; e_2) \;\texttt{in}\; e_3) \equiv (\texttt{let!}\; x = e_1 \;\texttt{in}\; (\texttt{let!}\; y = e_2 \;\texttt{in}\; e_3)) : T\,C}\;\text{STLC-EQ-BIND-BIND}$$

As well as congruence rules for each of the typing rules in Definition 3.2.4. For example, the congruence rules for STLC-FUN-E and STLC-PAIR-E are:

$$\frac{\Gamma \vdash e_1 \equiv e_1' : A \to B \qquad \Gamma \vdash e_2 \equiv e_2' : A}{\Gamma \vdash e_1\, e_2 \equiv e_1'\, e_2' : B}\;\text{STLC-CONGR-FUN-E}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : A \times B \qquad \Gamma, x : A, y : B \vdash e_2 \equiv e_2' : C}{\Gamma \vdash (\texttt{let}\,(x, y) = e_1 \;\texttt{in}\; e_2) \equiv (\texttt{let}\,(x, y) = e_1' \;\texttt{in}\; e_2') : C}\;\text{STLC-CONGR-PAIR-E}$$

The STLC-EQ-REFL, STLC-EQ-SYM and STLC-EQ-TRANS rules establish that $\Gamma \vdash \_ \equiv \_ : A$ is an equivalence relation for all expressions $e \in \texttt{Expr}$ for which $\Gamma \vdash e : A$.

The STLC-EQ-FUN-BETA, STLC-EQ-PAIR-BETA, STLC-EQ-FUN-ETA and STLC-EQ-PAIR-ETA rules are the β and η equivalences for STLC with unit and products.

The STLC-EQ-BIND-BETA, STLC-EQ-BIND-ETA and STLC-EQ-BIND-BIND rules correspond to the 3 monad laws and do not depend on the interpretation of heap operations. The form of these rules is the same as those of the Simple Metalanguage by Moggi [1991]. The assumption $\Gamma, y : B \vdash e_3 : T\,C$ of the STLC-EQ-BIND-BIND rule ensures (by the Barendregt convention) that x is not a free variable in e_3.

These equivalences are sufficient to prove properties about the pure (not heap) part of the language, such as proving that swapping a pair twice is equivalent to not swapping it at all (contexts and types are omitted for brevity):

Example 3.3.8 (Swap swap)**:**

$$\begin{aligned}\texttt{swap\_pair}\,(\texttt{swap\_pair}\,(x, y)) &\equiv \texttt{swap\_pair}\,([I\,\langle x_1 \mapsto x, y_1 \mapsto y\rangle](y_1, x_1)) \\ &\equiv \texttt{swap\_pair}\,(y, x) \\ &\equiv [I\,\langle x_2 \mapsto y, y_2 \mapsto x\rangle](y_2, x_2) \\ &\equiv (x, y)\end{aligned}$$

In the example, the given equality can be derived for a given context Γ (such as $x : A, y : B$), and type $A \times B$, but the context and type have been omitted for brevity. The derivation of the equality is not shown, but can be derived using the equivalences in Definition 3.3.7. The shape of the equivalence rules in Definition 3.3.7 is very similar to the typing rules in Definition 3.2.4, as typing such an equivalence is sufficient to typing both expressions.

Theorem 3.3.9 (Equivalence Typeability)**:** If $\Gamma \vdash e_1 \equiv e_2 : A$ then $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$.

Proof. By induction on the derivation of $\Gamma \vdash e_1 \equiv e_2 : A$. The proof for each of the β- and η-equivalences rules are similar. For instance, the derivations for the left and right expressions of STLC-EQ-FUN-BETA are given by:

$$\frac{\dfrac{\Gamma, x : A \vdash e_1 : B}{\Gamma \vdash (\lambda x : A.\, e_1) : A \to B}\text{STLC-FUN-I} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash (\lambda x : A.\, e_1)\, e_2 : B}\text{STLC-FUN-E}$$

and

$$\frac{\dfrac{\Gamma \vdash I : \Gamma \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash I\,\langle x \mapsto e_2\rangle : \Gamma, x : A} \qquad \Gamma, x : A \vdash e_1 : B}{\Gamma \vdash [I\,\langle x \mapsto e_2\rangle]e_1 : B}\text{Theorem 3.3.4}$$

The η-rules are similar, but apply thinning rather than substitution. The congruence rules are proven by induction and applying the corresponding typing rule for each side. □

3.4 Denotational Semantics

We give a denotational semantics for $\lambda_{\texttt{loc}}$ using a set interpretation of types. By proving that the semantic substitution and syntactic substitution commute, we can show that the denotational semantics is sound with respect to the β and η equivalences of $\lambda_{\texttt{loc}}$.

Definition 3.4.1 (Denotation of Types)**:**

$$\boxed{[\![A]\!] \in \mathit{Set}}$$

$$[\![1]\!] \stackrel{\text{def}}{=} \{\star\} \qquad [\![A \times B]\!] \stackrel{\text{def}}{=} [\![A]\!] \times [\![B]\!] \qquad [\![A \to B]\!] \stackrel{\text{def}}{=} \{f : [\![A]\!] \to [\![B]\!]\} \qquad [\![\texttt{loc}]\!] \stackrel{\text{def}}{=} \mathcal{L}$$

$$[\![T\,A]\!] \stackrel{\text{def}}{=} \{f : \texttt{Heap} \rightharpoonup [\![A]\!] \times \texttt{Heap}\}$$

where:

$$\texttt{Heap} \stackrel{\text{def}}{=} \mathcal{L} \xrightarrow{\text{fin}} \sum_{\vdash A:\texttt{Pure}} [\![A]\!]$$

$\mathcal{L}$ is a countably infinite set of heap locations

The denotations of each type are defined as sets of elements as follows. The unit type is interpreted as the set of a single unit element, the pair type as a cartesian product and the function type as a set of total functions. This corresponds to the set interpretation of STLC with units and products [Pierce 2002].

The location type is interpreted as a countably infinite set of location values $\mathcal{L}$ such as the natural numbers. The monadic type $T\,A$ is interpreted as the set of partial functions from heaps to a return value of type A and a resulting heap. The partiality in the interpretation is used to interpret heap operations which may get stuck (such as freeing a location twice), rather than for recursion as in Fiore et al. [2022].

The set of heaps `Heap` is interpreted as a finite map from locations to a pure type A and a value in the interpretation of that type. The heap is specifically restricted to `Pure` types, as these types do not reference the `Heap` itself. Attempting to store a non-pure type (such as $T\,1$) in the heap would require a definition of `Heap` including at least:

$$\begin{aligned}\texttt{Heap} &\stackrel{\text{def}}{=} \mathcal{L} \xrightarrow{\text{fin}} (\texttt{Heap} \rightharpoonup 1 \times \texttt{Heap}) + \ldots \\ &= \mathcal{L} \xrightarrow{\text{fin}} ((\mathcal{L} \xrightarrow{\text{fin}} (\texttt{Heap} \rightharpoonup 1 \times \texttt{Heap}) + \ldots) \rightharpoonup \ldots) + \ldots\end{aligned}$$

This is not a valid definition as this definition of `Heap` uses a non-strictly positive occurrence of `Heap`, *i.e.* on the left side of a function arrow, which prevents a straightforward inductive definition. Hence, allowing the storing of the monadic type on the heap would require a more complex definition of `Heap`, potentially involving constructs such as step indexing and guarded recursion [Appel and McAllester 2001; Nakano 2000]. These methods are not used in this work, as the focus is on the labeled type system in Chapter 4. Moreover, the restriction to `Pure` types ensures that the interpretations of programs with heap operations are terminating: they either succeed or get *stuck*.

Note that only the monad type provides access to the heap, and can therefore be used for heap operations, whereas the other types do not. Alternatively, each type could have been defined using the monad (*i.e.* A represents $T\,A$), but that would require defining an implicit execution order for each operation and complicates the denotational semantics. It is still possible to derive such a system by redefining each expression constructor using the monadic bind and return, for instance: $\texttt{let}\,(x, y) = e_1\,\texttt{in}\,e_2$ becomes $\texttt{let!}\,xy = e_1\,\texttt{in}\,\texttt{let}\,(x, y) = xy\,\texttt{in}\,e_2$ and $\texttt{replace}_{A,B}\,e_1\,e_2$ becomes $\texttt{let!}\,l = e_1\,\texttt{in}\,\texttt{let!}\,v = e_2\,\texttt{in}\,\texttt{replace}_{A,B}\,l\,v$. We opt for a separate monadic type to simplify the denotational semantics of the pure part of the language for both this system and the labeled system in Chapter 4 which takes advantage of the lack of execution order.

The denotational semantics of expressions are defined only for typeable expressions, by defining the denotation of typing derivations $\Gamma \vdash e : A$ instead of directly on expressions $e \in \texttt{Expr}$. They are defined as functions from the denotation of the context $[\![\Gamma]\!]$ to the denotation of the type $[\![A]\!]$, where the context $[\![\Gamma]\!]$ are represented as functions from the variables in the context to the values in the denotations of the types of those variables in the context.

Definition 3.4.2 (Denotation of Contexts)**:**

$$\boxed{[\![\Gamma]\!] \in \mathit{Set}}$$

$$[\![\cdot]\!] \stackrel{\text{def}}{=} \{[]\} \qquad [\![\Gamma, x : A]\!] \stackrel{\text{def}}{=} \{\gamma\,\langle x \mapsto v\rangle \mid \gamma \in [\![\Gamma]\!] \wedge x \in [\![A]\!]\}$$

We opt for a denotation of contexts similar to that of Fiore et al. [2022], rather than using cartesian products, as it simplifies finding the denotation of a specific variable.

Using these contexts, we define the denotation of derivations inductively as functions from the denotation of the context to the denotation of the type of the expressions:

Definition 3.4.3 (Denotation of Derivations)**:**

$$\boxed{[\![\Gamma \vdash e : A]\!] : [\![\Gamma]\!] \to [\![A]\!]}$$

$$\left[\!\!\left[\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} \gamma\, x$$

$$[\![\Gamma \vdash () : 1]\!]_\gamma \stackrel{\text{def}}{=} ()$$

$$\left[\!\!\left[\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \to B}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} v \in [\![A]\!] \mapsto [\![\Gamma, x : A \vdash e : B]\!]_{\gamma\langle x \mapsto v\rangle}$$

$$\left[\!\!\left[\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} [\![\Gamma \vdash e_1 : A \to B]\!]_\gamma\, ([\![\Gamma \vdash e_2 : A]\!]_\gamma)$$

$$\left[\!\!\left[\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} ([\![\Gamma \vdash e_1 : A]\!]_\gamma\, , [\![\Gamma \vdash e_2 : B]\!]_\gamma)$$

$$\left[\!\!\left[\frac{\Gamma \vdash e_1 : A \times B \qquad \Gamma, x : A, y : B \vdash e_2 : C}{\Gamma \vdash \mathtt{let}\,(x, y) = e_1\,\mathtt{in}\,e_2 : C}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} [\![\Gamma, x : A, y : B \vdash e_2 : C]\!]_{\gamma\langle (x,y) \mapsto [\![\Gamma \vdash e_1 : A \times B]\!]_\gamma\rangle}$$

$$\left[\!\!\left[\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{return}\, e : T\, A}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} h \mapsto ([\![\Gamma \vdash e : A]\!]_\gamma\, , h)$$

$$\left[\!\!\left[\frac{\Gamma \vdash e_1 : T\, A \qquad \Gamma, x : A \vdash e_2 : T\, B}{\Gamma \vdash \mathtt{let!}\, x : A = e_1\,\mathtt{in}\, e_2 : T\, B}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} \begin{array}{l} h \mapsto \mathtt{let}\,(x, h_1) = [\![\Gamma \vdash e_1 : T\, A]\!]_\gamma\; h\,\mathtt{in} \\ \qquad [\![\Gamma, x : A \vdash e_2 : T\, B]\!]_{\gamma\langle x \mapsto x\rangle}\; h_1 \end{array}$$

$$\left[\!\!\left[\frac{\Gamma \vdash e : A \qquad \vdash A : \mathtt{Pure}}{\Gamma \vdash \mathtt{ref}_A\, e : T\,\mathtt{loc}}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} \begin{array}{l} h \mapsto \mathtt{let}\, l = \mathtt{newloc}\, h\,\mathtt{in} \\ \qquad (l, h \uplus \{l \mapsto (A, [\![\Gamma \vdash e : A]\!]_\gamma)\}) \end{array}$$

$$\left[\!\!\left[\frac{\Gamma \vdash e : \mathtt{loc} \qquad \vdash A : \mathtt{Pure}}{\Gamma \vdash \mathtt{free}_A\, e : T\, A}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} h \mapsto \mathtt{let}\, l = [\![\Gamma \vdash e : \mathtt{loc}]\!]_\gamma \mathtt{in} \begin{cases} (\pi_2(h\, l), h \setminus \{l\}) & \textit{if } \pi_1(h\, l) = A \\ \bot & \textit{otherwise} \end{cases}$$

$$\left[\!\!\left[\frac{\begin{array}{ll}\Gamma \vdash e_1 : \mathtt{loc} & \Gamma \vdash e_2 : B \\ \vdash A : \mathtt{Pure} & \vdash B : \mathtt{Pure}\end{array}}{\Gamma \vdash \mathtt{replace}_{A,B}\, e_1\, e_2 : T\, A}\right]\!\!\right]_\gamma \stackrel{\text{def}}{=} \begin{array}{l} h \mapsto \mathtt{let}\, l = [\![\Gamma \vdash e_1 : \mathtt{loc}]\!]_\gamma\;\mathtt{in} \\ \qquad \mathtt{let}\, v = [\![\Gamma \vdash e_2 : B]\!]_\gamma\;\mathtt{in} \\ \qquad \begin{cases} (\pi_2\,(h\, l), h\,\langle l \mapsto (B, v)\rangle) & \textit{if } \pi_1(h\, l) = A \\ \bot & \textit{otherwise} \end{cases} \end{array}$$

The interpretation of the variable rule corresponds to taking the value of the variable in the context.

The unit is always interpreted as the single unit element.

The interpretations of function abstraction and function application are the same as for STLC, and the interpretation of pair introduction is the same as for STLC with products. The product elimination finds the interpretation of the first expression e_1 in $[\![A]\!] \times [\![B]\!]$, assigns x to the value in $[\![A]\!]$ and y to the value in $[\![B]\!]$ in an extended context, and then interprets the second expression e_2 in the extended context.

The `return` operation is interpreted as a total function from heaps to the interpretation of the inner expression and the unaltered heap. Similarly, the `let!` operation is similar to the `let` operation for pairs, but additionally threads the heap through the interpretations of the two expressions. For brevity, the partiality of interpreting the two expressions is left implicit.

The `ref` operation is also interpreted as a total function from heaps to locations and an updated heap. It first determines an unused location using the mathematical `newloc` function, which deterministically returns a location not in the heap. The specific implementation of the `newloc` operation is not relevant, as long as it is deterministic and the returned value is not in the heap. The new location is returned and the interpretation of the expression is added to the heap. By the assumption $\vdash A : \texttt{Pure}$ in the derivation, the interpretation $[\![\Gamma \vdash e : A]\!]$ in the context can be placed in the heap along with type A.

The `free` operation is interpreted as a partial function which does the opposite to `ref`. It first interprets the expression to a location l, then checks that the type of the value at that location is A. If it is, the value is returned and the location is removed from the returned heap. Otherwise the operation gets stuck and returns $\bot$.

The interpretation of the `replace` operation is a combination of `free` and `ref`. It first interprets the expression to a location l, then checks that the type of the value at that location is A. If it is, the value is returned and the location is updated with the new value and type B. Otherwise the operation gets stuck and returns $\bot$.

The denotational semantics of a typing derivation describe the mathematical behavior of the program. For instance, the denotational semantics of the `swap_pair` function corresponds to a mathematical function that swaps two elements in a pair:

Example 3.4.4 (Denotation of `swap_pair`)**:**

$$
\begin{aligned}
&[\![\Gamma \vdash \texttt{swap\_pair} : A \times B \to B \times A]\!]_\gamma \\
&= v \in [\![A \times B]\!] \mapsto [\![\Gamma, e : A \times B \vdash \texttt{swap\_pair}\, e : B \times A]\!]_{\gamma\langle e \mapsto v\rangle} \\
&= (v_a, v_b) \in [\![A]\!] \times [\![B]\!] \mapsto [\![\Gamma, e : A \times B, x : A, y : B \vdash (y, x) : B \times A]\!]_{\gamma\langle (x,y) \mapsto (v_a, v_b)\rangle} \\
&= (v_a, v_b) \in [\![A]\!] \times [\![B]\!] \mapsto (v_b, v_a)
\end{aligned}
$$

The denotations of programs such as `safe` and `unsafe`, which use the monadic type $T\,A$, return partial functions from heaps to a return value and new heap:

Example 3.4.5:

$$
\begin{aligned}
[\![\Gamma \vdash \texttt{duplicate} : \texttt{loc} \to \texttt{loc} \times \texttt{loc}]\!]_\gamma &= v \in \mathcal{L} \mapsto (v, v) \\
[\![\Gamma \vdash \texttt{free\_pair} : \texttt{loc} \times \texttt{loc} \to T\,(1 \times 1)]\!]_\gamma &= (l_1, l_2) \mapsto h \mapsto \begin{cases} (\pi_2\,(h\,l_1), \pi_2\,(h\,l_2)) & \text{if } \pi_1(h\,l_1) = 1 \\ & \text{and } \pi_1(h\,l_2) = 1 \\ & \text{and } l_1 \neq l_2 \\ \bot & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash \texttt{safe} : 1 \to T\,1]\!]_\gamma &= \star \mapsto h \mapsto (\star, h) \\
[\![\Gamma \vdash \texttt{unsafe} : 1 \to T\,1]\!]_\gamma &= \star \mapsto h \mapsto \bot
\end{aligned}
$$

The denotation of `duplicate` is similar to `swap_pair` and is a function from a location to a pair of the same location. The denotation of `free_pair` is a function taking a pair of locations and a heap, and removing the locations from the heap if they contain a unit. As only allocated locations can be freed, the denotation is not defined when the locations are the same or if one of the locations does not contain a unit.

Using this denotation, the denotations for `safe` and `unsafe` are defined as functions that return the same heap and get stuck ($\bot$ for any heap) respectively, as `safe` allocates and frees 2 new locations, whereas `unsafe` only allocates one location and attempts to free it twice. As `unsafe` is still typeable, the type system does not prevent memory errors such as double free. In Chapter 4 we refine the type system to exclude such programs by making them untypeable.

As typing derivations in $\lambda_{\texttt{loc}}$ are unique (Theorem 3.2.5), the denotational semantics of two typing derivations are equal as well.

Theorem 3.4.6 (Derivation Irrelevance)**:** If d, e are two derivations of $\Gamma \vdash e : A$ then $[\![d]\!] = [\![e]\!]$.

Proof. By Theorem 3.2.5. □

As the denotational semantics of a typing judgment does not depend on the specific derivation, it is sufficient to write $[\![\Gamma \vdash e : A]\!]$ instead of $[\![d]\!]$ for a specific derivation d of $\Gamma \vdash e : A$. Furthermore, it is an important property to ensure that the denotational interpretation of a program typed in the empty context $\vdash e : A$ depends only on the expression $e \in \texttt{Expr}$ and its type $A \in \texttt{Type}$, specified in the syntax, rather than choices made by a type-checker in finding a derivation.

Semantic Equivalence

In Example 3.3.8 we showed that applying `swap_pair` twice is equivalent to not applying it at all. Indeed, the denotational semantics of $\texttt{swap\_pair}(\texttt{swap\_pair}(x, y))$ is defined as:

$$\begin{aligned}
&[\![\Gamma \vdash \texttt{swap_pair}\,(\texttt{swap_pair}\,(x, y)) : A \times B]\!]_\gamma \\
&= (((v_a, v_b) \mapsto (v_b, v_a)) \circ ((v_a, v_b) \mapsto (v_b, v_a)))\ [\![\Gamma \vdash (x, y) : A \times B]\!]_\gamma \\
&= [\![\Gamma \vdash (x, y) : A \times B]\!]_\gamma
\end{aligned}$$

In a more general sense, we want to show that the denotational semantics is sound with respect to the equivalence relation (Definition 3.3.7), meaning that any β or η equivalent programs have the same denotational semantics. This is not straightforward to prove, as the equivalence rules use substitution. For instance, consider the STLC-EQ-FUN-BETA rule:

$$\begin{aligned}
[\![\Gamma \vdash (\lambda x : A.\, e_1)\, e_2 : B]\!]_\gamma &= [\![\Gamma \vdash (\lambda x : A.\, e_1) : A \to B]\!]_\gamma\ [\![\Gamma \vdash e_2 : A]\!]_\gamma \\
&= \lambda v \in [\![A]\!].\ [\![\Gamma, x : A \vdash e_1 : B]\!]_{\gamma\langle x \mapsto v\rangle}\ [\![\Gamma \vdash e_2 : A]\!]_\gamma \\
&= [\![\Gamma, x : A \vdash e_1 : B]\!]_{\gamma\left\langle x \mapsto [\![\Gamma \vdash e_2 : A]\!]_\gamma \right\rangle} \\
&= \left([\![\Gamma, x : A \vdash e_1 : B]\!] \circ \left(\gamma \mapsto \gamma \left\langle x \mapsto [\![\Gamma \vdash e_2 : B]\!]_\gamma \right\rangle\right)\right)\ \gamma
\end{aligned}$$

Here the type of $\gamma \mapsto \gamma \left\langle x \mapsto [\![\Gamma \vdash e_2 : B]\!]_\gamma \right\rangle$ is $[\![\Gamma]\!] \to [\![\Gamma, x : A]\!]$. What remains is to prove that this substitution of γ followed by the denotation is the same as the denotation of the substitution. This property is captured by the right hand side of the diagram below, namely, taking the denotation after syntactic substitution is the same as first taking the denotation and then applying semantic substitution.

$$\begin{array}{ccccc}
e & \xleftarrow[untyped]{} & \Gamma \vdash e\colon A & \xrightarrow{[\![\cdot]\!]} & [\![\Gamma \vdash e\colon A]\!] \\
\downarrow{\scriptstyle \delta} & & \downarrow{\scriptstyle \Delta \vdash \delta : \Gamma} & & \downarrow{\scriptstyle [\![\Delta \vdash \delta : \Gamma]\!]} \\
[\delta]e & \xleftarrow[untyped]{} & \Delta \vdash [\delta]e\colon A & \xrightarrow{[\![\cdot]\!]} & [\![\Delta \vdash [\delta]e\colon A]\!]
\end{array}$$

Both semantic thinning and semantic substitution can be represented as functions between interpretations of contexts as follows:

Definition 3.4.7 (Denotation of Thinning)**:**

$$\boxed{[\![\Gamma \supseteq \Gamma']\!] : [\![\Gamma]\!] \to [\![\Gamma']\!]}$$

$$[\![\cdot \supseteq \cdot]\!]\ [] = [] \qquad \left[\!\!\left[\frac{\Gamma \supseteq \Gamma'}{\Gamma, x : A \supseteq \Gamma', x : A}\right]\!\!\right] (\gamma\,\langle x \mapsto v\rangle) = ([\![\Gamma \supseteq \Gamma']\!]\,\gamma)\,\langle x \mapsto v\rangle$$

$$\left[\!\!\left[\frac{\Gamma \supseteq \Gamma'}{\Gamma, x : A \supseteq \Gamma'}\right]\!\!\right] (\gamma\,\langle x \mapsto v\rangle) = [\![\Gamma \supseteq \Gamma']\!]\ \gamma$$

Definition 3.4.8 (Denotation of Substitution)**:**

$$\boxed{[\![\Gamma \vdash \delta : \Gamma']\!] : [\![\Gamma]\!] \to [\![\Gamma']\!]}$$

$$[\![\Gamma \vdash \emptyset : \cdot]\!]\,[] \stackrel{\text{def}}{=} [] \qquad \left[\!\!\left[\frac{\Gamma \vdash \delta : \Gamma' \qquad \Gamma \vdash e : A}{\Gamma \vdash \delta\,\langle x \mapsto e\rangle : \Gamma', x : A}\right]\!\!\right]\gamma \stackrel{\text{def}}{=} ([\![\Gamma \vdash \delta : \Gamma']\!]\,\gamma)\,\langle x \mapsto [\![\Gamma \vdash e : A]\!]\,\gamma\rangle$$

The denotational semantics of substitutions is defined inductively using the empty substitution $\emptyset$ and the update of a substitution with a single new variable. The order of the notation for $\Gamma \vdash \delta : \Gamma'$ is chosen to match the order of the denotational semantics, as $[\![\Gamma \vdash \delta : \Gamma']\!] \in [\![\Gamma]\!] \to [\![\Gamma']\!]$ has the same order as for typing derivations: $[\![\Gamma \vdash e : A]\!] \in [\![\Gamma]\!] \to [\![A]\!]$.

Remark: The notation $\gamma\,\langle x \mapsto v\rangle$ in these definitions correspond to deconstructing a context $\gamma' \in [\![\Gamma, x : A]\!]$ into $\gamma \in [\![\Gamma]\!]$ and $v \in [\![A]\!]$ where $v = \gamma'\,x$ and γ is γ' but with x undefined.

In the case of STLC-EQ-FUN-BETA, the denotation of the substitution corresponds to the required function between context interpretations:

$$[\![\Gamma \vdash I\,\langle x \mapsto_2\rangle : \Gamma, x : A]\!] = \gamma \mapsto \gamma\left\langle x \mapsto [\![\Gamma \vdash e_2 : A]\!]_\gamma\right\rangle$$

Using this interpretation, and the commutativity of the diagram above, we can prove that the denotational semantics is sound with respect to STLC-EQ-FUN-BETA.

Proving the commutativity of the diagrams takes a similar approach to the proof that type preserving substitution preserves typeability of an expression in a context (Theorem 3.3.4). First, we show the same property for thinning: that thinning a context and then taking the interpretation commutes with taking the interpretation and then applying the interpretation of the thinning (Lemma 3.4.9). Next, we expand this thinning property from typing judgments to typed substitutions (Lemma 3.4.10), and then show that extending a substitution with a single variable and taking the interpretation commute (Lemma 3.4.11). Finally, we use these lemmas to prove that typed syntactic substitution and semantic substitution commute (Theorem 3.4.12).

Lemma 3.4.9 (Semantic Thinning)**:** $[\![\Gamma' \supseteq \Gamma; \Gamma \vdash e : A]\!] = [\![\Gamma' \supseteq \Gamma]\!]\,;[\![\Gamma \vdash e : A]\!]$

Proof. By induction on the typing derivation $\Gamma \vdash e : A$ generalized over all Γ'. □

Lemma 3.4.10 (Denotation of Substitution Thinning)**:** If $\Gamma_1 \supseteq \Gamma'$ and $\Gamma' \vdash \delta : \Gamma$ then $[\![\Gamma_1 \supseteq \Gamma'; \Gamma' \vdash \delta : \Gamma]\!] = [\![\Gamma_1 \sqsubseteq \Gamma']\!]\,;[\![\Gamma' \vdash \delta : \Gamma]\!]$

Proof. By induction on the typing derivation $\Gamma' \vdash \delta : \Gamma$ using Lemma 3.4.9 in the STLC-SUBST-CONS case. □

Lemma 3.4.11 (Denotation of Substitution Abstraction)**:** If $\Gamma \vdash \delta : \Gamma'$, $\gamma \in [\![\Gamma]\!]$ and $v \in [\![A]\!]$ then:

$$[\![\Gamma, x : A \vdash \delta\,\langle x \mapsto x\rangle : \Gamma', x : A]\!]\,(\gamma\,\langle x \mapsto v\rangle) = ([\![\Gamma \vdash \delta : \Gamma']\!]\,\gamma)\,\langle x \mapsto v\rangle$$

Proof. By the definition of $[\![\Gamma, x : A \vdash \delta\,\langle x \mapsto x\rangle : \Gamma', x : A]\!]$ and applying Lemma 3.4.10. □

Theorem 3.4.12 (Semantic Substitution)**:** $[\![\Gamma' \vdash \delta : \Gamma; \Gamma \vdash e : A]\!] = [\![\Gamma' \vdash \delta : \Gamma]\!]\,;[\![\Gamma \vdash e : A]\!]$

Proof. By induction on the typing derivation $\Gamma \vdash e : A$ using Lemma 3.4.11 in the STLC-FUN-I, STLC-PAIR-E and STLC-BIND cases. □

The final property for this system is to show that the denotational semantics is sound with respect to the equivalences in § 3.3.

