A type-theory for higher-order amortized analysis

A dissertation submitted towards the degree
Doctor of Engineering
of the
Faculty of Mathematics and Computer Science
of
Saarland University

by
Vineet Rajani

Saarbrücken
February, 2020

Date of Colloquium:	15/04/2020
Dean:	Univ.-Prof. Dr. Thomas Schuster
Reporters:	Prof. Dr. Deepak Garg
	Prof. Dr. Derek Dreyer
	Prof. Dr. Gilles Barthe
Chairman of the	
Examination board:	Prof. Dr. Jan Reineke
Scientific Assistant:	Dr. Andrew Hirsch

ABSTRACT

Verification of worst-case bounds (on the resource usage of programs) is an important problem in computer science. The usefulness of such verification depends on the precision of the underlying analysis. For precision, sometimes it is useful to consider the average cost over a *sequence of operations*, instead of separately considering the cost of each individual operation. This kind of an analysis is often referred to as *amortized resource analysis*. Typically, programs that optimize their internal state to reduce the cost of future executions benefit from such approaches. Analyzing resource usage of a standard functional (FIFO) queue implemented using two functional (LIFO) lists is a classic example of amortized analysis.

In this thesis we present λ^{amor}, a type-theory for amortized resource analysis of higher-order functional programs. A typical amortized analysis works by storing a ghost state called the *potential* with data structures. The key idea underlying amortized analysis is to show that, the available potential with the program is sufficient to account for the resource usage of that program. Verification in λ^{amor} is based on internalizing this idea into a type theory. We achieve this by providing a general type-theoretic construct to represent potential at the level of types and then building an affine type-theory around it. With λ^{amor} we show that, type-theoretic amortized analysis can be performed using well understood concepts from sub-structural and modal type theories. Yet, it yields an extremely expressive framework which can be used for resource analysis of higher-order programs, both in a strict and lazy setting. We show embeddings of two very different styles (one based on *effects* and the other on *coeffects*) of type-theoretic resource analysis frameworks into λ^{amor}. We show that λ^{amor} is sound (using a logical relations model) and complete for cost analysis of PCF programs (using one of the embeddings).

Next, we apply ideas from λ^{amor} to develop another type theory (called λ^{cg}) for a very different domain – Information Flow Control (IFC). λ^{cg} uses a similar type-theoretic construct (which λ^{amor} uses for the potential) to represent confidentiality label (the ghost state for IFC).

Finally, we abstract away from the specific ghost states (potential and confidentiality label) and describe how to develop a type-theory for a general ghost state with a monoidal structure.

ZUSAMMENFASSUNG

Die Verifikation von"Worst-Case" Schranken für Ressourcennutzung ist ein wichtiges Problem in der Informatik. Der Nutzen einer solchen Verifikation hängt von der Präzision der Analyse ab. Aus Gründen der Präzision ist es manchmal nützlich, die durchschnittlichen Kosten einer Folge von Operationen zu berücksichtigen, statt die Kosten jeder einzelnen Operation getrennt zu betrachten. Diese Art von Analyse wird oft als amortisierte Ressourcenanalyse bezeichnet. Typischerweise profitieren Programme, die ihren Zustand optimieren, um die Kosten zukünftiger Ausführungen zu reduzieren, von solchen Ansätzen. Die Analyse der Ressourcennutzung einer mit zwei (LIFO) Listen implementierten funktionalen (FIFO) Schlange ist ein klassisches Beispiel für eine amortisierte Analyse.

In dieser Arbeit präsentieren wir λ^{amor}, eine Typentheorie für die amortisierte Analyse der Ressourcennutzung höherstufiger Programme. Eine typische amortisierte Analyse speichert einen "ghost state", der als Potenzial bezeichnet wird, zusammen mit den Datenstrukturen. Die Kernidee der amortisierten Analyse ist es, zu zeigen, dass das dem Programm zur Verfügung stehende Potenzial ausreicht, um die Ressourcennutzung des Programms zu erfassen. Die Verifikation in λ^{amor} basiert auf der Realisierung dieser Idee in einer Typentheorie. Wir erreichen dies indem wir ein allgemeines typentheoretisches Konstrukt zur Darstellung des Potenzials auf der Ebene von Typen definieren und anschließend eine affine Typentheorie aufbauen. Mit λ^{amor} zeigen wir, dass eine typentheoretische amortisierte Analyse mit gut verstandenen Konzepten aus substrukturellen und modalen Typentheorien durchgeführt werden kann. Trotzdem ergibt sich ein äußerst aussagekräftiges Framework, das für die Ressourcenanalyse von höherstufigen Programmen, sowohl ein einem "strikten", als auch in einem "lazy" Setting, verwendet werden kann. Wir präsentieren Einbettungen zweier stark verschiedener Arten von typentheoretischen Ressourcenanalyseframeworks (eines basiert auf Effekten, das andere auf Koeffekten) in λ^{amor}. Wir zeigen, dass λ^{amor} korrekt (sound) ist (mithilfe eines "Logical relations" Modells) und, dass es vollständig für PCF-Programme ist (unter Verwendung einer der Einbettungen).

Als nächstes verwenden wir Ideen von λ^{amor}, um eine andere Typentheorie (genannt λ^{cg}) für einen ganz anderen Anwendungsfall - Informationsflusskontrolle (IFC) - zu entwickeln. λ^{cg} verwendet ähnliche typentheoretische Konstrukte wie

λ^{amor} für das Potenzial verwendet, um die Vertraulichkeitsmarkierungen (den "ghost state" für IFC) darzustellen.

Schließlich abstrahieren wir von den spezifischen "ghost states" (Potenzial und Vertraulichkeitsmarkierungen) und entwickeln eine Typentheorie für einen allgemeinen "ghost state" mit einer monoidalen Struktur.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Deepak Garg. He has been a constant source of inspiration and support throughout my Ph.D. He not only taught me the foundations required for doing meaningful research in the formal aspects of computer science, but also helped me in honing my skills with every project that we did. I always look up to him on matters related to research and otherwise. It has been a true privilege working with him all these years, cannot thank him enough.

Next, I would like to thank Derek and Gilles for being a part of my thesis committee and asking insightful questions during the review. Their questions not only helped improve this thesis but also provided interesting directions for future work. I also want to extend my sincere thanks to all my collaborators and teachers, they have all played a pivotal role in shaping my understanding.

Special thanks to all my friends and fellow colleagues in the Saarland Informatics Campus for making my stay at MPI-SWS absolutely wonderful. Thanks to Jan Menz for helping me with the German version of the abstract.

Finally, heartfelt gratitude for the endless love and support of my family and friends. None of this would have been possible without them.

CONTENTS

1 INTRODUCTION 1
- 1.1 Background and motivation 1
- 1.2 Limitations of prior work 3
- 1.3 Thesis statement 4
- 1.4 Overview 4
- 1.5 Contributions 6
- 1.6 Scope and limitations 7
- 1.7 Outline 7

I Type theory for amortized analysis 9

2 λ^{AMOR^-} 11
- 2.1 Syntax 11
- 2.2 Semantics 13
- 2.3 Type system 14

3 META-THEORY OF λ^{AMOR^-} 19

4 EXAMPLES 23
- 4.1 Map 23
- 4.2 Append 24
- 4.3 Church encoding 25
- 4.4 Eager functional queue 27
- 4.5 Okasaki's implicit queue 28

5 EMBEDDING UNIVARIATE RAML 31
- 5.1 Brief primer on Univariate RAML 31
- 5.2 Type-directed translation 33
- 5.3 Semantic properties of the embedding 36

6 FROM λ^{AMOR^-} TO λ^{AMOR} (FULL) 41
- 6.1 Changes to the type system: syntax and type rules 42
- 6.2 Semantic model of types 45

7 EMBEDDING dℓPCF 47
7.1 Brief primer on dℓPCF 47
7.2 Translating dℓPCF to λ^{amor} 49
7.3 Decompiling K_{PCF} triples to dℓPCF terms 53
7.4 Re-deriving dℓPCF's soundness 54

8 RELATED WORK FOR λ^{AMOR} 57

II Type theory for information flow control 61

9 APPLICATION TO INFORMATION FLOW CONTROL 63
9.1 Information flow control and granularity of tracking 63
9.2 Brief synopsis: type theory for coarse-grained IFC 64

10 λ^{CG}: TYPE THEORY FOR COARSE-GRAINED IFC 67
10.1 Type system 67
10.2 Semantic model of λ^{cg} 71
10.2.1 Unary interpretation 72
10.2.2 Binary interpretation 74
10.2.3 Meta-theoretic properties 76

11 λ^{FG}: TYPE THEORY FOR FINE-GRAINED IFC 79
11.1 Type system 79
11.2 Semantic model of λ^{fg} 82
11.2.1 Unary interpretation 82
11.2.2 Binary interpretation 84
11.2.3 Meta-theoretic properties 84

12 TRANSLATING λ^{FG} TO λ^{CG} 87
12.1 Type translation 87
12.2 Type-directed term translation 88
12.3 Properties of the translation 90

13 TRANSLATING λ^{CG} TO λ^{FG} 93
13.1 Type translation 93
13.2 Type-directed term translation 94
13.3 Properties of the translation 94

14 RELATED WORK FOR λ^{CG} 97

III Epilogue 101

15 ABSTRACTING THE GHOST STATE 103
15.1 Difference in the proof theories of λ^{amor} and λ^{cg} 103
15.2 Reconciling the differences 104

16 CONCLUSION AND FUTURE WORK 107
16.1 Concluding remarks 107
16.2 Some directions for future work 107
16.2.1 Future directions for λ^{amor} 108
16.2.2 Future directions for λ^{cg} 108

BIBLIOGRAPHY 109

1 INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

Verification of worst-case resource bounds is an important problem in computer science. However, for many data structures (both imperative and functional), the cost of an operation depends on the internal state at the time of the operation. In these cases, it is often more useful to establish an upper bound on a *sequence of* n *operations*, and then take the average cost over the n operations. This kind of an analysis is often called an *amortized resource analysis* [55]. The analysis can be understood from the classic example of eager functional queues. Eager functional queues are implemented using two stacks, say S_1 and S_2. Enqueue is implemented as a push on S_1 (which takes constant time). Dequeue is implemented as a pop from S_2 if it is non-empty but if S_2 is empty then it involves transferring contents from S_1 to S_2, thereby reversing the contents of S_1, and then popping S_2. Such a reversal changes the LIFO semantics of a stack into the FIFO semantics of a queue. Consequently, the final pop returns the very first element of the queue, thereby simulating a dequeue operation.

Such an eager functional queue can be easily implemented in a standard functional language with lists and pairs. Enqueue is encoded as a function which adds the new element to the front of the first list, l_1 (via a cons operation). This is shown in the Listing 1.1.

$$
\begin{aligned}
&enq : \tau \to (L\tau \otimes L\tau) \to (L\tau \otimes L\tau) \\
&enq \triangleq \lambda\ a\ \ q. \\
&\qquad\qquad \mathsf{let}\langle\langle l_1, l_2\rangle\rangle = q\ \mathsf{in}\ \ \langle\langle a :: l_1, l_2\rangle\rangle
\end{aligned}
$$

Listing 1.1: Encoding of enqueue

Dequeue on the other hand is a bit involved, it works by case analyzing the second list (denoted by l_2). If l_2 is nil then we transfer the contents of l_1 into l_2. This is represented by an abstract function $move$, whose trivial details we elide here, and then popping the resulting second list. Dequeue cannot be performed if both the lists are empty, represented by $\perp$. In the other case, when l_2 is non-empty, we just pop the top element from it. This encoding of dequeue is shown in Listing 1.2.

```
$dq : (L\tau \otimes L\tau) \rightarrow (L\tau \otimes L\tau)$
$dq \triangleq \lambda\ q.$
 $\mathsf{let}\langle\langle l_1, l_2\rangle\rangle = q\ \mathsf{in}$
 $\mathsf{match}\ \ l_2\ \ \mathsf{with}$
  $|nil \mapsto \mathsf{let}\ \ l_r = move\ \ l_1\ \ l_2\ \ \mathsf{in}$
   $\mathsf{match}\ \ l_r\ \ \mathsf{with}$
     $|nil \mapsto \perp$
     $|h_r :: l_r' \mapsto \langle\langle nil, l_r'\rangle\rangle$
  $|h_2 :: l_2' \mapsto \langle\langle l_1, l_2'\rangle\rangle$
```

Listing 1.2: Encoding of dequeue

Let us now consider a cost model where we only count a unit cost for every push and pop on the list. Under such a cost model, we can see that enqueue is a constant time operation as it involves just a single push on the first list. Dequeue, on the other hand, is a linear time operation with a precise cost of $2 * n + 1$ units where n is the length of the first list. The cost of $2 * n$ units come from a pop and push involved in the reversal of the stack (as part of the $move$ function), and the cost of 1 unit comes from the final pop from the second stack.

The problem that we want to analyze is the following: starting from an empty queue what is the worst case bound for a sequence of m enqueues/dequeues? This does not sound too hard, as we just saw that dequeue's worst-case bound is *linear*. Therefore a worst-case bound for this problem is $O(mn)$ which is $O(m^2)$ (assuming m is greater than n). This quadratic bound is correct but extremely imprecise as using amortized analysis we can obtain a much tighter bound for this problem. The key idea is to make use of the fact that we can never perform more dequeues than enqueues as we are starting from an empty queue. As a result, we can try to account for the cost of dequeue at the time of enqueue itself. It does not matter whether the actual dequeue happens or not. This is sound because we are only interested in the worst case bounds. In particular, by adding the cost of the $move$ function (two units per element) we increase the cost of enqueue to three units (which is still a constant) and reduce the cost of dequeue to one unit (which is also a constant now). This makes the total cost of this problem linear in the number of operations (m) which is way precise than the quadratic bound that we came up with earlier.

This is the general intuition of how amortized analysis works. This intuition with slight variations has been used by approaches like the method of potential [18, 55], the method of credits [18, 55] and the method of debits [49]. Common to these approaches is the notion of a *ghost state* (which we refer to as *potential* in this thesis) attached to data structures. The basic idea underlying these approaches is to show that the available potential is sufficient to account for the cost of the involved operations.

Let us go back to our example of the eager functional queue to see this in action. *Enqueue* and *dequeue* now require a potential of (at least) three units and one unit, respectively, to account for their respective amortized costs. *Enqueue* uses one of the three units to account for the cost of the push and stores the remaining two with the newly pushed element on the stack (to be used later for dequeue). *Dequeue*, which involves moving elements from one stack to the other, uses the potential (of two units per element) to cover the cost of the *move*. The cost of doing an actual pop is covered by the potential of one unit required by dequeue.

Our goal in this thesis is to internalize these reasoning principles into a type theory and to build a general framework for static verification of amortized bounds using such potentials.

1.2 LIMITATIONS OF PRIOR WORK

Developing a type theory for verification of amortized bounds is not a new research problem. It has been studied in prior work [20, 27–30, 32] but prior approaches suffer from two significant technical limitations. The first limitation pertains to the lack of a *general type-theoretic construct* for associating potentials to arbitrary types. In prior work, this association is limited to specific types (e.g. integers, and lists and trees over first-order data) only. This is not only dissatisfactory from a foundational perspective, but it also limits expressivity.