Theorem 3.4.13 (Soundness *w.r.t.* Equivalences)**:** If $\Gamma \vdash e_1 \equiv e_2 : A$ then $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

Proof. By induction on the derivation of $\Gamma \vdash e_1 \equiv e_2$. The β-rules are proven using the semantic substitution theorem (Theorem 3.4.12), the η-rules by derivation irrelevance (Theorem 3.4.6) and the congruence rules by induction. For instance, the proof for STLC-EQ-FUN-ETA is as follows:

$$
\begin{aligned}
&[\![\Gamma \vdash (\lambda x : A.e) : A \to B]\!]_\gamma \\
&= v \in [\![A]\!] \mapsto [\![\Gamma, x : A \vdash e\, x : B]\!]_{\gamma\langle x \mapsto v\rangle} \\
&= v \in [\![A]\!] \mapsto [\![\Gamma, x : A \vdash e : A \to B]\!]_{\gamma\langle x \mapsto v\rangle} \left([\![\Gamma, x : A \vdash x : A]\!]_{\gamma\langle x \mapsto v\rangle}\right) \\
&= v \in [\![A]\!] \mapsto [\![\Gamma, x : A \supseteq \Gamma; \Gamma \vdash e : A \to B]\!]_{\gamma\langle x \mapsto v\rangle}(v) && \text{(By Theorem 3.4.6)} \\
&= v \in [\![A]\!] \mapsto [\![\Gamma \vdash e : A \to B]\!]_\gamma(v) && \text{(By Lemma 3.4.9)} \\
&= [\![\Gamma \vdash e : A \to B]\!]_\gamma
\end{aligned}
$$

□

Chapter 4

Labeled Lambda Calculus

In the previous section we described a system and type theory for a simply types lambda calculus with locations $\lambda_{\texttt{loc}}$. This system permitted both weakening (leaving variables unused) and contraction (using variables more than once), and therefore could not rule out the possibility of memory errors such as *use-after-free*. In this section, we extend the types of $\lambda_{\texttt{loc}}$ with resource tracking, such that well-typed programs in the extended language λ_* do not get *stuck*.

Specifically, we prove a semantic safety theorem for the extended language corresponding to the semantic soundness theorem by Milner [1978], which states that the interpretation of well-typed terms is valid. In $\lambda_{\texttt{loc}}$, the only denotations which can be 'wrong' in this sense are those of heap operations $T\,A$, as their interpretation is undefined when heap operations get stuck. Hence, the goal of λ_* is to ensure that well-typed programs of type $T\,A$ correspond to *total*, rather than *partial* functions from heaps to a result value and updated heap, which we prove at the end of this section (Corollary 4.4.27).

The type system of λ_* is based on the logic of bunched implications (§ 2.2), rather than linear logic, meaning that it can represent functions which share resources between their arguments and captured variables. The language uses both additive ($\wedge$, $\Rightarrow$) and multiplicative ($*, \mathrel{-\!\!*}$) products and functions which describe shared and disjoint resources respectively.

Previous works on incorporating bunched implications in type theories have been made using a tree structure of contexts, called bunches, such as in the work by Pym et al. [O'Hearn and Pym 1999]. The denotational interpretation of such bunches consists of forming a tree of $*$ and $\wedge$ connectives. We instead build our type theory of bunched implications using a resource labeling of types, inspired by the labeled sequent calculus of BBI by Hóu et al. [2015] for a classical version of BI. In this approach, each type is annotated with a label, representing the resources held by the value of that type. In our case, these labels represent the partial heaps and each type is annotated with a partial heap in which the type is valid. Intuitively, the types represent separation logic propositions, whereas the labeling of a type represents the proposition holding for the specific heap corresponding to the label. The labeling of types prevents the need for the tree structure of bunches, and therefore supports simple validity conditions for both types and contexts in terms of the denotation of $\lambda_{\texttt{loc}}$ (Definitions 4.4.19 and 4.4.22).

Rather than defining λ_* as a language with separate denotational semantics, we take an approach similar to that of type-preserving compilation [Morrisett et al. 1999] and erasure of refinement types [Ghalayini and Krishnaswami 2023]. For the labeled system λ_*, we define types and expressions (§ 4.1) which erase (compile) to the unlabeled system $\lambda_{\texttt{loc}}$ (§ 4.4) while preserving well-typedness (Theorem 4.4.5). The denotational semantics of the labeled system is then defined as the denotation of the erased system (§ 4.4). The additional guarantees of the labeled system are finally proven with respect to the denotation of the erased system in a separate regularity theorem (Theorem 4.4.26), from which we conclude semantic safety (Corollary 4.4.27) corresponding to the statement 'well-typed programs don't go wrong'.

4.1 Syntax

The syntax of the labeled calculus λ_* extends that of the $\lambda_{\texttt{loc}}$ by defining both an additive and multiplicative version of functions, products and units. The labeled system similarly has the monadic type for heap operations, but uses typed linear references $\texttt{Ref}\,A$ instead of the untyped locations $\texttt{loc}$ of $\lambda_{\texttt{loc}}$. The types and expressions of λ_* are defined as follows:

Definition 4.1.1 (Types)**:**

$$\begin{aligned} A \in \texttt{Type} ::= {} & 1_{\mathsf{a}} \mid A \wedge B \mid A \Rightarrow B && \text{(Additive)} \\ \mid {} & 1_{\mathsf{m}} \mid A * B \mid A \mathrel{-\!\!*} B && \text{(Multiplicative)} \\ \mid {} & T\ A \mid \texttt{Ref}\ A && \text{(References)} \end{aligned}$$

Definition 4.1.2 (Expressions)**:**

$$\begin{aligned} e \in \texttt{Expr} ::= {} & x \mid \langle\rangle \mid \texttt{let}\,\langle\rangle = e_1\,\texttt{in}\,e_2 \mid \langle e_1, e_2\rangle \mid \texttt{let}\,\langle x, y\rangle = e_1\,\texttt{in}\,e_2 \mid \lambda^* x : A.\, e \mid e_1^*\, e_2 && \text{(Multiplicative)} \\ \mid {} & [] \mid [e_1, e_2] \mid \texttt{let}\,[x, y] = e_1\,\texttt{in}\,e_2 \mid \lambda x : A.\, e \mid e_1\, e_2 && \text{(Additive)} \\ \mid {} & \texttt{return}\,e \mid \texttt{let!}\,x = e_1\,\texttt{in}\,e_2 && \text{(Monadic Operations)} \\ \mid {} & \texttt{ref}_A\,e \mid \texttt{replace}_{A,B}\,e_1\,e_2 \mid \texttt{free}_A\,e \end{aligned}$$

Unit Types. In λ_* there are two types of units. The first is the additive unit 1_{a}, which can hold any resources and corresponds to $\top$ in separation logic. It is constructed using the [] expression, and like the unit 1 in $\lambda_{\texttt{loc}}$, does not have an elimination form. The second type of unit is the multiplicative unit 1_{m}, which holds no resources – similar to `emp` in separation logic – and is constructed using $\langle\rangle$. The multiplicative unit can be eliminated using the $\texttt{let}\,\langle\rangle = e_1\,\texttt{in}\,e_2$ expression, which is used to witness that the expression e_1 describes no resources.

Product Types. There are also both additive and multiplicative products. The additive product $A \wedge B$ is constructed using the $[e_1, e_2]$ expression, and describes pairs in which both elements describe the same resources. The elimination form $\texttt{let}\,[x, y] = e_1\,\texttt{in}\,e_2$ is used to access the components of the pair simultaneously, after which x and y both hold the same resources as e_1. The multiplicative product $A * B$, constructed using $\langle e_1, e_2\rangle$, instead describes pairs in which the resources described by A and B are disjoint. The elimination form $\texttt{let}\,\langle x, y\rangle = e_1\,\texttt{in}\,e_2$ is used to access the components of the pair separately, after which x and y together describe the same resources as e_1.

Function Types. λ_* also has two function types, corresponding to the additive and multiplicative implications in BI respectively. The additive function type $A \Rightarrow B$ is constructed using the $\lambda x : A.\, e$ expression, and is used to describe functions where the function and argument may share the same resources. The additive function type can be eliminated using $e_1\, e_2$, where the resources described by the function e_1, the argument e_2 and the result are all the same.

The multiplicative function type $A \mathrel{-\!\!*} B$, corresponding to $\mathrel{-\!\!*}$ in separation logic and $\multimap$ in BI, is constructed using the $\lambda^* x : A.\, e$ expression, and is used to describe functions where the function and argument describe disjoint resources. The multiplicative function type can be eliminated using $e_1^*\, e_2$, where the resources described by the function and argument must be disjoint, and the resources described by the result are the union of both the function and argument.

The additive function type can be used in combination with additive pairs, whereas the multiplicative function can be used in combination with multiplicative pairs. For instance, the following function which ignores the argument can only be written using the additive function pair:

$$\texttt{let}\,[x, y] = e_1\,\texttt{in}\,(\lambda x : A.\, y)\, x$$

The same program with the multiplicative pair and function type is incorrect, as the resources of the result (those of y) are not the union of those of the function (those of y) and the argument (x):

$$\texttt{let}\,\langle x, y\rangle = e_1\,\texttt{in}\,(\lambda^* x : A.\, y)^*\, x$$

Monadic Type. λ_* also supports monadic types, which are used to encapsulate heap computations that produce values of a certain type. The monadic type $T\,A$ represents a heap computation that, when executed, yields a value of type A. The resources of the monadic type are the partial heap necessary to execute the computation, and the resources of the resulting type A are the partial heap after the computation has been executed. As such, the monadic type $T\,A$ closely resembles the weakest precondition in separation logic.

As the monadic type is a monad, it supports the `return` operation which creates a heap computation that results in the same expression and does not change the heap, and a monadic bind operation $\texttt{let!}\, x : A = e_1\, \texttt{in}\, e_2$ which sequences two heap computations e_1 of type $T\,A$ and e_2 of type $T\,B$.

Reference Types. The reference type $\texttt{Ref}\, A$ represents a reference to a memory location that stores a value of type A. The resources of the reference type are both the location itself, and the resources of the value of type A stored at that location. The reference type $\texttt{Ref}\, A$ is constructed using the $\texttt{ref}_A\, e$ expression, which allocates a new memory location and initializes it with the value e of type A. As creating a reference is a heap operation, the resulting type $T\,(\texttt{Ref}\, A)$ is wrapped in the monadic type.

References can be updated using the $\texttt{replace}_{A,B}\, e_1\, e_2$ operation (similar to `replace` in Rust), which updates the value stored at the memory location referenced by e_1 with the new value e_2 of type B. As the replace operation performs *strong updates* (*i.e.* updates that change the type of the reference), and the resources of the value stored at the location change, the operation returns not only the previous value, but also a new reference of the updated type B. The operation swaps the resources of the previous value on the heap out of the reference type, and the resources of the new value into the reference type. This differs from the $\lambda_{\texttt{loc}}$, where the `replace` operation only returns the previous value, as the location type `loc` does not contain information about the type stored at that location.

Finally, linear references can be freed by giving up the reference of type $\texttt{Ref}\, A$, resulting in the value of type A previously stored at that location. This deallocates the resources corresponding to the location itself, and returns the resources of the A previously stored at that location.

4.2 Typing

Similar to the simply typed system, λ_* makes a distinction between the types which can be stored on the heap, and those which cannot. More specifically, any type involving heap computations $T\,A$ cannot be stored on the heap to ensure that all programs are terminating. Allowing $T\,A$ to be stored on the heap would allow for unbounded recursion through Landin's knot [Landin 1964]. As in $\lambda_{\texttt{loc}}$, each type is assigned a kind, either `Stateful` for types containing heap computations, or `Pure` for types that do not, and can therefore be stored on the heap.

Definition 4.2.1 (Type Kinds)**:**

$$\boxed{\vdash A : k}$$

$$\vdash 1_a : k \qquad \frac{\vdash A : k \qquad \vdash B : k}{\vdash (A \wedge B) : k} \qquad \frac{\vdash A : k \qquad \vdash B : k}{\vdash (A \Rightarrow B) : k} \qquad \vdash 1_m : k \qquad \frac{\vdash A : k \qquad \vdash B : k}{\vdash (A * B) : k}$$

$$\frac{\vdash A : k \qquad \vdash B : k}{\vdash (A \mathrel{-\!\!*} B) : k} \qquad \frac{\vdash A : k}{\vdash T\,A : \texttt{Stateful}} \qquad \frac{\vdash A : \texttt{Pure}}{\vdash (\texttt{Ref}\, A) : k}$$

The main difference in the definition of $\vdash A : k$ between λ_* and $\lambda_{\texttt{loc}}$ is the addition of the $\texttt{Ref}\, A$ type constructor instead of the untyped `loc` type. The $\texttt{Ref}\, A$ type also describes the type of the value stored at the location, and therefore requires that A is pure, to ensure it can be stored in the heap. Unlike the monadic type $T\,A$, the reference type $\texttt{Ref}\, A$ is not restricted to the `Stateful` type kind, as it represents a location, which itself can be stored on the heap. Even though the reference type $\texttt{Ref}\, A$ describes part of the heap, its denotational semantics does not. The specifics of this will be discussed in § 4.4.

The discussion on type constructors in the syntax section discussed the concept of *resources* informally. In the λ_*, we take the approach of Hóu et al. [2015] and Ghari [2017] to include include the resources in the judgment rules in the form of labels and worlds.

Labels and Worlds. Similar to how variables in the λ-calculus can be interpreted as placeholders for values or expressions, labels in the labeled calculus can be interpreted as placeholders for resources.

Definition 4.2.2 (Worlds)**:**

$$a, b, c : \texttt{Label}$$
$$w \in \texttt{World} ::= \epsilon \mid a$$

Where `Label` is a countably infinite set of labels.

Whereas the expression language (`Expr`) is relatively large, the only possible worlds are the labels, and a special world ϵ, which represents no resources. In the concept of separation logic, the labels can be thought of as representing any partial heap, whereas the empty world ϵ specifically represents the empty heap.

Each type in the labeled system $A \in \texttt{Type}$ is labeled with a world $w \in \texttt{World}$, denoted $A \mathbin{@} w$ and called a *labeled type*. This allows the type system to track which resources are held by values or expressions of a given type and ensure that resources are managed correctly. The intuitive definition of $A \mathbin{@} w$ with respect to separation logic, is that the labeled type $A \mathbin{@} w$ for a given heap h representing world w represents the values of type A which satisfy h. If A were a proposition this would correspond to $h \models A$.

In the partial heap model of separation logic, disjoint partial heaps $h_1 \# h_2$ can be combined into a single larger heap $h_1 \uplus h_2$. Similarly, two worlds can be merged into a single larger world. However, as labels are variables, it is not possible to decide whether two labels represent disjoint partial heaps. Instead, the labeled system takes a top-down approach, and adds *constraints* which represent the fact that a world can be split into two disjoint worlds.

Definition 4.2.3 (Constraints)**:** A *constraint* is a triple of worlds $w_1, w_2, w_3 \in \texttt{World}$ such that world w_3 can be split into worlds w_1 and w_2, denoted $(w_1, w_2 \triangleright w_3)$.

In the partial heap interpretation of worlds, as in the model of separation logic, the constraint $(w_1, w_2 \triangleright w_3)$ is interpreted as w_1 and w_2 being disjoint heaps h_1 and h_2 which can be combined into a larger heap h_3. Hence, the $(w_1, w_2 \triangleright w_3)$ corresponds to the equality $h_1 \uplus h_2 = h_3$.

The type system for the labeled system tracks not only the types of variables, but also the defined labels and constraints on worlds. As such, the context is defined in 3 parts as follows:

Definition 4.2.4 (Contexts)**:**

$$\Sigma ::= \cdot \mid \Sigma, l \quad (\textit{Where } l \in \texttt{Label})$$
$$\Theta ::= \cdot \mid \Theta, (w_1, w_2 \triangleright w_3) \quad (\textit{Where } w_1, w_2, w_3 \in \texttt{World})$$
$$\Gamma ::= \cdot \mid \Gamma, x : A \mathbin{@} w \quad (\textit{Where } A \in \texttt{Type}\,, w \in \texttt{World})$$

The labeled system adds 2 additional contexts. The first is the Σ context. It is similar to the Γ context in STLC, but contains only variables of a special `Label` type, which is interpreted as partial heaps.

The second context is the Θ context, which is used to track the constraints between labels, describing the ways in which the corresponding partial heaps can be split and merged.

Finally, there is the variable context Γ, which is similar to that of STLC and $\lambda_{\texttt{loc}}$, but additionally labels each type A with a world w. The labeled type $A \mathbin{@} w$ can be interpreted as representing the values of type A which are valid for the partial heap described by world w.

Notation: The notation $\Omega := \Sigma; \Theta; \Gamma$ represents the contexts Σ, Θ and Γ. Additionally, we use the notations 'Ω, a', '$\Omega, (w_1, w_2 \triangleright w_3)$' and '$\Omega, x : A \mathbin{@} w$' to add labels, constraints and assumptions to Σ, Θ and Γ respectively. The notations $\Omega.\Sigma$, $\Omega.\Theta$ and $\Omega.\Gamma$ are used to refer to the Σ, Θ and Γ contexts of Ω respectively.

Notation: The set of worlds which are defined by a given label context Σ consists of the labels defined by Σ and the empty world and is denoted $\Sigma_\epsilon \stackrel{\texttt{def}}{=} \Sigma \cup \{\epsilon\}$

Similar to how typing an expression $e \in \texttt{Expr}$ in STLC is only possible in contexts Γ which give types to all the free variables in e, the typing judgment in the labeled system is only valid for contexts in which all the labels in both the constraint Θ and assumption Γ contexts are defined in the label context Σ. This condition is the well-formedness condition of contexts and is formally specified as follows:

Definition 4.2.5 (Constraint Context Well-formedness)**:**

$$\boxed{\Sigma \vdash \Theta\ \texttt{wf}}$$

STLC-SEP-THETA-EMP-WF
$$\Sigma \vdash \cdot\ \texttt{wf}$$

STLC-SEP-THETA-CONSTR-WF
$$\frac{\Sigma \vdash \Theta\ \texttt{wf} \qquad w_1, w_2, w_3 \in \Sigma_\epsilon}{\Sigma \vdash \Theta, (w_1, w_2 \triangleright w_3)\ \texttt{wf}}$$

Definition 4.2.6 (Assumption Context Well-formedness)**:**

$$\boxed{\Sigma \vdash \Gamma\ \texttt{wf}}$$

STLC-SEP-GAMMA-EMP-WF
$$\Sigma \vdash \cdot\ \texttt{wf}$$

STLC-SEP-GAMMA-ASSUMPTION-WF
$$\frac{\Sigma \vdash \Gamma\ \texttt{wf} \qquad w \in \Sigma_\epsilon \qquad \vdash A : \texttt{Stateful}}{\Sigma \vdash \Gamma, x : A \mathbin{@} w\ \texttt{wf}}$$

The additional assumption $\vdash A : \texttt{Stateful}$ ensures that all references $\texttt{Ref}\, A$ in the context contain only pure types A which may be stored on the heap.

Definition 4.2.7 (Context Well-formedness)**:**

$$\boxed{\Sigma; \Theta; \Gamma\ \texttt{wf}}$$

STLC-SEP-CTX-WF
$$\frac{\Sigma \vdash \Theta\ \texttt{wf} \qquad \Sigma \vdash \Gamma\ \texttt{wf}}{\Sigma; \Theta; \Gamma\ \texttt{wf}}$$

Using these three contexts, we define the typing judgment $\Omega \vdash e : A \mathbin{@} w$ as that the expression $e \in \texttt{Expr}$ has type $A \in \texttt{Type}$ in context Ω and is valid in world $w \in \texttt{World}$ by the following rules:

Definition 4.2.8 (Typing Rules)**:**

$$\boxed{\Omega \vdash e : A \mathbin{@} w}$$

Logical Rules:

STLC-SEP-VAR
$$\frac{x : A \mathbin{@} w \in \Omega.\Gamma}{\Omega \vdash x : A \mathbin{@} w}$$

STLC-SEP-A-UNIT-I
$$\frac{w \in \Omega.\Sigma_\epsilon}{\Omega \vdash [] : 1_a \mathbin{@} w}$$

STLC-SEP-A-FUN-I
$$\frac{w \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful} \qquad \Omega, x : A \mathbin{@} w \vdash e : B \mathbin{@} w}{\Omega \vdash \lambda x : A.\, e : A \Rightarrow B \mathbin{@} w}$$

STLC-SEP-A-FUN-E
$$\frac{\Omega \vdash e_1 : A \Rightarrow B \mathbin{@} w \qquad \Omega \vdash e_2 : A \mathbin{@} w}{\Omega \vdash e_1\, e_2 : B \mathbin{@} w}$$

STLC-SEP-A-PAIR-I
$$\frac{\Omega \vdash e_1 : A \mathbin{@} w \qquad \Omega \vdash e_2 : B \mathbin{@} w}{\Omega \vdash [e_1,\, e_2] : A \wedge B \mathbin{@} w}$$

STLC-SEP-A-PAIR-E
$$\frac{\Omega \vdash e_1 : A \wedge B \mathbin{@} w \qquad \Omega, x : A \mathbin{@} w, y : B \mathbin{@} w \vdash e_2 : C \mathbin{@} w'}{\Omega \vdash \texttt{let}\, [x,\, y] = e_1\, \texttt{in}\, e_2 : C \mathbin{@} w'}$$

STLC-SEP-M-UNIT-I
$$\Omega \vdash \langle\rangle : 1_m \mathbin{@} \epsilon$$

STLC-SEP-M-UNIT-E
$$\frac{\Omega \vdash e_1 : 1_m \mathbin{@} w' \qquad \Omega, (\epsilon, \epsilon \triangleright w') \vdash e_2 : C \mathbin{@} w}{\Omega \vdash \texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2 : C \mathbin{@} w}$$

STLC-SEP-M-FUN-I
$$\frac{w \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful} \qquad \Omega, a, c, (w, a \triangleright c), x : A \mathbin{@} a \vdash e : B \mathbin{@} c}{\Omega \vdash \lambda^* x : A.\, e : A \multimap B \mathbin{@} w}$$

STLC-SEP-M-FUN-E
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : A \multimap B \mathbin{@} w_1 \qquad \Omega \vdash e_2 : A \mathbin{@} w_2}{\Omega \vdash e_1^*\, e_2 : B \mathbin{@} w}$$

$$\text{STLC-SEP-M-PAIR-I}\quad \frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : A \mathbin{@} w_1 \qquad \Omega \vdash e_2 : B \mathbin{@} w_2}{\Omega \vdash \langle e_1,\, e_2 \rangle : A * B \mathbin{@} w}$$

$$\text{STLC-SEP-M-PAIR-E}\quad \frac{w' \in \Omega.\Sigma_\epsilon \qquad \Omega \vdash e_1 : A * B \mathbin{@} w \qquad \Omega, a, b, x : A \mathbin{@} a, y : B \mathbin{@} b, (a, b \triangleright w) \vdash e_2 : C \mathbin{@} w'}{\Omega \vdash \mathtt{let}\, \langle x,\, y \rangle = e_1\, \mathtt{in}\, e_2 : C \mathbin{@} w'}$$

$$\text{STLC-SEP-RETURN}\quad \frac{\Omega \vdash e : A \mathbin{@} w}{\Omega \vdash \mathtt{return}\, e : T\, A \mathbin{@} w}$$

$$\text{STLC-SEP-BIND}\quad \frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : T\, A \mathbin{@} w_1 \qquad \Omega, a, c, (w_2, a \triangleright c), x : A \mathbin{@} a \vdash e_2 : T\, B \mathbin{@} c}{\Omega \vdash \mathtt{let!}\, x = e_1\, \mathtt{in}\, e_2 : T\, B \mathbin{@} w}$$

$$\text{STLC-SEP-NEWREF}\quad \frac{\Omega \vdash e : A \mathbin{@} w \qquad \vdash A : \mathtt{Pure}}{\Omega \vdash \mathtt{ref}_A\, e : T\, (\mathtt{Ref}\, A) \mathbin{@} w}$$

$$\text{STLC-SEP-REPLACE}\quad \frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : \mathtt{Ref}\, A \mathbin{@} w_1 \qquad \Omega \vdash e_2 : B \mathbin{@} w_2 \qquad \vdash B : \mathtt{Pure}}{\Omega \vdash \mathtt{replace}_{A,B}\, e_1\, e_2 : T\, (A * \mathtt{Ref}\, B) \mathbin{@} w}$$

$$\text{STLC-SEP-FREE}\quad \frac{\Omega \vdash e : \mathtt{Ref}\, A \mathbin{@} w}{\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w}$$

Structural Rules:

$$\text{STLC-SEP-SYM}\quad \frac{(w_1, w_2 \triangleright w_3) \in \Omega.\Theta \qquad \Omega, (w_2, w_1 \triangleright w_3) \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

$$\text{STLC-SEP-UNIT-1}\quad \frac{w_1 \in \Omega.\Sigma_\epsilon \qquad \Omega, (w_1, \epsilon \triangleright w_1) \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

$$\text{STLC-SEP-UNIT-2}\quad \frac{(w_1, \epsilon \triangleright w_2) \in \Omega.\Theta \qquad \Omega, (w_2, \epsilon \triangleright w_1) \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

$$\text{STLC-SEP-CAST}\quad \frac{(w, \epsilon \triangleright w') \in \Omega.\Theta \qquad \Omega \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w'}$$

$$\text{STLC-SEP-ASSOC}\quad \frac{w \in \Omega.\Sigma_\epsilon \qquad (w_1, w_2 \triangleright w_{12}) \in \Omega.\Theta \qquad (w_{12}, w_3 \triangleright w_{123}) \in \Omega.\Theta \qquad \Omega, k, (w_2, w_3 \triangleright k), (w_1, k \triangleright w_{123}) \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

$$\text{STLC-SEP-ASSOC-UNIT-1}\quad \frac{(w_1, \epsilon \triangleright w_1') \in \Omega \qquad (w_1', w_2 \triangleright w_3) \in \Omega.\Theta \qquad \Omega, (w_1, w_2 \triangleright w_3) \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

$$\text{STLC-SEP-ASSOC-UNIT-2}\quad \frac{(w_3, \epsilon \triangleright w_3') \in \Omega.\Theta \qquad (w_1, w_2 \triangleright w_3) \in \Omega.\Theta \qquad \Omega, (w_1, w_2 \triangleright w_3') \vdash e : A \mathbin{@} w}{\Omega \vdash e : A \mathbin{@} w}$$

The labeled system has 2 types of typing rules: the *logical rules* which are dependent on the structure of the expression, and *structural rules* which reason about worlds without changing the expression.

Logical Rules. In the labeled system, assumptions are annotated with worlds, and the variable rule STLC-SEP-VAR can only be applied when the world of the assumption and conclusion are the same.