The second limitation is the improper or complete lack of *linearity* in the type system, which limits expressiveness. One fundamental requirement (for soundness) is that stored potential must not be duplicated. A natural way of doing this is to make the type system linear or, more precisely, affine. Some existing type systems for amortized resource analysis use affineness, but only in very limited contexts. For example, AARA [28] treats first-order arguments (with potentials) as affine, but not returned functions. As a result, it forces that *all* arguments of a Curried function be applied atomically to prevent duplication of potential captured in a partially applied function. In other cases, prior work targets call-by-need semantics where affineness is not needed since a closure is never evaluated twice, even if it is duplicated and forced twice. For example, a formalization of Okasaki's method of debits [49], [20] uses this approach. However, such an approach does not work in a call-by-name or call-by-value setting where non-affine potentials are unsound.

Both these limitations highlight a significant gap in the space of type-theoretic development for amortized analysis. It is unclear at this moment if linearity/affineness is the right tool for this job, let alone provide a fully general way of type-theoretic amortized analysis.

1.3 THESIS STATEMENT

To overcome these limitations in Part I this thesis we present λ^{amor}, the first fully general affine type theory for verification of amortized bounds. Verification in λ^{amor} is based on four key technical pillars: a new modal type for representing potential, use of affine types for preventing duplication of potential, light-weight type refinements for expressivity and use of monads to localize cost. All of these except the new modal type are well-understood concepts from modal and sub-structural type systems. The key statement/hypothesis on which this thesis is built is the sufficiency of these four constructs for a very general type theory to verify amortized bounds.

1.4 OVERVIEW

Cost in λ^{amor} is tracked as an effect captured in monads, something which is well understood from prior work like [20]. Cost bearing computations are described using monads, where the cost of the computation is specified as a grade on the monadic type. $\mathbb{M}\,\kappa\,\tau$, for instance, is the type of a computation which when forced produces a value of type τ and incurs a cost κ in doing so (κ is a refinement of type real).

In addition to cost, λ^{amor} also captures potential, which is used to pay for the cost of computations. This is captured using a novel modal type constructor, $[p]\,\tau$. The p in the $[p]\,\tau$ describes the potential associated with an inhabitant of type τ. The potential p is actually a *ghost resource*. It has no term-level manifestation and is merely a proof artifact required for the meta-theory. This means an inhabitant of type $[p]\,\tau$ is just an inhabitant of the underlying type τ. However, this ability to capture potential with an individual type is extremely advantageous, and is really at the core of making this framework compositional and scalable to the higher-order setting. For instance, these potential-carrying modal types make it possible to capture the remaining potential from a partial function application on the type itself which can be passed around to other program parts, an ability that no prior work possesses (we explain this with an example of list append in Chapter 4).

Potential is a limited resource and, hence, must be tracked correctly. For instance, an unrestricted use of a value of type $[p]\,\tau$ would give us an unbounded amount of potential which can be used to type check any program in the system irrespective of the actual cost. This is prevented using affine types. However, affine types without exponentials (!) are too restrictive, but if added their use must only be limited to types that do not capture potentials (otherwise we would end up with the same problem of getting duplicate potential). To handle this, in λ^{amor} we make use of the dependent sub-exponential ($!_{a<I}\tau$) from Bounded Linear Logic [24] and its generalization in dℓPCF [39]. $!_{a<I}\tau$ represents I copies of a term of type τ with a free in it, and the free a inside τ can range from 0 to $I-1$. Morally $!_{a<I}\tau$ is equivalent to $\tau[0/a] \otimes \ldots \otimes \tau[(n-1)/a]$.

Finally, we use light-weight refinements to capture dependencies between input sizes and costs.

The four pillars described above makes λ^{amor} quite expressive. We can give *precise* types to fairly intricate encodings like Church numerals. We also embed a core of Univariate RAML [29] (an effect-based cost analysis framework which is also based on the method of potentials) and dℓPCF [39] (a very different style of cost analysis which is based on coeffects) in a way that internalizes the cost into the types. The embedding of RAML and dℓPCF shows that two very different styles of cost tracking can both be described in λ^{amor}. But, additionally, we also use the embedding of dℓPCF to get a very strong relative completeness[1] result which dℓPCF could achieve (non-compositionally) for whole programs only. So, λ^{amor} can be seen as a *compositional* extension of dℓPCF too. The compositionality works because λ^{amor} can record costs in the types while in dℓPCF cost is recorded only in the typing derivation. For both the embeddings, we show that they preserve types, cost and semantics of the source programs. This is done by developing cross-language models for each of the embeddings.

We show that this type theory can be interpreted in Kripke logical-relations where the Kripke worlds represent resources (à la semantics of BI [51]). The key new insight is how we treat the type construct for the potentials ($[p]\,\tau$ type that we mentioned above). The model makes the ghost nature of the potentials explicit by showing that they only affect the worlds and not the values. We use the model to prove the soundness of the type system and also to derive additional properties for the RAML and dℓPCF's embeddings.

1 Completeness is relative to an oracle which can determine the truth of simple index inequalities defined over finite sums.

Next, generalizing beyond amortized cost analysis, we show how the potential construct ($[p]\,\tau$) and the monad of λ^{amor} can be adapted for a completely different analysis, namely, Information Flow Control (IFC). The idea is to replace potentials with *confidentiality labels* (confidentiality annotations) that now act as the ghost state. Confidentiality labels (unlike potentials) are *relational* ghost resources i.e. they represent ghost information across two different executions of a program. Additionally, affineness has no use in information flow control. Despite such glaring differences in the nature of the ghost state, we could use familiar ideas from λ^{amor} to develop the type system for information flow control. In particular, the rules for manipulating ghost resources are quite similar. We call this type system λ^{cg}. Besides demonstrating the generality of our type theory's constructs, we make additional contributions with λ^{cg}: 1) To prove λ^{cg} sound, we develop the first semantic model for IFC type systems with full-higher order state, something which had not been done prior to our work, and 2) We show that λ^{cg} is very expressive by embedding a standard IFC type system (an idealization of FlowCaml [50]) into it. We also develop an embedding in the other direction, thus establishing equi-expressiveness. We prove that these translations are type- and semantics-preserving. We develop cross-language models to prove some of these results.

Finally, we tie the two type theories (λ^{amor} and λ^{cg}) together by showing that the two ghost states (potential and confidentiality label) are instances of a more general ghost state with a monoidal structure. We describe how to obtain a type theory for such a monoidal ghost state.

1.5 CONTRIBUTIONS

We summarize the key technical contributions of the thesis:

1. We present λ^{amor}, a compositional type theory for amortized cost analysis of higher-order functional programs. λ^{amor} is built from well understood concepts from sub-structural and modal type theories. Yet, λ^{amor} is sufficient to perform both effect- and coeffect-based cost analysis. We give a set-theoretic interpretation to the types of λ^{amor} using Kripke logical-relations and use the interpretation to prove the type-theory sound.

2. We give an embedding of Univariate RAML [29] and dℓPCF [39] in λ^{amor}. We prove that these embeddings are not only type preserving but also semantics and cost preserving. We show this by deriving alternate proofs of RAML's and dℓPCF's soundness in λ^{amor}. Both these proofs are based on cross-language

logical relations, while the original proofs (for both RAML and dℓPCF) are syntactic.

3. Our embedding of dℓPCF shows that λ^{amor} is *relative-complete* for cost analysis of PCF programs. Moreover, our analysis is compositional, unlike dℓPCF's.

4. We show how the basic design principles of λ^{amor} can be adapted for a completely different purpose, namely, information flow analysis. We develop a type theory for this (λ^{cg}), which makes additional contributions.

5. Finally, we abstract away the structural differences between the two type theories by showing that both λ^{amor} and λ^{cg} are instances of a more general type theory for an abstract ghost state with a monoidal structure.

1.6 SCOPE AND LIMITATIONS

The focus of this thesis is to develop the theoretical foundations for type-based analysis of amortized costs and information flow control. Implementation of these type theories, while an interesting goal, is out of the scope of this thesis. Nonetheless, we expect that in a restricted setting like polynomial cost, one could use ideas from prior work, like AARA [28] and RAML [27, 29], to implement λ^{amor} efficiently. Similarly implementation of the type theory for IFC can be done following ideas from an existing IFC type system like SLIO [13].

1.7 OUTLINE

We organize the rest of this thesis into four parts.

In Part I we describe the type theory for amortized analysis. We begin with a subset of λ^{amor} without the sub-exponential (called λ^{amor^-}) in Chapter 2. We describe the meta-theory of λ^{amor^-} in Chapter 3. Even without the sub-exponential, λ^{amor^-} turns out to be quite expressive. We demonstrate this via encodings of a variety of examples from different domains in Chapter 4. λ^{amor^-} can also encode the whole of Univariate RAML [29, 32]. We describe this encoding in Chapter 5. Then we add the dependent sub-exponential to λ^{amor^-} and describe the development of λ^{amor} (full) in Chapter 6. λ^{amor} is extremely expressive; we obtain a very strong relative completeness result by embedding dℓPCF in Chapter 7. We conclude the amortized analysis part with a description of related work in Chapter 8.

In Part II we develop a similar type theory for the domain of Information Flow Control (IFC). We begin with a high level description of the generality of the type-theoretic constructs of λ^{amor} and how we apply them to the domain of IFC in Chapter 9. We describe λ^{cg}, a type theory for coarse-grained IFC in Chapter 10. The obtained type theory, λ^{cg}, is very expressive, which we show by translating an existing fine-grained IFC type system into λ^{cg}. To do this, we first describe the λ^{fg} type system, a variant of an existing fine-grained IFC type system, in Chapter 11. Then we show that λ^{fg} can be embedded in λ^{cg} in Chapter 12. After this, we show that even the reverse encoding of λ^{cg} into λ^{fg} is also possible, thereby establishing equi-expressiveness of the two IFC type systems. The reverse translation from λ^{cg} to λ^{fg} is described in Chapter 13. Finally, we describe related work for the IFC part in Chapter 14.

In part III we describe an abstract monoidal structure that is common to the ghost states of λ^{amor} and λ^{cg} in Chapter 15. We describe the differences in the treatment of the ghost states in λ^{amor} and λ^{cg} and explain how to reconcile them. We conclude the thesis in Chapter 16 with some directions for future work.

Additional details of everything presented in this thesis is available in the technical report [52].

Part I

Type theory for amortized analysis

2

λ^{amor^-}

In this chapter we describe the λ^{amor^-} system (a version of λ^{amor} without the sub-exponential). We begin by describing the syntax of λ^{amor^-}. Then we look at the evaluation and typing rules.

2.1 SYNTAX

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \mathbf{1} \mid b \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \tau_1 \oplus \tau_2 \mid\, !\tau \mid [p]\,\tau \mid \mathbb{M}\,\kappa\,\tau \mid L^n\,\tau \\
 & & & \alpha \mid \forall \alpha : K.\tau \mid \forall i : S.\tau \mid \lambda_t i : S.\tau \mid \tau\ I \mid \exists i : S.\tau \mid C \Rightarrow \tau \mid C \& \tau \\
\text{Expressions} & e & ::= & v \mid x \mid e_1\ e_2 \mid \langle\langle e_1, e_2 \rangle\rangle \mid \mathsf{let} \langle\langle x, y \rangle\rangle = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\ x.e \mid \\
 & & & \langle e, e \rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}\ e, x.e, y.e \mid \\
 & & & \mathsf{let}\,!\,x = e_1\ \mathsf{in}\ e_2 \mid e :: e \mid \mathsf{match}\ e\ \mathsf{with}\ |\mathit{nil} \mapsto e_1\ |h :: t \mapsto e_2 \mid e\ [] \mid \\
 & & & \mathsf{xlet}\,x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{clet}\,x = e_1\ \mathsf{in}\ e_2 \\
\text{Values} & v & ::= & () \mid c \mid \lambda x.e \mid \langle\langle v_1, v_2 \rangle\rangle \mid \langle v, v \rangle \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid\, !\,e \mid \mathit{nil} \mid \\
 & & & \Lambda.e \mid \mathsf{ret}\,e \mid \mathsf{bind}\,x = e_1\ \mathsf{in}\ e_2 \mid \uparrow^\kappa \mid \mathsf{release}\,x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{store}\,e \\
\text{Index} & I, \kappa, p, n & ::= & i \mid N \mid R^+ \mid I + I \mid I - I \mid \lambda_s i : S.I \mid I\ I \\
\text{Constraints} & C & ::= & I = I \mid I < I \mid C \wedge C \\
\text{Sort} & S & ::= & \mathbb{N} \mid \mathbb{R}^+ \mid S \to S \\
\text{Kind} & K & ::= & \mathsf{Type} \mid S \to K
\end{array}
$$

Figure 2.1: λ^{amor^-}'s syntax

λ^{amor^-} treats resource consumption as an an *effect*. As a result, all operations that consume resources (i.e. potential) in some form are classified as *impure* and the rest as *pure*. The syntax of the language is shown in Fig. 2.1. We describe the various syntactic categories of the calculus below.

Indices, sorts, kinds and constraints. λ^{amor^-} is a refinement type system. (Static) indices, à la DML [57], are used to track information like the length of a list and the cost of a computation. The length of list comes from the sort $\mathbb{N}$ of natural numbers. The potential and the cost both come from the sort $\mathbb{R}^+$ of non-negative real numbers. Besides this, the grammar for indices also consists of index variables, index-level functions and their applications (index level functions and their application is required for our Church encoding, described in Section 4.3). λ^{amor^-} also features kinds, denoted by K. Type represents the kind of the standard affine types of λ^{amor^-} and $S \to K$ represents the kind of sort-indexed type families. Finally, constraints (denoted by C) are predicates $(=, <, \wedge)$ over indices.

Types. λ^{amor^-} uses an affine type system. In the pure fragment, the most important type is the modal type denoted by $[p]\,\tau$. $[p]\,\tau$ can be thought of as the type of a value with potential p and type τ. We have the unit type (denoted by $\mathbf{1}$) and an abstract base type (denoted by b) to represent types like integers or booleans. We have the standard types from affine λ-calculus, which include types for functions ($\multimap$), sums ($\oplus$), pairs (both $\otimes$ and $\&$) and the exponential (!), which can be assigned to expressions that can be duplicated. We also have the size-refined list type ($L^n\tau$), where the size n of the list is drawn from the language of indices (described earlier). We have universal quantification over types and indices denoted by $\forall\alpha : K.\tau$ and $\forall i : S.\tau$ respectively, and similarly we also have existential quantification over indices denoted by $\exists i : S.\tau$. The constraint type (denoted by $C \Rightarrow \tau$) specifies that if constraint C holds then the underlying term has the type τ. The other constraint type, denoted by $C\&\tau$, specifies that the constraint C holds and the type of the underlying term is τ. For instance, it can be used to specify the type of the non-empty list as $(n > 0)\&(L^n\tau)$. Lastly, we have sort-indexed type families, which are type-level functions from sorts to kinds. In the impure fragment, the only type we have is the type of a graded monad, denoted by $\mathbb{M}\,\kappa\,\tau$. Intuitively, $\mathbb{M}\,\kappa\,\tau$ is the type of a computation that has a cost of κ units (or needs a potential of κ units) and produces a value of type τ. Technically, $\mathbb{M}\,\kappa\,\tau$ is a graded monad [23].