The introduction and elimination rules for the additive connectives $1_{\mathtt{a}}$, $\Rightarrow$ and $\wedge$ match those of the intuitionistic typing rules in Definition 3.2.4, except that they use the same world w for all added assumptions and conclusions. The only difference is the conclusion of STLC-SEP-A-PAIR-E, in which the conclusion can be of any labeled type $C \mathbin{@} w'$ in a world that is not necessarily w, for the same reason as that the type of the conclusion can be any type C, possibly unrelated to A or B in $\lambda_{\mathtt{loc}}$.

The STLC-SEP-M-UNIT-I rule ensures that any multiplicative unit is in the empty world ϵ. This corresponds to how the only heap that satisfies `emp` in separation logic is the empty heap. The STLC-SEP-M-UNIT-E rule uses the property that any multiplicative unit is in the empty world to add the constraint $(\epsilon, \epsilon \triangleright w')$, which corresponds to the equation $\emptyset \uplus \emptyset = h$ and is equivalent to $h = \emptyset$. By the structural rules, this means that types of world w' and ϵ can be used interchangeably.

The STLC-SEP-M-FUN-I rule can similarly be interpreted using the partial heap model of separation logic. It states that an abstraction is a multiplicative function $A \multimap B$ valid in a world w (represented by heap h_w) when both w is a valid world, for any two heaps h_a and h_c such that $h_w \uplus h_a = h_c$, and h_a is a valid heap for $x : A$, then h_c is a valid heap for expression $e : B$. This corresponds to the partial heap model $h \models A \mathrel{-\!\!*} B$ for separation logic. Both this rule and the STLC-SEP-A-FUN-I rule require that the type A is stateful to ensure that all references in A, and therefore also the extended context, contain only pure types that can be stored on the heap.

The STLC-SEP-M-FUN-E rule does the opposite, as it splits the world w into two worlds w_1 and w_2, where w_1 is valid for the multiplicative function, and w_2 is valid for the argument to conclude that the application is valid in w. This interpretation follows from reading the heap condition for $h \models A \multimap B$ backwards, namely where w represents the heap $h \uplus h_2$, w_1 represents h and w_2 represents h_2.

The STLC-SEP-M-PAIR-I and STLC-SEP-M-PAIR-E rules similarly follow from the heap interpretation of separation logic.

As the `return` operation does not change the heap, the world w in the assumption and the conclusion of the STLC-SEP-RETURN rule are the same.

The STLC-SEP-BIND rule is the only rule which 'executes' heap operations and therefore changes the heap. It is a combination of the STLC-SEP-M-FUN-E and STLC-SEP-M-FUN-I rules, where the world w is split into two worlds w_1 and w_2 for the monadic value and the continuation respectively. The main difference is that the monadic value $T\,A \mathbin{@} w_1$ is executed, resulting in a value $A \mathbin{@} a$, where the heap represented by world a is the heap after executing heap operations and is therefore not necessarily the same as w_1. In the continuation, we have access to the updated part of the heap a, and the unused part of the heap w_2.

The worlds w for the STLC-SEP-NEWREF, STLC-SEP-REPLACE and STLC-SEP-FREE rules represent the part of the heap which is updated by the operation (before the operation). From the STLC-SEP-NEWREF rule, it follows that the `Ref` A type describes not only the new location, but also the resources owned by the value $A \mathbin{@} w$ at that location. As in $\lambda_{\texttt{loc}}$, the types stored on the heap must be pure. The types of the references `Ref` A and `Ref` B are additionally forced to be the same types as specified by the operations. This ensures that the types of the values in the heap match up, and the operations do not get stuck, unlike in $\lambda_{\texttt{loc}}$. Unlike in the base language, `replace` not only returns the value previously stored at the location, but also the location itself, updated with the newly stored type. This is because reference types are linear. As all types in well-formed contexts are stateful, reference types contain only pure types, meaning that only the types to be stored have to be checked as pure in the STLC-SEP-NEWREF and STLC-SEP-REPLACE rules.

Structural Rules. In addition to the logical rules, the labeled system has structural rules, which are used to reason about worlds, rather than the expressions themselves. The STLC-SEP-SYM and STLC-SEP-ASSOC rules reflect that disjoint union is both commutative and associative. The STLC-SEP-ASSOC-UNIT-1 and STLC-SEP-ASSOC-UNIT-2 rules are special cases of the STLC-SEP-ASSOC rule in which one of the worlds is the empty world ϵ and hence we can use the existing label w_2 instead of k.

The STLC-SEP-UNIT-1 and STLC-SEP-UNIT-2 rules reflect that the empty heap (represented by world ϵ) is a unit of the disjoint union. Finally, the STLC-SEP-CAST rule uses the property that the empty heap is a unit of the disjoint union operation to allow the worlds w and w' to be used interchangeably by casting types from world w to world w'.

The main difference between the structural rules by Hóu et al. [2015] and our labeled system is the choice of rules for equal worlds represented by the constraint $(w_1, \epsilon \triangleright w_2)$. Whereas their system uses a substitution rule for labels to handle multiple conclusions, we opt for a casting approach similar to the subsumption rule for subtyping [Pierce 2002]. Additionally we add special cases of the STLC-SEP-ASSOC rule to model the substitution in constraints. This simplifies the proof that well-typed substitutions preserve well-typedness, as substitution of a specific label does not commute with general label substitution.

Example 4.2.9 (Associativity)**:** A simple example program which uses the structural rules is the function which associates a multiplicative triple from the right to the left. As the resources of the argument and result are the same, we use an additive function instead of a multiplicative function.

$$\texttt{assoc} \stackrel{\text{def}}{=} \lambda xyz : (A * (B * C)).\texttt{let}\,\langle x,\, yz\rangle = xyz\,\texttt{in} \quad (1)$$
$$\texttt{let}\,\langle y,\, z\rangle = yz\,\texttt{in} \quad (2)$$
$$\langle\langle x,\, y\rangle,\, z\rangle \quad (3)$$

As typing derivations in the labeled system are large due to the size of the context, will instead describe the steps of the derivation $a \vdash \texttt{assoc} : (A * (B * C)) \Rightarrow (A * B) * C \mathbin{@} a$ rather than giving the derivation itself. The additional label a ensures that `assoc` is typeable in any world. The steps of the derivation are as follows:

1. First the rule STLC-SEP-A-FUN-I is applied, which adds $xyz : A * (B * C) \mathbin{@} a$ to the context.
2. Next the rule STLC-SEP-M-PAIR-E is used to eliminate the `let` binding on line 1. We use $w = a$ and $w' = a$. This results in two new labels b, c, the constraint $(b, c \triangleright a)$ and the assumptions $x : A \mathbin{@} b$ and $yz : B * C \mathbin{@} c$.
3. The STLC-SEP-M-PAIR-E rule is used again to eliminate the `let` binding on line 2. We use $w = c$ and $w' = a$. This results in two new labels d, e, the constraint $(d, e \triangleright c)$ and the assumptions $y : B \mathbin{@} d$ and $z : C \mathbin{@} e$.
4. The STLC-SEP-M-PAIR-I rule cannot immediately be applied, as there is no constraint which splits the world a into e and another world. Instead we apply STLC-SEP-ASSOC which creates a new label f, and constraints $(b, d \triangleright f)$ and $(f, e \triangleright a)$.
5. The rule STLC-SEP-M-PAIR-I is applied using the constraint $(f, e \triangleright a)$, and the right hand side is typed using the STLC-SEP-VAR rule.
6. Finally we must type $\langle x,\, y\rangle : A * B \mathbin{@} f$, which is done using the STLC-SEP-M-PAIR-I rule and $(b, d \triangleright f)$ constraint.

These steps for deriving $a \vdash \texttt{assoc} : (A * (B * C)) \Rightarrow (A * B) * C \mathbin{@} a$ are more succinctly denoted in the following table:

Typed Expression	Rule	Added to Context
$\lambda xyz : A * (B * C).\ \ldots\ : (A * B) * C \mathbin{@} a$	STLC-SEP-A-FUN-I	$xyz : A * (B * C) \mathbin{@} a$
$\texttt{let}\,\langle x,\, yz\rangle = xyz\,\texttt{in}\ \ldots : (A * B) * C \mathbin{@} a$	STLC-SEP-M-PAIR-E	$b, cd, (b, cd \triangleright a), x : A \mathbin{@} b, yz : B * C \mathbin{@} c$
$-\ xyz : A * (B * C) \mathbin{@} a$	STLC-SEP-VAR	
$\texttt{let}\,\langle y,\, z\rangle = yz\,\texttt{in}\ \ldots : (A * B) * C \mathbin{@} a$	STLC-SEP-M-PAIR-E	$d, e, (d, e \triangleright c), y : B \mathbin{@} d, z : C \mathbin{@} e$
$-\ yz : B * C \mathbin{@} c$	STLC-SEP-VAR	
	STLC-SEP-ASSOC	$f, (b, d \triangleright f), (f, e \triangleright a)$
$\langle\langle x,\, y\rangle,\, z\rangle : (A * B) * C \mathbin{@} a$	STLC-SEP-M-PAIR-I	
$-\ \langle x,\, y\rangle : A * B \mathbin{@} f$	STLC-SEP-M-PAIR-I	
$-\ -\ x : A \mathbin{@} b$	STLC-SEP-VAR	
$-\ -\ y : B \mathbin{@} d$	STLC-SEP-VAR	
$-\ z : C \mathbin{@} e$	STLC-SEP-VAR	

In the table, the first column represents the expression and the labeled type with which it should be typed. The second column states the applied rule for the expression, and the third column the labels, constraints and assumptions added to the context during the typing of this part of the expression. As examples may be several lines long, the table is split into sections, one for each line of the program, with ... representing the remaining lines. The typing of subexpressions is denoted by an indent '–'. The application of structural rules is denoted by a line with no expression.

Next we consider the function duplicate from simply typed $\lambda_{\texttt{loc}}$, written both using a multiplicative and additive pair:

Example 4.2.10 (Duplicate)**:**

$$\texttt{duplicate\_a} \stackrel{\text{def}}{=} \lambda x : \texttt{Ref}\,1_{\mathsf{m}}.\,[x,\, x] \quad \text{(Additive)}$$
$$\texttt{duplicate\_m} \stackrel{\text{def}}{=} \lambda x : \texttt{Ref}\,1_{\mathsf{m}}.\,\langle x,\, x\rangle \quad \text{(Multiplicative)}$$

In the labeled typing system, the additive version can be typed $a \vdash \texttt{duplicate\_a} : \texttt{Ref}\, 1_{\mathsf{m}} \Rightarrow \texttt{Ref}\, 1_{\mathsf{m}} \wedge \texttt{Ref}\, 1_{\mathsf{m}}$, but the multiplicative version cannot. The additive version is typed as follows:

Typed Expression	Rule	Added to Context
$\lambda x : \texttt{Ref}\, 1_{\mathsf{m}}.\, [x,\, x]$: $\texttt{Ref}\, 1_{\mathsf{m}} \Rightarrow \texttt{Ref}\, 1_{\mathsf{m}} \wedge \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-A-FUN-I	$x : \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$
$-\, [x,\, x]$ $: \texttt{Ref}\, 1_{\mathsf{m}} \wedge \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-A-PAIR-I	
$-\,-\, x$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-VAR	
$-\,-\, x$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-VAR	

Attempting to type the multiplicative version `duplicate_m` results in the following table corresponding to the typing derivation:

Typed Expression	Rule	Added to Context
$\lambda x : \texttt{Ref}\, 1_{\mathsf{m}}.\, \langle x,\, x\rangle$: $\texttt{Ref}\, 1_{\mathsf{m}} \Rightarrow \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-A-FUN-I	$x : \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$
	'no rule exists'	$(a, a \triangleright a)$
$-\, \langle x,\, x\rangle$ $: \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-M-PAIR-I	
$-\,-\, x$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-VAR	
$-\,-\, x$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-VAR	

Unlike in the additive case, the STLC-SEP-M-PAIR-I rule has an additional assumption $(a, a \triangleright a)$. Hence, we would need a combination of structural rules at the placeholder 'no rule exists' to add such a constraint to the context, however, no such combination is possible. In the partial heap model, such a rule would imply that for any heap h, $h \uplus h = h$, which only holds for the empty heap $\emptyset$, and therefore not for any heap describing a reference.

Similarly, the `free_pair` function can only be typed when using a multiplicative pair as follows:

$$\begin{aligned}\texttt{free\_pair} \stackrel{\text{def}}{=} \lambda p : \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}}.\, &\texttt{let}\, \langle l_1,\, l_2\rangle = p\, \texttt{in}\\ &\texttt{let!}\, x = \texttt{free}_{1_{\mathsf{m}}}\, l_1\, \texttt{in}\\ &\texttt{let!}\, y = \texttt{free}_{1_{\mathsf{m}}}\, l_2\, \texttt{in}\\ &\texttt{return}\, \langle x,\, y\rangle\end{aligned}$$

When attempting to type check the `let!` binding, the resources of the first expression are used up, and replaced by the resources of the return value. As such, a reference with the same resources (such as the same location), cannot be used multiple times. This is enforced in the STLC-SEP-BIND rule by splitting the world into two resources w_1 and w_2, and ensuring that the only way to create the result world is to only use the resources not used by the first expression w_2 and the resources of the result a. In the case of `free_pair`, this means that the world for l_1 cannot be reused for l_2. A typing derivation for `free_pair` can be constructed as follows:

Typed Expression	Rule	Added to Context
$\lambda p : \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}}.\, \ldots : \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}} \Rightarrow T\,(1_{\mathsf{m}} * 1_{\mathsf{m}}) \mathbin{@} a$	STLC-SEP-A-FUN-I	$p : \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$
$\texttt{let}\, \langle l_1,\, l_2\rangle = p\, \texttt{in}\, \ldots$ $: T\,(1_{\mathsf{m}} * 1_{\mathsf{m}}) \mathbin{@} a$	STLC-SEP-M-PAIR-E	$b, c, (b, c \triangleright a), l_1 : \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} b, l_2 : \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} c$
$-\, p$ $: \texttt{Ref}\, 1_{\mathsf{m}} * \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} a$	STLC-SEP-VAR	
$\texttt{let!}\, x = \texttt{free}_{1_{\mathsf{m}}}\, l_1\, \texttt{in}\, \ldots$ $: T\,(1_{\mathsf{m}} * 1_{\mathsf{m}}) \mathbin{@} a$	STLC-SEP-BIND	$d, a', (d, c \triangleright a'), x : 1_{\mathsf{m}} \mathbin{@} d$
$-\, \texttt{free}_{1_{\mathsf{m}}}\, l_1$ $: T\, 1_{\mathsf{m}} \mathbin{@} b$	STLC-SEP-FREE	
$-\,-\, l_1$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} b$	STLC-SEP-VAR	
	STLC-SEP-SYM	$(c, d \triangleright a')$
$\texttt{let!}\, y = \texttt{free}_{1_{\mathsf{m}}}\, l_2\, \texttt{in}\, \ldots$ $: T\,(1_{\mathsf{m}} * 1_{\mathsf{m}}) \mathbin{@} a'$	STLC-SEP-BIND	$e, a'', (e, d \triangleright a''), y : 1_{\mathsf{m}} \mathbin{@} e$
$-\, \texttt{free}_{1_{\mathsf{m}}}\, l_2$ $: T\, 1_{\mathsf{m}} \mathbin{@} c$	STLC-SEP-FREE	
$-\,-\, l_2$ $: \texttt{Ref}\, 1_{\mathsf{m}} \mathbin{@} c$	STLC-SEP-VAR	
	STLC-SEP-SYM	$(d, e \triangleright a'')$
$\texttt{return}\, \langle x,\, y\rangle$ $: T\,(1_{\mathsf{m}} * 1_{\mathsf{m}}) \mathbin{@} a''$	STLC-SEP-RETURN	
$-\, \langle x,\, y\rangle$ $: 1_{\mathsf{m}} * 1_{\mathsf{m}} \mathbin{@} a''$	STLC-SEP-M-PAIR-I	
$-\,-\, x$ $: 1_{\mathsf{m}} \mathbin{@} d$	STLC-SEP-VAR	
$-\,-\, y$ $: 1_{\mathsf{m}} \mathbin{@} e$	STLC-SEP-VAR	

As `duplicate` can only create an additive pair, whereas `free_pair` requires a multiplicative pair, the `unsafe` program typeable in $\lambda_{\texttt{loc}}$ cannot be typed in λ_*. The `safe` program which uses the multiplicative pair can however be typed in both $\lambda_{\texttt{loc}}$ and λ_*. A typeable definition of the `safe` program in λ_* is as follows:

$$
\begin{aligned}
\texttt{safe} \stackrel{\text{def}}{=} \lambda u : 1_{\mathsf{m}}.\, & \texttt{let}\,\langle\rangle = u\,\texttt{in} \\
& \texttt{let!}\, l_1 = \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle\,\texttt{in} \\
& \texttt{let!}\, l_2 = \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle\,\texttt{in} \\
& \texttt{let!}\, r = \texttt{free\_pair}\,\langle l_1, l_2\rangle\,\texttt{in} \\
& \texttt{let}\,\langle x, y\rangle = r\,\texttt{in} \\
& \texttt{let}\,\langle\rangle = x\,\texttt{in} \\
& \texttt{let}\,\langle\rangle = y\,\texttt{in} \\
& \texttt{return}\,()
\end{aligned}
$$

The definition of `safe` is similar to that for $\lambda_{\texttt{loc}}$, but requires the additional lines in blue to prove that u and r do not describe any resources. The typing derivation for `safe` is depicted by the following table:

Typed Expression		Rule	Added to Context
$\lambda u : 1_{\mathsf{m}}. \ldots$	$: 1_{\mathsf{m}} \Rightarrow T\,1_{\mathsf{m}}\,@\,a$	STLC-SEP-A-FUN-I	$u : 1_{\mathsf{m}}\,@\,a$
$\texttt{let}\,\langle\rangle = u\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,a$	STLC-SEP-M-UNIT-E	$(\epsilon, \epsilon \triangleright a)$
$- u$	$: 1_{\mathsf{m}}\,@\,a$	STLC-SEP-VAR	
$\texttt{let!}\, l_1 = \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-BIND	$b, a', (b, \epsilon \triangleright a'), l_1 : \texttt{Ref}\,1_{\mathsf{m}}\,@\,b$
$- \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle$	$: \texttt{Ref}\,1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-NEWREF	
$- - \langle\rangle$	$: 1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-M-UNIT-I	
		STLC-SEP-SYM	$(\epsilon, b \triangleright a')$
$\texttt{let!}\, l_2 = \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,a'$	STLC-SEP-BIND	$c, a'', (c, b \triangleright a''), l_2 : \texttt{Ref}\,1_{\mathsf{m}}\,@\,c$
$- \texttt{ref}_{1_{\mathsf{m}}}\,\langle\rangle$	$: \texttt{Ref}\,1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-NEWREF	
$- - \langle\rangle$	$: 1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-M-UNIT-I	
		STLC-SEP-UNIT-1	$(a'', \epsilon \triangleright a'')$
		STLC-SEP-SYM	$(b, c \triangleright a'')$
$\texttt{let!}\, r = \texttt{free\_pair}\,\langle l_1, l_2\rangle\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,a''$	STLC-SEP-BIND	$d, a''', (d, \epsilon \triangleright a'''), r : 1_{\mathsf{m}} * 1_{\mathsf{m}}\,@\,d$
$- \texttt{free\_pair}\,\langle l_1, l_2\rangle$	$: T\,(1_{\mathsf{m}} * 1_{\mathsf{m}})\,@\,a''$	STLC-SEP-VAR	
$- - \texttt{free\_pair}$	$: \texttt{Ref}\,1_{\mathsf{m}} * \texttt{Ref}\,1_{\mathsf{m}} \Rightarrow T\,(1_{\mathsf{m}} * 1_{\mathsf{m}})\,@\,a''$	*By derivation above*	
$- - \langle l_1, l_2\rangle$	$: \texttt{Ref}\,1_{\mathsf{m}} * \texttt{Ref}\,1_{\mathsf{m}}\,@\,a''$	STLC-SEP-VAR	
$- - - l_1$	$: \texttt{Ref}\,1_{\mathsf{m}}\,@\,b$	STLC-SEP-VAR	
$- - - l_2$	$: \texttt{Ref}\,1_{\mathsf{m}}\,@\,c$	STLC-SEP-VAR	
		STLC-SEP-CAST	*using* $(d, \epsilon \triangleright a''')$
$\texttt{let}\,\langle x, y\rangle = r\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,d$	STLC-SEP-M-PAIR-E	$e, f, (e, f \triangleright d), x : 1_{\mathsf{m}}\,@\,e, y : 1_{\mathsf{m}}\,@\,f$
$- r$	$: 1_{\mathsf{m}} * 1_{\mathsf{m}}\,@\,d$	STLC-SEP-VAR	
$\texttt{let}\,\langle\rangle = x\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,d$	STLC-SEP-M-UNIT-E	$(\epsilon, \epsilon \triangleright e)$
$- x$	$: 1_{\mathsf{m}}\,@\,e$	STLC-SEP-VAR	
$\texttt{let}\,\langle\rangle = y\,\texttt{in} \ldots$	$: T\,1_{\mathsf{m}}\,@\,d$	STLC-SEP-M-UNIT-E	$(\epsilon, \epsilon \triangleright f)$
$- y$	$: 1_{\mathsf{m}}\,@\,f$	STLC-SEP-VAR	
		STLC-SEP-ASSOC-UNIT-1	$(\epsilon, f \triangleright d)$, $(\epsilon, \epsilon \triangleright d)$
		STLC-SEP-CAST	*using* $(\epsilon, \epsilon \triangleright d)$
$\texttt{return}\,\langle\rangle$	$: T\,1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-RETURN	
$- \langle\rangle$	$: 1_{\mathsf{m}}\,@\,\epsilon$	STLC-SEP-M-UNIT-I	

In the derivation above, we type `free_pair` in the world a'', whereas in the earlier derivation of `free_pair`, we assumed a general world a. One method for resolving this is to redo the derivation of `free_pair` using the world a'' and the current context instead of a. Another method would be to use thinning to remove all but the label a'' from the context, and then use substitution to substitute a'' for a in the derivation of `free_pair`, similar to applying thinning and substitution for variables in the simply typed $\lambda_{\texttt{loc}}$. The specific definitions and lemmas associated with thinning and substitution of λ_* are given in the next section (§ 4.3).

An important result which follows immediately from the typing rules is that when typing in a well-formed context, the result world is always present in the context, and the result type is stateful:

Lemma 4.2.11 (Result Type and World Exist)**:** If Ω `wf` and $\Omega \vdash e : A\,@\,w$ then $w \in \Omega.\Sigma_\epsilon$ and $\vdash A : \texttt{Stateful}$.

Proof. By induction on the typing derivation. In the inductive cases, we ensure that the contexts in the assumptions are also well-formed.

For instance, in STLC-SEP-A-FUN-I and STLC-SEP-M-FUN-I the assumptions $w \in \Omega.\Sigma_\epsilon$ and $\vdash A :$ `Stateful` are used to ensure that $\Omega, x : A\,@\,w \vdash e : B\,@\,w$ `wf` and $\Omega, a, c, (w, a \triangleright c), x : A\,@\,a \vdash e : B$ `wf`

hold respectively using the STLC-SEP-GAMMA-ASSUMPTION-WF rule and STLC-SEP-THETA-CONSTR-WF in the case of STLC-SEP-M-FUN-I.

In rules such as STLC-SEP-NEWREF, the assumption $\vdash A : \texttt{Pure}$ ensures that the result type $T\,(\texttt{Ref}\,A)$ is stateful.

In rules such as STLC-SEP-M-PAIR-I the result world w exists as $\Omega.\Sigma\ \texttt{wf}\Omega.\Theta$ and w is in a constraint $(w_1, w_2 \triangleright w) \in \Omega.\Theta$. □

Remark: From this point on we assume that typing is always done in a well-formed context. More specifically, any derivation of the form $\Omega \vdash e : A \mathbin{@} w$ assumes that $\Omega\ \texttt{wf}$.

4.3 Substitution and Equivalences

As in the $\lambda_{\texttt{loc}}$, the β-equivalences in λ_* require a definition of substitution which preserves well-typedness. The thinning and substitutions are defined as judgments between contexts, rather than as weakening (single variable thinning) and single-variable substitution. In the case of the λ_*, using a simultaneous substitution is especially important, as the multiplicative abstraction rule STLC-SEP-M-FUN-I introduces not only a new variable, but also two new labels and a constraint. Hence, for defining the β-equivalence for multiplicative functions, not only the variable, but also the labels and constraints must be substituted simultaneously.

By the remark above, we only consider derivations in well-typed contexts, and therefore also only define thinning and substitutions in terms of well-typed contexts. The thinning judgment for the labeled calculus has the same form as that of the unlabeled calculus, but additionally allows for adding new labels and constraints.

Definition 4.3.1 (Thinning)**:**

THINNING-EMP
$$\cdot \supseteq \cdot$$

THINNING-SIGMA-TAKE
$$\frac{\Omega \supseteq \Omega'}{\Omega, a \supseteq \Omega', a}$$

THINNING-SIGMA-DROP
$$\frac{\Omega \supseteq \Omega'}{\Omega, a \supseteq \Omega'}$$

THINNING-THETA-TAKE
$$\frac{\Omega \supseteq \Omega' \qquad w_1, w_2, w_3 \in \Omega'.\Theta}{\Omega, (w_1, w_2 \triangleright w_3) \supseteq \Omega', (w_1, w_2 \triangleright w_3)}$$

THINNING-THETA-DROP
$$\frac{\Omega \supseteq \Omega' \qquad w_1, w_2, w_3 \in \Omega.\Theta}{\Omega, (w_1, w_2 \triangleright w_3) \supseteq \Omega'}$$

THINNING-GAMMA-TAKE
$$\frac{\Omega \supseteq \Omega' \qquad w \in \Omega'.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful}}{\Omega, x : A \mathbin{@} w \supseteq \Omega', x : A \mathbin{@} w}$$

THINNING-GAMMA-DROP
$$\frac{\Omega \supseteq \Omega' \qquad w \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful}}{\Omega, x : A \mathbin{@} w \supseteq \Omega'}$$

In $\lambda_{\texttt{loc}}$ there was only a single context Γ, which could be extended with variables. In λ_*, thinnings can be extended with labels (THINNING-SIGMA-TAKE, THINNING-SIGMA-DROP), constraints (THINNING-THETA-TAKE, THINNING-THETA-DROP) and labeled assumptions (THINNING-GAMMA-TAKE, THINNING-GAMMA-DROP). The additional assumptions for THINNING-THETA-TAKE and THINNING-GAMMA-TAKE ensure that the stronger context Ω' remains well-formed, whereas the THINNING-THETA-DROP and THINNING-GAMMA-DROP assumptions ensure that the larger context Ω is well-formed.

Lemma 4.3.2 (Thinning well-formed)**:** If $\Omega \supseteq \Omega'$ then $\Omega\ \texttt{wf}$ and $\Omega'\ \texttt{wf}$.

Proof. By induction on the derivation of $\Omega \supseteq \Omega'$. □

The most common thinning is the identity thinning, which does not add any labels, constraints or variables. Due to the additional constraints, the identity thinning does not immediately follow from the definition of the context, as thinnings are only defined for well-formed contexts. Instead, the identity thinning follows from the definition of well-formed contexts.

Lemma 4.3.3 (Identity Thinning)**:** If $\Omega\ \texttt{wf}$ then $\Omega \supseteq \Omega$.

Proof. By induction on the derivation of $\Omega\ \texttt{wf}$. □

As in the simply typed $\lambda_{\texttt{loc}}$, thinning a context preserves well-typedness of expressions.

Lemma 4.3.4 (Thinning Lemma)**:** If $\Omega \vdash e : A \mathbin{@} w$ and $\Omega' \supseteq \Omega$ then $\Omega' \vdash e : A \mathbin{@} w$ (denoted $\Omega' \supseteq \Omega; \Omega \vdash e : A \mathbin{@} w$).

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$. The proof uses the following 3 sub-lemmas, each proven by induction on the thinning:
(i) If $a : \texttt{Label} \in \Omega'.\Sigma$ and $\Omega \supseteq \Omega'$ then $a \in \Omega.\Sigma$
(ii) If $(w_1, w_2 \triangleright w_3) \in \Omega'.\Theta$ and $\Omega \supseteq \Omega'$ then $(w_1, w_2 \triangleright w_3) \in \Omega.\Theta$
(iii) If $x : A \mathbin{@} w \in \Omega'.\Gamma$ and $\Omega \supseteq \Omega'$ then $x : A \mathbin{@} w \in \Omega.\Gamma$ □

Remark: By the last remark of the previous section, the thinning lemma also includes the assumption $\Omega\ \texttt{wf}$ and has the conclusion $\Omega'\ \texttt{wf}$. In this specific case, both are well-formed due to the assumption $\Omega' \supseteq \Omega$.

In the labeled system, not only variables, but also labels can be substituted, which is reflected in the syntactic substitution:

Definition 4.3.5 (Syntactic Substitution)**:** Define $\delta ::= \delta_v, \delta_l$ where $\delta_v : \texttt{Var} \xrightarrow{\text{fin}} \texttt{Expr}$ and $\delta_l : \texttt{Label} \xrightarrow{\text{fin}} \texttt{World}$.

The syntactic substitution consists of a substitution on variables δ_v, similar to the unlabeled calculus, and a substitution on labels δ_l, which maps labels in one context to worlds in another context.

Remark: For both the syntactic variable and label substitution, the empty substitution $\emptyset$, identity substitution I and update substitution $\delta_v \langle x \mapsto e \rangle$ and $\delta_l \langle a \mapsto w \rangle$ are defined the same as for $\lambda_{\texttt{loc}}$.

The substitution of variables can be applied to expressions:

Definition 4.3.6 (Expression Substitution)**:**

$$[\delta_v]x \stackrel{\text{def}}{=} \delta_v\, x \qquad [\delta_v][] \stackrel{\text{def}}{=} []$$
$$[\delta_v]\langle\rangle \stackrel{\text{def}}{=} \langle\rangle \qquad [\delta_v](\texttt{let}\,\langle\rangle = e_1\,\texttt{in}\,e_2) \stackrel{\text{def}}{=} \texttt{let}\,\langle\rangle = [\delta_v]e_1\,\texttt{in}\,[\delta_v]e_2$$
$$[\delta_v](\lambda x : A.\, e) \stackrel{\text{def}}{=} \lambda x : A.\, [\delta_v\, \langle x \mapsto x\rangle]e \qquad [\delta_v](\lambda^* x : A.\, e) \stackrel{\text{def}}{=} \lambda^* x : A.\, [\delta_v\, \langle x \mapsto x\rangle]e$$
$$[\delta_v](e_1\, e_2) \stackrel{\text{def}}{=} ([\delta_v]e_1)\,([\delta_v]e_2) \qquad [\delta_v](e_1^*\, e_2) \stackrel{\text{def}}{=} ([\delta_v]e_1)^*\,([\delta_v]e_2)$$
$$[\delta_v]([e_1,\, e_2]) \stackrel{\text{def}}{=} [[\delta_v]e_1,\, [\delta_v]e_2] \qquad [\delta_v]\langle e_1,\, e_2\rangle \stackrel{\text{def}}{=} \langle[\delta_v]e_1,\, [\delta_v]e_2\rangle$$
$$[\delta_v](\texttt{let}\,[x,\, y] = e_1\,\texttt{in}\,e_2) \stackrel{\text{def}}{=} (\texttt{let}\,[x,\, y] = [\delta_v]e_1\,\texttt{in}\,[\delta_v\, \langle x \mapsto x, y \mapsto y\rangle]e_2)$$
$$[\delta_v](\texttt{let}\,\langle x,\, y\rangle = e_1\,\texttt{in}\,e_2) \stackrel{\text{def}}{=} (\texttt{let}\,\langle x,\, y\rangle = [\delta_v]e_1\,\texttt{in}\,[\delta_v\, \langle x \mapsto x, y \mapsto y\rangle]e_2)$$

Similar to how the substitution of variables is extended to substituting expressions, the substitution of labels is extended to substituting worlds.

Definition 4.3.7 (World Substitution)**:**

$$[\delta_l]\epsilon \stackrel{\text{def}}{=} \epsilon \qquad [\delta_l]a \stackrel{\text{def}}{=} \delta_l\, a$$

When substituting a world, the substitution maps labels to worlds according to the substitution, but maps the empty world ϵ to the empty world. This is because the empty world ϵ corresponds to the constant empty heap $\emptyset$ and therefore should not be substituted. Intuitively, the world ϵ can be thought of as similar to the constructor $\langle\rangle$ for a unit, which under substitution also remains unchanged.

With these definitions of syntactic variable (δ_v) and labels (δ_l) substitutions, the substitution judgment is defined as:

Definition 4.3.8 (Typed Substitutions)**:**

$$\text{SUBST-NIL}\ \frac{\Omega'\ \texttt{wf}}{\Omega' \vdash \emptyset, \emptyset : \cdot} \qquad \text{SUBST-SIGMA}\ \frac{\Omega' \vdash \delta_v, \delta_l : \Omega \qquad w \in \Omega'.\Sigma_\epsilon}{\Omega' \vdash \delta_v, \delta_l \langle a \mapsto w \rangle : \Omega, a}$$

$$\text{SUBST-THETA}\ \frac{\Omega' \vdash \delta_v, \delta_l : \Omega \qquad w_1, w_2, w_3 \in \Omega.\Sigma_\epsilon \qquad ([\delta_l]w_1, [\delta_l]w_2 \triangleright [\delta_l]w_3) \in \Omega'.\Theta}{\Omega' \vdash \delta_v, \delta_l : \Omega, (w_1, w_2 \triangleright w_3)}$$

$$\text{SUBST-GAMMA}\ \frac{\Omega' \vdash \delta_v, \delta_l : \Omega \qquad w \in \Omega.\Sigma_\epsilon \qquad \Omega' \vdash [\delta_v]e : A \mathbin{@} [\delta_l]w}{\Omega' \vdash \delta_v \langle x \mapsto e \rangle, \delta_l : \Omega, x : A \mathbin{@} w}$$

The substitution judgment is slightly different from that of $\lambda_{\texttt{loc}}$. The first difference is that the empty substitution requires the context to be well-formed. This ensures that applying the empty substitution to a typing derivation results in a well-formed derivation, as per the last remark of the previous section. In $\lambda_{\texttt{loc}}$, any context that can be constructed from the grammar is well-formed, so the well-formed context assumption is always true. The second difference is that labels can be added to the substitution with SUBST-SIGMA, which requires that the world w is a valid world in the context Ω'. This assumption is also required for the SUBST-GAMMA rule, but follows from the other assumptions and Lemma 4.2.11. The last difference is the addition of the SUBST-THETA rule, which ensures that any constraint in the right context also holds in the left context after substituting. As a result, the derivation rules using constraints are still valid after substituting labels.

The approach to proving that substitutions preserve typing derivations is similar to $\lambda_{\texttt{loc}}$. We first prove that substitutions can be weakened, then use this to show how substitutions can be extended with new variables, labels and constraints, and finally prove the substitution theorem itself.

Lemma 4.3.9 (Thinning Substitution)**:** If $\Omega_1 \supseteq \Omega'$ and $\Omega' \vdash \delta_v, \delta_l : \Omega$ then $\Omega_1 \vdash \delta_v, \delta_l : \Omega$ (denoted $\Omega_1 \supseteq \Omega'; \Omega' \vdash \delta_v, \delta_l : \Omega$).

Proof. By induction on the derivation of $\Omega' \vdash \delta_v, \delta_l : \Omega$ followed by applying the thinning lemma (Lemma 4.3.4) and applicable sub-lemmas in each case. □

Lemma 4.3.10 (Substitution Abstractions)**:** If $\Omega' \vdash \delta_v, \delta_l : \Omega$ then:

1. $\Omega', a \vdash \delta_v, \delta_l \langle a \mapsto a \rangle : \Omega, a$
2. $\Omega', ([\delta_l]w_1, [\delta_l]w_2 \triangleright [\delta_l]w_3) \vdash \delta_v, \delta_l : \Omega, (w_1, w_2 \triangleright w_3)$ for all $w_1, w_2, w_3 \in \Sigma_\epsilon$
3. $\Omega', x : A \mathbin{@} [\delta_l]w \vdash \delta_v \langle x \mapsto x \rangle, \delta_l : \Omega, x : A \mathbin{@} w$ for all $w \in \Sigma_\epsilon$

Proof. The proofs of all 3 theorems consist of applying Lemma 4.3.9 followed by applying the corresponding derivation rule in Definition 4.3.8. □

Theorem 4.3.11 (Derivation Substitution)**:** If $\Omega' \vdash \delta_v, \delta_l : \Omega$ and $\Omega \vdash e : A \mathbin{@} w$ then $\Omega' \vdash [\delta_v]e : A \mathbin{@} [\delta_l]w$ (denoted $\Omega' \vdash \delta_v, \delta_l : \Omega; \Omega \vdash e : A \mathbin{@} w$).

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$ and applying the abstraction lemmas Lemma 4.3.10 in cases that introduce new variables, labels or constraints. □

The combination of the thinning and substitution lemmas allow for reusing typing derivations in different contexts. For instance, in the case of `safe`, we can first use thinning to remove all but the label a'' from the context, and then use the substitution $\delta_v, \delta_l = [], [a \mapsto a'']$ to turn the derivation $a \vdash \texttt{free\_pair} : \texttt{Ref}\ 1_\texttt{m} * \texttt{Ref}\ 1_\texttt{m} \Rightarrow T\,(1_\texttt{m} * 1_\texttt{m}) \mathbin{@} a$ into $a'' \vdash \texttt{free\_pair} : \texttt{Ref}\ 1_\texttt{m} * \texttt{Ref}\ 1_\texttt{m} \Rightarrow T\,(1_\texttt{m} * 1_\texttt{m}) \mathbin{@} a''$, which can then be used in the derivation of `safe`.

As the substitution judgments allow for substituting variables, labels and constraints simultaneously, they can be used to define the β equivalences for the labeled system, for instance by substituting w_2 for a, w_3 for c and e_2 for x in the case of STLC-SEP-M-FUN-I followed by STLC-SEP-M-FUN-E. The β and η equivalence rules for the labeled system are as follows:

Definition 4.3.12 (Equivalences)**:** For a well formed context $\Omega\ \mathtt{wf}$, the equivalence judgment $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$ states that e_2 and e_2 are equivalent expressions of type A at world w in context Ω. The β and η equivalence rules are as follows:

$$\boxed{\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w}$$

STLC-SEP-EQ-A-FUN-BETA
$$\frac{\Omega, x : A \mathbin{@} w \vdash e_1 : B \qquad \Omega \vdash e_2 : A \mathbin{@} w}{\Omega \vdash (\lambda x : A.\, e_1)\, e_2 \equiv [I\,\langle x \mapsto e_2\rangle]e_1 : B \mathbin{@} w}$$

STLC-SEP-EQ-A-FUN-ETA
$$\frac{\Omega \vdash e : A \Rightarrow B \mathbin{@} w \qquad x \notin \mathtt{fv}\, e}{\Omega \vdash (\lambda x : A.\, e\, x) \equiv e : A \Rightarrow B \mathbin{@} w}$$

STLC-SEP-EQ-A-PAIR-BETA
$$\frac{\Omega \vdash e_1 : A \mathbin{@} w \qquad \Omega \vdash e_2 : B \mathbin{@} w \qquad \Omega, x : A \mathbin{@} w, y : B \mathbin{@} w \vdash e_3 : C \mathbin{@} w'}{\Omega \vdash (\mathtt{let}\,[x,\, y] = [e_1,\, e_2]\,\mathtt{in}\, e_3) \equiv [I\,\langle x \mapsto e_1, y \mapsto e_2\rangle]e_3 : C \mathbin{@} w'}$$

STLC-SEP-EQ-A-PAIR-ETA
$$\frac{\Omega \vdash e : A \wedge B \mathbin{@} w \qquad x, y \notin \mathtt{fv}\, e}{\Omega \vdash (\mathtt{let}\,[x,\, y] = e\,\mathtt{in}\,[x,\, y]) \equiv e : A \wedge B \mathbin{@} w}$$

STLC-SEP-EQ-M-UNIT-BETA
$$\frac{\Omega \vdash \langle\rangle : 1_{\mathsf{m}} \mathbin{@} \epsilon \qquad \Omega \vdash e : A \mathbin{@} w'}{\Omega \vdash (\mathtt{let}\,\langle\rangle = \langle\rangle\,\mathtt{in}\, e) \equiv e : A \mathbin{@} w'}$$

STLC-SEP-EQ-M-UNIT-ETA
$$\frac{\Omega \vdash e : 1_{\mathsf{m}} \mathbin{@} w}{\Omega \vdash (\mathtt{let}\,\langle\rangle = e\,\mathtt{in}\,\langle\rangle) \equiv e : 1_{\mathsf{m}} \mathbin{@} w}$$

STLC-SEP-EQ-M-FUN-BETA
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a \vdash e_1 : B \mathbin{@} c \qquad \Omega \vdash e_2 : A \mathbin{@} w_2}{\Omega \vdash (\lambda^* x : A.\, e_1)^*\, e_2 \equiv [I\,\langle x \mapsto e_2\rangle]e_1 : B \mathbin{@} w}$$

STLC-SEP-EQ-M-FUN-ETA
$$\frac{\Omega \vdash e : A \mathrel{-\!\!*} B \mathbin{@} w \qquad x \notin \mathtt{fv}\, e}{\Omega \vdash (\lambda^* x : A.\, e^*\, x) \equiv e : A \mathrel{-\!\!*} B \mathbin{@} w}$$

STLC-SEP-EQ-M-PAIR-BETA
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Sigma \qquad \Omega \vdash e_1 : A \mathbin{@} w_1 \qquad \Omega \vdash e_2 : B \mathbin{@} w_2 \qquad \Omega, a, b, (a, b \triangleright w), x : A \mathbin{@} a, y : B \mathbin{@} b \vdash e_3 : C \mathbin{@} w}{\Omega \vdash (\mathtt{let}\,\langle x,\, y\rangle = \langle e_1,\, e_2\rangle\,\mathtt{in}\, e_3) \equiv [I\,\langle x \mapsto e_1, y \mapsto e_2\rangle]e_3 : C \mathbin{@} w}$$

STLC-SEP-EQ-M-PAIR-ETA
$$\frac{\Omega \vdash e : A * B \mathbin{@} w \qquad x, y \notin \mathtt{fv}\, e}{\Omega \vdash (\mathtt{let}\,\langle x,\, y\rangle = e\,\mathtt{in}\,\langle x,\, y\rangle) \equiv e : A * B \mathbin{@} w}$$

STLC-SEP-EQ-BIND-BETA
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : A \mathbin{@} w_1 \qquad \Omega, a, c, (w_2, a \triangleright c), x : A \mathbin{@} a \vdash e_2 : T\, B \mathbin{@} c}{\Omega \vdash (\mathtt{let!}\, x = \mathtt{return}\, e_1\,\mathtt{in}\, e_2) \equiv [I\,\langle x \mapsto e_1\rangle]e_2 : T\, B \mathbin{@} w}$$

STLC-SEP-EQ-BIND-ETA
$$\frac{\Omega \vdash e : T\, A \mathbin{@} w \qquad x \notin \mathtt{fv}\, e}{\Omega \vdash (\mathtt{let!}\, x = e\,\mathtt{in}\,\mathtt{return}\, x) \equiv e : T\, A \mathbin{@} w}$$

STLC-SEP-EQ-BIND-BIND
$$\frac{(w_1, w_2 \triangleright w_{12}) \in \Omega.\Theta \qquad (w_{12}, w_3 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : T\, A \mathbin{@} w_1 \qquad \Omega, a, c, (w_2, a \triangleright c), x : A \mathbin{@} a \vdash e_2 : T\, B \mathbin{@} c \qquad \Omega, a, c, (w_3, a \triangleright c), x : B \mathbin{@} c \vdash e_3 : T\, C \mathbin{@} w}{\Omega \vdash (\mathtt{let!}\, x = (\mathtt{let!}\, y = e_1\,\mathtt{in}\, e_2)\,\mathtt{in}\, e_3) \equiv (\mathtt{let!}\, y = e_1\,\mathtt{in}\,\mathtt{let!}\, x = e_2\,\mathtt{in}\, e_3) : T\, C \mathbin{@} w}$$

Additionally, the equivalence judgment forms an equivalence relation:

STLC-SEP-EQ-REFL
$$\frac{\Omega \vdash e : A \mathbin{@} w}{\Omega \vdash e \equiv e : A \mathbin{@} w}$$

STLC-SEP-EQ-SYM
$$\frac{\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w}{\Omega \vdash e_2 \equiv e_1 : A \mathbin{@} w}$$

STLC-SEP-EQ-TRANS
$$\frac{\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w \qquad \Omega \vdash e_2 \equiv e_3 : A \mathbin{@} w}{\Omega \vdash e_1 \equiv e_3 : A \mathbin{@} w}$$

Finally, each of the typing rules – both logical and structural – induces a congruence rule for the equivalence relation. The congruence rule has the same assumptions as the typing rule, but replaces typing judgments $\Omega \vdash e : A \mathbin{@} w$ with equivalence judgments $\Omega \vdash e \equiv e' : A \mathbin{@} w$. For instance, STLC-SEP-M-FUN-E induces the congruence rule:

$$\text{STLC-SEP-EQ-M-FUN-E-CONGR}\quad \frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 \equiv e_1' : A \multimap B \mathbin{@} w_1 \qquad \Omega \vdash e_2 \equiv e_2' : A \mathbin{@} w_2}{\Omega \vdash e_1^*\, e_2 \equiv e_1'^*\, e_2' : B \mathbin{@} w}$$

The β- and η-equivalences for the additive connectives are the same as those for $\lambda_{\texttt{loc}}$. For the multiplicative connectives however, the β and η equivalences are more complex, as they must also take into account the labels and constraints. Additionally, not only the logical rules, but also the structural rules have congruence rules for the equivalence relation. The first derivable property of the equivalence judgment is that both sides are well-typed:

Theorem 4.3.13 (Soundness *w.r.t.* Equivalence)**:** If Ω `wf` and $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$ then $\Omega \vdash e_1 : A \mathbin{@} w$ and $\Omega \vdash e_2 : A \mathbin{@} w$.

Proof. By induction on the derivation of $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$. The proof takes the same approach as that of the base language (3.4.13).

For instance, consider STLC-SEP-EQ-M-FUN-BETA. The derivation of the left side is given by:

$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \dfrac{w_1 \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful} \qquad \Omega, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a \vdash e_1 : B \mathbin{@} c}{\Omega \vdash \lambda^* x : A.\, e_1 : A \multimap B \mathbin{@} w_1} \qquad \Omega \vdash e_2 : A \mathbin{@} w_2}{\Omega \vdash (\lambda^* x : A.\, e_1)^*\, e_2 : B \mathbin{@} w}$$

The assumption $w_1 \in \Omega.\Sigma_\epsilon$ follows from Ω `wf` and the assumption $(w_1, w_2 \triangleright w) \in \Omega.\Theta$. The assumption $\vdash A : \texttt{Stateful}$ follows from $\Omega \vdash e_2 : A \mathbin{@} w_2$ and Lemma 4.2.11.

Similar to the base language, the derivation of the right hand side follows from the substitution theorem (Theorem 4.3.11) and has the following form:

$$\frac{\Omega \vdash I \langle x \mapsto e_2 \rangle, I \langle a \mapsto w_2, c \mapsto w \rangle : \Omega, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a \qquad \Omega, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a \vdash e_1 : B \mathbin{@} c}{\Omega \vdash [I \langle x \mapsto e_2 \rangle] e_1 : B \mathbin{@} w}$$

In this derivation, the substitution $\Omega \vdash I \langle x \mapsto e_2 \rangle, I \langle a \mapsto w_2, c \mapsto w \rangle : \Omega, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a$ is derived by the rules in 4.3.8, using the remaining assumptions of STLC-SEP-EQ-M-FUN-BETA. □

The β-equivalence rules in Definition 4.3.12 have the same shape as those for the $\lambda_{\texttt{loc}}$. However, due to the structural rules, the assumptions of the equivalence do not immediately follow from the derivation of the β-redex (*e.g.* $\Omega \vdash (\lambda x : A.\, e_1)\, e_2 : B \mathbin{@} w$), as in the simply typed $\lambda_{\texttt{loc}}$. An example snippet of such a derivation is the following:

$$\frac{\dfrac{(w, \epsilon \triangleright w') \in \Omega \qquad \dfrac{\vdots}{\Omega, x : A \mathbin{@} w' \vdash e_1 : B \mathbin{@} w'}}{\Omega \vdash \lambda x : A.\, e_1 : A \Rightarrow B \mathbin{@} w'} \qquad \dfrac{\ldots}{\Omega \vdash e_2 : A \mathbin{@} w}}{\Omega \vdash (\lambda x : A.\, e_1)\, e_2 : B \mathbin{@} w}$$

In this derivation snippet, the STLC-SEP-EQ-A-FUN-BETA rule cannot be applied, as the assumption $\Omega, x : A \mathbin{@} w \vdash e_1 : B \mathbin{@} w$ is not in any part of the derivation. The β-equivalence rules therefore seem overly restrictive, as they require the typing derivation of the β-redex to contain no structural rules. In Chapter 5 we return to this problem and show that β-equivalence rules requiring only the typeability of the left side are admissible rules by defining β-reductions. For instance, the following alternative β rule for $A \Rightarrow B$ is admissible:

$$\frac{\Omega \vdash (\lambda x : A.\, e_1)\, e_2 : B \mathbin{@} w}{\Omega \vdash (\lambda x : A.\, e_1)\, e_2 \equiv [I \langle x \mapsto e_2 \rangle] e_1 : B \mathbin{@} w}$$

The form of these admissible rules corresponds to the *preservation* theorem for β-reduction (Theorem 5.4.3).

4.4 Denotational Semantics

Similar to $\lambda_{\texttt{loc}}$, the denotational semantics of the labeled calculus could be defined directly on the labeled system λ_*.

Instead, we observe that the distinction between additive and multiplicative connectives is not relevant to the execution of a program. Therefore, we adopt a refinement type approach, where the labeled types of the λ_* are erased to the types in $\lambda_{\texttt{loc}}$. Similarly, the expressions and derivations can be erased, and the denotational semantics of λ_* are defined as the denotational semantics of $\lambda_{\texttt{loc}}$ for the erased derivations.

Next, we demonstrate that well-typed expressions in λ_* offer stronger guarantees than those in the simply typed $\lambda_{\texttt{loc}}$ by proving an additional *regularity* lemma, which asserts that well-typed programs do not get *stuck*. Our approach is inspired by Ghalayini and Krishnaswami [2023], who use erasure to show that explicit refinement types can be erased to a simply typed lambda calculus without refinements.

The approach consists of three steps. First, we define an erasure operation from types and expressions of the λ_* to $\lambda_{\texttt{loc}}$. Next, we show that this erasure preserves typing derivations and substitutions. Finally, we define a logical predicate that relates types in λ_* to interpretations of types in $\lambda_{\texttt{loc}}$, and use this to show that well-typed expressions in λ_* do not get stuck.

Erasure

The erasure of types in the labeled λ_* collapses the additive and multiplicative connectives into the corresponding connectives of $\lambda_{\texttt{loc}}$ as follows:

Definition 4.4.1 (Type Erasure)**:**

$$|1_a| \stackrel{\text{def}}{=} 1 \qquad |1_m| \stackrel{\text{def}}{=} 1 \qquad |A \Rightarrow B| \stackrel{\text{def}}{=} |A| \to |B| \qquad |A \mathrel{-\!\!*} B| \stackrel{\text{def}}{=} |A| \to |B| \qquad |A \wedge B| \stackrel{\text{def}}{=} |A| \times |B|$$

$$|A * B| \stackrel{\text{def}}{=} |A| \times |B| \qquad |T\,A| \stackrel{\text{def}}{=} T\ |A| \qquad |\texttt{Ref}\ A| \stackrel{\text{def}}{=} \texttt{loc}$$

Both additive and multiplicative units are erased to the unit type 1, the additive and multiplicative functions are erased to the function type, and the additive and multiplicative products are erased to the product type. The monadic type is similarly erased to the monadic type. The main difference is the reference type, which is erased to the location type `loc` (similar to erasing a pointer $A*$ to a void pointer `void`$*$ in C). The erasure of types is compositional, meaning that the erasure of a type is defined in terms of the erasure of the components of the type. To ensure that erased types of `Pure` types are can still be placed in the heap, the erasure of a type respects its kind:

Lemma 4.4.2: If $A \in \texttt{Type}$ and $\vdash A : k$ for some kind k, then $\vdash |A| : k$.

Proof. By induction on the derivation of $\vdash A : k$ and the definition of erasure for types. □

Definition 4.4.3 (Expression Erasure)**:**

$$|x| \stackrel{\text{def}}{=} x \qquad |[]| \stackrel{\text{def}}{=} () \qquad |\lambda x : A.\, e| \stackrel{\text{def}}{=} \lambda x.\ |e| \qquad |e_1\, e_2| \stackrel{\text{def}}{=} |e_1|\ |e_2| \qquad |[e_1,\, e_2]| \stackrel{\text{def}}{=} (|e_1|\,, |e_2|)$$

$$|\texttt{let}\,[x,\, y] = e_1\,\texttt{in}\,e_2| \stackrel{\text{def}}{=} \texttt{let}\,(x, y) = |e_1|\ \texttt{in}\ |e_2| \qquad |\langle\rangle| \stackrel{\text{def}}{=} () \qquad |\texttt{let}\,\langle\rangle = e_1\,\texttt{in}\,e_2| \stackrel{\text{def}}{=} |e_2|$$

$$|\lambda^* x : A.\, e| \stackrel{\text{def}}{=} \lambda x.\ |e| \qquad |e_1^*\, e_2| \stackrel{\text{def}}{=} |e_1|\ |e_2| \qquad |\langle e_1,\, e_2\rangle| \stackrel{\text{def}}{=} (|e_1|\,, |e_2|)$$

$$|\texttt{let}\,\langle x,\, y\rangle = e_1\,\texttt{in}\,e_2| \stackrel{\text{def}}{=} \texttt{let}\,(x, y) = |e_1|\ \texttt{in}\ |e_2| \qquad |\texttt{return}\,e| \stackrel{\text{def}}{=} \texttt{return}\ |e|$$

$$|\texttt{let!}\ e_1 = x\,\texttt{in}\,e_2| \stackrel{\text{def}}{=} \texttt{let!}\ |e_1| = x\ \texttt{in}\ |e_2| \qquad |\texttt{ref}_A\, e| \stackrel{\text{def}}{=} \texttt{ref}_{|A|}\ |e|$$

$$\left|\texttt{replace}_{A,B}\, e_1\, e_2\right| \stackrel{\text{def}}{=} \texttt{let}\, l = |e_1|\ \texttt{in}\,\texttt{let!}\ v = (\texttt{replace}_{|A|,|B|}\ l\ |e_2|)\,\texttt{in}\,(v, l) \qquad |\texttt{free}_A\, e| \stackrel{\text{def}}{=} \texttt{free}_{|A|}\ |e|$$

The erasure of expressions is also compositional, and follows from the erasure of types. The operations with both additive and multiplicative versions in the λ_*, such as function abstraction and application, are

both erased to the same corresponding operation in $\lambda_{\texttt{loc}}$. The `return`, `let!`, `ref` and `free` operations are all similarly erased to the corresponding operations. As mutable operations are described by the monadic type, the expression e_1 in the $\texttt{let}\,\langle\rangle = e_1\,\texttt{in}\,e_2$ expression does not have any side effects. As such, the erasure simply ignores the first expression e_1 and returns the erasure of the second expression e_2. Finally, the `replace` operation is the only operation not directly erased, as the type of the `replace` operation in λ_* ($\texttt{Ref}\,A * B \Rightarrow T\,(A * \texttt{Ref}\,B)$) does not match the type of the corresponding operation in $\lambda_{\texttt{loc}}$ ($\texttt{loc} \times B \to T\,A$) after erasure. As such, we erase `replace` to an expression which first executes `replace` in the $\lambda_{\texttt{loc}}$, followed by pairing the result with a copy of the location l.