Expressions and values. There are term-level constructors for all types (in the universe Type) except for the modal type ($[p]\,\tau$). The inhabitants of type $[p]\,\tau$ are exactly those of type τ. The potential is really a ghost at the level of terms. We describe the expression and value forms for some of the types here. The term-level constructors for the constraint type ($C \Rightarrow \tau$), type and index level quantification ($\forall\alpha : K.\tau$, $\forall i : S.\tau$) are all denoted by $\Lambda.e$. The constraints, type and index variables show up only at the level of types. There is also a fixed point operator (fix) which is used to encode recursion. The impure fragment has several terms, including a return ($\mathsf{ret}\; e$) and bind ($\mathsf{bind}\; x = e_1 \;\mathsf{in}\; e_2$) for the graded monad. There is a construct for storing potential with a term, namely, $\mathsf{store}\, e$. Dually, we have a construct which can be used to release the stored potential from a term, $\mathsf{release}\, x = e_1 \;\mathsf{in}\; e_2$. Note that, $\mathsf{store}\, e$ and $\mathsf{release}\, x = e_1 \;\mathsf{in}\; e_2$ are meaningful only for the type system: they indicate when potentials need to be stored and released, respectively. Operationally, they are uninteresting: $\mathsf{store}\, e$ evaluates exactly like $\mathsf{ret}\, e$, while $\mathsf{release}\, x = e_1 \;\mathsf{in}\; e_2$ evaluates exactly like $\mathsf{bind}\, x = e_1 \;\mathsf{in}\; e_2$. This is completely consistent with the ghost nature of the potential. Finally, we have a construct for consuming resources: The tick denoted by $\uparrow^\kappa$, it indicates the consumption of κ resources. Programmers place it model different kind of costs as in prior work [20].

2.2 SEMANTICS

$$\boxed{\text{Forcing reduction relation: } e \Downarrow^{\kappa}_{t} v}$$

$$\frac{e \Downarrow_t v}{\mathsf{ret}\, e \Downarrow^{0}_{t+1} v}\ \text{E-return} \qquad \frac{e_1 \Downarrow_{t_1} v_1 \qquad v_1 \Downarrow^{\kappa_1}_{t_2} v_1' \qquad e_2[v_1'/x] \Downarrow_{t_3} v_2 \qquad v_2 \Downarrow^{\kappa_2}_{t_4} v_2'}{\mathsf{bind}\, x = e_1 \;\mathsf{in}\; e_2 \Downarrow^{\kappa_1+\kappa_2}_{t_1+t_2+t_3+t_4+1} v_2'}\ \text{E-bind}$$

$$\frac{}{\uparrow^{\kappa} \Downarrow^{\kappa}_{1} ()}\ \text{E-tick} \qquad \frac{e_1 \Downarrow_{t_1} v_1 \qquad e_2[v_1/x] \Downarrow_{t_2} v_2 \qquad v_2 \Downarrow^{\kappa}_{t_3} v_2'}{\mathsf{release}\, x = e_1 \;\mathsf{in}\; e_2 \Downarrow^{\kappa}_{t_1+t_2+t_3+1} v_2'}\ \text{E-release}$$

$$\frac{e \Downarrow_t v}{\mathsf{store}\, e \Downarrow^{0}_{t+1} v}\ \text{E-store}$$

Figure 2.2: Evalaution rules for impure fragment

λ^{amor^-} is a call-by-name calculus with eager[1] evaluation. The pure evaluation judgment ($e \Downarrow v$) relates a λ^{amor^-} expression to the value the expression evaluates to. All monadic forms are treated as values in the pure evaluation. The rules for the pure fragment are standard and hence omitted here but we describe them in the technical report [52]. The forcing evaluation judgment ($e \Downarrow^{\kappa} v$, where κ indicates the amount of resources consumed) is a relation between terms of type $\mathbb{M}\, \kappa\, \tau$ and values of type τ. Big-step evaluation rules for the impure fragment of λ^{amor^-} are given in Fig. 2.2. The E-return rule states that if e reduces with the pure reduction to v then so does $\mathsf{ret}\, e$ with 0 cost. At the level of evaluation rules, E-store behaves exactly like E-return emphasizing the ghost nature of the potential. E-bind is the standard

1 & pairs are evaluated eagerly but since all cost effects are performed in a monad so it does not matter. ! is lazy as in a standard affine λ-calculus.

monadic composition of e_1 with e_2, except for the cost annotations – the cost of the bind is the sum of the costs of forcing e_1 and e_2. E-release works in a similar way. $\uparrow^\kappa$ is the only cost-consuming construct in the language. The E-tick rule states that $\uparrow^\kappa$ reduces to $()$ and it consumes κ resources.

2.3 TYPE SYSTEM

The typing judgment of λ^{amor^-} is written $\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau$. Here, Ψ is a context mapping type-level variables to their kinds, Θ is a context mapping index-level variables to their sorts, Δ is a context of constraints on the index variables, Ω and Γ are the non-linear and linear typing contexts respectively, both mapping term-level variables to their types. We use the notation $\Gamma_1 + \Gamma_2$ to describe disjoint union of the linear contexts Γ_1 and Γ_2. Selected typing rules are described in Fig. 2.3, and the full set of rules can be found in the technical report [52].

T-tensorI describes the type rule for the introduction form for the tensor pair $\langle\langle e_1, e_2\rangle\rangle$ - if e_1 and e_2 are typed τ_1 and τ_2 under linear contexts Γ_1 and Γ_2, respectively, then $\langle\langle e_1, e_2\rangle\rangle$ is typed $(\tau_1 \otimes \tau_2)$ under the context $(\Gamma_1 + \Gamma_2)$. Dually, T-tensorE describes the typing for the elimination form of the tensor pair – if expression e is of type $(\tau_1 \otimes \tau_2)$ in the context Γ_1 and a continuation e' is of type τ' in the context Γ_2 along with both elements of tensor pair available via variables x and y, then the expression $\mathsf{let}\langle\langle x, y\rangle\rangle = e \mathsf{\ in\ } e'$ is of type τ' under the context $(\Gamma_1 + \Gamma_2)$. T-expI type checks $!e$ with type $!\tau$ if e can be type-checked with type τ, under an empty linear context (unbounded terms can not depend on finite resources). We can of course use weakening (T-weaken) to type the exponential under a non-empty linear context if required. The subtyping relation (<:) is described below, but we skip describing the standard details of the $\sqsubseteq$ relation which can be found in the technical report [52]. T-expE is the rule for the elimination form of $!\tau$ – the important thing to note here is that the continuation e' has unbounded access to e via the non-linear variable x.

T-ret is the type rule for the return of the monad – $\mathsf{ret}\, e$ basically takes a well-typed expression and returns it unmodified, and hence has a cost of 0 units, represented by the type $\mathbb{M}\, 0\, \tau$. Dually, T-bind describes the typing rule for the monadic bind, which is basically a sequencing construct. Hence, the cost in the conclusion is the sum of the costs in the premises. T-tick type checks $\uparrow^\kappa$ at a monad of unit type that has a cost of κ units. T-store is the typing rule for the store construct, which is used to associate potential with a type. If p units of potential are attached to a type τ then the cost of doing so is p units. Finally, we have T-release as the dual rule for T-store. It takes the stored potential p_1 on the first expression and makes it available to the continuation (notice that the conclusion is typed with p_2 only while the continuation is typed with $p_1 + p_2$).

Subtyping. Selected subtyping rules are described in Fig. 2.4. As mentioned earlier λ^{amor^-} also has type-level functions and applications. We have added subtyping rules to convert from the application form $((\lambda_t i : S.\tau)\ I)$ to the substitution form $(\tau[I/i])$ and vice versa. Rule sub-potArrow helps in distributing the potential on the function to the potential over the argument and the return value. sub-potZero helps cast a value of type τ to a value of type $[0]\,\tau$. This reinforces the ghost nature of the potential at the level of terms. The subtyping of the modal type $[p]\,\tau$ is covariant in the types but contra-variant in the potential because it is sound to throw away potential (if a term has p units of potential then it also has less than p units of potential). The subtyping for the monadic type is covariant in both the type and the cost (because it is always safe to over-estimate the cost of a term). There are additional typing rules for sorts and kinds which are fairly standard so we omit them here, but describe them in the technical report [52].

Theorem 1 formulates the soundness criteria for λ^{amor^-}. Intuitively, it says that, if e is a closed term which has a statically approximated cost of κ units (as specified in the monadic type $\mathbb{M}\,\kappa\,\tau$) and forcing it actually consumes κ' units of resources, then $\kappa' \leqslant \kappa$. We prove this theorem using a semantic argument described in Chapter 3.

Theorem 1 (Soundness)**.** $\forall e, \kappa, \kappa', \tau \in \mathsf{Type}.$
$\vdash e : \mathbb{M}\,\kappa\,\tau \wedge e \Downarrow^{\kappa'}_{-} - \implies \kappa' \leqslant \kappa$

Theorem 1 is the typical way of stating soundness of a type-based cost analysis *without* potentials. λ^{amor^-} also has potentials, so we can state the soundness in an alternate way as described in Theorem 2. Here, instead of representing the requirement as a cost on the monad, we represent it as a potential in the negative position. The Theorem 2 shows that the runtime cost of forcing the term after a unit application is upper-bounded by the input potential.

Theorem 2 (Soundness)**.** $\forall e, \kappa, \kappa', \tau \in \mathsf{Type}.$
$\vdash e : [\kappa]\,\mathbf{1} \multimap \mathbb{M}\,0\,\tau \wedge e() \Downarrow \_ \Downarrow^{\kappa'} \_ \implies \kappa' \leqslant \kappa$

We give a semantic proof of both these theorems using a technique of step-indexed Kripke logical relation, which we describe in the next chapter.

$\boxed{\text{Typing judgment: } \Psi;\Theta;\Delta;\Omega;\Gamma \vdash e : \tau}$

$$\frac{\Psi;\Theta;\Delta;\Omega;\Gamma_1 \vdash e_1 : \tau_1 \qquad \Psi;\Theta;\Delta;\Omega;\Gamma_2 \vdash e_2 : \tau_1}{\Psi;\Theta;\Delta;\Omega;\Gamma_1 + \Gamma_2 \vdash \langle\langle e_1, e_2 \rangle\rangle : (\tau_1 \otimes \tau_2)} \text{ T-tensorI}$$

$$\frac{\Psi;\Theta;\Delta;\Omega;\Gamma_1 \vdash e : (\tau_1 \otimes \tau_2) \qquad \Psi;\Theta;\Delta;\Omega;\Gamma_2, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Psi;\Theta;\Delta;\Omega;\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\langle\langle x, y \rangle\rangle = e \;\mathsf{in}\; e' : \tau} \text{ T-tensorE}$$

$$\frac{\Psi;\Theta;\Delta;\Omega;. \vdash e : \tau}{\Psi;\Theta;\Delta;\Omega;. \vdash !e : !\tau} \text{ T-ExpI} \qquad \frac{\Psi;\Theta;\Delta;\Omega;\Gamma_1 \vdash e : !\tau \qquad \Psi;\Theta;\Delta;\Omega, x : \tau;\Gamma_2 \vdash e' : \tau'}{\Psi;\Theta;\Delta;\Omega;\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,!\,x = e \;\mathsf{in}\; e' : \tau'} \text{ T-ExpE}$$

$$\frac{\Psi;\Theta;\Delta;\Omega, x : \tau;. \vdash e : \tau}{\Psi;\Theta;\Delta;\Omega;. \vdash \mathsf{fix}\; x.e : \tau} \text{ T-fix}$$

$$\frac{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash e : \tau \qquad \Psi;\Theta;\Delta \vdash \Gamma' \sqsubseteq \Gamma \qquad \Psi;\Theta;\Delta \vdash \Omega' \sqsubseteq \Omega \qquad \Psi;\Theta;\Delta \vdash \tau <: \tau'}{\Psi;\Theta;\Delta;\Omega';\Gamma' \vdash e : \tau'} \text{ T-weaken}$$

$$\frac{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash e : \tau}{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash \mathsf{ret}\; e : \mathbb{M}\, 0\, \tau} \text{ T-ret}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Omega;\Gamma_1 \vdash e_1 : \mathbb{M}\, \kappa_1\, \tau_1 \\ \Psi;\Theta;\Delta;\Omega;\Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}\, \kappa_2\, \tau_2 \qquad \Theta;\Delta \vdash \kappa_1 : \mathbb{R}^+ \qquad \Theta;\Delta \vdash \kappa_2 : \mathbb{R}^+\end{array}}{\Psi;\Theta;\Delta;\Omega;\Gamma_1 + \Gamma_2 \vdash \mathsf{bind}\; x = e_1 \;\mathsf{in}\; e_2 : \mathbb{M}(\kappa_1 + \kappa_2)\, \tau_2} \text{ T-bind}$$

$$\frac{\Theta;\Delta \vdash \kappa : \mathbb{R}^+}{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash \uparrow^{\kappa} : \mathbb{M}\, \kappa\, \mathbf{1}} \text{ T-tick} \qquad \frac{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash e : \tau \qquad \Theta;\Delta \vdash p : \mathbb{R}^+}{\Psi;\Theta;\Delta;\Omega;\Gamma \vdash \mathsf{store}\; e : \mathbb{M}\, p\, ([p]\, \tau)} \text{ T-store}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Omega;\Gamma_1 \vdash e_1 : [p_1]\, \tau_1 \\ \Psi;\Theta;\Delta;\Omega;\Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}(p_1 + p_2)\, \tau_2 \qquad \Theta;\Delta \vdash p_1 : \mathbb{R}^+ \qquad \Theta;\Delta \vdash p_2 : \mathbb{R}^+\end{array}}{\Psi;\Theta;\Delta;\Omega;\Gamma_1 + \Gamma_2 \vdash \mathsf{release}\; x = e_1 \;\mathsf{in}\; e_2 : \mathbb{M}\, p_2\, \tau_2} \text{ T-release}$$

Figure 2.3: Selected typing rules for λ^{amor^-}

$$\frac{\Psi;\Theta;\Delta \vdash \tau <: \tau' \qquad \Theta;\Delta \models p' \leqslant p}{\Psi;\Theta;\Delta \vdash [p]\,\tau <: [p']\,\tau'}\ \text{sub-potential}$$

$$\frac{\Psi;\Theta;\Delta \vdash \tau <: \tau' \qquad \Theta;\Delta \models \kappa \leqslant \kappa'}{\Psi;\Theta;\Delta \vdash \mathbb{M}\,\kappa\,\tau <: \mathbb{M}\,\kappa'\,\tau'}\ \text{sub-monad}$$

$$\frac{\Theta;\Delta \vdash p : \mathbb{R}^+ \qquad \Theta;\Delta \vdash p' : \mathbb{R}^+}{\Psi;\Theta;\Delta \vdash [p](\tau_1 \multimap \tau_2) <: ([p']\,\tau_1 \multimap [p'+p]\,\tau_2)}\ \text{sub-potArrow}$$

$$\frac{}{\Psi;\Theta;\Delta \vdash \tau <: [0]\,\tau}\ \text{sub-potZero} \qquad \frac{\Psi;\Theta, i : S;\Delta \vdash \tau <: \tau'}{\Psi;\Theta;\Delta \vdash \lambda_t i : S.\tau <: \lambda_t i : S.\tau'}\ \text{sub-familyAbs}$$

$$\frac{\Theta;\Delta \vdash I : S}{\Psi;\Theta;\Delta \vdash (\lambda_t i : S.\tau)\ I <: \tau[I/i]}\ \text{sub-familyApp1}$$

$$\frac{\Theta;\Delta \vdash I : S}{\Psi;\Theta;\Delta \vdash \tau[I/i] <: (\lambda_t i : S.\tau)\ I}\ \text{sub-familyApp2}$$

Figure 2.4: Selected subtyping rules

3 META-THEORY OF λ^{amor^-}

In this chapter we describe a model for λ^{amor^-}'s types along with the key meta-theoretic properties. This model not only gives a semantic interpretation to the types of λ^{amor^-}, but it is also used to prove the soundness of the type system.