The erasure of types can be extended to contexts in the intuitive way, by erasing the types of each of the variables in the context and ignoring labels and constraints entirely:

Definition 4.4.4 (Context Erasure)**:** Erasure on assumption contexts Γ is defined as:

$$|\cdot| \stackrel{\text{def}}{=} \cdot \qquad\qquad |\Gamma, x : A \mathbin{@} w| \stackrel{\text{def}}{=} |\Gamma|, x : |A|$$

Erasure on contexts Ω is defined as:

$$|\Omega| \stackrel{\text{def}}{=} |\Omega.\Gamma|$$

After defining erasure of contexts, expressions and types, we can state the first important property of erasure: that erasure preserves typing derivations. This ensures that the denotation of erased derivations can be used as the denotation of the derivations of the labeled system.

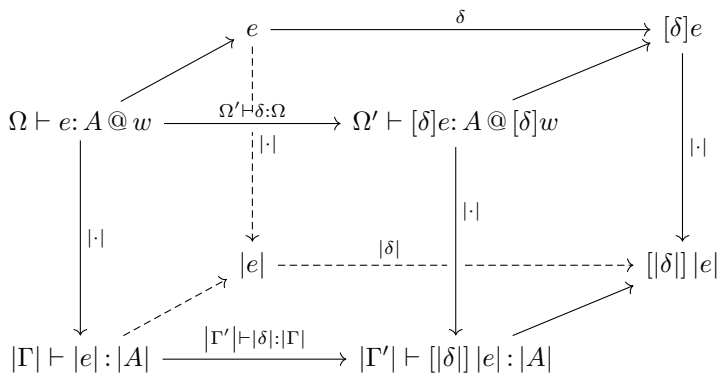
Theorem 4.4.5 (Erasure preserves typing)**:** If $\Omega \vdash e : A \mathbin{@} w$ then there exists a derivation $|\Omega| \vdash |e| : |A|$ written $|\Omega \vdash e : A \mathbin{@} w|$.

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$ and erasure of expressions and types. □

This notion of erasure corresponds to the *erasure functor* for type refinement systems [Zeilberger 2016]. These systems use a categorical perspective of typing, where contexts and types are considered as objects, and typing derivations are considered as morphisms (arrows between objects). In this perspective, both the refined (labeled) and unrefined (unlabeled) type systems can be seen as categories. The erasure operation acts as a functor between these categories. This means it maps each context and type in the refined system (objects in the category) to a corresponding context and type in the unrefined system. Similarly, it maps each derivation in the refined system (morphisms in the category) to corresponding derivations in the unrefined system.

Erasure of Substitution and Equivalences

In addition to typing judgments $\Omega \vdash e : A \mathbin{@} w$, the judgments for thinning, typed substitutions and equivalences are also defined in both λ_* and $\lambda_{\texttt{loc}}$. In this subsection, we show that the erasure operation can be extended to these judgments, and that erasure commutes with thinning and substitution. The subsection concludes with the erasure theorem for equality judgments, which can be combined with Theorem 3.4.13 to also prove soundness *w.r.t.* equivalences for λ_*. The below figure depicts the commutative properties of erasure and substitution. The down direction represents erasure to $\lambda_{\texttt{loc}}$, the back direction erasure of derivations to only expressions, and the right direction applying substitutions.



The proofs of the substitution theorems for both λ_* (Theorem 4.3.11) and $\lambda_{\texttt{loc}}$ (Theorem 3.3.4) use a corresponding lemma for thinning. As such, the commutativity proof of erasure (down) and substitution (right) also uses a corresponding lemma for thinning. This commutativity proof can be depicted in a similar diagram to the figure above, but replacing substitutions $\Omega' \vdash \delta : \Omega$ with thinnings $\Omega' \supseteq \Omega$, resulting in the figure below. As thinning does not change the expression e, the back direction has been omitted.

$$\begin{array}{ccc} \Omega \vdash e \colon A \mathbin{@} w & \xrightarrow{\Omega' \supseteq \Omega} & \Omega' \vdash e \colon A \mathbin{@} w \\ \Big\downarrow{\scriptstyle |\cdot|} & & \Big\downarrow{\scriptstyle |\cdot|} \\ |\Gamma| \vdash |e| : |A| & \xrightarrow{|\Gamma'| \supseteq |\Gamma|} & |\Gamma'| \vdash |e| : |A| \end{array}$$

The commutativity of the above diagram is proven in the following two lemmas, which show that $|\Gamma'| \supseteq |\Gamma|$ exists and prove the commutativity respectively.

Lemma 4.4.6 (Erasure of Thinning)**:** Let $\Omega \supseteq \Omega'$ be a thinning, then there is a corresponding thinning in the erased system $|\Omega| \supseteq |\Omega'|$ denoted $|\Omega \supseteq \Omega'|$.

Proof. By induction on the derivation of $\Omega \supseteq \Omega'$ and applying the erasure rules for contexts. □

Lemma 4.4.7 (Commutativity of Thinning and Erasure)**:** Let $\Omega \supseteq \Omega'$ be a thinning and $\Omega \vdash e : A \mathbin{@} w$ be a typing derivation then: $|\Omega' \supseteq \Omega; \Omega \vdash e : A \mathbin{@} w| = |\Omega' \supseteq \Omega| \, ; |\Omega \vdash e : A \mathbin{@} w|$

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$. □

Next, the commutativity of the back face of the cube e to $[|\delta|] \, |e|$ is proven. It consists of first defining the erasure of syntactic substitution, followed by proving the commutativity of syntactic substitution and erasure.

Definition 4.4.8 (Erasure of Syntactic Substitution)**:** $|\delta_v| \stackrel{\texttt{def}}{=} \{x \mapsto |e| \mid x \mapsto e \in \delta_v\}$

Lemma 4.4.9 (Commutativity of Expression Substitution and Erasure)**:** $|[\delta_v]e| = [|\delta_v|] \, |e|$

Proof. By induction on e. □

The last face of the cube is the front face, which consists of defining the erasure of typed substitutions, and proving commutativity of typed substitution and erasure. The front face of the cube requires that syntactic erasure and substitution commute, as substitution followed by erasure (right then down) results in a derivation of the form $|\Gamma| \vdash |[\delta]e| : |A|$, whereas first erasure and then substitution results in a derivation of the form $|\Gamma| \vdash [|\delta|] \, |e| : |A|$. By commutativity of the back face however, these are derivations of the same judgment. The following 3 lemmas define the erasure of typed substitutions and prove the commutativity of typed substitution and erasure.

Lemma 4.4.10 (Erasure of Substitution)**:** If $\Omega' \vdash \delta_v, \delta_l : \Omega$ then $|\Omega'| \vdash |\delta_v| : |\Omega|$ denoted $|\Omega' \vdash \delta_v, \delta_l : \Omega|$.

Proof. By induction on $\Omega' \vdash \delta_v, \delta_l : \Omega$. □

Lemma 4.4.11 (Commutativity of Substitution Thinning and Erasure)**:** $|\Omega_1 \supseteq \Omega'; \Omega' \vdash \delta_v, \delta_l : \Omega| = |\Omega_1 \supseteq \Omega'| \, ; |\Omega' \vdash \delta_v, \delta_l : \Omega|$.

Proof. By induction on $\Omega' \vdash \delta_v, \delta_l : \Omega$. The proof has the same structure as the proof of Lemma 4.4.7, but uses the commutativity lemma for thinning Lemma 4.4.7 instead of the thinning lemma for substitution Lemma 4.3.9. □

Lemma 4.4.12 (Commutativity of Substitution and Erasure)**:**

$$|\Omega' \vdash \delta_v, \delta_l : \Omega; \Omega \vdash e : A \mathbin{@} w| = |\Omega' \vdash \delta_v, \delta_l : \Omega| \, ; |\Omega \vdash e : A \mathbin{@} w|$$

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$, using Lemma 4.4.11 in the cases that introduce new variables, labels or constraints. □

The last theorem for erasure relates equivalences in λ_* to those in $\lambda_{\texttt{loc}}$. The theorem ensures that the soundness *w.r.t.* equivalences of the denotational semantics of $\lambda_{\texttt{loc}}$ can be lifted to the labeled λ_*, thereby ensuring that equivalent expressions in the λ_* have the same denotation.

Theorem 4.4.13 (Erasure of Equivalence)**:** If $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$ then $|\Omega| \vdash |e_1| \equiv |e_2| : |A|$.

Proof. By induction on the derivation of $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$, applying Lemma 4.4.9 in the cases that involve substitution of variables. □

Denotations

Similar to the denotational semantics of $\lambda_{\texttt{loc}}$, the denotational semantics of λ_* gives a mathematical interpretation to types, contexts and typing derivations. Rather than defining the denotational semantics directly on typing derivations, they are instead defined in terms of the erasure to $\lambda_{\texttt{loc}}$. Notably, as the denotational semantics of $\lambda_{\texttt{loc}}$ does not take into account resources, the denotational semantics of λ_* does not directly guarantee that programs are safe (*i.e.* do not get *stuck*). To show this, we will define a *heap validity* condition, which asserts the resource ownership of labeled types, and show that this condition is preserved between the context Ω and the value of labeled type $A \in w$ of typing derivations.

Definition 4.4.14 (Denotations of λ_*)**:** The denotations of λ_* are defined in terms of erasure as follows:

1. For $A \in \texttt{Type}$ define $[\![A]\!] \stackrel{\text{def}}{=} [\![|A|]\!]$
2. For $\Omega \in \texttt{Ctx}$ define $[\![|\Omega|]\!]$
3. For $\Omega \vdash e : A \mathbin{@} w$ define $[\![\Omega \vdash e : A \mathbin{@} w]\!] \stackrel{\text{def}}{=} [\![|\Omega \vdash e : A \mathbin{@} w|]\!]$

As a consequence of defining the denotational semantics in terms of $\lambda_{\texttt{loc}}$, the properties of the denotational semantics of $\lambda_{\texttt{loc}}$ also apply to λ_*:

Theorem 4.4.15 (Derivation Irrelevance)**:** If d, e are derivations of $\Omega \vdash e : A \mathbin{@} w$ then $[\![d]\!] = [\![e]\!]$.

Proof. By Theorem 3.4.6. □

Theorem 4.4.16 (Semantic Thinning)**:** If $\Omega' \supseteq \Omega$ and $\Omega \vdash e : A \mathbin{@} w$ then $[\![\Omega' \supseteq \Omega; \Omega \vdash e : A \mathbin{@} w]\!] = [\![\Omega' \supseteq \Omega]\!] \,;\, [\![\Omega \vdash e : W \mathbin{@} w]\!]$.

Proof. By Lemma 4.4.7 and Lemma 3.4.9. □

Theorem 4.4.17 (Semantic Substitution)**:** If $\Omega' \vdash \delta_v, \delta_l : \Omega$ and $\Omega \vdash e : A \mathbin{@} w$, Then

$$[\![\Omega' \vdash \delta_v, \delta_l : \Omega; \Omega \vdash e : A \mathbin{@} w]\!] = [\![\Omega' \vdash \delta_v, \delta_l : \Omega]\!] \,;\, [\![\Omega \vdash e : A \mathbin{@} w]\!]$$

Proof. By Lemma 4.4.12 and Theorem 3.4.12. □

Similar to the soundness *w.r.t.* equivalence for the $\lambda_{\texttt{loc}}$, the typeable β- and η-equivalent expressions in the labeled λ_* have the same denotation.

Theorem 4.4.18 (Soundness *w.r.t.* Equivalences)**:** If $\Omega \vdash e_1 \equiv e_2 : A \mathbin{@} w$ then $[\![\Omega \vdash e_1 : A \mathbin{@} w]\!] = [\![\Omega \vdash e_2 : A \mathbin{@} w]\!]$.

Proof. By Lemma 4.4.12 and Theorem 3.4.13. □

Regularity

The above definition of denotational semantics for the labeled system is weak, as it does not take into account any of the labels or constraints. More specifically, this denotational semantics is not sufficient to show that well-typed programs in the labeled system do not get *stuck*.

To show this, we define an additional property of the refined types based on the resource semantics of separation logic (§ 2.3). The property additionally states that heap operations do not get *stuck* for any heap containing the required resources for the computation. By extending this property as a validity property of both contexts and labeled types, we can show that well-typed programs maintain the validity of resources, and therefore that well-typed heap computations $T\,A$ do not get *stuck*. This property is called *regularity* and resembles a similar property in the work of Ghalayini and Krishnaswami [2023] to prove that type refinements are preserved through erasure.

The additional property is defined as a logical predicate $h, a \models A$ for heap h and value a in the interpretation of type A. The relation asserts that the proposition corresponding to type A is true for partial heap h and value a in the interpretation of type A.

Definition 4.4.19 (Heap Validity of Types)**:** Given a heap $h \in \texttt{Heap}$, type $A \in \texttt{Type}$ *s.t.* $\vdash A : \texttt{Stateful}$ and value $a \in [\![A]\!]$, we define the forcing relation $h, v \models A$ as:

$$
\begin{aligned}
h, v &\models 1_a \stackrel{\text{def}}{=} \text{True} \\
h, () &\models 1_m \stackrel{\text{def}}{=} h = \emptyset \\
h, (x, y) &\models A \wedge B \stackrel{\text{def}}{=} (h, x \models A) \wedge (h, y \models B) \\
h, (x, y) &\models A * B \stackrel{\text{def}}{=} \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge (h_1, x \models A) \wedge (h_2, y \models B) \\
h, f &\models A \Rightarrow B \stackrel{\text{def}}{=} \forall x \in X_A.\, (h, x \models A) \implies (h, f\, x \models B) \\
h, f &\models A \mathrel{-\!\!*} B \stackrel{\text{def}}{=} \forall h_1, x.\, (h_1, x \models A) \implies ((h \uplus h_1), f\, x \models B) \\
h, p &\models T\,A \stackrel{\text{def}}{=} \forall h_f.\, h_f \# h \implies \exists h', a.\, p(h \uplus h_f) = (a, h' \uplus h_f) \wedge (h', a \models A) \\
h, l &\models \texttt{Ref}\ A \stackrel{\text{def}}{=} \exists h', v.\, h = \{l \mapsto (|A|, v)\} \uplus h' \wedge (h', v \models A)
\end{aligned}
$$

The relation $h, v \models A$ closely resembles that of separation logic (§ 2.3), but includes an additional value v. The two main differences are the monadic type $T\,A$ and the reference type $\texttt{Ref}\ A$.

The relation for the reference type $\texttt{Ref}\ A$ corresponds to the pointsto connective $l \mapsto v$ in separation logic, but the location l is given as the value, rather than fixed. Similarly, the value v which is fixed in separation logic, is generalized to a value of type A in the labeled calculus. As such, the reference type $\texttt{Ref}\ A$ describes not only the resources of the location (the singleton heap $\{l \mapsto (|A|, v)\}$), but also those of the value of type A stored at that location $(h', v \models A)$. As only pure types can be stored in the heap, the logical predicate is only defined for stateful types.

The monadic type $T\,A$ corresponds to the weakest precondition $\texttt{wp}\, e\, \{A\}$ in separation logic, but where the expression e is modeled by the value $p : \texttt{Heap} \rightharpoonup [\![A]\!] \times \texttt{Heap}$ which describes the heap operations, and includes a totality condition $p(h \uplus h_f) = (a, h' \uplus h_f)$ which asserts that the heap computation p does not get *stuck* for any heap containing at least the resources h, and that the heap computation does not alter the resources in the frame heap h_f.

The addition of the frame heap h_f in the definition of the forcing relation for $h, f \models T\,A$ ensures that the frame rule of separation logic holds for the relation. Namely, if $h, (f, b) \models T\,A * B$ then $h, f' \models T\,(A * B)$, where f' is the heap operation that executes f and then pairs the result with b, and is defined as $f' : h \mapsto \texttt{let}\,(a, h') = f\,h\,\texttt{in}\,((a, b), h')$.

The heap validity condition is then extended to contexts by giving denotations of well-formed label contexts, constraint contexts and variable contexts that respect the heap validity condition. Similar to how the assumption context Γ in the unlabeled system is interpreted as any map from the variables defined in Γ to values in the interpretation of the corresponding type, the label context Σ is interpreted as any map from the labels defined in Σ to partial heaps.

Definition 4.4.20 (Denotation of Labels)**:**

$$\boxed{[\![\Sigma]\!] : \mathit{Set}}$$

$$[\![\cdot]\!] \stackrel{\text{def}}{=} \{[]\} \qquad\qquad [\![\Sigma, a]\!] \stackrel{\text{def}}{=} \{\sigma\,\langle a \mapsto h\rangle \mid \sigma \in [\![\Sigma]\!] \wedge h : \texttt{Heap}\}$$

Substitutions $\sigma : \Sigma \to \texttt{Heap}$ on labels are implicitly extended to $\sigma : \Sigma_\epsilon \to \texttt{Heap}$ by taking $\sigma\,\epsilon \stackrel{\text{def}}{=} \emptyset$.

The second context Θ describes the constraints between worlds, and hence the corresponding partial heaps in the interpretation. The well-formed constraint context $\Sigma \vdash \Theta\ \mathtt{wf}$ is therefore the subset of maps from labels to partial heaps that satisfy the constraints in Θ, where each constraint $(w_1, w_2 \triangleright w)$ is interpreted as an equality $\sigma\, w_1 \uplus \sigma\, w_2 = \sigma\, w_3$ on partial heaps.

Definition 4.4.21 (Denotation of Constraint Well-formedness)**:**

$$\boxed{[\![\Sigma \vdash \Theta\ \mathtt{wf}]\!] : \mathit{Set}}$$

$$[\![\Sigma \vdash \cdot\ \mathtt{wf}]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!] \qquad [\![\Sigma \vdash \Theta, (w_1, w_2 \triangleright w)\ \mathtt{wf}]\!] \stackrel{\text{def}}{=} \{\sigma \in [\![\Sigma \vdash \Theta\ \mathtt{wf}]\!] \mid \sigma\, w_1 \uplus \sigma\, w_2 = \sigma\, w\}$$

The denotation of a well-formed constraint context describes the subset of the denotation of label contexts that satisfy the constraints in the constraint context. As such, any label substitution σ valid for $\Sigma \vdash \Theta\ \mathtt{wf}$ is also valid for Σ. Hence, the denotation of the constraint context forms a subset of the label context: $[\![\Sigma \vdash \Theta\ \mathtt{wf}]\!] \subseteq [\![\Sigma]\!]$.

The labeled assumption context Γ describes not only the variables and their types, but also the world in which each variable is valid. For a given labeled assumption $x : A\,@\,w$ this corresponds to the heap validity condition $(\sigma\, w), (\gamma\, x) \models A$. The denotation of a well-formed labeled assumption context $\Sigma \vdash \Gamma\ \mathtt{wf}$ is then the set of mappings from variables to values that satisfy the heap validity condition for each variable.

Definition 4.4.22 (Denotation of Assumption Well-formedness)**:**

$$\boxed{[\![\Sigma \vdash \Gamma\ \mathtt{wf}]\!] : [\![\Sigma]\!] \to \mathit{Set}}$$

$$[\![\Sigma \vdash \cdot\ \mathtt{wf}]\!]_\sigma \stackrel{\text{def}}{=} \{[]\} \qquad [\![\Sigma \vdash \Gamma, x : A\,@\,w\ \mathtt{wf}]\!]_\sigma \stackrel{\text{def}}{=} \{\gamma\,\langle x \mapsto v\rangle \mid \gamma \in [\![\Sigma \vdash \Gamma\ \mathtt{wf}]\!]_\sigma \wedge v \in [\![A]\!] \wedge \sigma\, w, h \models A\}$$

As the heap validity condition of an assumption (marked in blue) depends on the heap corresponding the world w of the assumption, the denotation of well-formed labeled assumption contexts is dependent on the denotation of the label context Σ, which maps worlds to partial heaps.

Similar to how the well-formedness of contexts is defined as the well-formedness of the constraint context and the assumption context, the denotations of well-formed contexts can be defined as a dependent pair of a valid mapping σ from labels to heaps respecting the constraints, and a valid mapping γ from variables to values *w.r.t.* those heaps.

Definition 4.4.23 (Denotation of Context Well-formedness)**:** The denotation of a well-formed context $\Omega\ \mathtt{wf}$ is defined as:

$$[\![\Omega\ \mathtt{wf}]\!] \stackrel{\text{def}}{=} (\sigma : [\![\Omega.\Sigma \vdash \Omega.\Theta\ \mathtt{wf}]\!]) \times [\![\Omega.\Sigma \vdash \Omega.\Gamma\ \mathtt{wf}]\!]_\sigma$$

Similar to how the denotations of constraint contexts are subsets of the denotations of label contexts, the denotations of well-formed assumption contexts in λ_* are subsets of the denotations of assumption contexts in $\lambda_{\mathtt{loc}}$. This is formalized in the following lemma:

Lemma 4.4.24 (Erasure of Assumption Context Denotation)**:** For all $\sigma \in [\![\Sigma]\!]$ it holds that $[\![\Sigma \vdash \Gamma]\!]_\sigma \subseteq [\![|\Gamma|]\!]$.

Proof. By induction on the derivation of $\Sigma \vdash \Gamma\ \mathtt{wf}$. □

Corollary 4.4.25: If $\Omega\ \mathtt{wf}$ and $(\sigma, \gamma) \in [\![\Omega\ \mathtt{wf}]\!]$ then $\gamma \in [\![|\Omega|]\!]$

The corollary ensures that valid mappings from variables to values γ in the denotation of the labeled context Ω can be used as the argument to the denotation of an erased derivation. The final theorem states that in a valid context, the heap validity condition is preserved by typing derivations in the labeled system.

Theorem 4.4.26 (Regularity)**:** Let $\Omega\ \mathtt{wf}$, $(\sigma, \gamma) \in [\![\Omega\ \mathtt{wf}]\!]$ and $\Omega \vdash e : A \mathbin{@} w$ be given. Then:

$$\sigma\, w, [\![|\Omega \vdash e : A \mathbin{@} w|]\!]_\gamma \models A$$

Proof. By induction on the derivation of $\Omega \vdash e : A \mathbin{@} w$. As the cases are similar, we show the cases for STLC-SEP-M-FUN-I, STLC-SEP-M-FUN-E and STLC-SEP-FREE as examples:

*(*STLC-SEP-M-FUN-I*).* We have to prove: $\sigma\, w, [\![|\Omega \vdash \lambda^* x : A.\, e : A \mathrel{-\!\!*} B \mathbin{@} w|]\!]_\gamma \models A \mathrel{-\!\!*} B$.
The denotation can be expanded as:

$$\begin{aligned}[\![|\Omega \vdash \lambda^* x : A.\, e : A \mathrel{-\!\!*} B \mathbin{@} w|]\!]_\gamma &= [\![|\Omega| \vdash \lambda x : |A| \,.\, |e| : |A| \to |B|]\!]_\gamma \\ &= v \in [\![|A|]\!] \mapsto [\![|\Omega|\,, x : |A| \vdash |e| : |B|]\!]_\gamma \langle x \mapsto v\rangle \\ &= v \in [\![|A|]\!] \mapsto [\![|\Omega, a, c, (a, w \triangleright c)\,, x : A \mathbin{@} a \vdash e : B \mathbin{@} c|]\!]_{\gamma\langle x \mapsto v\rangle}\end{aligned}$$

Additionally, the conclusion $\sigma\, w, [\![|\Omega \vdash \lambda^* x : A.\, e : A \mathrel{-\!\!*} B \mathbin{@} w|]\!]_\gamma \models A \mathrel{-\!\!*} B$ can be expanded to:

$$\forall h_v \in \mathtt{Heap}, v \in [\![|A|]\!] \,.\, h_v, v \models A \implies (h \uplus h_v), ([\![|\Omega, a, c, (a, w \triangleright c)\,, x : A \mathbin{@} a \vdash e : B \mathbin{@} c|]\!]_{\gamma\langle x \mapsto v\rangle}) \models B$$

Let h_v and v *s.t.* $h_v, v \models A$ be given. Choose $\sigma' ::= \sigma\, \langle a \mapsto h_v, c \mapsto (\sigma\, w \uplus h_v)\rangle$.
Then $\sigma' \in [\![\Omega.\Sigma, a, c \vdash \Omega.\Theta, (w, a \triangleright c)]\!]$. Moreover, as $h_v, v \models A$ it holds that:
$\gamma\, \langle x \mapsto v\rangle \in [\![\Omega.\Sigma, a, c \vdash \Omega.\Gamma, x : A \mathbin{@} a]\!]_{\sigma'}$ and hence $\sigma', \gamma\, \langle x \mapsto v\rangle \in [\![\Omega, a, c, (w, a \triangleright c)\,, x : A \mathbin{@} a\ \mathtt{wf}]\!]$.

By the induction hypothesis it follows that $\sigma'\, c, [\![|\Omega, a, c, (a, w \triangleright c)\,, x : A \mathbin{@} a \vdash e : B \mathbin{@} c|]\!]_{\gamma\,\langle x \mapsto v\rangle} \models B$, and the conclusion follows by $\sigma'\, c = h \uplus h_v$.

*(*STLC-SEP-M-FUN-E*).* We have to prove: $\sigma\, w, [\![|\Omega \vdash e_1\, e_2 : B \mathbin{@} w|]\!]_\gamma \models B$. By $\sigma, \gamma \in [\![\Omega\ \mathtt{wf}]\!]$ and $(w_1, w_2 \triangleright w) \in \Omega.\Theta$ it holds that $\sigma\, w_1 \uplus \sigma\, w_2 = \sigma\, w$. By the induction hypothesis it holds that:
$\sigma\, w_1, [\![|\Omega \vdash e_1 : A \mathrel{-\!\!*} B \mathbin{@} w_1|]\!]_\gamma \models A \mathrel{-\!\!*} B$ and $\sigma\, w_2, [\![|\Omega \vdash e_2 : A \mathbin{@} w_2|]\!]_\gamma \models A$. The conclusion follows from Definition 4.4.19, the definition of $[\![|\Omega \vdash e_1\, e_2 : B \mathbin{@} w|]\!]$ and $\sigma\, w_1 \uplus \sigma\, w_2 = \sigma\, w$.