Our model for λ^{amor^-}'s types is based on the technique of step-indexed Kripke logical relations [3]. The model for λ^{amor^-} (Fig. 3.1) is described by defining three mutually recursive relations: value relation, expression relation and substitution relations for the linear and non-linear context. These relations make use of two ghost states, the (available) potential (denoted by p) and the step-index (denoted by T). The step-index is a purely technical device that we use to make our relation well-founded. The use of step-index is completely standard. The potential, on the other hand, is the main interesting aspect of our model. As mentioned earlier, the purpose of potential is to account for resource usage. Technically, the potential can be viewed as a Kripke world.

The value relation (denoted by $[\![.]\!]$) gives an interpretation to λ^{amor^-} types in terms of sets of triples of the form (p, T, v). The potential p specifies an upper-bound on the potential required to construct the value v. The value relation is defined by nested induction on types and the step-index.

The interpretation for the $\mathbf{1}$ (unit) type includes the only inhabitant denoted by $()$, along with an arbitrary step-index and a potential. The interpretation for the base type is similar. The interpretation for the list type is defined by a further induction on list size: for a list of size 0 the value relation contains a *nil* value with any step-index and any potential, while for a list of size $s+1$, the value relation consists of $(p, T, v :: l)$ s.t. the potential p suffices to give interpretation to the head (v) at type τ and the tail (l) at type $L^s\tau$. For a tensor ($\otimes$) pair, both components can be used. Therefore, the potential required to construct a tensor pair should be at least equal to the sum of the potentials required to construct the components. For a with (&) pair, either but not both of the components can be used. So we take the max[1] of the potentials.

1 the max is not really needed because the model admits monotonicity on potentials

$$
\begin{array}{lll}
\llbracket \mathbf{1} \rrbracket & \triangleq & \{(p, T, ())\} \\
\llbracket b \rrbracket & \triangleq & \{(p, T, v) \mid v \in \llbracket b \rrbracket\} \\
\llbracket L^0 \tau \rrbracket & \triangleq & \{(p, T, \mathit{nil})\} \\
\llbracket L^{s+1} \tau \rrbracket & \triangleq & \{(p, T, v :: l) \mid \exists p_1, p_2 . p_1 + p_2 \leqslant p \wedge (p_1, T, v) \in \llbracket \tau \rrbracket \wedge (p_2, T, l) \in \llbracket L^s \tau \rrbracket\} \\
\llbracket \tau_1 \otimes \tau_2 \rrbracket & \triangleq & \{(p, T, \langle\langle v_1, v_2 \rangle\rangle) \mid \\
 & & \quad \exists p_1, p_2 . p_1 + p_2 \leqslant p \wedge (p_1, T, v_1) \in \llbracket \tau_1 \rrbracket \wedge (p_2, T, v_2) \in \llbracket \tau_2 \rrbracket\} \\
\llbracket \tau_1 \,\&\, \tau_2 \rrbracket & \triangleq & \{(p, T, \langle v_1, v_2 \rangle) \mid (p, T, v_1) \in \llbracket \tau_1 \rrbracket \wedge (p, T, v_2) \in \llbracket \tau_2 \rrbracket\} \\
\llbracket \tau_1 \oplus \tau_2 \rrbracket & \triangleq & \{(p, T, \mathsf{inl}(v)) \mid (p, T, v) \in \llbracket \tau_1 \rrbracket\} \cup \{(p, T, \mathsf{inr}(v)) \mid (p, T, v) \in \llbracket \tau_2 \rrbracket\} \\
\llbracket \tau_1 \multimap \tau_2 \rrbracket & \triangleq & \{(p, T, \lambda x.e) \mid \\
 & & \quad \forall p', e', T' {<} T . (p', T', e') \in \llbracket \tau_1 \rrbracket_{\mathcal{E}} \implies (p + p', T', e[e'/x]) \in \llbracket \tau_2 \rrbracket_{\mathcal{E}}\} \\
\llbracket !\tau \rrbracket & \triangleq & \{(p, T, !e) \mid (0, T, e) \in \llbracket \tau \rrbracket_{\mathcal{E}}\} \\
\llbracket [n]\, \tau \rrbracket & \triangleq & \{(p, T, v) \mid \exists p' . p' + n \leqslant p \wedge (p', T, v) \in \llbracket \tau \rrbracket\}\} \\
\llbracket \mathbb{M}\, n\, \tau \rrbracket & \triangleq & \{(p, T, v) \mid \\
 & & \quad \forall n', v', T' {<} T . v \Downarrow^{n'}_{T'} v' \implies \exists p' . n' + p' \leqslant p + n \wedge (p', T - T', v') \in \llbracket \tau \rrbracket\} \\
\llbracket \forall \alpha . \tau \rrbracket & \triangleq & \{(p, T, \Lambda.e) \mid \forall \tau', T' {<} T . (p, T', e) \in \llbracket \tau[\tau'/\alpha] \rrbracket_{\mathcal{E}}\} \\
\llbracket \forall i . \tau \rrbracket & \triangleq & \{(p, T, \Lambda.e) \mid \forall I, T' {<} T . (p, T', e) \in \llbracket \tau[I/i] \rrbracket_{\mathcal{E}}\} \\
\llbracket C \Rightarrow \tau \rrbracket & \triangleq & \{(p, T, \Lambda.e) \mid . \models C \implies (p, T, e) \in \llbracket \tau \rrbracket_{\mathcal{E}}\} \\
\llbracket C \& \tau \rrbracket & \triangleq & \{(p, T, v) \mid . \models C \wedge (p, T, v) \in \llbracket \tau \rrbracket\} \\
\llbracket \exists s . \tau \rrbracket & \triangleq & \{(p, T, v) \mid \exists s' . (p, T, v) \in \llbracket \tau[s'/s] \rrbracket\} \\
\llbracket \lambda_t i . \tau \rrbracket & \triangleq & f \text{ where } \forall I.\ f\ I = \llbracket \tau[I/i] \rrbracket \\
\llbracket \tau\ I \rrbracket & \triangleq & \llbracket \tau \rrbracket\ I \\
 & & \\
\llbracket \tau \rrbracket_{\mathcal{E}} & \triangleq & \{(p, T, e) \mid \forall\ T' {<} T, v . e \Downarrow_{T'} v \implies (p, T - T', v) \in \llbracket \tau \rrbracket\} \\
 & & \\
\llbracket \Gamma \rrbracket_{\mathcal{E}} & \triangleq & \{(p, T, \gamma) \mid \exists f : \mathcal{V}ars \to \mathcal{P}ots . \\
 & & \quad (\forall x \in \mathit{dom}(\Gamma) . (f(x), T, \gamma(x)) \in \llbracket \Gamma(x) \rrbracket_{\mathcal{E}}) \wedge (\sum_{x \in \mathit{dom}(\Gamma)} f(x) \leqslant p)\} \\
 & & \\
\llbracket \Omega \rrbracket_{\mathcal{E}} & \triangleq & \{(0, T, \delta) \mid (\forall x \in \mathit{dom}(\Omega) . (0, T, \delta(x)) \in \llbracket \tau \rrbracket_{\mathcal{E}})\}
\end{array}
$$

Figure 3.1: Model of λ^{amor^-} types

Inhabitants of the sum $(\tau_1 \oplus \tau_2)$ type can be inhabitants of either τ_1 using (inl) or τ_2 using (inr). Thus, the required potential should be enough to handle both the cases.

Next, we explain the interpretation of the arrow type: $(p, T, \lambda x.e)$ is in the interpretation of $\tau_1 \multimap \tau_2$ if for any expression e' in the (expression) interpretation of the input type τ_1 (with some potential p' and smaller step index T'), we have $(\lambda x.e)e'$ or equivalently $e[e'/x]$ in the (expression) interpretation of the result type τ_2 with the total potential i.e. $p + p'$ (p' coming from the substitution) and smaller step-index T' (as the application will consume at least one step).

The interpretation of polymorphic and the constraint type $(C \Rightarrow \tau)$ is based on similar reasoning as that for the arrow type. However, an important point about the interpretation of type-level quantification is the use of the step-index. Since λ^{amor^-} has impredicative quantification over types, we use the step-index to break the circularity in the definition and make the relation well-founded. Such a use of step-index is not new. It has been used in prior work like [48].

Next we explain the value relation for the exponential type: $!e$ is in the interpretation of $!\tau$ with some arbitrary potential and step-index iff e is in the (expression) interpretation of τ the same step-index and 0 potential. It is important that the inhabitants of τ do not have any potential with them, because otherwise we can end-up with infinite potential due to replication.

Next is the modal type $[n]\,\tau$: (p, T, v) is in the interpretation of $[n]\,\tau$ iff the required potential p is sufficient to account for n and the potential required for v. Note that the same value v is in the interpretation of both τ and $[n]\,\tau$, this justifies the ghost nature of the potential at the term level.

Next comes the type for the graded monad. The idea is that the total required resources, $p + \kappa$ (p which is required by the monadic value and κ which the monadic value needs for forcing), should be enough to account for the actual cost of forcing (κ') plus the potential (p') that is required for the resulting value (v').

Finally, we explain the interpretation for the type family $(\lambda_t i.\tau)$. The type family is a type-level function denoted by f s.t. when applied to some index I it yields a set which *is* the interpretation of $\tau[I/i]$.

The remaining cases of the value relation described in Fig. 3.1 should be self-explanatory.

The expression relation (denoted by $[\![.]\!]_{\mathcal{E}}$) is defined by a set of triples consisting of a potential, p, a step-index, T, and an expression e. Such a triple is in the interpretation at type τ iff the value obtained after the pure reduction of e is in the value interpretation of τ with the same potential (pure evaluations do not consume any resources). This works because we use the monad to isolate cost effects. As a result, all the cost checking is localized to the value relation of the monadic type (described above).

Finally, we define the substitution relations for both the linear context (Γ) and the non-linear context (Ω). The two key points about the interpretation of Γ are: 1) there exists a function mapping each variable to a potential value s.t. the substituted value along with the corresponding potential is in the value relation of the type of that variable and 2) the required potential p of the context is sufficient to account for the required potential for the substitutions of all the variables. The interpretation for Ω is much simpler. It only demands that the substituted value is in the interpretation of the type of the variable at 0 potential.

The main meta-theoretic property of the model is described using the fundamental theorem (Theorem 3). It basically states that if e is a syntactically well-typed expression at type τ (obtained via typing rules) then e is also a semantically well-typed term at the same type τ (i.e. is in the expression relation at type τ).

Theorem 3 (Fundamental theorem for λ^{amor^-})**.** $\forall \Theta, \Omega, \Gamma, e, \tau, T, p_l, \gamma, \delta, \sigma, \iota.$
$\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \wedge (p_l, T, \gamma) \in \llbracket \Gamma\ \sigma\iota \rrbracket_{\mathcal{E}} \wedge (0, T, \delta) \in \llbracket \Omega\ \sigma\iota \rrbracket_{\mathcal{E}} \implies$
$(p_l, T, e\ \gamma\delta) \in \llbracket \tau\ \sigma\iota \rrbracket_{\mathcal{E}}$.

The proof of this theorem is by induction on the given typing judgment $\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau$ with an additional induction on the step-index in the proof of the fixpoint combinator. Theorem 1 and Theorem 2 are direct corollaries of this fundamental theorem.

We can derive several interesting corollaries about the execution cost directly from this fundamental theorem. For instance, for an open term which only partially uses the input potential and saves the rest with the result, we can derive the cost bounds as stated in Corollary 4. Basically we derive an upper-bound on the cost of execution of e applied to unit (written $e\ ()$). The total available potential here is $q + p_l$ (q units are required by e and p_l units are given to us from the linear substitution γ). The total remaining potential after the execution is $q' + p_v$ (refer to the interpretation of the modal type described earlier). The corollary basically shows that the consumed (available minus remaining) potential is a good upper-bound on the cost of execution (denoted by J). We will show an interesting use of this corollary for giving an alternate (semantic) proof of soundness of Univariate RAML in Chapter 5.

Corollary 4. $\forall \Gamma, e, q, q', \tau, p_l, \gamma, J, v_t, v.$
$.;.;.;.;\Gamma \vdash e : [q]\,\mathbf{1} \multimap \mathbb{M}\,0\,([q']\,\tau) \wedge (p_l, \_, \gamma) \in \llbracket \Gamma \rrbracket_{\mathcal{E}} \wedge e\ ()\ \gamma \Downarrow v_t \Downarrow^{J} v \implies$
$\exists p_v.\ (p_v, \_, v) \in \llbracket \tau \rrbracket \wedge J \leqslant (q + p_l) - (q' + p_v)$

4 EXAMPLES

In this chapter we describe various examples of type-based amortized analysis in λ^{amor^-}. All the examples described below have been type checked in λ^{amor^-} but we do not describe the typing derivations here. They can be found in the technical report [52].

4.1 MAP

For our first example, we show the standard list map function assuming that the cost of applying the mapping function is a fixed c units. We show that such a function can be mapped over a list of length n each of whose elements comes with a potential of c units. We show how these requirements can be encoded purely in the types of λ^{amor^-}. The type and the term for map are described as follows:

```
$map : \forall n, c.!(\tau_1 \multimap \mathbb{M}\, c\, \tau_2) \multimap L^n([c]\,\tau_1) \multimap \mathbb{M}\, 0\, (L^n \tau_2)$
fix $map.\Lambda.\Lambda.\lambda g l.$
 let $!g_u = g$ in
   match $l$ with
    $|\, nil \mapsto$ ret $nil$
    $|\, h :: t \mapsto$
      release $h_e = h$ in
        bind $h_n = g_u\ h_e$ in
          bind $t_n = map[][]\ !g_u\ t$ in
            ret $h_n :: t_n$
```

Listing 4.1: map in λ^{amor^-}

The type of map is polymorphic in the length of the list (n) and the cost (c) required for every application of the mapping function. The type of the mapping function is given by $!(\tau_1 \multimap \mathbb{M}\, c\, \tau_2)$. There is an exponential as the function has to be applied on all elements of the given list. The c in the return type of the mapping function, $\mathbb{M}\, c\, \tau_2$, is the cost of each application. This is the standard way of encoding an effectful function using a cost monad. Alternatively, we could also have stipulated a mapping function of the type $!([c]\, \tau_1 \multimap \mathbb{M}\, 0\, \tau_2)$. λ^{amor^-} supports both encodings. We have found the latter to be more expressive in some cases, e.g., the Church encoding in Section 4.3 and embedding of a relatively complete type system in Chapter 7. The list type denoted by $L^n([c]\, \tau_1)$ indicates that every element in the list of length n carries a potential of c units. The return type of map, $\mathbb{M}\, 0\, (L^n \tau_2)$, indicates that a list of length n and type τ_2 is returned and there is no additional cost requirement (as the potential coming from the list elements suffices for the cost of the mapping function). The term for the map function is usual. It returns a nil when the input list is empty, otherwise it returns a list of elements obtained after applying the given mapping function on each element of the given list.