*(*STLC-SEP-FREE*)* We have to prove: $1\sigma\, w, [\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]_\gamma \models T\, A$. First unfold the definition of $[\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]$ as follows:

$$\begin{aligned}[\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]_\gamma &= [\![|\Omega| \vdash \mathtt{free}_{|A|}\ |e| : T\ |A|]\!]_\gamma \\ &= h \mapsto \mathtt{let}\, l = [\![|\Omega| \vdash |e| : \mathtt{loc}]\!]_\gamma\ \mathtt{in} \begin{cases}(\pi_2(h\, l), h \setminus \{l\}) & \text{if } \pi_1(h\, l) = A \\ \bot & \text{otherwise}\end{cases} \\ &= h \mapsto \mathtt{let}\, l = [\![|\Omega \vdash e : \mathtt{Ref}\, A \mathbin{@} w|]\!]_\gamma\ \mathtt{in} \begin{cases}(\pi_2(h\, l), h \setminus \{l\}) & \text{if } \pi_1(h\, l) = A \\ \bot & \text{otherwise}\end{cases}\end{aligned}$$

Similarly unfold the definition of $\sigma\, w, [\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]_\gamma \models T\, A$ as:

$$T\, A \overset{\text{def}}{=} \forall h_f \,.\, h_f \# \sigma\, w \implies \exists h', a.\ [\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]_\gamma\, (\sigma\, w \uplus h_f) = (a, h' \uplus h_f) \wedge (h', a \models A)$$

Let h_f be given such that $h_f \# \sigma\, w$ and let $l = [\![|\Omega \vdash e : \mathtt{Ref}\, A|]\!]_\gamma$. By the induction hypothesis: $\sigma\, w, l \models \mathtt{Ref}\, A$ and hence there is a $v \in [\![|A|]\!]$ *s.t.* $\sigma\, w\, l = (|A|\,, v)$ and $(\sigma\, w \setminus \{l\}), v \models A$. Choose $a = v$ and $h' = \sigma\, w \setminus \{l\}$. Then the conclusion follows by the definition of $[\![|\Omega \vdash \mathtt{free}_A\, e : T\, A \mathbin{@} w|]\!]_\gamma\ (\sigma\, w \uplus h_f)$. □

Considering only top-level heap computations $T\, A$ (*i.e.* those that do not require any resources), the regularity theorem ensures that such computations do not get *stuck* in any heap. This is formalized as the *semantic safety* property of λ_*:

Corollary 4.4.27 (Semantic Safety)**:** If $\cdot \vdash e : T\, A \mathbin{@} \epsilon$ then for $p = [\![\cdot \vdash e : T\, A \mathbin{@} \epsilon]\!]_\emptyset$ it holds that:

$$\forall h \in \mathtt{Heap} \,.\, p\, h \neq \bot$$

Namely, p is a *total* rather than *partial* function on heaps.

Additionally, if $A = B \wedge 1_{\mathsf{m}}$ for some $B \in \mathtt{Type}$ then:

$$\forall h \,.\, \pi_2\, (p\, h) = h \qquad \textit{and} \qquad \emptyset, \pi_1\, (p\, h) \models B$$

The safety condition of the labeled system consists of two parts. The first is a weaker condition of safety, which states that heap computations that require no external resources do not get *stuck*. It ensures the absence of memory issues such as *use-after-free* and reading the wrong type of value from a location.

The second part of the safety condition is a stronger condition of safety, which states that when the result type holds no resources, the heap before and after the computation are the same. As such, the condition ensures the absence of memory leaks. Namely, all allocated references in the computation p have also been freed. Reading the second part $\emptyset, v \models B$ as a predicate $\phi\, v$ as a predicate on only the return value v, the second safety condition resembles the *adequacy* theorem in separation logic framework such as Iris [Jung et al. 2018].

In separation logic based program logics, where the safety/adequacy theorem is used in a left-to-right direction to prove that single programs are safe. In type theories such as for λ_* however, the safety/adequacy theorem is best used in a right-to-left direction, stating that programs that get *stuck* cannot be typed in the labeled system:

Corollary 4.4.28 (Semantic Safety of λ_*)**:** Let $e \in \texttt{Expr}$ and $A \in \texttt{Type}$ in $\lambda_{\texttt{loc}}$ such that $\cdot \vdash e : T\, A$.
Let $p = [\![\cdot \vdash e : T\, A]\!]_\emptyset : \texttt{Heap} \rightharpoonup (A \times \texttt{Heap})$ be the heap computation corresponding to e.
If there is a heap $h \in \texttt{Heap}$ *s.t.* $p\, h = \bot$, then:

$$\forall A^* \in \texttt{Type}\,, e^* \in \texttt{Expr}.\ |e^*| = e \implies \cdot \not\vdash e^* : T\,(A^*)\,@\,\epsilon$$

For instance, the corollary states that the `unsafe` program typeable in the unlabeled system is not typeable in the labeled system, as the corresponding heap computation gets *stuck*.

Chapter 5

Preservation

In the previous section, we used a denotational approach for proving safety, namely, the interpretation of well-typed programs does not get stuck. This corresponds to the semantic approach of type soundness described by Milner [1978]. Another approach is to use an operational semantics, which defines the steps (also called *reductions*) that a program takes during evaluation. The approach was introduced by Wright and Felleisen [1994] as a method for proving type soundness syntactically, rather than using a mathematical interpretation (as with denotational semantics).

They define reductions of the form $e \to e'$, which describe that taking a step in the evaluation of e results in e', and introduce an additional state `wrong`, which represents evaluations that fail, such as trying to add a function to a number. The syntactic approach to type soundness states that if a program e is well-typed, then there is no evaluation of e which results in the state `wrong`. This corresponds to the notion that "well-typed programs don't go wrong" introduced by Milner [1978]. The syntactic approach to type soundness is commonly proven using two lemmas: *preservation*, which states that if e is well-typed and $e \to e'$, then e' is also well-typed with the same type, and *progress*, which states that if e is well-typed, then either e is a value (*i.e.* cannot take any further steps and is not `wrong`) or there is an e' such that $e \to e'$.

The first property, *preservation*, is generally proven using an inversion lemma, which given a well-typed expression, describes that sub-expressions are also well-typed (See for example Pierce [2002]). For a simply typed lambda calculus with units, this inversion lemma would have the form:

Lemma 5.0.1 (Inversion of STLC)**:**

1. If $\Gamma \vdash x : A$ then $x : A \in \Gamma$.
2. If $\Gamma \vdash () : A$ then $A = 1$.
3. If $\Gamma \vdash \lambda x : A.\, e : R$ then $R = A \to B$ for some $A, B \in \texttt{Type}$ and $\Gamma, x : A \vdash e : B$.
4. If $\Gamma \vdash e_1\, e_2 : B$ then there is an $A \in \texttt{Type}$ such that $\Gamma \vdash e_1 : A \to B$ and $\Gamma \vdash e_2 : A$.

The proof of this inversion lemma relies on the fact that each type of expression can only be typed according to a single rule, and hence inversion follows by a case analysis of the expression.

Such an inversion lemma can easily be extended to single structural rules such as subtyping (Pierce [2002, Ch 15]), as each case only needs to consider their corresponding logical rule and a single additional structural rule.

The labeled system λ_* on the other hand contains 7 different structural rules, each of which extend the context or change the world. As such, a direct approach to proving an inversion lemma for an operational semantics of the labeled system is infeasible.

As an example, consider expressions of the form $e_1^*\, e_2$, which applies a multiplicative function e_1 with argument e_2. The corresponding logical rule is STLC-SEP-M-FUN-E:

$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash e_1 : A \mathrel{-\!\!*} B \mathbin{@} w_1 \qquad \Omega \vdash e_2 : A \mathbin{@} w_2}{\Omega \vdash e_1^*\, e_2 : B \mathbin{@} w}$$

As such, the preferred inversion lemma for this form of expression would be:

$$\Omega \vdash e_1^*\, e_2 : B \mathbin{@} w \implies \exists\, (w_1, w_2 \triangleright w) \in \Omega.\, \Omega \vdash e_1 : A \mathrel{-\!\!*} B \mathbin{@} w_1 \wedge \Omega \vdash e_2 : A \mathbin{@} w_2$$

However, taking into account STLC-SEP-ASSOC rule, the worlds w_1 or w_2 may be a newly introduced label k, and therefore $(w_1, w_2 \triangleright w) \in \Omega.\Theta$ may not necessarily hold. As such, an inversion lemma would need to consider newly introduced labels and constraints by each of the structural rules.

In this section, we discuss an approach for proving *preservation* for possible operational semantics λ_*. Rather than defining an operational semantics for heaps and heap operations, we limit ourselves to evaluating the pure (non-heap changing) fragment of the labeled system, by only considering β-reductions. The approach consists of translating derivations in the λ_* system to a layered system, which restricts the points at which structural rules can be applied, and therefore allows for a form of inversion lemma (for the most restrictive layer). Using this layered system, we prove the *preservation* property for β-reductions.

The section first introduces the β-reductions for λ_* (§ 5.1). Next we reduce the number of structural rules in λ_* by introducing a concept of *context strengthening* (§ 5.2), which we use to define the layered type system (§ 5.3). We show that derivations in the λ_* type system can be translated to those in the layered system using an approach based on the localization of structural rules in the labeled sequent calculus [Hóu et al. 2015]. Finally, we prove the *preservation* property for β-reductions by translating derivations into the layered type system (§ 5.4).

5.1 β-reductions

In this section, we define the β-reductions for the labeled system λ_*. The β-reductions describe the evaluation steps for function applications and let-bindings. We consider both additive and multiplicative function applications, as well as pairings and unit values. As β-reductions can be applied in any part of the expression, we define the reduction in two steps: (1) the head reduction $e \to_\beta^{\mathsf{hd}} e'$, where the reduction from e to e' is at the topmost expression, and $e \to_\beta e'$ where the reduction is possibly in a sub-expression of e.

Definition 5.1.1 (β head reduction)**:** For $e, e' \in \texttt{Expr}$, the head reduction $e \to_\beta^{\mathsf{hd}} e'$ is defined as:

$$\boxed{e \to_\beta^{\mathsf{hd}} e'}$$

$$\lambda x : A.\, e_1\, e_2 \to_\beta^{\mathsf{hd}} [I\, \langle x \mapsto e_2 \rangle] e_1 \qquad \lambda^* x : A.\, e_1\, e_2 \to_\beta^{\mathsf{hd}} [I\, \langle x \mapsto e_2 \rangle] e_1$$

$$\texttt{let}\, [x,\, y] = [e_1,\, e_2]\, \texttt{in}\, e_3 \to_\beta^{\mathsf{hd}} [I\, \langle x \mapsto e_1, y \mapsto e_2 \rangle] e_3$$

$$\texttt{let}\, \langle x,\, y \rangle = \langle e_1,\, e_2 \rangle\, \texttt{in}\, e_3 \to_\beta^{\mathsf{hd}} [I\, \langle x \mapsto e_1, y \mapsto e_2 \rangle] e_3 \qquad \texttt{let}\, \langle\rangle = \langle\rangle\, \texttt{in}\, e \to_\beta^{\mathsf{hd}} e$$

$$\texttt{let!}\, x = \texttt{return}\, e_1\, \texttt{in}\, e_2 \to_\beta^{\mathsf{hd}} [I\, \langle x \mapsto e_1 \rangle] e_2$$

Definition 5.1.2 (β-reductions)**:** The β-reductions $e \mapsto e'$ for λ_* are the reductions where a single head reduction is applied at any sub-expression of e. As there are many expression constructors, we only show the first few. The β-reductions for the other expression constructors are defined similarly.

$$\boxed{e \to_\beta e'}$$

$$\frac{e \to_\beta^{\mathsf{hd}} e'}{e \to_\beta e'} \qquad \frac{e_1 \to_\beta e_1'}{\texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2 \to_\beta \texttt{let}\, \langle\rangle = e_1'\, \texttt{in}\, e_2} \qquad \frac{e_2 \to_\beta e_2'}{\texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2 \to_\beta \texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2'}$$

$$\frac{e_1 \to_\beta e_1'}{[e_1,\, e_2] \to_\beta [e_1',\, e_2]} \qquad \frac{e_2 \to_\beta e_2'}{[e_1,\, e_2] \to_\beta [e_1,\, e_2']} \qquad \frac{e_1 \to_\beta e_1'}{\texttt{let}\, [x,\, y] = e_1\, \texttt{in}\, e_2 \to_\beta \texttt{let}\, [x,\, y] = e_1'\, \texttt{in}\, e_2}$$

$$\frac{e_2 \to_\beta e_2'}{\texttt{let}\, [x,\, y] = e_1\, \texttt{in}\, e_2 \to_\beta \texttt{let}\, [x,\, y] = e_1\, \texttt{in}\, e_2'} \qquad \frac{e \to_\beta e'}{\lambda x : A.\, e \to_\beta \lambda x : A.\, e'} \qquad \frac{e_1 \to_\beta e_1'}{e_1\, e_2 \to_\beta e_1'\, e_2}$$

$$\frac{e_2 \to_\beta e_2'}{e_1\, e_2 \to_\beta e_1\, e_2'} \qquad \ldots$$

The β-reduction rules for both the additive λ-abstraction $\lambda x : A.\, e$ and multiplicative λ-abstraction $\lambda^* x : A.\, e$ correspond to the β-reductions of the simply typed lambda calculus. In the β-reductions, the only reduction concerning monadic operations is for $\texttt{let!}\,\texttt{return}\,x = e_1\,\texttt{in}\,e_2$, which is added because $\texttt{return}\,x$ does not perform any heap operations.

Example 5.1.3 (Swap Twice)**:** Let $\texttt{swap} : A * A \Rightarrow A * A$ be the program defined as:

$$\texttt{swap} \stackrel{\text{def}}{=} \lambda p : A * A.\, \texttt{let}\,\langle x, y\rangle = p\,\texttt{in}\,\langle y, x\rangle$$

Then the program that applies `swap` to a pair twice can be β-reduced as:

$$\begin{aligned}
&\texttt{swap}\,(\texttt{swap}\,\langle x, y\rangle)\\
&\to_\beta \texttt{swap}\,(\texttt{let}\,\langle x', y'\rangle = \langle x, y\rangle\,\texttt{in}\,\langle y', x'\rangle)\\
&\to_\beta \texttt{swap}\,\langle y, x\rangle\\
&\to_\beta \texttt{let}\,\langle x', y'\rangle = \langle y, x\rangle\,\texttt{in}\,\langle y', x'\rangle\\
&\to_\beta \langle x, y\rangle
\end{aligned}$$

With respect to this example, the *preservation* property at the end of this section can be used to prove both: if $\Omega \vdash \texttt{swap}\,(\texttt{swap}\,\langle x, y\rangle) : A * A \mathbin{@} w$ then $\Omega \vdash \langle x, y\rangle : A \mathbin{@} w$, and additionally that β-reduction preserves denotations: $[\![\Omega \vdash \texttt{swap}\,(\texttt{swap}\,\langle x, y\rangle) : A * A \mathbin{@} w]\!] = [\![\Omega \vdash \langle x, y\rangle : A \mathbin{@} w]\!]$.

Remark: As the β-reductions do not reduce the heap operations `ref`, `replace` and `free`, they do not form a complete operational semantics for λ_*. A possible approach to extending β-reductions to a full operational semantics would be to add a second parameter σ to the reduction, resulting in reductions of the form $(e, \sigma) \to (e', \sigma')$, where σ, σ' describe the state of the heap before and after the step. This is the approach taken by for instance L3 [Morrisett et al. 2005].

5.2 Strengthening

The first step towards proving the preservation property is to reduce the number of structural rules that have to be considered, by combining the structural rules into more general structural rules. For this purpose, the structural rules are divided into two groups:

1. The structural rules which change only the context but not the result type and world.
2. The structural rules which do not change the context, but do change the result type and world.

Of the structural rules in the labeled calculus, only the STLC-SEP-CAST rule falls into the second category, whereas the remaining fall into the former. In fact, all structural rules that fall into the first category also do not change the assumption context Γ, and therefore describe only how labels and constraints can be added to the label and constraint contexts Σ and Θ. This property is captured by the concept of *strengthening*, which describes how each structural rule in the first category changes the label and constraint contexts.

Definition 5.2.1 (Constraint Strengthening)**:** Given well-formed constraint contexts $\Sigma \vdash \Theta\ \texttt{wf}$ and $\Sigma' \vdash \Theta'\ \texttt{wf}$, we say that $\Sigma; \Theta$ is a strengthening of $\Sigma'; \Theta'$, denoted $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$ if it is derivable by the following rules:

$$\boxed{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta}$$

$$\text{STR-ID}\ \frac{\Sigma \vdash \Theta\ \texttt{wf}}{\Sigma; \Theta \rightsquigarrow \Sigma; \Theta} \qquad \text{STR-SYM}\ \frac{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta \qquad (w_1, w_2 \triangleright w_3) \in \Theta}{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta, (w_2, w_1 \triangleright w_3)} \qquad \text{STR-UNIT-1}\ \frac{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta \qquad w_1 \in \Sigma_\epsilon}{\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta', (w_1, \epsilon \triangleright w_1)}$$

$$\text{STR-UNIT-2}\ \frac{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta \qquad (w_1, \epsilon \triangleright w_2) \in \Theta}{\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta, (w_2, \epsilon \triangleright w_1)}$$

$$\text{STR-ASSOC}\quad \frac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma_\epsilon \qquad (w_1, w_2 \triangleright w_{12}) \in \Theta \qquad (w_{12}, w_3 \triangleright w) \in \Theta}{\Sigma';\Theta' \rightsquigarrow \Sigma, k;\Theta, (w_2, w_3 \triangleright k), (w_1, k \triangleright w)}$$

$$\text{STR-ASSOC-UNIT-1}\quad \frac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma_\epsilon \qquad (w_1, \epsilon \triangleright w_1') \in \Theta \qquad (w_1', w_2 \triangleright w_3) \in \Theta}{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta, (w_1, w_2 \triangleright w_3)}$$

$$\text{STR-ASSOC-UNIT-2}\quad \frac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma_\epsilon \qquad (w_3, \epsilon \triangleright w_3') \in \Theta \qquad (w_1, w_2 \triangleright w_3) \in \Theta}{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta, (w_1, w_2 \triangleright w_3')}$$

The strengthening relation $\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta$ consists of a reflexivity rule STR-ID, as well as a single rule for each of the structural rules other than cast. The shape of the strengthening rule is derived from the shape of each structural rule. For instance, the STLC-SEP-SYM rule:

$$\frac{(w_1, w_2 \triangleright w_3) \in \Theta \qquad \Sigma;\Theta, (w_2, w_1 \triangleright w_3);\Gamma \vdash e : A \mathbin{@} w'}{\Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w'}$$

This rule states that for any constraint $(w_1, w_2 \triangleright w_3)$, the symmetrical constraint $(w_2, w_1 \triangleright w_3)$ can also be added to the context, which is similarly captured by the STR-SYM rule. The other strengthening rules are structured in a similar manner.

The main property of the strengthening relation is that it can be used instead of individual structural rules in typing derivations. Indeed, the following typing rule which applies a strengthening is admissible:

Lemma 5.2.2 (Strengthening Lemma)**:** The following rule is admissible for λ_*:

$$\text{APPLY-STR}\quad \frac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad \Sigma' \vdash \Gamma' \ \mathtt{wf} \qquad w \in \Sigma'_\epsilon \qquad \Sigma;\Theta;\Gamma' \vdash e : A \mathbin{@} w}{\Sigma';\Theta';\Gamma' \vdash e : A \mathbin{@} w}$$

Proof. By induction on the derivation of $\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta$, applying the corresponding structural rule in each case. □

Remark: As mentioned in a prior remark after Lemma 4.2.11, any typing derivation $\Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w$ assumes that $\Sigma;\Theta;\Gamma\ \mathtt{wf}$. Due to this convention, the assumption $\Sigma' \vdash \Gamma'\ \mathtt{wf}$ in the lemma above can be omitted.

Similar to how derivations with the strengthening rule can be replaced by standard structural rules, derivations in well typed contexts can be rewritten to use the strengthening rule instead of (non-cast) structural rules. For instance, the rule STLC-SEP-SYM corresponds to applying APPLY-STR using STR-SYM as follows:

$$\frac{\dfrac{(w_1, w_2 \triangleright w_3) \in \Theta \qquad \dfrac{}{\Sigma;\Theta \rightsquigarrow \Sigma;\Theta}}{\Sigma;\Theta \rightsquigarrow \Sigma;\Theta, (w_2, w_1 \triangleright w_3)} \qquad w \in \Sigma_\epsilon \qquad \Sigma;\Theta, (w_2, w_1 \triangleright w_3);\Gamma \vdash e : A \mathbin{@} w'}{\Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w}$$

Here the assumptions marked in blue correspond exactly to the premises of the STLC-SEP-SYM rule (and well-formedness of the context). The remaining structural rules other than cast can also be rewritten in a similar manner, as captured by the following corollary:

Corollary 5.2.3: Every application of a structural rule in a derivation $\Omega \vdash e : A \mathbin{@} w$ (given $\Omega\ \mathtt{wf}$) can be rewritten using only the following two structural rules:

$$\text{APPLY-STR}\quad \frac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma'_\epsilon \qquad \Sigma;\Theta;\Gamma' \vdash e : A \mathbin{@} w}{\Sigma';\Theta';\Gamma' \vdash e : A \mathbin{@} w} \qquad \text{CAST}\quad \frac{(w, \epsilon \triangleright w') \in \Theta \qquad \Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w}{\Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w'}$$

The introduction of the strengthening relation reduces the number of structural rules for λ_* from 7 to 2, but does not solve the problem of introducing new labels and constraints which complicate the inversion lemma. Before solving this issue, we first discuss and prove several properties of the strengthening relation.

Lemma 5.2.4 (Strengthening is Transitive)**:** If $\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_2; \Theta_2$ and $\Sigma_2; \Theta_2 \rightsquigarrow \Sigma_3; \Theta_3$ then $\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_3; \Theta_3$.

Proof. By induction on the derivation of $\Sigma_2; \Theta_2 \rightsquigarrow \Sigma_3; \Theta_3$. □

The first additional property of the strengthening relation is the transitivity of strengthening, which can be used to combine multiple applications of APPLY-STR into a single application, a consequence of transitivity which is discussed in § 5.3.

It was previously indicated that the strengthening relation can be considered similar to a thinning as, reading down the derivation tree, thinning adds extra labels and constraints, whereas strengthening removes specific labels and constraints. In fact, the labels and constraints removed by a strengthening can be added again by a thinning.

Lemma 5.2.5 (Inversion of Strengthening)**:** If $\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta$ and $\Sigma' \vdash \Gamma'$ `wf` then $\Sigma; \Theta; \Gamma' \supseteq \Sigma'; \Theta'; \Gamma'$.

Proof. By induction on the derivation of $\Sigma'; \Theta' \rightsquigarrow \Sigma; \Theta$. □

The inversion property of strengthening on its own does not appear particularly useful at first glance, as it merely adds back in the labels and constraints removed by the strengthening. However, the property is essential for restricting the places that context-changing structural rules can be applied in § 5.3.

The final property describes how strengthening interacts with thinning. Specifically, it states that strengthening and thinning commute, meaning that the thinning lemma (Lemma 4.3.4) also applies to APPLY-STR.

Lemma 5.2.6 (Strengthening Thinning)**:** If $\Sigma_1; \Theta_1; \Gamma_1 \supseteq \Sigma_2; \Theta_2; \Gamma_2$ and $\Sigma_2; \Theta_2 \rightsquigarrow \Sigma_3; \Theta_3$ then there exist Σ_4 and Θ_4 *s.t.*

$$\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_4; \Theta_4 \quad \textit{and} \quad \Sigma_4; \Theta_4; \Gamma_1 \supseteq \Sigma_3; \Theta_3; \Gamma_2$$

Proof. By induction on the derivation of $\Sigma_2; \Theta_2 \rightsquigarrow \Sigma_3; \Theta_3$. As all the cases are similar, we only give the base case STR-ID and one of the inductive cases STR-SYM:

(STR-ID) We have $\Sigma_1; \Theta_1; \Gamma_1 \supseteq \Sigma_2; \Theta_2; \Gamma_2$ and $\Sigma_2 \vdash \Theta_2$ `wf`.
It is necessary to prove that there are $\Sigma_4; \Theta_4$ *s.t.* $\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_4; \Theta_4$ and $\Sigma_4; \Theta_4; \Gamma_1 \supseteq \Sigma_2; \Theta_2; \Gamma_2$.
Take $\Sigma_4 = \Sigma_1$ and $\Theta_4 = \Theta_1$, then the result follows from Lemma 4.3.3 and STR-ID.

(STR-SYM) We have $\Sigma_1; \Theta_1; \Gamma_1 \supseteq \Sigma_2; \Theta_2; \Gamma_2$ and $\Sigma_2; \Theta_2 \rightsquigarrow \Sigma_3; \Theta_3$ and $(w_1, w_2 \triangleright w_3) \in \Theta_3$.
It is necessary to prove that there are Σ_4, Θ_4 *s.t.* $\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_4; \Theta_4$ and $\Sigma_4; \Theta_4; \Gamma_1 \supseteq \Sigma_3; \Theta_3, (w_2, w_1 \triangleright w_3); \Gamma_2$
By induction we have that there are Σ_4', Θ_4' *s.t.* (1) $\Sigma_1; \Theta_1 \rightsquigarrow \Sigma_4'; \Theta_4'$ and (2) $\Sigma_4'; \Theta_4'; \Gamma_1 \supseteq \Sigma_3; \Theta_3; \Gamma_2$.
By (2) we have that $(w_1, w_2 \triangleright w_3) \in \Theta_4'$. Pick $\Sigma_4 = \Sigma_4'$ and $\Theta_4 = \Theta_4'$, then the conclusions follow from STR-SYM and THINNING-THETA-TAKE.

The remaining cases have the same structure as STR-SYM and are therefore omitted. □

5.3 Layering

Having reduced the number of structural rules from 7 to merely 2 by introducing strengthenings, we move to decreasing the number of points at which these structural rules can be applied, and therefore the points at which inversion is tedious.

The specific approach is to layer the typing rules into 3 different judgments: those where the last applied rule is a logical rule $\vdash_L$, those where it is a casting rule $\vdash_C$ and those where it is an application of the strengthening rule $\vdash_S$.