Note that no version of the prior work RAML [27, 29, 30] can encode this example as it uses a higher-order function. Prior work AARA [28] can encode this example if aggregate potential of $n * c$ is associated with the list as a whole. This is because it cannot associate potential with arbitrary types.

4.2 APPEND

Our next example is an encoding of a list append function where we assume to incur a unit cost for every cons (::) operation.

```
$append : \forall n_1, n_2. L^{n_1}([1]\, \tau) \multimap L^{n_2}\tau \multimap \mathbb{M}\, 0\, (L^{n_1+n_2}\tau)$
$\text{fix } append.\Lambda.\Lambda.\lambda l_1\, l_2.\ \text{match } l_1 \text{ with}$
    $\mid nil \mapsto \text{ret}(l_2)$
    $\mid h :: t \mapsto$
      $\text{release } h_e = h \text{ in}$
        $\text{bind } t_e = append[\,][\,]\ t\ l_2 \text{ in}$
          $\text{bind } - = \uparrow^1 \text{ in ret } h_e :: t_e$
```

Listing 4.2: append in λ^{amor^-}

The type of $append$ is polymorphic in the lengths of the two lists. Every element of the first list comes with a potential of one unit, indicated by the type $(L^{n_1}([1]\, \tau))$. This potential is released and consumed for every cons operation and hence no additional potential is required, nor is any potential left after the operation finishes. Note the return type $\mathbb{M}\, 0\, (L^{n_1+n_2}\tau)$, which has a 0 cost. The term of the append function is self-explanatory. It releases the potential which is available at the head of the list and consumes it using the tick construct, modeling the cost for performing a cons.

An interesting aspect of this example regards partial application. If append is partially applied (with just the first list) then the closure created will capture potential in it. So, if this closure gets used more than once this will lead to duplicating the stored potential. This cannot happen in λ^{amor^-} because of affineness. Prior work including RAML [27, 29, 30] and AARA [28, 33] cannot handle this kind of partial application. However, if this example is rewritten s.t. the potential is associated with only the last argument then [33] would be able to type check it. [28], on the other hand completely ignores partial applications and forces atomic full application for Curried functions.

4.3 CHURCH ENCODING

Our next example shows how to type Church numerals and operations on them. Typing these constructions require non-trivial use of type and index families. The type we give to Church numerals is both general and expressive enough to encode and give precise cost to operations like addition, multiplication and exponentiation.

To begin, let us first consider the typing of Church numerals without any cost. To recap, Church numerals encode natural numbers using function applications. For example, a Church zero is defined as $\lambda f.\lambda x.x$ (with zero applications), a Church one as $\lambda f.\lambda x.f\ x$ (with one application), a Church two as $\lambda f.\lambda x.f\ f\ x$ (with two applications) and so on. To type a Church numeral, we must specify a type for f. We assume that we have an $\mathbb{N}$-indexed family of types α and f maps $\alpha\ i$ to $\alpha\ (i+1)$ for every i. Then, the nth Church numeral, given such a function f, maps $\alpha\ 0$ to $\alpha\ n$.

Next, we consider costs. Here, we are interested in counting a unit cost for every *function application*. We want to encode the precise costs of operations like addition, multiplication in their types. Classically these operations are defined using, for instance, a successor function for f in the case of addition, an addition function for f in the case of multiplication and so on. Therefore, in the type of Church nat we must also account for the cost of f in a general way to allow for such compositional definitions. This cost is specified using a cost family C from $\mathbb{N}$ to $\mathbb{R}^+$. The cost of applying f depends on the index of the argument (called j_n below). Then, given such a f, the nth Church numeral maps $\alpha\ 0$ to $\alpha\ n$ with cost $C\ 0 + \ldots + C\ (n-1) + n$, where each $C\ i$ is the cost of using f the ith time and the last n is the cost of the n applications in the definition of the nth Church numeral. Our type for Church numerals captures exactly this intuition. The full type of a Church number is given as follows:

$$\begin{aligned}&\mathsf{Nat} = \lambda_t n.\forall\alpha : \mathbb{N} \to \mathsf{Type}.\forall C : \mathbb{N} \to \mathbb{R}^+.\\ &\quad !(\forall j_n.((\alpha\ j_n \otimes [C\ j_n]\ \mathbf{1}) \multimap \mathbb{M}\,0\,(\alpha\ (j_n+1)))) \multimap\\ &\quad \mathbb{M}\,0\,((\alpha\ 0 \otimes [(C\ 0 + \ldots + C\ (n-1) + n)]\ \mathbf{1}) \multimap \mathbb{M}\,0\,(\alpha\ n))\end{aligned}$$

We describe a term for the Church one (denoted by $\overline{1}$) that corresponds to the type $\mathsf{Nat}\ 1$. Since $\overline{1}$ consists of only one application, we only need an input potential of $(C\ 1)+1$ in the type, all of which gets consumed. For simplification, define a notation to indicate consumption of a unit potential with an application: $e_1 \uparrow^1 e_2 \triangleq \mathsf{bind} - = \uparrow^1 \mathsf{in}\ e_1\ e_2$. The Church one is defined as follows:

```
1̄ : Nat 1
1̄ ≜ Λ.Λ.λf.
        ret (λx. let !f_u = f in
                   let ⟨⟨y_1, y_2⟩⟩ = x in
                     release − = y_2 in
                       bind a = store() in
                         f_u 0 ↑^1 ⟨⟨y_1, a⟩⟩)
```

Listing 4.3: Encoding of the Church numeral "1" in λ^{amor^-}

The term corresponding to the Church one takes the input pair x and obtains the value and the potential from it. It then releases the potential and stores it on a, which is then used to apply f just once.

Let us now see the type and the encoding for Church addition. Church addition is defined using a successor function ($succ$) which is also defined and type-checked in λ^{amor^-}, but whose details we elide here. It is just enough to know that the cost of successor under the chosen cost model is two units, $succ : \forall n.\,[2]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\mathsf{Nat}[n] \multimap \mathbb{M}\,0\,\mathsf{Nat}[n+1])$. An encoding of Church addition (add) in λ^{amor^-} is described in Listing 4.4. The type of add takes the required potential $(4*n_1+2)$ units along with two Church naturals ($\mathsf{Nat}\ n_1$ and $\mathsf{Nat}\ n_2$) as arguments and computes their sum. The potential of $(4*n_1+2)$ units corresponds to the precise cost of performing the Church addition under the chosen cost model. The whole type is parameterized on n_1 and n_2.

```
add : ∀n_1, n_2. [(4 ∗ n_1 + 2)] 1 ⊸ M 0 (Nat n_1 ⊸ M 0(Nat n_2 ⊸ M 0 Nat (n_1 + n_2)))
    add ≜ Λ.Λ.λp.
              ret (λN̄_1. ret (λN̄_2.
                 release − = p in
                   bind a = E_1 in E_2))
```

$$
\begin{aligned}
E_1 &\triangleq \overline{N_1}\ \ \square\ \ \square\ \uparrow^1\ !(\Lambda.\lambda t.\,\mathsf{let}\,\langle\!\langle y_1, y_2\rangle\!\rangle = t\ \mathsf{in}\\
&\qquad\qquad\qquad \mathsf{release} - = y_2\ \mathsf{in}\\
&\qquad\qquad\qquad\quad \mathsf{bind}\, b_1 = (\mathsf{bind}\, b_2 = \mathsf{store}()\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad (succ\ \square\ b_2))\ \mathsf{in}\ \ b_1\ \uparrow^1 y_1)\\
\\
E_2 &\triangleq\ \mathsf{bind}\, b = \mathsf{store}()\ \mathsf{in}\ \ a\ \uparrow^1 \langle\!\langle \overline{N_2}, b\rangle\!\rangle
\end{aligned}
$$

Listing 4.4: Encoding of the Church addition in λ^{amor^-}

Besides add and $succ$ we have encoded the Church multiplication and exponentiation operations (described along with their typing derivations in the technical report [52]). Their definitions follow similar composition patterns as for $succ$ and add. Having such a general type for Church numerals which can encode the precise cost of Church operations shows the expressive power of λ^{amor^-}. We are not aware of such a general encoding in a pure monadic system without potentials.

4.4 EAGER FUNCTIONAL QUEUE

As explained in Chapter 1, eager functional queues are implemented using two stacks represented by lists, say l_1 and l_2. Enqueue is implemented as a push on l_1. Dequeue is implemented as a pop from l_2 if it is non-empty. If l_2 is empty, then the contents of l_1 are transferred to l_2 and the new l_2 is popped. The transfer from l_1 to l_2 reverses l_1, thus changing the stack's LIFO semantics to a queue's FIFO semantics. We describe the encoding of this functional queue, assuming a unit cost for every list cons operation.

The amortized analysis of functional queues works by accounting for the cost of dequeuing an element at the time it is enqueued. This is sound because an enqueued element can be dequeued at most once. Concretely, the enqueue operation takes a potential of 3 units, 1 of which is used by the enqueue operation itself and the remaining 2 are stored with the element in the list l_1 to be used later in the dequeue operation if required. This is reflected in the type of enqueue. The term for enqueue is obvious so we skip it here.

$enq : \forall m, n.\, [3]\,\mathbf{1} \multimap \tau \multimap L^n([2]\,\tau) \multimap L^m\tau \multimap \mathbb{M}\,0\,(L^{n+1}([2]\,\tau) \otimes L^m\tau)$

The dequeue operation (denoted by dq below) is a bit more involved. The constraints in the type of dequeue reflect a) dequeue can only be performed on a non-empty queue, i.e., if $m + n > 0$ and b) the sum of the lengths of the resulting list is only 1 less than the length of the input lists, i.e., $\exists m', n'.((m' + n' + 1) = (m + n))$. The full type and the term for the dequeue operation are described in Listing 4.5. Dequeue makes use of a function $move$ which performs the job of inserting the elements of first list into the second one in reverse order. We skip the description of move. Type-checked terms for enqueue, dequeue and move can be found in the technical report [52].

$$\begin{aligned}
\mathsf{dq} : \forall m, n.(m+n > 0) \Rightarrow\ & L^{m}([2]\,\tau) \multimap \\
& L^{n}\tau \multimap \\
& \mathbb{M}\,0\,(\exists m', n'.((m'+n'+1) = (m+n))\&(L^{m'}[2]\,\tau \otimes L^{n'}\tau))
\end{aligned}$$

```
dq ≜ Λ.Λ.Λ.λ l₁ l₂. match l₂ with
 |nil ↦ bind l_r = move [][] l₁ nil in
    match l_r with
      |nil ↦ fix x.x
      |h_r :: l'_r ↦ ret Λ.⟨⟨nil, l'_r⟩⟩
 |h₂ :: l'₂ ↦ ret Λ.⟨⟨l₁, l'₂⟩⟩
```

Listing 4.5: Dequeue operation for eager functional queue in λ^{amor^-}

4.5 OKASAKI'S IMPLICIT QUEUE

Next we describe an encoding of a lazy data structure, namely, Okasaki's implicit queue[49]. An implicit queue is an instance of implicit recursive slowdown [49], which is an efficient way of encoding algorithms by incrementally computing (encoded using laziness) over data. An implicit queue can be a shallow queue consisting of zero or one element, or, it can be a deep queue consisting of three parts namely front, middle and rear. Okasaki represents the front part as consisting of one or two elements, middle part as a suspended implicit queue of *pairs* and the rear part as consisting of zero or one element. Okasaki uses the method of debits [49] to analyze the amortized cost of operations like head, tail and snoc, counting the number of recursive calls in them.

To encode implicit queue in λ^{amor^-}, we describe the different ways of constructing it using six value constructors. We show that by adding these constructors along with a construct to case analyze them to λ^{amor^-}, we are not only able to succinctly represent Okasaki's implicit queue but also show how to encode the method of debits for amortized analysis of snoc, head and tail.

The types for the six value constructors ($C0$ - $C5$) are described in Fig. 4.1. $C0$ and $C1$ correspond to the two ways of creating a shallow queue, while $C2$ to $C5$ correspond to the four ways of representing a deep queue. Constructors corresponding to the deep queue also carry a potential argument in their first position. They correspond to the debit invariants that Okasaki uses for the cost analysis of head, tail and snoc. We use a different cost model than Okasaki. We count unit cost for every case analysis on the implicit queue, because it is easier to represent. It turns out that the same debit invariants are sufficient for our cost model too. This is because every recursive call is always preceded by a case analysis in this implementation, resulting in the same amortized cost in both the cost models.

$$\begin{aligned}
&C0 : \mathit{Queue}\ \tau\\
&C1 : \tau \multimap \mathit{Queue}\ \tau\\
&C2 : [1]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\tau \otimes \mathit{Queue}(\tau \otimes \tau)) \multimap \mathit{Queue}\ \tau\\
&C3 : [0]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\tau \otimes \mathit{Queue}(\tau \otimes \tau) \otimes \tau) \multimap \mathit{Queue}\ \tau\\
&C4 : [2]\,\mathbf{1} \multimap \mathbb{M}\,0\,((\tau \otimes \tau) \otimes \mathit{Queue}(\tau \otimes \tau)) \multimap \mathit{Queue}\ \tau\\
&C5 : [1]\,\mathbf{1} \multimap \mathbb{M}\,0\,((\tau \otimes \tau) \otimes \mathit{Queue}(\tau \otimes \tau) \otimes \tau) \multimap \mathit{Queue}\ \tau
\end{aligned}$$

Figure 4.1: Value constructors for Okasaki's implicit queue

As an example, we describe the implementation of a function which we use to obtain both the head and tail of a queue in Listing 4.6. It has an amortized cost of three units as indicated by the type. It basically works by case analyzing the input queue and returning the head and tail after accounting for the cost. We release the input potential of three units and consume one to account for the cost of case analysis. The remaining two units are either discarded or are consumed by the different cases in the implementation. The cases corresponding to the shallow queue are very simple: they both discard the remaining potential. When the input queue is $C0$, then we return false denoted by $\mathsf{fix}\ x.x$ (as it is not possible to take the head and tail of an empty queue). When the input queue is $C1\ x$ then we return a pair of x and the empty queue.