Definition 5.3.1 (Layered Typing Rules)**:** The layered typing system of λ_* contains 3 types of typing judgments: $\Omega \vdash_L e : A \mathbin{@} w$ for the logical rules, $\Omega \vdash_C e : A \mathbin{@} w$ for the casting rules and $\Omega \vdash_S e : A \mathbin{@} w$ for the structural rules. Each of the typing judgments has the additional assumption Ω `wf`.

$$\boxed{\Omega \vdash_L e : A \mathbin{@} w}$$

LAYER-VAR
$$\frac{x : A \mathbin{@} w \in \Omega.\Gamma}{\Omega \vdash_L x : A \mathbin{@} w}$$

LAYER-A-UNIT-I
$$\frac{w \in \Omega.\Sigma_\epsilon}{\Omega \vdash_L [] : 1_a \mathbin{@} w}$$

LAYER-A-FUN-I
$$\frac{w \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful} \qquad \Omega, x : A \mathbin{@} w \vdash_C e : B \mathbin{@} w}{\Omega \vdash_L \lambda x : A.\, e : A \Rightarrow B \mathbin{@} w}$$

LAYER-A-FUN-E
$$\frac{\Omega \vdash_C e_1 : A \Rightarrow B \mathbin{@} w \qquad \Omega \vdash_C e_2 : A \mathbin{@} w}{\Omega \vdash_L e_1\, e_2 : B \mathbin{@} w}$$

LAYER-A-PAIR-I
$$\frac{\Omega \vdash_C e_1 : A \mathbin{@} w \qquad \Omega \vdash_C e_2 : B \mathbin{@} w}{\Omega \vdash_L [e_1,\, e_2] : A \wedge B \mathbin{@} w}$$

LAYER-A-PAIR-E
$$\frac{\Omega \vdash_C e_1 : A \wedge B \mathbin{@} w \qquad \Omega, x : A \mathbin{@} w, y : B \mathbin{@} w \vdash_C e_2 : C \mathbin{@} w'}{\Omega \vdash_L \texttt{let}\, [x,\, y] = e_1\, \texttt{in}\, e_2 : C \mathbin{@} w'}$$

LAYER-M-UNIT-I
$$\Omega \vdash_L \langle\rangle : 1_m \mathbin{@} \epsilon$$

LAYER-M-UNIT-E
$$\frac{\Omega \vdash_C e_1 : 1_m \mathbin{@} w' \qquad \Omega, (\epsilon, \epsilon \triangleright w') \vdash_S e_2 : C \mathbin{@} w}{\Omega \vdash_L \texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2 : C \mathbin{@} w}$$

LAYER-M-FUN-I
$$\frac{w \in \Omega.\Sigma_\epsilon \qquad \vdash A : \texttt{Stateful} \qquad \Omega, a, c, (w, a \triangleright c), x : A \mathbin{@} a \vdash_S e : B \mathbin{@} c}{\Omega \vdash_L \lambda^* x : A.\, e : A \multimap B \mathbin{@} w}$$

LAYER-M-FUN-E
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash_C e_1 : A \multimap B \mathbin{@} w_1 \qquad \Omega \vdash_C e_2 : A \mathbin{@} w_2}{\Omega \vdash_L e_1^*\, e_2 : B \mathbin{@} w}$$

LAYER-M-PAIR-I
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash_C e_1 : A \mathbin{@} w_1 \qquad \Omega \vdash_C e_2 : B \mathbin{@} w_2}{\Omega \vdash_L \langle e_1,\, e_2 \rangle : A * B \mathbin{@} w}$$

LAYER-M-PAIR-E
$$\frac{w' \in \Omega.\Sigma_\epsilon \qquad \Omega \vdash_C e_1 : A * B \mathbin{@} w \qquad \Omega, a, b, x : A \mathbin{@} a, y : B \mathbin{@} b, (a, b \triangleright w) \vdash_S e_2 : C \mathbin{@} w'}{\Omega \vdash_L \texttt{let}\, \langle x,\, y \rangle = e_1\, \texttt{in}\, e_2 : C \mathbin{@} w'}$$

LAYER-RETURN
$$\frac{\Omega \vdash_C e : A \mathbin{@} w}{\Omega \vdash_L \texttt{return}\, e : T\, A \mathbin{@} w}$$

LAYER-BIND
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash_C e_1 : T\, A \mathbin{@} w_1 \qquad \Omega, a, c, (w_2, a \triangleright c), x : A \mathbin{@} a \vdash_S e_2 : T\, B \mathbin{@} c}{\Omega \vdash_L \texttt{let!}\, x = e_1\, \texttt{in}\, e_2 : T\, B \mathbin{@} w}$$

LAYER-NEWREF
$$\frac{\Omega \vdash_C e : A \mathbin{@} w \qquad \vdash A : \texttt{Pure}}{\Omega \vdash_L \texttt{ref}_A\, e : T\, (\texttt{Ref}\, A) \mathbin{@} w}$$

LAYER-REPLACE
$$\frac{(w_1, w_2 \triangleright w) \in \Omega.\Theta \qquad \Omega \vdash_C e_1 : \texttt{Ref}\, A \mathbin{@} w_1 \qquad \Omega \vdash_C e_2 : B \mathbin{@} w_2 \qquad \vdash B : \texttt{Pure}}{\Omega \vdash_L \texttt{replace}_{A,B}\, e_1\, e_2 : T\, (A * \texttt{Ref}\, B) \mathbin{@} w}$$

LAYER-FREE
$$\frac{\Omega \vdash_C e : \texttt{Ref}\, A \mathbin{@} w}{\Omega \vdash_L \texttt{free}_A\, e : T\, A \mathbin{@} w}$$

$$\boxed{\Omega \vdash_C e : A @ w}$$

$$\frac{(w, \epsilon \rhd w') \in \Omega \qquad \Omega \vdash_L e : A @ w}{\Omega \vdash_C e : A @ w'} \text{ LAYER-CAST}$$

$$\boxed{\Omega \vdash_S e : A @ w}$$

$$\frac{\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta' \qquad w \in \Sigma_\epsilon \qquad \Sigma'; \Theta'; \Gamma \vdash_C e : A @ w}{\Sigma; \Theta; \Gamma \vdash_S e : A @ w} \text{ LAYER-STR-APPLY}$$

In the layered typing system, each logical rule is followed by a single casting rule, and possibly an application of the strengthening rule. For the logical rules, the assumptions only allow for strengthening rules when new labels or constraints are introduced. This limited application of strengthening relies on the fact that strengthening can be moved down derivations. For instance, consider an application of the LAYER-A-PAIR-I rule where strengthening is applied to the left element:

$$\frac{\dfrac{\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta' \qquad \Sigma'; \Theta'; \Gamma \vdash e_1 : A @ w}{\Sigma; \Theta; \Gamma \vdash e_1 : A @ w} \qquad \Sigma; \Theta; \Gamma \vdash e_2 : B @ w}{\Sigma; \Theta; \Gamma \vdash [e_1, e_2] : A \wedge B @ w}$$

The strengthening $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$ results in a thinning $\Sigma'; \Theta' \supseteq \Sigma; \Theta$ which can be applied to the right element. Therefore the application of strengthening can be moved down, resulting in the following derivation:

$$\dfrac{\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta' \qquad \dfrac{\Sigma; \Theta; \Gamma \vdash e_1 : A @ w \qquad \dfrac{\dfrac{\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'}{\Sigma'; \Theta' \supseteq \Sigma; \Theta} \qquad \Sigma; \Theta; \Gamma \vdash e_2 : B @ w}{\Sigma'; \Theta'; \Gamma \vdash e_2 : B @ w}}{\Sigma'; \Theta'; \Gamma \vdash [e_1, e_2] : A \wedge B @ w}}{\Sigma; \Theta; \Gamma \vdash [e_1, e_2] : A \wedge B @ w}$$

The same step can then be used for the second element, in the case that it also uses a strengthening rule.

The resulting derivation of the example above uses an application of thinning. The thinning lemma (Lemma 4.3.4) ensures that the thinned derivation exists in the non-layered typing system, but does not guarantee that no additional strengthening rules are introduced. If it were to introduce such strengthening rules, then the rewriting above would merely shift the strengthening between the two elements rather than only moving it down. In practice, the proof of the thinning lemma does not change the applied rules, and neither does the corresponding lemma for the layered typing system.

Lemma 5.3.2 (Thinning Preserves Layers)**:** If $\Omega' \supseteq \Omega$ and:

1. If $\Omega \vdash_L e : A @ w$ then $\Omega' \vdash_L e : A @ w$.
2. If $\Omega \vdash_C e : A @ w$ then $\Omega' \vdash_C e : A @ w$.
3. If $\Omega \vdash_S e : A @ w$ then $\Omega' \vdash_S e : A @ w$.

Proof. By mutual induction on the typing derivations using the same structure as Lemma 4.3.4. The case for $\vdash_S$ applies Lemma 5.2.6. □

In the non-layered typing system, structural rules can be applied at any point. As such, the strongest translation from any derivation in the non-layered system is to a derivation in the strengthening layer ($\vdash_S$). As such, we first prove an auxiliary lemma which converts derivations in the logical layer ($\vdash_L$) to derivations in the strengthening layer ($\vdash_S$).

Lemma 5.3.3: If $\Sigma; \Theta; \Gamma \vdash_L e : A @ w$ then $\Sigma; \Theta; \Gamma \vdash_S e : A @ w$.

Proof. Let D be the derivation $\Omega \vdash_L e : A \mathbin{@} w$, then a derivation of $\Omega \vdash_S e : A \mathbin{@} w$ is given by:

$$\dfrac{\dfrac{w \in \Sigma_\epsilon \qquad \dfrac{}{\Sigma;\Theta \rightsquigarrow \Sigma;\Theta}}{\Sigma;\Theta \rightsquigarrow \Sigma;\Theta,(w,\epsilon \triangleright w)} \qquad w \in \Sigma_\epsilon \qquad \dfrac{(w,\epsilon \triangleright w) \in \Theta,(w,\epsilon \triangleright w) \qquad \dfrac{\dfrac{D}{\Sigma;\Theta;\Gamma \vdash_L e : A \mathbin{@} w}}{\Sigma;\Theta,(w,\epsilon \triangleright w);\Gamma \vdash_L e : A \mathbin{@} w}}{\Sigma;\Theta,(w,\epsilon \triangleright w);\Gamma \vdash_C e : A \mathbin{@} w}}{\Sigma;\Theta;\Gamma \vdash_S e : A \mathbin{@} w}$$

where $w \in \Sigma_\epsilon$ follows from Lemma 4.2.11. □

The main result of the layered type system, is that any typing judgment that holds in the non-layered system, also holds for the strengthening layer of the layered system. This ensures that the properties for judgments in the strengthening layer of the layered system, also apply to judgments in the original non-layered system. Specifically, the inversion theorem for β-redexes in § 5.4.

Theorem 5.3.4 (Derivation Layering)**:** If $\Omega \vdash e : A \mathbin{@} w$ then $\Omega \vdash_S e : A \mathbin{@} w$.

The translation from the non-layered system to the layered system is algorithmic, and consists of applying several rules to rewrite derivations to a form that matches the layering structure. Before starting the proof itself, it is useful to give 3 commonly used rewritings.

The first rewriting combines adjacent applications of strengthening rules, which is used for the case of applying APPLY-STR. The translation uses Corollary 5.2.3 to reduce the number of structural rules in derivations from 7 to 2.

Rewriting 1: A derivation of the form:

$$\dfrac{\Sigma_1;\Theta_1 \rightsquigarrow \Sigma_2;\Theta_2 \qquad w \in \Sigma_{1\epsilon} \qquad \dfrac{\Sigma_2;\Theta_2 \rightsquigarrow \Sigma_3;\Theta_3 \qquad w \in \Sigma_{2\epsilon} \qquad \Sigma_3;\Theta_3;\Gamma \vdash e : A \mathbin{@} w}{\Sigma_2;\Theta_2;\Gamma \vdash e : A \mathbin{@} w}\text{APPLY-STR}}{\Sigma_1;\Theta_1;\Gamma \vdash e : A \mathbin{@} w}\text{APPLY-STR}$$

is rewritten to:

$$\dfrac{\dfrac{\Sigma_1;\Theta_1 \rightsquigarrow \Sigma_2;\Theta_2 \qquad \Sigma_2;\Theta_2 \rightsquigarrow \Sigma_3;\Theta_3}{\Sigma_1;\Theta_1 \rightsquigarrow \Sigma_3;\Theta_3}\text{Lemma 5.2.4} \qquad w \in \Sigma_{1\epsilon} \qquad \Sigma_3;\Theta_3;\Gamma \vdash e : A \mathbin{@} w}{\Sigma_1;\Theta_1;\Gamma \vdash e : A \mathbin{@} w}\text{APPLY-STR}$$

The second rewriting swaps the order of a cast and non-cast structural rules, which is used in the case of the STLC-SEP-CAST rule as well as in later proofs.

Rewriting 2: A derivation of the form:

$$\dfrac{(w,\epsilon \triangleright w') \in \Theta' \qquad \dfrac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma'_\epsilon \qquad \Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w}{\Sigma';\Theta';\Gamma \vdash e : A \mathbin{@} w}\text{APPLY-STR}}{\Sigma';\Theta';\Gamma \vdash e : A \mathbin{@} w'}\text{CAST}$$

is rewritten to:

$$\dfrac{\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta \qquad w \in \Sigma'_\epsilon \qquad \dfrac{(w,\epsilon \triangleright w') \in \Theta \qquad \Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w}{\Sigma;\Theta;\Gamma \vdash e : A \mathbin{@} w'}\text{CAST}}{\Sigma';\Theta';\Gamma \vdash e : A \mathbin{@} w'}\text{APPLY-STR}$$

The condition $(w,\epsilon \triangleright w') \in \Theta$ holds by the fact that $\Sigma';\Theta' \rightsquigarrow \Sigma;\Theta$ induces a weakening $\Sigma;\Theta;\Gamma' \supseteq \Sigma';\Theta';\Gamma'$ for any well-formed Γ' *s.t.* $\Sigma' \vdash \Gamma'$ `wf` (Lemma 5.2.5).

The final common rewriting combines two casts, which is also used in the STLC-SEP-CAST rule, as well as proofs of later theorems.

Rewriting 3 (Combining Casts)**:** A derivation of the form:

$$\dfrac{(w_2, \epsilon \triangleright w_3) \in \Theta \qquad \dfrac{(w_1, \epsilon \triangleright w_2) \in \Theta \qquad \Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w_1}{\Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w_2}\textsc{cast}}{\Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w_3}\textsc{cast}$$

is rewritten to:

$$\dfrac{\dfrac{\Sigma; \Theta \rightsquigarrow \Sigma; \Theta}{\dfrac{(w_1, \epsilon \triangleright w_2) \in \Theta \qquad (w_2, \epsilon \triangleright w_3) \in \Theta}{\Sigma; \Theta \rightsquigarrow \Sigma; \Theta; (w_1, \epsilon \triangleright w_3)}} \qquad \dfrac{\Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w_1}{\Sigma; \Theta; \Gamma, (w_1, \epsilon \triangleright w_3) \vdash e : A \mathbin{@} w_1}\textsc{cast}}{\Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w_3}\textsc{apply-str}$$

Proof (Theorem 5.3.4). By induction on the typing derivation of $\Omega \vdash e : A \mathbin{@} w$. As the structure of the proof is similar for most logical rules, we only show several cases.

*(*STLC-SEP-CAST*).* After induction on the derivation of the inner expression, the derivation is rewritten with Rewriting 2 followed by Rewriting 3 and finally Rewriting 1.

*(*APPLY-STR*).* Assume that the structural rule is written in the form of APPLY-STR. Apply the induction hypothesis followed by Rewriting 1 to get the desired form.

*(*STLC-SEP-VAR*).* Use LAYER-VAR to get a derivation of $\Omega \vdash_L x : A \mathbin{@} w$, followed by applying Lemma 5.3.3.

*(*STLC-SEP-A-FUN-I*).* Apply the induction hypothesis to get a derivation of $\Omega, x : A \mathbin{@} w \vdash_S e : B \mathbin{@} w$ and decompose it into (1) $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$ and (2) $\Sigma'; \Theta'; \Gamma, x : A \mathbin{@} w \vdash_C e : B \mathbin{@} w$. By (1) we also have $w \in \Sigma'_\epsilon$ so apply LAYER-A-FUN-I to get $\Sigma'; \Theta'; \Gamma \vdash_L \lambda x : A.\, e : A \Rightarrow B \mathbin{@} w$, followed by applying Lemma 5.3.3 and Rewriting 1 to get $\Omega \vdash_S \lambda x : A.\, e : A \Rightarrow B \mathbin{@} w$.

*(*STLC-SEP-M-PAIR-I*).* Apply the induction hypothesis to get derivations of the form (1a) $\Sigma; \Theta; \Gamma \vdash_S e_1 : A \mathbin{@} w_1$ and (1b) $\Sigma; \Theta; \Gamma \vdash_S e_2 : B \mathbin{@} w_2$. Decompose (1a) into (2a) $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$ and (2b) $\Sigma'; \Theta'; \Gamma \vdash_C e_1 : A \mathbin{@} w_1$. Apply Lemma 5.3.2 to (1b) using (2a) to find $\Sigma'; \Theta'; \Gamma \vdash_S e_2 : A \mathbin{@} w_2$, which is then decomposed into (3a) $\Sigma'; \Theta' \rightsquigarrow \Sigma''; \Theta''$ and (3b) $\Sigma''; \Theta''; \Gamma \vdash_C e_2 : B \mathbin{@} w_2$. Similarly apply Lemma 5.3.2 to (2b) using (3a) to get $\Sigma''; \Theta''; \Gamma \vdash_S e_1 : A \mathbin{@} w_1$. Apply LAYER-M-PAIR-I to get $\Sigma''; \Theta''; \Gamma \vdash_L \langle e_1, e_2 \rangle : A * B \mathbin{@} w$ and then apply Lemma 5.3.3 followed by Rewriting 1 twice using (3a) and (2a) respectively.

*(*STLC-SEP-BIND*).* Apply the induction hypothesis to get a derivation of $\Sigma; \Theta; \Gamma \vdash_S e_1 : T\, A \mathbin{@} w_1$ and decompose it into (1a) $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$ and (1b) $\Sigma'; \Theta'; \Gamma \vdash_C e_1 : T\, A \mathbin{@} w_1$. Similarly apply the induction hypothesis to get a derivation of (2) $\Sigma; \Theta; \Gamma, a, c, (w_1, a \triangleright c), x : A \mathbin{@} a \vdash_S e_2 : T\, B \mathbin{@} c$. (1a) results in a thinning $\Sigma'; \Theta'; \Gamma \supseteq \Sigma; \Theta; \Gamma$, which can be extended to $\Sigma', a, c; \Theta', (w_2, a \triangleright c); \Gamma, x : A \mathbin{@} a \supseteq \Sigma', a, c; \Theta', (w_2, a \triangleright c); \Gamma, x : A \mathbin{@} a$ and applied to (2), resulting in (3) $\Sigma', a, c; \Theta', (w_1, a \triangleright c); \Gamma, x : A \mathbin{@} a \vdash_S e_2 : T\, B \mathbin{@} c$. Apply LAYER-BIND to get $\Sigma'; \Theta'; \Gamma \vdash_L \texttt{let!}\ x = e_1\ \texttt{in}\ e_2 : T\, B \mathbin{@} w$ followed by Lemma 5.3.3 and finally by Rewriting 1 using (1a).

The remaining cases are similar and therefore omitted. □

As all typing rules in the layered system are also typing rules in the original non-layered system (or admissible in the case of LAYER-STR-APPLY), valid typing judgments in the layered system are also valid judgments in the original system:

Corollary 5.3.5 (Unlayering)**:** If $\Sigma; \Theta; \Gamma \vdash_k e : A \mathbin{@} w$ for some $k \in \{L, C, S\}$ then $\Sigma; \Theta; \Gamma \vdash e : A \mathbin{@} w$

Proof. By induction on $\Sigma; \Theta; \Gamma \vdash_k e : A \mathbin{@} w$ and applying Lemma 5.2.2 for LAYER-STR-APPLY. □

5.4 Preservation

The most important property of the layered system is that for any expression e, there is at most 1 logical rule which applies for a judgment in the logical layer $\Omega \vdash_L e : A \mathbin{@} w$. As such, a standard inversion theorem can be given for the logical layer.

Theorem 5.4.1 (Logical Layer Inversion Theorem)**:** If $\Sigma; \Theta; \Gamma \vdash_L e : R \mathbin{@} w$ then:

1. If $e = x$ then $x : R \mathbin{@} w \in \Gamma$.
2. If $e = []$ then $R = 1_{\mathsf{a}}$ and $w \in \Sigma_\epsilon$
3. If $e = \lambda x : A.\, e_1$ there is a $B \in \texttt{Type}$ *s.t.* $R = A \Rightarrow B$ and $w \in \Sigma_\epsilon$ and $\Sigma; \Theta; \Gamma, x : A \mathbin{@} w \vdash_C e_1 : B \mathbin{@} w$
4. If $e = e_1\, e_2$ then there is an $A \in \texttt{Type}$ *s.t.* $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \Rightarrow R \mathbin{@} w$ and $\Sigma; \Theta; \Gamma \vdash_C e_2 : A \mathbin{@} w$.
5. If $e = [e_1, e_2]$ then there are $A, B \in \texttt{Type}$ *s.t.* $R = A \wedge B$ and $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \mathbin{@} w$ and $\Sigma; \Theta; \Gamma \vdash_C e_2 : B \mathbin{@} w$.
6. If $e = \texttt{let}\, [x, y] = e_1\, \texttt{in}\, e_2$ then there are $A, B \in \texttt{Type}$ and $w' \in \Sigma_\epsilon$ *s.t.* $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \wedge B \mathbin{@} w'$ and $\Sigma; \Theta; \Gamma, x : A \mathbin{@} w, y : B \mathbin{@} w \vdash_C e_2 : R \mathbin{@} w$.
7. If $e = \langle\rangle$ then $R = 1_{\mathsf{a}}$ and $w = \epsilon$.
8. If $e = \texttt{let}\, \langle\rangle = e_1\, \texttt{in}\, e_2$ then there is a $w' \in \Sigma_\epsilon$ *s.t.* $\Sigma; \Theta; \Gamma \vdash_C e_1 : 1_{\mathsf{a}} \mathbin{@} w'$ and $\Sigma; \Theta, (\epsilon, \epsilon \triangleright w'); \Gamma \vdash_S e_2 : R \mathbin{@} w$.
9. If $e = \lambda^* x : A.\, e_1$ then there is a $B \in \texttt{Type}$ *s.t.* $R = A \twoheadrightarrow B$ and $w \in \Sigma_\epsilon$ and $\Sigma, a, c; \Theta, (w, a \triangleright c); \Gamma, x : A \mathbin{@} a \vdash_S e_1 : B \mathbin{@} c$.
10. If $e = e_1^*\, e_2$ then there is an $A \in \texttt{Type}$ and $(w_1, w_2 \triangleright w) \in \Theta$ *s.t.* $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \twoheadrightarrow R \mathbin{@} w_1$ and $\Sigma; \Theta; \Gamma \vdash_C e_2 : A \mathbin{@} w_2$.
11. If $e = \langle e_1, e_2 \rangle$ then there are $A, B \in \texttt{Type}$ and $(w_1, w_2 \triangleright w) \in \Theta$ *s.t.* $R = A * B$, $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \mathbin{@} w_1$ and $\Sigma; \Theta; \Gamma \vdash_C e_2 : B \mathbin{@} w_2$.
12. If $e = \texttt{let}\, \langle x, y \rangle = e_1\, \texttt{in}\, e_2$ then there are $A, B \in \texttt{Type}$ and $w' \in \Sigma_\epsilon$ *s.t.* $\Sigma; \Theta; \Gamma \vdash_C e_1 : A * B \mathbin{@} w'$ and $\Sigma, a, b; \Theta, (a, b \triangleright w'); \Gamma, x : A \mathbin{@} a, y : B \mathbin{@} b \vdash_S e_2 : R \mathbin{@} w$.
13. If $e = \texttt{return}\, e_1$ then there is a $A \in \texttt{Type}$ *s.t.* $R = R\, A$ and $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \mathbin{@} w$.
14. If $e = \texttt{let!}\, x = e_1\, \texttt{in}\, e_2$ then there are $A, B \in \texttt{Type}$ and $(w_1, w_2 \triangleright w) \in \Theta$ *s.t.* $R = T\, B$, $\Sigma; \Theta; \Gamma \vdash_C e_1 : T\, A \mathbin{@} w_1$ and $\Sigma, a, c; \Theta, (w_2, a \triangleright c); \Gamma, x : A \mathbin{@} a \vdash_S e_2 : T\, B \mathbin{@} c$.
15. If $e = \texttt{ref}_A\, e_1$ then there is an $A \in \texttt{Type}$ *s.t.* $R = T\, (\texttt{Ref}\, A)$, $\vdash A : \texttt{Pure}$ and $\Sigma; \Theta; \Gamma \vdash_C e_1 : A \mathbin{@} w'$.
16. If $e = \texttt{replace}_{A,B}\, e_1\, e_2$ then there are $A, B \in \texttt{Type}$ and $(w_1, w_2 \triangleright w) \in \Theta$ *s.t.* $R = T\, (A * \texttt{Ref}\, B)$, $\vdash B : \texttt{Pure}$, $\Sigma; \Theta; \Gamma \vdash_C e_1 : \texttt{Ref}\, A \mathbin{@} w_1$ and $\Sigma; \Theta; \Gamma \vdash_C e_2 : B \mathbin{@} w_2$.
17. If $e = \texttt{free}_A\, e_1$ then there is an $A \in \texttt{Type}$ *s.t.* $R = T\, A$ and $\Sigma; \Theta; \Gamma \vdash_C e_1 : \texttt{Ref}\, A \mathbin{@} w$.

Proof. By definition of the typing rules in the logical layer. □

The inversion lemma for the logical layer cannot be directly applied to prove a preservation theorem, as there can still be casts and strengthenings between the two elements of a reduction. For instance, for $\Omega \vdash_L (\lambda x : A.\, e_1)\, e_2 : B \mathbin{@} w$, the inversion lemma gives only that $\Omega \vdash_C \lambda x : A.\, e_1 : A \Rightarrow B \mathbin{@} w$ in the cast layer, rather than the logical layer necessary to apply the inversion lemma again.

The next approach would be to extend the inversion lemma to the cast layer, which would allow us to apply it to both $(\lambda x : A.\, e_1)\, e_2$ and $\lambda x : A.\, e_1$. However, for instance in the STLC-SEP-EQ-M-PAIR-BETA rule, the world of the two elements of the pair combined is also used in typing the body of the let expression. As such, an inversion lemma for the cast layer would also not be sufficient for proving a preservation theorem.

Instead, we fall back to proving an inversion lemma for each of the β-redexes in the strengthening layer. This allows for removing the cast between the β-redex rules in the derivation. This possibly adds more strengthening rules, which can be moved down as in the proof of Theorem 5.3.4. At this point, standard inversion can be applied to get the required conclusions for each redex.

Theorem 5.4.2 (Head reduction preservation)**:** If $\Omega \vdash_L e : A \mathbin{@} w$ and $e \to_\beta^{\mathsf{hd}} e'$ then $\Omega \vdash e \equiv e' : A \mathbin{@} w$.

Proof. By case analysis on $e \to_\beta^{\mathsf{hd}} e'$ as follows. As the cases are similar, we only show the cases for $e = (\lambda^* x : B.\, e_1)^*\, e_2$. We also split Ω into $\Sigma; \Theta; \Gamma$.

Case $e = (\lambda^* x : A.\, e_1)^*\, e_2$**.** First apply Theorem 5.4.1 to $\Sigma; \Theta; \Gamma \vdash (\lambda^* x : B.\, e_1)^*\, e_2 : B \mathbin{@} w$ to find an (1a) $(w_1, w_2 \triangleright w) \in \Theta$ *s.t.* (1b) $\Sigma; \Theta; \Gamma \vdash_C \lambda^* x : B.\, e_1 : B \twoheadrightarrow A \mathbin{@} w_1$ and (1c) $\Sigma; \Theta; \Gamma \vdash_C e_2 : B \mathbin{@} w_2$.
Next by case analysis on $\Sigma; \Theta; \Gamma \vdash_C \lambda^* x : B.\, e_1 : B \twoheadrightarrow A \mathbin{@} w_1$ we find (2a) $(w_1', \epsilon \triangleright w_1) \in \Theta$ and (2b) $\Sigma; \Theta; \Gamma \vdash \lambda^* x : B.\, e_1 : B \twoheadrightarrow A \mathbin{@} w_1'$.
Apply Theorem 5.4.1 to (2b) to find (3a) $w_1' \in \Sigma_\epsilon$ and (3b) $\Sigma, a, c; \Theta, (w_1', a \triangleright c); \Gamma, x : B \mathbin{@} a \vdash_C e_1 : A \mathbin{@} c$.
Apply thinning to (3b) and (1b) to find (4a) $\Sigma, a, c; \Theta, (w_1', w_2 \triangleright w), (w_1', a \triangleright c); \Gamma, x : B \mathbin{@} a \vdash_C e_1 : A \mathbin{@} c$ and (4b) $\Sigma; \Theta, (w_1', w_2 \triangleright w); \Gamma \vdash_C e_2 : B \mathbin{@} w_2$.
Apply STLC-SEP-EQ-M-FUN-BETA to (4a) and (4b) to find $\Sigma; \Theta, , (w_1', w_2 \triangleright w); \Gamma \vdash (\lambda^* x : B.\, e_1)^*\, e_2 \equiv [I\,\langle x \mapsto e_2\rangle] e_1 : A \mathbin{@} w$.
Finally apply the congruence rule corresponding to the STLC-SEP-ASSOC-UNIT-1 rule using (2a) to find $\Sigma; \Theta; \Gamma \vdash (\lambda^* x : B.\, e_1)^*\, e_2 \equiv [I\,\langle x \mapsto e_2\rangle] e_1 : A \mathbin{@} w$.
The remaining cases are similar. □

The conclusions for each β-redex in the theorem above correspond to a β-equivalence rule in Definition 4.3.12 for $\Sigma'; \Theta'; \Gamma \vdash e : A \mathbin{@} w'$. As such, by applying the corresponding congruence rules for casting back to w and those corresponding to the strengthening $\Sigma; \Theta \rightsquigarrow \Sigma'; \Theta'$, we finally prove the necessary preservation properties:

Theorem 5.4.3 (Preservation)**:** If $\Omega \vdash e : A \mathbin{@} w$ and $e \to_\beta e'$ then $\Omega \vdash e \equiv e' : A \mathbin{@} w$.