The remaining cases (the ones corresponding to the deep queue) force the corresponding suspension by providing the right amount of potential and obtaining the head and tail from it. We only explain one of the cases corresponding to $C3$ here. From Fig. 4.1 we know that the suspension in $C3$ needs zero units of potential to be forced. So we store zero units of potential in p' and the remaining two units of potential from the input are stored in p_o (this will be used later in obtaining the tail). We then force the suspension denoted by x to obtain the front (f), middle (m) and rear (r). The front f is just returned as the head while tail is constructed using the constructor $C5$. Inside the suspension of $C5$ we have one unit of additional potential available to us via p''. We use this one unit of potential from p'' along with the two units of potential available from p_o to obtain a total of three units of potential to make a recursive call on the middle part to obtain the head and tail for the tail of the middle queue (m). This finishes the implementation corresponding to this case. The implementation for the C2 case is similar, while those for C4 and C5 do not make any recursive calls.

An important point of comparison with Okasaki's encoding is that Okasaki works in a non-affine setting (unlike ours) and hence he uses the same middle queue twice to obtain the head and tail, which we cannot since λ^{amor^-} is an affine language. This is the reason for writing a combined function to obtain both (individual head and tail functions are just written as projection functions on top of this combined function).

```
$\mathtt{headTail} : [3]\,\mathbf{1} \multimap \forall\alpha.\mathit{Queue}\ \ \alpha \multimap \mathbb{M}\,0\,(\alpha \otimes \mathit{Queue}\ \ \alpha)$
$\mathtt{headTail} \triangleq \mathsf{fix}\ \ HT.\lambda p.\Lambda.\lambda\ \ q.$
$- = \mathsf{release}\, p\ \mathsf{in}\ \ - = \uparrow^1\ \mathsf{in}\ \ \mathsf{ret}$
 $\mathsf{case}\ \ q\ \ \mathsf{of}$
  $|\mathit{C0} \mapsto \mathsf{fix}\ x.x$

  $|\mathit{C1}\ \ x \mapsto \mathsf{ret}\langle\langle x, \mathit{C0}\rangle\rangle$

  $|\mathit{C2}\ \ x \mapsto$
   $\mathsf{bind}\, p' = \mathsf{store}()\ \mathsf{in}\ \ \mathsf{bind}\, p_o = \mathsf{store}()\ \mathsf{in}$
    $\mathsf{bind}\, x' = x\ \ p'\ \mathsf{in}\ \mathsf{let}\langle\langle f, m\rangle\rangle = x'\ \mathsf{in}$
     $\mathsf{ret}\langle\langle f, (\mathit{C4}\ \ (\lambda p''.- = \mathsf{release}\, p_o\ \mathsf{in}\ - = \mathsf{release}\, p''\ \mathsf{in}\ \mathsf{bind}\, p_r = \mathsf{store}()\ \mathsf{in}\ HT\ \ p_r\ \ []\ \ m))\rangle\rangle$

  $|\mathit{C3}\ \ x \mapsto$
   $\mathsf{bind}\, p' = \mathsf{store}()\ \mathsf{in}\ \ \mathsf{bind}\, p_o = \mathsf{store}()\ \mathsf{in}$
    $\mathsf{bind}\, x' = x\ \ p'\ \mathsf{in}\ \ \mathsf{let}\langle\langle fm, r\rangle\rangle = x'\ \mathsf{in}\ \ \mathsf{let}\langle\langle f, m\rangle\rangle = fm\ \mathsf{in}$
    $\mathsf{ret}\langle\langle f, (\mathit{C5}\ \ (\lambda p''.- = \mathsf{release}\, p_o\ \mathsf{in}\ - = \mathsf{release}\, p''\ \mathsf{in}$
                $\mathsf{bind}\, p''' = \mathsf{store}()\ \mathsf{in}\ \ \mathsf{bind}\, ht = HT\ \ p'''\ \ []\ \ m\ \mathsf{in}\ \mathsf{ret}\langle\langle ht, r\rangle\rangle))\rangle\rangle$

  $|\mathit{C4}\ \ x \mapsto$
   $\mathsf{bind}\, p' = \mathsf{store}()\ \mathsf{in}\ \mathsf{bind}\, x' = x\ \ p'\ \mathsf{in}\ \mathsf{let}\langle\langle f, m\rangle\rangle = x'\ \mathsf{in}\ \mathsf{let}\langle\langle f_1, f_2\rangle\rangle = f\ \mathsf{in}$
   $\mathsf{ret}\langle\langle f_1, \mathit{C2}\ \ (\lambda p''.\,\mathsf{ret}\langle\langle f_2, m\rangle\rangle)\rangle\rangle$

  $|\mathit{C5}\ \ x \mapsto$
   $\mathsf{bind}\, p' = \mathsf{store}()\ \mathsf{in}\ \mathsf{bind}\, x' = x\ \ p'\ \mathsf{in}\ \mathsf{let}\langle\langle fm, r\rangle\rangle = x'\ \mathsf{in}\ \mathsf{let}\langle\langle f, m\rangle\rangle = fm\ \mathsf{in}\ \mathsf{let}\langle\langle f_1, f_2\rangle\rangle = f\ \mathsf{in}$

     $\mathsf{ret}\langle\langle f_1, (\mathit{C3}\ \ (\lambda p''.\,\mathsf{ret}\langle\langle\langle\langle f_2, m\rangle\rangle, r\rangle\rangle))\rangle\rangle$
```

Listing 4.6: Function to obtain head and tail

The technical report [52] contains full typing derivations for the $\mathtt{headTail}$, $\mathtt{head}$, $\mathtt{tail}$ and $\mathtt{snoc}$ operations.

5

EMBEDDING UNIVARIATE RAML

Resource Aware ML (RAML) [29, 32] is a type and effect system for amortized analysis of OCaml programs using the method of potentials [18, 55]. It basically works by associating potential with specific datatypes like list and trees. This potential is made available for consumption when an expression is eliminated. The potential in RAML is specified as a function of the *size* of the inputs. Many versions of RAML exist. For instance, [30] supports linear potentials, [29] supports univariate polynomial potentials and [27] supports multivariate polynomial potentials. Potentials in λ^{amor^-} are very different. They are more general and not just limited to the sizes of the inputs. Also, λ^{amor^-} do not restrict potentials to datastructures only. They can be associated to arbitrary types using the modal type constructor (as described earlier). The main motivation for showing an embedding of RAML is three fold: 1) we want to show that λ^{amor^-} is more expressive than RAML and thus can be used to analyze all examples that have been tried on RAML, 2) we want to show that the potential-handling approach of λ^{amor^-} is more general than RAML's and therefore RAML-style potentials can be captured in λ^{amor^-} and finally 3) we want to show that λ^{amor^-}, despite being a *call-by-name* framework, can embed RAML which is a *call-by-value* framework.

In this chapter we describe an embedding of Univariate RAML [29, 32] (which subsumes Linear RAML) into λ^{amor^-}. We leave embedding multivariate RAML to future work but anticipate no fundamental difficulties in doing so.

5.1 BRIEF PRIMER ON UNIVARIATE RAML

We give a brief primer of Univariate RAML [29, 32] here. The key feature of Univariate RAML is an ability to encode univariate polynomials in the size of the input data as potential functions. Such functions are expressed as non-negative linear combinations of binomial coefficients $\binom{n}{k}$, where n is the size of the input data structure and k is some natural number. Vector annotations on the list type $L^{\vec{q}}\tau$, for instance, are used as a representation of such univariate polynomials. The underlying potential on a list of size n and type $L^{\vec{q}}\tau$ can then be described as $\phi(\vec{q}, n) \triangleq \sum_{1 \leqslant i \leqslant k} \binom{n}{i} q_i$ where $\vec{q} = \{q_1 \dots q_k\}$. The authors of RAML show using the properties of binomial coefficients, that such a representation is amenable to an inductive characterization of polynomials which plays a crucial role in setting up the typing rules of their system. If $\vec{q} = \{q_1 \dots q_k\}$ is the potential vector associated with a list then $\lhd(\vec{q}) = \{q_1 + q_2, q_2 + q_3, \dots q_{k-1} + q_k, q_k\}$ is the potential vector associated with the tail of that list. Trees follow a treatment similar to lists. Base types (unit, bools, ints) have zero potential and the potential of a pair is just the sum of the potentials of the components. A snippet of the definition of the potential function $\Phi(a : A)$ (from [32]) is described below.

$$\Phi(a : A) \;=\; 0 \quad \text{where } A = \{\mathsf{unit}, \mathsf{int}, \mathsf{bool}\}$$

$$\Phi([] : L^{\vec{q}}A) \;=\; 0$$

$$\Phi((a_1, a_2) : (A_1, A_2)) \;=\; \Phi(a_1 : A_1) + \Phi(a_2 : A_2)$$

$$\Phi((a :: \ell) : L^{\vec{q}}A) \;=\; q_1 + \Phi(a : A) + \Phi(\ell : L^{\lhd\vec{q}}A)$$

$$\text{where } \vec{q} = \{q_1 \dots q_k\}$$

A type system is built around this basic idea with a typing judgment of the form $\Sigma; \Gamma \vdash^{q}_{q'} e_r : \tau$ where Γ is a typing context mapping free variables to their types, Σ is a context for function signatures mapping a function name to a type (this is separate from the typing context because RAML only has first-order functions that are declared at the top-level), q and q' denote the statically approximated available and remaining potential before and after the execution of e_r, respectively, and τ is the zero-order type of e_r. Vector annotations are specified on list and tree types (as mentioned above). Types of first-order functions follow an intuition similar to the typing judgment above. $\tau_1 \overset{q/q'}{\to} \tau_2$ denotes the type of a first-order RAML function which takes an argument of type τ_1 and returns a value of type τ_2. q units of potential are needed before this function can be applied and q' units of potential are left after this function has been applied. Intuitively, the cost of the function is upper-bounded by (q+potential of the input) - (q'+potential of the result). Fig. 5.1 describe typing rules for function application and list cons. The *app* rule type-checks the function application with an input and remaining potential of $(q + K^{app}_1)$ and $(q' - K^{app}_2)$[1] units, respectively. RAML divides the cost of application between K^{app}_1 and K^{app}_2 units. Of the available $q + K^{app}_1$ units, q units are required by the function itself and K^{app}_1 units are consumed

1 Every time a subtraction like $(I - J)$ appears, RAML implicitly assumes that there is a side condition $(I - J) \geqslant 0$.

before the application is performed. Likewise, of the remaining $q' - K_2^{app}$ units, q' units are made available from the function and K_2^{app} units are consumed after the application is performed. The *cons* rule requires an input potential of $q + p_1 + K^{cons}$ units of which p_1 units are added to the potential of the resulting list and K^{cons} units are consumed as the cost of performing this operation.

$$\frac{\tau_1 \overset{q/q'}{\to} \tau_2 \in \Sigma(f)}{\Sigma; x : \tau_1 \vdash^{q+K_1^{app}}_{q'-K_2^{app}} f\ x : \tau_2}\ \text{app} \qquad \frac{\vec{p} = (p_1, \dots, p_k)}{\Sigma; x_h : \tau, x_t : L^{(\lhd\ \vec{p})}\tau \vdash^{q+p_1+K^{cons}}_{q} cons(x_h, x_t) : L^{\vec{p}}\tau}\ \text{cons}$$

Figure 5.1: Selected type rules of Univariate RAML from [32]

Soundness of the type system is defined by Theorem 5. Soundness is defined for *top-level* RAML programs (formalized later in Definition 7), which basically consist of first-order function definitions (denoted by F) and the "main" expression e, which uses those functions. Stack (denoted by V) and heap (denoted by H) are used to provide bindings for free variables and locations in e.

Theorem 5 (Univariate RAML's soundness)**.** $\forall H, H', V, \Gamma, \Sigma, e, \tau, {}^s v, p, p', q, q', t.$
$P = F, e$ is a RAML top-level program and
$H \models V : \Gamma \wedge \Sigma, \Gamma \vdash^q_{q'} e : \tau \wedge V, H \vdash^p_{p'} e \Downarrow_t {}^s v, H' \implies p - p' \leqslant (\Phi_{H,V}(\Gamma) + q) - (q' + \Phi_H({}^s v : \tau))$

5.2 TYPE-DIRECTED TRANSLATION

As mentioned above, types in Univariate RAML include types for unit, booleans, integers, lists, trees, pairs and first-order functions. Without loss of generality we introduce two simplifications: 1) we abstract RAML's bool and int types into an arbitrary base type denoted by b and 2) we just choose to work with the list type only ignoring trees. These simplifications only make the development more concise as we do not have to deal with the redundancy of treating similar types again and again.

The translation from Univariate RAML to λ^{amor^-} is type-directed. We describe the type translation function (denoted by $(\!|.|\!)$) from RAML types to λ^{amor^-} types in Fig. 5.2.

Since RAML allows for full replication of unit and base types, we translate RAML's base type, b, into !b of λ^{amor^-}. But translation of the unit type does not need a !, as $\mathbf{1}$ and $!\mathbf{1}$ are isomorphic in λ^{amor^-}. Unlike the unit and base type of RAML, the list type does have some potential associated with it, indicated by $\vec{q}$. Therefore, we translate RAML's list type into a pair type composed of a modal unit type carrying the required potential and a λ^{amor^-} list type. Since the list type in λ^{amor^-} is refined with size, we

$$\begin{aligned}
(\!|\mathsf{unit}|\!) &= \mathbf{1} \\
(\!|b|\!) &= !b \\
(\!|L^{\vec{q}}\ \tau|\!) &= \exists s.([\phi(\vec{q}, s)]\ \mathbf{1} \otimes L^s (\!|\tau|\!))
\end{aligned}
\qquad
\begin{aligned}
(\!|(\tau_1, \tau_2)|\!) &= ((\!|\tau_1|\!) \otimes (\!|\tau_2|\!)) \\
(\!|\tau_1 \overset{q/q'}{\to} \tau_2|\!) &= [q]\ \mathbf{1} \multimap (\!|\tau_1|\!) \multimap \mathbb{M}\, 0\, ([q']\ (\!|\tau_2|\!))
\end{aligned}$$

Figure 5.2: Type translation of Univariate RAML

add an existential on the pair to quantify the size of the list. The potential captured by the unit type must equal the potential associated with the RAML list (this is indicated by the function $\phi(\vec{q}, s)$). The function $\phi(\vec{q}, s)$ corresponds to the one that RAML uses to compute the total potential associated with a list of s elements, which we described above. Note the difference in how potentials are managed in RAML vs how they are managed in the translation. In RAML, the potential for an element gets added to the potential of the tail with every cons operation and, dually only the, potential of the head element is consumed in the match operation. The translation, however, does not assign potential on a per-element basis, instead the aggregate potential is captured using the ϕ function and the translations of the cons and the match expressions work by adding or removing potential from this aggregate. We believe a translation which works with per element potential is also feasible but we would need an additional index to identify the elements of the list in the list data type.

We translate a RAML pair type into a tensor ($\otimes$) pair. This is in line with how pairs are treated in RAML (both elements of the pair are available on elimination). Finally, a function type $\tau_1 \overset{q/q'}{\to} \tau_2$ in RAML is translated into the function type $[q]\ \mathbf{1} \multimap (\!|\tau_1|\!) \multimap \mathbb{M}\, 0\, ([q']\ (\!|\tau_2|\!))$. As in RAML, the translated function type also requires a potential of q units for application and a potential of q' units remains after the application. The monadic type is required because we cannot release/store potential without going into the monad. The translation of typing contexts is defined pointwise using the type translation function.

We use this type translation function to produce a translation for Univariate RAML expressions by induction on RAML's typing judgment. The translation judgment is $\Sigma; \Gamma \vdash^{q}_{q'} e_r : \tau \quad \rightsquigarrow \quad e_a$. It basically means that a well-typed RAML expression e_r is translated into a $\lambda^{\mathrm{amor}^-}$ expression e_a. The translated expression is of the type $[q]\ \mathbf{1} \multimap \mathbb{M}\, 0\, ([q'](\!|\tau|\!))$. We only describe the app rule here (Fig. 5.3). Since we know that the desired term must have the type $[q + K^{app}_1]\ \mathbf{1} \multimap \mathbb{M}\, 0\, ([q' - K^{app}_2](\!|\tau|\!))$, the translated term is a function which takes an argument, u, of the desired modal type and releases the potential to make it available for consumption. The continuation then consumes K^{app}_1 potential that leaves $q - K^{app}_1$ potential remaining for $\mathsf{bind}\, P = \mathsf{store}()\ \mathsf{in}\ E_1$. We then store q units of potential with the unit and use it to perform a function application. We get a result of type $\mathbb{M}\,0\,([q']\,(\!|\tau_2|\!))$. We release these q' units of potential and consume K_2^{app} units from it. This leaves us with a remaining potential of $q' - K_2^{app}$ units. We store this remaining potential with f_2 and box it up in a monad to get the desired type. Translations of other RAML terms (which we do not describe here) follow a similar approach. The entire translation is intuitive and relies extensively on the ghost operations store and release at appropriate places.

$$\frac{}{\Sigma; . \vdash_q^{q+K^{unit}} () : \mathsf{unit} \rightsquigarrow \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K^{unit}} \text{ in } \mathsf{bind}\ a = \mathsf{store}() \text{ in } \mathsf{ret}(a)}\ \text{unit}$$

$$\frac{}{\Sigma; . \vdash_q^{q+K^{base}} c : b \rightsquigarrow \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K^{base}} \text{ in } \mathsf{bind}\ a = \mathsf{store}(!c) \text{ in } \mathsf{ret}(a)}\ \text{base}$$

$$\frac{}{\Sigma; x : \tau \vdash_q^{q+K^{var}} x : \tau \rightsquigarrow \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K^{var}} \text{ in } \mathsf{bind}\ a = \mathsf{store}\, x \text{ in } \mathsf{ret}(a)}\ \text{var}$$

$$\frac{\tau_1 \overset{q/q'}{\to} \tau_2 \in \Sigma(f)}{\Sigma; x : \tau_1 \vdash_{q'-K_2^{app}}^{q+K_1^{app}} f\ x : \tau_2 \rightsquigarrow \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K_1^{app}} \text{ in } \mathsf{bind}\ P = \mathsf{store}() \text{ in } E_1}\ \text{app}$$

where

$E_1 = \mathsf{bind}\, f_1 = (f\ P\ x) \text{ in } \mathsf{release}\, f_2 = f_1 \text{ in } \mathsf{bind} - = \uparrow^{K_2^{app}} \text{ in } \mathsf{bind}\, f_3 = \mathsf{store}\, f_2 \text{ in } \mathsf{ret}\, f_3$

$$\frac{}{\begin{array}{c}\Sigma; \emptyset \vdash_q^{q+K^{nil}} \mathit{nil} : L^{\vec{p}}\tau \rightsquigarrow \\ \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K^{nil}} \text{ in } \mathsf{bind}\ a = \mathsf{store}() \text{ in } \mathsf{bind}\ b = \mathsf{store}\langle\!\langle a, \mathit{nil}\rangle\!\rangle \text{ in } \mathsf{ret}(b)\end{array}}\ \text{nil}$$

$$\frac{\vec{p} = (p_1, \ldots, p_k)}{\Sigma; x_h : \tau, x_t : L^{(\lhd\, \vec{p})}\tau \vdash_q^{q+p_1+K^{cons}} \mathsf{cons}(x_h, x_t) : L^{p}\tau \rightsquigarrow \lambda u.\mathsf{release} - = u \text{ in } \mathsf{bind} - = \uparrow^{K^{cons}} \text{ in } E_0}\ \text{cons}$$

where

$E_0 = x_t; x.\, \mathsf{let}\langle\!\langle x_1, x_2\rangle\!\rangle = x \text{ in } E_1$

$E_1 = \mathsf{release} - = x_1 \text{ in } \mathsf{bind}\ a = \mathsf{store}() \text{ in } \mathsf{bind}\ b = \mathsf{store}\langle\!\langle a, x_h :: x_2\rangle\!\rangle \text{ in } \mathsf{ret}(b)$

Figure 5.3: Expression translation: Univariate RAML to λ^{amor}

We show that the translation is type-preserving by proving that the obtained λ^{amor^-} terms are well-typed (Theorem 6). The proof of this theorem works by induction on RAML's type derivation.

Theorem 6 (Type preservation: Univariate RAML to λ^{amor^-})**.** If $\Sigma; \Gamma \vdash^{q}_{q'} e : \tau$ in Univariate RAML then there exists e' such that $\Sigma; \Gamma \vdash^{q}_{q'} e : \tau \rightsquigarrow e'$ and there is a derivation of $.; .; .; (\!|\Sigma|\!), (\!|\Gamma|\!) \vdash e' : [q]\,\mathbf{1} \multimap \mathbb{M}\,0\,([q'](\!|\tau|\!))$ in λ^{amor^-}.

As mentioned earlier, RAML only has first-order functions which are defined at the top-level. So, we need to lift this translation to the top-level. Definition 7 defines the top-level RAML program along with the translation.

Definition 7 (Top level RAML program translation)**.** Given a top-level RAML program
$P \triangleq F, e_{main}$ where $F \triangleq f1(x) = e_{f1}, \ldots, fn(x) = e_{fn}$ s.t.
$\Sigma, x : \tau_{f1} \vdash^{q_1}_{q'_1} e_{f1} : \tau'_{f1} \ \ldots \ \ \Sigma, x : \tau_{fn} \vdash^{q_n}_{q'_n} e_{fn} : \tau'_{fn}$ and $\Sigma, \Gamma \vdash^{q}_{q'} e_{main} : \tau$
where $\Sigma = f1 : \tau_{f1} \overset{q_1/q'_1}{\to} \tau'_{f1}, \ldots, fn : \tau_{fn} \overset{q_n/q'_n}{\to} \tau'_{fn}$
The translation of P, denoted by $\overline{P}$, is defined as $(\overline{F}, e_t)$ where
$\overline{F} = \text{fix } f_1.\lambda u.\lambda x.e_{t1}, \ldots, \text{fix } f_n.\lambda u.\lambda x.e_{tn}$ s.t.
$\Sigma, x : \tau_{f1} \vdash^{q_1}_{q'_1} e_{f1} : \tau'_{f1} \rightsquigarrow e_{t1} \ \ldots \ \ \Sigma, x : \tau_{fn} \vdash^{q_n}_{q'_n} e_{fn} : \tau'_{fn} \rightsquigarrow e_{tn}$ and
$\Sigma, \Gamma \vdash^{q}_{q'} e_{main} : \tau \rightsquigarrow e_t$

5.3 SEMANTIC PROPERTIES OF THE EMBEDDING

Besides type-preservation, we additionally : 1) prove that our translation preserves semantics and cost of the source RAML term and 2) re-derive RAML's soundness result using λ^{amor^-}'s fundamental theorem (Theorem 3) and properties of the translation. This is a sanity check to ensure that our type translation preserves cost meaningfully (otherwise, we would not be able to recover RAML's soundness theorem in this way).

Semantics and cost preservation is formally stated in Theorem 8, which can be read as follows: if e_s is a closed source (RAML) term which translates to a target (λ^{amor^-}) term e_t and if the source expression evaluates to a value (and a heap H, because RAML uses imperative boxed data structures) then the target term after applying to a unit (because the translation is always a function) can be evaluated to a value ${}^{t}v_f$ via pure ($\Downarrow$) and forcing ($\Downarrow^{J}$) relations s.t. the source and the target values are the same and the cost of evaluation in the target is at least as much as the cost of evaluation in the source.

Theorem 8 (Semantics and cost preservation)**.** $\forall H, e, {}^{s}v, p, p', q, q'$.
$.; . \vdash^{q}_{q'} e_s : b \rightsquigarrow e_t \wedge ., . \vdash^{p}_{p'} e \Downarrow {}^{s}v, H \implies$
$\exists {}^{t}v_f, J.e_t() \Downarrow \_ \Downarrow^{J} {}^{t}v_f \wedge {}^{s}v = {}^{t}v_f \wedge p - p' \leqslant J$

The proof of Theorem 8 is via a cross-language relation between RAML and λ^{amor^-} terms. The relation is complex because it has to relate RAML's imperative data structures (like list which is represented as a chain of pointers in the heap) with λ^{amor^-}'s purely functional datastructures. The cross-language relation relating Univariate RAML and λ^{amor^-} terms is described in Fig. 5.4

$$
\begin{array}{lcl}
\lfloor \mathsf{unit} \rfloor^{H}_{\mathcal{V}} & \triangleq & \{(T, {}^{s}v, {}^{t}v) \mid {}^{s}v \in [\![\mathsf{unit}]\!] \wedge {}^{t}v \in [\![\mathbf{1}]\!] \wedge {}^{s}v = {}^{t}v\} \\
\lfloor b \rfloor^{H}_{\mathcal{V}} & \triangleq & \{(T, {}^{s}v, !\,{}^{t}v) \mid {}^{s}v \in [\![b]\!] \wedge {}^{t}v \in [\![b]\!] \wedge {}^{s}v = {}^{t}v\} \\
\lfloor (\tau_1, \tau_2) \rfloor^{H}_{\mathcal{V}} & \triangleq & \{(T, \ell, \langle\langle {}^{t}v_1, {}^{t}v_2 \rangle\rangle) \mid H(\ell) = ({}^{s}v_1, {}^{s}v_2) \wedge (T, {}^{s}v_1, {}^{t}v_1) \in \lfloor \tau_1 \rfloor_{\mathcal{V}} \wedge (T, {}^{s}v_2, {}^{t}v_2) \in \lfloor \tau_2 \rfloor_{\mathcal{V}}\} \\
\lfloor L^{\vec{q}} \tau \rfloor^{H}_{\mathcal{V}} & \triangleq & \{(T, \ell_s, \langle\langle (), l_t \rangle\rangle) \mid (T, \ell_s, l_t) \in \lfloor L\ \tau \rfloor^{H}_{\mathcal{V}}\} \\
\textit{where} & & \\
\lfloor L\ \tau \rfloor^{H}_{\mathcal{V}} & \triangleq & \{(T, \mathsf{NULL}, \mathit{nil})\} \cup \\
 & & \{(T, \ell, {}^{t}v :: l_t) \mid H(\ell) = ({}^{s}v, \ell_s) \wedge (T, {}^{s}v, {}^{t}v) \in \lfloor \tau \rfloor_{\mathcal{V}} \wedge (T, \ell_s, l_t) \in \lfloor L\ \tau \rfloor_{\mathcal{V}}\} \\
\lfloor \tau_1 \xrightarrow{q/q'} \tau_2 \rfloor^{H} & \triangleq & \{(T, f(x) = e_s, \mathsf{fix}\ f.\lambda u.\lambda x.e_t) \mid \forall {}^{s}v', {}^{t}v', T' < T\ . \\
 & & (T', {}^{s}v', {}^{t}v') \in \lfloor \tau_1 \rfloor^{H}_{\mathcal{V}} \implies (T', e_s, e_t[()/u][{}^{t}v'/x][\mathsf{fix}\ f.\lambda u.\lambda x.e_t/f]) \in \lfloor \tau_2 \rfloor^{\{x \mapsto {}^{s}v'\}, H}_{\mathcal{E}}\}
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor \tau \rfloor^{V,H}_{\mathcal{E}} & \triangleq & \{(T, e_s, e_t) \mid \forall H', {}^{s}v, p, p', t < T\ .\ V, H \vdash^{p}_{p'} e_s \Downarrow_t {}^{s}v, H' \implies \\
 & & \exists {}^{t}v_t, {}^{t}v_f, J. e_t \Downarrow_{-} {}^{t}v_t \Downarrow^{J}_{-} {}^{t}v_f \wedge (T - t, {}^{s}v, {}^{t}v_f) \in \lfloor \tau \rfloor^{H'}_{\mathcal{V}} \wedge p - p' \leqslant J\}
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor \Gamma \rfloor^{H}_{\mathcal{V}} & = & \{(T, V, \delta_t) \mid \forall x : \tau \in \mathit{dom}(\Gamma).(T, V(x), \delta_t(x)) \in \lfloor \tau \rfloor^{H}_{\mathcal{V}}\} \\
\lfloor \Sigma \rfloor^{H}_{\mathcal{V}} & = & \{(T, \delta_{sf}, \delta_{tf}) \mid (\forall f : (\tau_1 \xrightarrow{q/q'} \tau_2) \in \mathit{dom}(\Sigma).(T, \delta_{sf}(f)\ \delta_{sf}, \delta_{tf}(f)\ \delta_{tf}) \in \lfloor (\tau_1 \xrightarrow{q/q'} \tau_2) \rfloor^{H})\}
\end{array}
$$

Figure 5.4: Cross language model: Univariate RAML to λ^{amor}

We define a value relation (denoted by $\lfloor . \rfloor^{H}_{\mathcal{V}}$) for relating a RAML value with a λ^{amor^-} value. It is defined by induction on the source (univariate RAML) types. Notice that the value relation is indexed with a heap H. This is done to accommodate heap-based implementation of lists in RAML. From the type translation we know that a RAML list is translated into a pair consisting of a unit and a λ^{amor^-} list (we do not have an explicit intro form for existential types). Therefore, the value relation for the list type must relate a RAML list denoted by ℓ_s with a λ^{amor^-} pair denoted by $\langle\langle (), l_t \rangle\rangle$ s.t. ℓ_s and l_t are related at the $L\ \tau$ (list type without the potential). The value interpretation of $L\ \tau$ relates a RAML NULL value to a *nil* in λ^{amor^-} and relates the two lists pointwise at type τ after dereferencing the location ℓ in the heap H. The other cases of the value relation are obvious.