Proof. The proof follows from Theorem 5.3.4 and Lemma 5.4.4 below. □

By applying Theorem 4.3.13 and Theorem 4.4.18 to the result of the preservation theorem, we derive both the standard preservation property as in for instance Harper [2016], and that β-reduction preserves the denotational semantics of the program.

Even though Theorem 5.4.3 considers general derivations in λ_*, the proof depends on the layered structure of typing derivations, as specified in the following lemma.

Lemma 5.4.4 (Layered Preservation)**:**

1. If $\Omega \vdash_L e : A \mathbin{@} w$ and $e \to_\beta e'$ then $\Omega \vdash e \equiv e' : A \mathbin{@} w$.
2. If $\Omega \vdash_C e : A \mathbin{@} w$ and $e \to_\beta e'$ then $\Omega \vdash e \equiv e' : A \mathbin{@} w$.
3. If $\Omega \vdash_S e : A \mathbin{@} w$ and $e \to_\beta e'$ then $\Omega \vdash e \equiv e' : A \mathbin{@} w$.

Proof. The proof consists of mutual induction on the derivation of $\Omega \vdash_j e : A \mathbin{@} w$ for $j \in \{L, C, S\}$. The proof for these layers have the following structure:

($\Omega \vdash_L e : A \mathbin{@} w$) If $e \to_\beta e'$ is $e \to_\beta^{\mathsf{hd}} e'$, then apply Theorem 5.4.2. Otherwise, apply induction and the congruence rule corresponding to the logical rule.

($\Omega \vdash_C e : A \mathbin{@} w$) Apply induction, followed by the congruence rule for STLC-SEP-CAST.

($\Omega \vdash_S e : A \mathbin{@} w$) By case analysis on the derivation, we find Σ', Θ' *s.t.* $\Omega.\Sigma; \Omega.\Theta \rightsquigarrow \Sigma'; \Theta'$ and $\Sigma'; \Theta'; \Omega.\Gamma \vdash_C e : A \mathbin{@} w$. Using induction we find $\Sigma'; \Theta'; \Omega.\Gamma \vdash e \equiv e' : A \mathbin{@} w$. The case concludes by induction on $\Omega.\Sigma; \Omega.\Theta \rightsquigarrow \Sigma'; \Theta'$, applying the congruence rule of the structural rule corresponding to the case of the *strengthening*. □

Chapter 6

Related Work

Bunched Implications. The logic of bunched implications was introduced by O'Hearn and Pym [1999] as a logic containing both additive and multiplicative versions of implication, as complements to additive and multiplicative conjunction in linear logic [Girard 1995] respectively. They introduce the concept of *bunched contexts*, where contexts can be combined using either additively ';' or multiplicatively ','. This results in the following two proof rules:

$$\frac{\Gamma; A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

Where $\to$ and $\multimap$ are the additive and multiplicative implication respectively. As bunches can be combined in two ways, they form a tree-like structure. In later work, Pym [2002] presents the $\alpha\lambda$-calculus in direct Curry-Howard correspondence with their logic of bunched implications. O'Hearn [1999] separately discusses the resource sharing interpretations of $\alpha\lambda$ and a related affine variant which admits weakening of both additive and multiplicative bunches.

To the author's knowledge, the most recent works on bunched implications as type theory for λ-calculi are Berdine and O'Hearn [2006], Collinson and Pym [2006], and Collinson et al. [2008]. In Berdine and O'Hearn [2006], the authors present a language with bunched implications and linear references. Their heap operations are modeled in a continuation passing style, rather than the monadic style used in our work. Additionally, they restrict the values on the heap to integers and a *true* value that do not carry specific resources, whereas we restrict the values on the heap to those that do not access the heap, including other references. Finally, they present a denotational model directly on the bunched calculus.

In Collinson et al. [2008], the authors extend the $\alpha\lambda$ calculus with polymorphism similar to $\lambda 2$ for the λ-calculus. They discuss two types of polymorphism, one where the type variables are intuitionistic as in λP, and one where the type variables themselves form a bunched context. For the case of bunched type variables, they present a categorical semantics, and prove preservation (subject-reduction) and strong normalization of $\beta\eta\zeta$-reductions, where ζ reductions lift `let` bindings out of subexpressions. Collinson and Pym [2006] similarly describe a form of polymorphism, but on variables representing memory regions rather than types. Similar to Berdine and O'Hearn [2006], they add heap operations in a continuation passing style, and give a denotational semantics for their language. Unlike our work, none of these languages erase to a language without BI-based types, but rather define a denotational semantics directly on the BI-based language and bunched contexts.

Labeled Calculi. Labeled sequent calculi are sequent calculi where each assumption or conclusion is additionally assigned a label. They can be applied to describe both more restrictive logics, such as intuitionistic and substructural logics [Balat and Galmiche 2000], and modal logics [Baldoni 2000; Ghari 2017]. Combing commonalities between these labeled systems, Viganò [2000] described a general labeled sequent calculus framework for describing non-classical logics. The semantics of their framework is based on a general Kripke semantics of possible worlds [Kripke 1963], where the labels in the calculus represent *possible worlds* in the Kripke semantics.

In the original paper introducing bunched implications [O'Hearn and Pym 1999], published before the framework, the authors present such a Kripke resource semantics of BI. Hóu et al. [2015] develop a labeled sequent calculus for BBI – a classical variant of BI – based on this Kripke semantics and inspired

by labeled sequent calculi in modal logic. They prove that the labeled calculus is equivalent to the Hilbert system for BBI [Galmiche and Larchey-Wendling 2006].

Galmiche et al. [2019] similarly describe a labeled sequent calculus for BI based on the Kripke resource semantics, and study the relation between the labeled and bunched sequent calculi for BI. They show that the labeled sequent calculus is sound *w.r.t.* the bunched calculus but were only able to prove that proofs in the bunched calculus of a certain form – namely satisfying a *tree property* – could be converted to proofs in the labeled calculus.

In this work, we defined a labeled intuitionistic logic based on the labeled sequent calculi in these prior works. Unlike these works, we used this logic to define a λ-calculus similar to $\alpha\lambda$, but without bunches, for which we give and verify a denotational semantics. Hence, whereas prior works on BI focuses on the logic side of the Curry-Howard correspondence of BI, our work focuses on the computational side.

Separation Logic. Reynolds [2002] introduced separation logic as an extension of Hoare logic for better reasoning about programs with shared state, such as low level programs with shared pointers. It introduces the separating conjunction $*$, which allows reasoning about disjoint parts of the heap. Realizing that separation logic is an instance of Bunched Implications, they also introduced the $-\!\!*$ connective.

In separation logic, programs are verified by proving Hoare triples of the form $\{P\}e\{Q\}$, which states that if P holds for a certain heap before executing e, then Q holds for the updated heap after executing e. This 'executing' behavior of the program e is formally specified by the operational semantics of the programming language. An important rule in separation logic is the frame rule, which states that the parts of the heap not mentioned in the pre- and post-condition are not affected by the execution of the program. Based on this property, O'Hearn [2007] extended separation logic with a parallel rule to reason about concurrent programs.

The logic of concurrent separation logic has been formalized in theorem provers, such as VST-Floyd [Cao et al. 2018] and Iris [Jung et al. 2018] for verifying separation logic Hoare triples for programs.

In such separation logic frameworks, there is a clear distinction between the language of programs, CLight – a subset of C – for VST-Floyd and for instance HeapLang for Iris, and the verification of these programs using separation logic. This allows for a clear separation between the programs themselves and the verification of these programs, which can for instance be used to prove multiple properties for the same program, for instance for different inputs or different levels of abstraction [Somers and Krebbers 2024].

Our system on the other hand specifies the separation logic in the type system of the language, rather than as a separate logic, allowing for a more direct correspondence between the programs and their verification. This is similar to the approach taken by for instance Morrisett et al. [2005] in the L3 language, where the type system is extended with linear types to describe resource ownership.

Resource Ownership Type Systems. There are two main approaches to adding resource ownership to type systems themselves. The first approach is to add specific resource types to the type system, such as capability types in L3 [Morrisett et al. 2005]. The second approach is to add resources as type refinements to the language, as is the case in CN [Pulte et al. 2023] and RefinedC [Sammler et al. 2021].

L3 introduces first-class resource types called capabilities to the type system, to track the permission to access memory locations separately from the pointers themselves. They use a linear type system to ensure that capabilities are not duplicated or discarded, and employ a specific type constructor to describe types such as pointers which can be explicitly duplicated and discarded. Due to the linear type system, the language only contains the linear function type.

RefinedC [Sammler et al. 2021] is a type refinement system for a subset of the C language. In RefinedC, types in this subset of C (called Cesium) are annotated with refinements, which describe not only restrictions on values, but also ownership of values. Type checking in RefinedC is done by semantic typing, namely by defining refinement types and typing judgments in terms of Iris' separation logic, where typing rules are proven as lemmas. The typing rules are defined in terms of a fragment of separation logic, and type checking corresponds to finding a proof in this fragment. The specific separation logic fragment used to permit proof search without backtracking does not include the additive implication. Hence, their approach to typing uses neither bunches as in $\alpha\lambda$ nor labels as in our work, but rather a proof procedure for a fragment of separation logic.

Another refinement type system for C with resources is CN [Pulte et al. 2023]. It is based on Cesium [Memarian et al. 2016], a description of a large part of the ISO C standard by translation into a smaller Core calculus. Rather than defining refinements on the C language itself, it defines these typing refinements with respect to this Core calculus. CN internally uses SMT solving as part of their type checking by translating their separation logic formulas into SMT formulas. Their type refinements consists of a combination of first-class linear resource ownership types (as in L3) combined by separation logic connectives $*$ and $-\!\!*$, and refinements in the form of boolean predicates. They restrict the use of the additive implication and conjunction to only boolean predicates, rather than the general additive BI connectives considered in our work.

Chapter 7

Conclusion and Future Work

In this thesis, we introduced λ_*, a λ-calculus with bunched implications, extended with linear references that support strong updates via a monadic type. λ_* includes two types of functions and products: those that share resources, as in intuitionistic logic, and those that do not, as in linear logic. Rather than describing the typing system using the tree-shaped bunched contexts used in the original BI logic and associated $\alpha\lambda$-calculus [Pym 2002], we employ a *labeling* approach commonly used in sequent calculi for modal logics [Baldoni 2000; Ghari 2017; Viganò 2000], resulting in a type system that admits both weakening and contraction.

The types and expressions in λ_* erase to those in a simply typed $\lambda_{\texttt{loc}}$ with monadic heap operations and non-typed references that do not track resources. A denotational semantics for $\lambda_{\texttt{loc}}$ was provided in which the monadic type was interpreted as partial functions from heaps to a result and an updated heap. Such a function is undefined when a heap operation fails and therefore gets *stuck*. We defined β- and η-equivalences on simply typed terms as an equivalence judgment, ensuring that typeable β, η-equivalent programs have the same interpretation in the denotational semantics. Additionally, we proved an analog to the *preservation* property (also known as *subject-reduction*) of operational semantics for the equivalence judgment by demonstrating that the typeability of the β-redex is sufficient to prove typeable β-equivalence. This proof follows from a standard inversion lemma on the form of expressions [Pierce 2002].

Subsequently, we demonstrated that λ_* can be interpreted as a refinement type system of $\lambda_{\texttt{loc}}$. This type refinement preserves both typing judgments and equivalence judgments, allowing us to reuse the denotational semantics of $\lambda_{\texttt{loc}}$ for λ_*. By defining a logical predicate on labeled types in λ_*, we further showed that well-typed programs are semantically safe, meaning that no heap operations fail in the denotational semantics.

Finally, for the pure (non heap changing) part of λ_*, we defined β-reductions and proved the *preservation* property by adding layering to the typing rules of λ_*, restricting the places at which structural rules can be applied. The *preservation* theorem for the pure fragment provides a strong initial step towards proving the syntactic soundness of the bunched implication calculus with respect to a potential operational semantics. We conclude the thesis with a discussion of this and other possible future work.

Future Work

Operational Semantics. In our work we focused on a denotational semantics of typing judgments. By proving regularity, we showed that the denotational interpretation of well-typed expressions in the labeled system do not get stuck. An alternative semantics is an *operational semantics*, in which the execution steps of untyped expressions are defined in the form of reductions, where *stuck* operations such as attempting to free a non-existent location would have no further reductions. In an operational semantics, the safety property states that well-typed programs do not get *stuck*. Safety is commonly proven using a combination of *preservation*, which states that reductions maintain well-typedness, and *progress*, which states that well-typed programs are either values, or can take another step [Harper 2016; Pierce 2002]. We have proven the former property for the reductions that do not change the heap (*i.e.* β-reductions) but not for the heap-changing operations. Defining an operational semantics, proving safety

w.r.t. the operational semantics and relating the operational and denotational semantics are possibilities for future work.

Higher-Order Store and Recursion. To ensure that the interpretations of programs in λ_* are terminating, and to simplify the denotational model of heaps, we excluded the monadic type $T\,A$ from being stored on the heap, excluding higher-order store.

As including higher-order store would permit general recursion due to Landin's Knot [Landin 1964], extending λ_* would require a more complex denotational semantics to account for both non-terminating programs and programs that get stuck. A possible approach to account for general recursion would be to use step indexing and guarded recursion using a later modality [Appel and McAllester 2001; Appel et al. 2007].

This approach can be used to model general recursion in both operational semantics [Appel et al. 2007] and denotational semantics [Møgelberg and Paviotti 2019]. The approach for operational semantics has been mechanized by Iris [Jung et al. 2018]. Recently, Frumin et al. [2022] similarly mechanized an approach for denotational semantics, building on the Iris framework. It would be interesting to study how the labeling approach for resources in our work would interact with step indexing to support higher order store.

Dependent Types. Another direction for future work would be to extend the calculus with dependent products and dependent functions to correspond with predicate, rather than propositional BI. In such a dependent type system, the linear references $\texttt{Ref}\,A$ could instead be replaced by locations $\texttt{loc}$ as in $\lambda_{\texttt{loc}}$, and linear capabilities $l \mapsto A$ corresponding to location l containing a value of type A.

This resembles the pointers and capabilities in the L3 language [Morrisett et al. 2005]. In L3, references consist of two parts: the location of the heap itself $\texttt{Ptr}\,\rho$ and a linear capability to access the location $\texttt{Cap}\,\rho\,A$, where the name ρ relates pointers and capabilities of the same location. These could be modeled by $\texttt{loc}$ and $l \mapsto A$ in a dependent version of our system respectively.

Similarly, the *pointsto* connective of separation logic $l \mapsto_A v$ – describing that a location contains a *specific value* v of type A – could be modeled as a capability with type. In a dependent system with equalities, the dependent pair $(\Sigma v' : A) \times (v = v')$ consists of a value v' of type A and a proof that v is equal to v'. Hence, the *pointsto* connective for a specific value could be modeled as:
$l \mapsto_A v \stackrel{\text{def}}{=} l \mapsto (\Sigma v' : A) \times (v = v')$.

A key challenge in extending λ_* with dependent types is finding a suitable notion of equality. For instance, the capability $l \mapsto A$ and the equality $v = v'$ have influence on the computation, and hence should be possible to erase. A naive approach would be to erase both to unit values, and to erase all dependent pairs and products to standard pairs and products. However, this approach would result in a significant number of irrelevant units. A more sophisticated approach would be that of Ghalayini and Krishnaswami [2023], who describe a dependently typed calculus with clear separation between types and propositions, that allows propositions and type dependencies to be erased similar to how we erase resource tracking through labels. By combining our approaches, future work could create a separation logic style calculus where both resource tracking (including capabilities and equalities) and type dependency can be erased without introducing unnecessary units.

Type Checking and Label Inference. In our work, we defined a type system for the λ_*, a λ-calculus with bunched function types. In this type theory, typing derivations are not unique due to the addition of structural rules. Whereas the next logical rule to apply during type checking is driven by the form of the expression, where to apply structural rules and which worlds and constraints to use for each rule are not.

As such, creating a type checker for the labeled system would require a mechanism to infer both when to apply structural rules, and which worlds to assign to expressions. As this approach is closely related to proof search in logics – but fixing the order in which logical rules are applied – we believe that approaches used for proof search in such logics could be adapted to type checking, and possibly type inference, for λ_*.

Most closely related are the approaches for proof search in sequent calculi for bunched implications. Specifically, Hóu et al. [2015] describe a labeled sequent calculus for Boolean BI (BBI), a classical variant of BI. They describe an approach to proof search in which their labeled sequent calculus is converted to a system without structural rules, by generating constraints when applying logical rules. These constraints are then solved and checked at the end of the proof search. This approach resembles that of constraint-

based type inference used by for instance ML [Pottier and Rémy. 2004] and Haskell [Peyton Jones 2019]. Future work could combine these approaches to provide both type inference and the world and constraint inference necessary to type check programs in λ_*.

References

Carlo Angiuli and Daniel Gratzer. 2024. *Principles of Dependent Type Theory*. Retrieved Oct. 1, 2024 from https://www.danielgratzer.com/courses/type-theory-s-2024/lecture-notes.pdf.

Andrew W. Appel and David McAllester. Sept. 2001. "An indexed model of recursive types for foundational proof-carrying code." *ACM Trans. Program. Lang. Syst.*, 23, 5, (Sept. 2001), 657–683. DOI: 10.1145/504709.504712.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. "A very modal model of a modern, major, general type system." In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, 109–122. DOI: 10.1145/1190216.1190235.

Vincent Balat and Didier Galmiche. 2000. *Labelled Proof Systems for Intuitionistic Provability*. Springer, Dordrecht, 1–32. ISBN: 978-94-011-4040-9. DOI: 10.1007/978-94-011-4040-9_1.

Matteo Baldoni. 2000. *Normal Multimodal Logics with Interaction Axioms*. Springer, Dordrecht, 33–57. ISBN: 978-94-011-4040-9. DOI: 10.1007/978-94-011-4040-9_2.

Hendrik Pieter Barendregt. 1985. *The Lambda Calculus - its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Vol. 103. North-Holland. ISBN: 978-0-444-86748-3.

Josh Berdine and Peter W. O'Hearn. 2006. "Strong Update, Disposal, and Encapsulation in Bunched Typing." In: *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006*, 81–98. DOI: 10.1016/J.ENTCS.2006.04.006.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Dec. 2017. "Linear Haskell: practical linearity in a higher-order polymorphic language." *Proc. ACM Program. Lang.*, 2, POPL, Article 5, (Dec. 2017), 29 pages. DOI: 10.1145/3158093.

Cristiano Calcagno, Philippa Gardner, and Matthew Hague. 2005. "From Separation Logic to First-Order Logic." In: *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Proceedings*, 395–409. DOI: 10.1007/978-3-540-31982-5_25.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs." *Journal of Automated Reasoning*, 61, 1-4, 367–422. DOI: 10.1007/S10817-018-9457-5.

Matthew Collinson and David J. Pym. 2006. "Bunching for Regions and Locations." In: *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006*, 171–197. DOI: 10.1016/J.ENTCS.2006.04.010.

Matthew Collinson, David J. Pym, and Edmund Robinson. Dec. 2008. "Bunched polymorphism." *Mathematical Structures in Computer Science*, 18, 6, (Dec. 2008), 1091–1132. DOI: 10.1017/S0960129508007159.

M Fiore, A. M. Pitts, and G. Winskel. 2022. *Lecture Notes on Denotational Semantics 2022/23*. University of Cambridge, Department of Computer Science and Technology. Retrieved Oct. 7, 2024 from https://www.cl.cam.ac.uk/teaching/2223/DenotSem/DenotSemNotes.pdf.

Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. "RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly." *Proceedings of the ACM on Programming Languages*, 8, PLDI, 1656–1679. DOI: 10.1145/3656444.

Free Software Foundation. 2024. *Void pointers (GNU C language manual)*. Retrieved Oct. 16, 2024 from https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Void-Pointers.html.

Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. "A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency." *Proceedings of the ACM on Programming Languages*, 6, OOPSLA2, 841–869. DOI: 10.1145/3563318.

Didier Galmiche and Dominique Larchey-Wendling. 2006. "Expressivity Properties of Boolean BI Through Relational Models." In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 357–368. ISBN: 978-3-540-49995-4. DOI: 10.1007/11944836_33.

Didier Galmiche, Michel Marti, and Daniel Méry. 2019. "Relating Labelled and Label-Free Bunched Calculi in BI Logic." In: *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*, 130–146. DOI: 10.1007/978-3-030-29026-9_8.

Jad Elkhaleq Ghalayini and Neel Krishnaswami. 2023. "Explicit Refinement Types." *Proceedings of the ACM on Programming Languages*, 7, ICFP, 187–214. DOI: 10.1145/3607837.

Meghdad Ghari. 2017. "Labeled sequent calculus for justification logics." *Annals of Pure and Applied Logic*, 168, 1, 72–111. DOI: 10.1016/J.APAL.2016.08.006.

Jean-Yves Girard. 1995. *Linear Logic: its syntax and semantics*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1–42.

Carl A. Gunter. 1993. *Semantics of programming languages - structures and techniques*. Foundations of computing. MIT Press. ISBN: 978-0-262-07143-7.

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)* Cambridge University Press. ISBN: 9781107150300.

J. Roger Hindley and Jonathan P. Seldin. 2008. *Lambda-Calculus and Combinators: An Introduction*. (2nd ed.). Cambridge University Press. ISBN: 9780521898850.

Zhé Hóu, Rajeev Goré, and Alwen Tiu. June 2015. "A labelled sequent calculus for BBI: proof theory and proof search." *Journal of Logic and Computation*, 28, 4, (June 2015), 809–872. DOI: 10.1093/logcom/exv033.

Samin S. Ishtiaq and David J. Pym. 1999. "Kripke Resource Models of a Dependently-Typed, Bunched λ-Calculus." In: *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, 235–249. DOI: 10.1007/3-540-48168-0_17.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. "VeriFast: a powerful, sound, predictable, fast verifier for C and java." In: *Proceedings of the Third International Conference on NASA Formal Methods* (NFM'11). Springer-Verlag, Pasadena, CA, 41–55. ISBN: 9783642203978.

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, United Kingdom. ISBN: 9780521826143.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." *Journal of Functional Programming*, 28, e20. DOI: 10.1017/S0956796818000151.

Saul A. Kripke. 1963. "Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi." *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9, 67–96. DOI: 10.1002/malq.19630090502.

P. J. Landin. 1964. "The Mechanical Evaluation of Expressions." *The Computer Journal*, 6, 4, 308–320. DOI: 10.1093/COMJNL/6.4.308.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. "Into the depths of C: elaborating the de facto standards." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 1–15. DOI: 10.1145/2908080.2908081.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Aug. 2020. "Cosmo: a concurrent separation logic for multicore OCaml." 4, ICFP, Article 96, (Aug. 2020), 29 pages. DOI: 10.1145/3408978.

Robin Milner. 1978. "A theory of type polymorphism in programming." *Journal of Computer and System Sciences*, 17, 3, 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Rasmus Ejlers Møgelberg and Marco Paviotti. 2019. "Denotational semantics of recursive types in synthetic guarded domain theory." *Math. Struct. Comput. Sci.*, 29, 3, 465–510. DOI: 10.1017/S0960129518000087.

Eugenio Moggi. 1989. *An Abstract View of Programming Languages*. Publisher: Division of Informatics, The University of Edinburgh. Retrieved Oct. 1, 2024 from http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-113/.

Eugenio Moggi. 1991. "Notions of Computation and Monads." *Information and Computation*, 93, 1, 55–92. DOI: 10.1016/0890-5401(91)90052-4.

Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. 2005. "L^3: A Linear Language with Locations." In: *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings* (Lecture Notes in Computer Science). Vol. 3461. Springer, 293–307. DOI: 10.1007/11417170_22.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. "From system F to typed assembly language." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21, 3, 527–568. DOI: 10.1145/319301.319345.

Hiroshi Nakano. 2000. "A Modality for Recursion." In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, 255–266. DOI: 10.1109/LICS.2000.855774.

Peter W. O'Hearn. 1999. "Resource Interpretations, Bunched Implications and the *alpha lambda*-Calculus." In: *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, 258–279. DOI: 10.1007/3-540-48959-2_19.

Peter W. O'Hearn. 2007. "Resources, concurrency, and local reasoning." *Theoretical Computer Science*, 375, 1-3, 271–307. DOI: 10.1016/J.TCS.2006.12.035.

Peter W. O'Hearn and David J. Pym. 1999. "The Logic of Bunched Implications." *The Bulletin of Symbolic Logic*, 5, 2, 215–244. DOI: 10.2307/421090.

Simon Peyton Jones. June 2019. *Type inference as constraint solving: how GHC's type inference engine actually works*. Zurihac keynote talk. (June 2019). https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8.

François Pottier and Didier Rémy.. Dec. 2004. "The Essence of ML Type Inference." In: *Advanced Topics in Types and Programming Languages*. The MIT Press, (Dec. 2004). Chap. 10. ISBN: 9780262281591. DOI: 10.7551/mitpress/1104.003.0016.

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. "CN: Verifying Systems C Code with Separation-Logic Refinement Types." *Proceedings of the ACM on Programming Languages*, 7, POPL, 1–32. DOI: 10.1145/3571194.

David J. Pym. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Vol. 26. Springer. ISBN: 978-1-4020-0745-3.

David J. Pym, Jonathan M. Spring, and Peter O'Hearn. Sept. 2019. "Why Separation Logic Works." *Philosophy & Technology*, 32, 3, (Sept. 2019), 483–516. DOI: 10.1007/s13347-018-0312-8.

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. June 2023. "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs." 7, PLDI, Article 151, (June 2023), 25 pages. DOI: 10.1145/3591265.

replace in std::mem - Rust. Retrieved Oct. 5, 2024 from https://doc.rust-lang.org/std/mem/fn.replace.html.

John C. Reynolds. 2002. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, 55–74. DOI: 10.1109/LICS.2002.1029817.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. "RefinedC: automating the foundational verification of C code with refined ownership types." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, 158–174. DOI: 10.1145/3453483.3454036.

Thomas Somers and Robbert Krebbers. Oct. 2024. "Verified Lock-Free Session Channels with Linking." *Proceedings of the ACM on Programming Languages*, 8, OOPSLA2, Article 292, (Oct. 2024), 30 pages. DOI: 10.1145/3689732.

Luca Viganò. 2000. *Labelled non-classical logics*. Springer. ISBN: 978-0-7923-7749-8.

Philip Wadler. 1993. "A Taste of Linear Logic." In: *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings* (Lecture Notes in Computer Science). Vol. 711. Springer, 185–210. DOI: 10.1007/3-540-57182-5_12.

Philip Wadler. 1990. "Linear Types can Change the World!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. North-Holland, 561.

Andrew K. Wright and Matthias Felleisen. 1994. "A Syntactic Approach to Type Soundness." *Information and Computation*, 115, 1, 38–94. DOI: 10.1006/INCO.1994.1093.

Noam Zeilberger. 2016. *Principles of Type Refinement*. Retrieved Oct. 8, 2024 from http://noamz.org/oplss16/refinements-notes.pdf.