RAML only has first-order functions. This is incorporated in the model by defining a separate relation for the arrow type: $\lfloor \tau_1 \xrightarrow{q/q'} \tau_2 \rfloor^{H}$ (notice the absence of the subscript

$\mathcal{V}$). From the type translation we know that RAML's arrow type ($\tau_1 \overset{q/q'}{\rightarrow} \tau_2$) is translated to a λ^{amor^-} type $[q]\,\mathbf{1} \multimap (\!|\tau_1|\!) \multimap \mathbb{M}\,0\,([q']\,(\!|\tau_2|\!))$. Also, the first-order only restriction does not preclude RAML from having recursion. Therefore, a RAML first-order function, $f(x) = e_s$, is related to a λ^{amor^-} fixpoint over a function, $\text{fix } f.\lambda u.\lambda x.e_t$. The rest of the definition basically says that if we supply related values as arguments to the two functions then e_s is related to $e_t[()/u][\text{fix } f.\lambda u.\lambda x.e_t/f]$ with those arguments as substitutions for x under the expression relation (described next). Note that the potential on the source type does not play any role in the cross-language model. The potential is only relevant for the type translation (as explained above).

RAML's expression evaluation is defined wrt substitutions for free variables and locations in a RAML expression as mentioned earlier. The substitutions for variables are performed using a stack V and, for locations, substitution is performed using a heap H. Like the value relation, the expression relation is also indexed with a heap H but, additionally, it is also indexed with a stack V for reasons we just explained. It basically relates a RAML expression to a λ^{amor^-} expression s.t. if the given RAML expression terminates to a value v and heap H' by consuming $p - p'$ resources, then the related λ^{amor^-} term must also terminate to some value v_f s.t. the two resulting values are related under the value relation at the obtained heap (H'). Also the cost consumed (J) in λ^{amor^-} is *at least* the cost consumed by the evaluation of related RAML expression.

We also need a relation for relating substitutions for zero-order terms ($\lfloor\Gamma\rfloor_{\mathcal{V}}^{H}$) and first-order functions ($\lfloor\Sigma\rfloor_{\mathcal{V}}^{H}$). The $\lfloor\Gamma\rfloor_{\mathcal{V}}^{H}$ relation is obvious but $\lfloor\Sigma\rfloor_{\mathcal{V}}^{H}$ needs a comment. First-order functions can refer to other functions and to themselves (because of recursion). The self-reference would be a bound variable but reference to other functions would involve a free occurrence of a function name. This is the reason we apply the substitution (both the source and target) when we relate the functions at the $\lfloor\tau_1 \overset{q/q'}{\rightarrow} \tau_2\rfloor^{H}$ relation. We have not explained the use of step-index in the model yet. It is used for a particular proof which we explain later.

We prove the model sound by proving the fundamental theorem (Theorem 9).

Theorem 9 (Fundamental theorem of RAML to λ^{amor^-} translation).

$\forall \Sigma, \Gamma, q, q', \tau, e_s, e_t, I, V, H, \delta_t, \delta_{sf}, \delta_{tf}, T.$
$\Sigma; \Gamma \vdash_{q'}^{q} e_s : \tau \rightsquigarrow e_t \wedge (T, V, \delta_t) \in \lfloor\Gamma\rfloor_{\mathcal{V}}^{H} \wedge (T, \delta_{sf}, \delta_{tf}) \in \lfloor\Sigma\rfloor_{\mathcal{V}}^{H} \implies$
$(T, e_s\delta_{sf}, e_t\ ()\ \delta_t\delta_{tf}) \in \lfloor\tau\rfloor_{\mathcal{E}}^{V,H}$

Note that the theorem relates the given RAML expression e_s to a unit-applied translation of e_s. This is because from Theorem 6 we know that a well-typed translated term is always a function at the top-level. The proof of this theorem works by induction on RAML's typing derivation. There are two consequences of this fundamental theorem: a) we have shown that the translation preserves semantics and b) the cost of execution of the translated term in λ^{amor^-} is lower bounded by the cost of the execution in RAML. The extra cost could be due to administrative reductions.

Finally, we re-derive RAML's soundness (Theorem 5) in λ^{amor^-} using λ^{amor^-}'s fundamental theorem and the properties of the translation. To prove this theorem, we obtain a translated term corresponding to the term e (of Theorem 5) via our translation. Then, using Theorem 8, we show that the cost of forcing the unit application of the target is lower-bounded by $p - p'$. After that, we use Corollary 4 to obtain the upper-bound on $p - p'$ as required in the statement of Theorem 5.

6

FROM λ^{AMOR^-} TO λ^{AMOR} (FULL)

Recall the Church encoding from Section 4.3. A Church numeral always applies the function argument a finite number of times. However, the type that we assigned to Church numeral specified an unbounded number of copies for the function argument. Similarly, the index j_n can only take n unique values in the range 0 to $n-1$, but it was left unrestricted in the type that we saw earlier. Both these limitations are due to λ^{amor^-}'s lack of ability to specify these constraints at the level of types. These limitations, however, can be avoided by refining the exponential type $(!\tau)$ a bit. In particular, we add dependent sub-exponentials, denoted by $!_{a<I}\tau$, that can not only specify a bound on the number of copies of the underlying term but can also specify the constraints on the index-level substitutions that are needed in the Church encoding. $!_{i<n}\tau$ represents n copies of τ in which i is uniquely substituted with all values from 0 to $n-1$.

Such dependent sub-exponentials have been used in the prior work. dℓPCF [39][1], for instance, uses it to obtain relative completeness of typing for PCF programs, which means every PCF program can be type checked in dℓPCF, where the cost of the PCF program gets internalized in dℓPCF's typing derivation. This is a very powerful result. However, cost analysis in dℓPCF works only for whole programs. This is because dℓPCF does not internalize cost into the types but rather tracks it only on the typing judgment. As a result, in order to verify the cost of e_2 in the let expression, say $\text{let } x = e_1 \text{ in } e_2$, we would need the whole typing derivation of e_1 as cost is encoded on the judgment in dℓPCF.

Contrast this with λ^{amor} where cost requirements are described in the types ($\mathbb{M}\,\kappa\,\tau$ for instance). In this case, the cost of e_2 can be verified just by knowing the type of e_1 (the whole typing derivation of e_1 is not required to type check e_2. Therefore e_1 can be verified separately).

1 The bounded exponential was first introduced in Bounded Linear Logic [24], but it was deliberately restricted to polynomial bounds only. dℓPCF [39] generalized the bounds, we use this generalized form here.

We show that by adding such an indexed sub-exponential to λ^{amor^-}, we can not only obtain the same relative completeness[2] result that dℓPCF obtains, but also provide a compositional alternative to the dℓPCF style of cost analysis. We describe the addition of $!_{i<n}\tau$ to λ^{amor^-} in this chapter. We call the resulting system λ^{amor}.

6.1 CHANGES TO THE TYPE SYSTEM: SYNTAX AND TYPE RULES

We take the same language as described earlier in Chapter 2 but replace the exponential type with an indexed sub-exponential type. There are no changes to the term syntax or semantics of the language. We just extend the index language with two specific counting functions described below.

$$\begin{array}{llll}
\text{Index} & I, J, K & ::= & \ldots \mid \sum_{a<J} I \mid \bigcirc\!\!\!\!\triangle_{a}^{J,K} I \mid \ldots \\
\text{Types} & \tau & ::= & \ldots \mid !_{a<I}\tau \mid \ldots \\
\text{Non-linear context} & \Omega & ::= & . \mid \Omega, x :_{a<I} \tau \\
\text{for term variables} & & &
\end{array}$$

$$\Omega_1 + \Omega_2 \triangleq \begin{cases} \Omega_2 & \Omega_1 = . \\ (\Omega_1' + \Omega_2/x), x :_{c<I+J} \tau & \Omega_1 = \Omega_1', x :_{a<I} \tau[a/c] \wedge (x :_{b<J} \tau[I+b/c]) \in \Omega_2 \\ (\Omega_1' + \Omega_2), x :_{a<I} \tau & \Omega_1 = \Omega_1', x :_{a<I} \tau \wedge (x :_{-} -) \notin \Omega_2 \end{cases}$$

$$\sum_{a<I} \Omega \triangleq \begin{cases} . & \Omega = . \\ (\sum_{a<I} \Omega), x :_{c<\sum_{a<I} J} \sigma & \Omega = \Omega', x :_{b<J} \sigma[(\sum_{d<a} J[d/a] + b)/c] \end{cases}$$

Figure 6.1: Changes to the type system syntax

We describe the changes introduced to the type and index language in Fig. 6.1. Since the sub-exponential type helps in specifying the number of copies of a term, we find inclusion of two specific counting functions to the index language very useful, both of which have been inspired from prior work [39]. The first one is a function for computing a bounded sum over indices, denoted by $\sum_{a<J} I$. It basically describes summation of I with a ranging from 0 to $J-1$ inclusive, i.e., $I[0/a] + \ldots + I[J-1/a]$. The other function is used for computing the number of nodes in a graph structure

2 Use of indexed sub-exponential is just one way of obtaining relative completeness. There could be other approaches, which we do not get into here.

like a forest of recursion trees. This is called the forest cardinality and denoted $\bigcirc\hspace{-0.75em}\triangle_a^{J,K} I$. The forest cardinality $\bigcirc\hspace{-0.75em}\triangle_a^{J,K} I$ counts the number of nodes in the forest (described by I) consisting of K trees starting from the Jth node. Nodes are assumed to be numbered in a pre-order fashion. It can be formally defined as in Fig. 6.2 and is used to count and identify children in the recursion tree of a fix construct.

$$\begin{aligned}
\bigcirc\hspace{-0.75em}\triangle_a^{I,0} K &= 0 \\
\bigcirc\hspace{-0.75em}\triangle_a^{I,J+1} K &= \bigcirc\hspace{-0.75em}\triangle_a^{I,J} K + \left(\bigcirc\hspace{-0.75em}\triangle_a^{I+1+\bigcirc\hspace{-0.75em}\triangle_a^{I,J} K,\, K[I+\bigcirc\hspace{-0.75em}\triangle_a^{I,J} K/a]} K\right)
\end{aligned}$$

Figure 6.2: Formal definition of forest cardinality from [39]

The typing judgment is still the same: $\Psi;\Theta;\Delta;\Omega;\Gamma \vdash e : \tau$. However, the definition of Ω is now different. The non-linear context Ω now carries the constraint on the index variable described on the ":" as in $x :_{a<I} \tau$ (Fig. 6.1). It specifies that there are I copies of x with type τ in which the free a is substituted with unique values in the range from 0 to $I - 1$. The non-linear context also differs in the definition of splitting. The definition of $+$ (splitting, also referred to as the binary sum) for Ω allows for the same variable to be present in the two contexts but by allowing splitting over the index ranges. Binary sum of Ω_1 and Ω_2 in λ^{amor^-} was just a disjoint union of the two contexts. However, here in λ^{amor}, it permits Ω_1 and Ω_2 to have common variables but their multiplicities should add up. We also introduce a notion of bounded sum for the non-linear context denoted by $\sum_{a<I} \Omega$. Both binary and bounded sum over non-linear contexts are described in Fig. 6.1.

We only describe the type rules for the sub-exponential and the fixpoint in Fig. 6.3 as these are the only rules that change. T-subExpI is the rule for the introduction form of the sub-exponential. It says that if an expression e has type τ under a non-linear context Ω and $a < I$ s.t. e does not use any linear resources (indicated by an empty Γ) then $!e$ has type $!_{a<I}\tau$ under context $\sum_{a<I} \Omega$. As before, we can always use the weakening rule to add linear resources to the conclusion. T-subExpE is similar to T-expE defined earlier but additionally it also carries the index constraint coming from the type of e_1 in the context for e_2.

The fixpoint expression (fix $x.e$) encodes recursion by allowing e to refer to fix $x.e$ via x. T-fix defines the typing for such a fixpoint construct. It is a refinement of the corresponding rule in Fig. 2.3. The refinements serve two purposes: 1) they make the total number of recursive calls explicit (this is represented by L) and 2) they identify each instance of the recursive call in a pre-order traversal of the recursion tree. This is represented by the index $(b + 1 + \bigcirc\hspace{-0.75em}\triangle_b^{b+1,a} I)$ (representing the ath child of the bth node in the pre-order traversal). Using these two refinements, the T-fix rule in Fig. 6.3 can be read as follows: if for all I copies of x in the context we can type check e with τ, then we can also type check the top-most instance of fix $x.e$ with type $\tau[0/b]$ (0 denotes the starting node in the pre-order traversal of the entire recursion tree). Contrast the rules described in Fig. 6.3 with the corresponding rules for λ^{amor^-} described earlier in Fig. 2.3.

$$\frac{\Psi; \Theta, a; \Delta, a < I; \Omega; . \vdash e : \tau}{\Psi; \Theta; \Delta; \sum_{a<I} \Omega; . \vdash !e : !_{a<I}\tau} \text{ T-subExpI}$$

$$\frac{\Psi; \Theta; \Delta; \Omega_1; \Gamma_1 \vdash e : (!_{a<I}\tau) \qquad \Psi; \Theta; \Delta; \Omega_2, x :_{a<I} \tau; \Gamma_2 \vdash e' : \tau'}{\Psi; \Theta; \Delta; \Omega_1 + \Omega_2; \Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !\, x = e \;\mathsf{in}\; e' : \tau'} \text{ T-subExpE}$$

$$\frac{\Psi; \Theta, b; \Delta, b < L; \Omega, x :_{a<I} \tau[(b + 1 + \bigcirc\!\!\!\!\triangle_b^{b+1,a} I)/b]; . \vdash e : \tau \qquad L \geqslant \bigcirc\!\!\!\!\triangle_b^{0,1} I}{\Psi; \Theta; \Delta; \sum_{b<L} \Omega; . \vdash \mathsf{fix}\;\; x.e : \tau[0/b]} \text{ T-fix}$$

Figure 6.3: Changes to the type rules

We also introduce a new subtyping rule, sub-bSum. sub-bSum helps move the potential from the outside to the inside of a sub-exponential. This is sound because 1) potentials are really ghosts at the term level. Therefore terms of type $[\sum_{a<I} K]\, !_{a<I}\tau$ and $!_{a<I}\, [K]\, \tau$ are both just exponentials and 2) there is only a change in the position but no change of potential in going from $[\sum_{a<I} K]\, !_{a<I}\tau$ to $!_{a<I}\, [K]\, \tau$. We have proved that this new subtyping rule is sound wrt the model of λ^{amor} types by proving that if τ is a subtype of τ' according to the syntactic subtyping rules then the interpretation of τ is a subset of the interpretation of τ'. This is formalized in Lemma 10. σ and ι represent the substitutions for the type and index variables respectively.

$$\frac{}{\Psi; \Theta; \Delta \vdash [\sum_{a<I} K]\, !_{a<I}\tau <: !_{a<I}\, [K]\, \tau} \text{ sub-bSum}$$

It is noteworthy that sub-bSum is the only rule in λ^{amor} which specifies how the two modalities, namely, the sub-exponential ($!_{a<I}\tau$) and potential capturing modal type ($[p]\, \tau$) interact with each other. People familiar with monads and comonads might wonder, why such an interaction between the sub-exponential and the monad is not required? We believe this is because we can always internalize the cost on the

type using the store construct, so just relating exponential and potential modal type suffices. However, studying such interactions could be an interesting direction for future work.

Lemma 10 (Value subtyping lemma). $\forall \Psi, \Theta, \Delta, \tau \in \mathsf{Type}, \tau', \sigma, \iota.$
$\Psi; \Theta; \Delta \vdash \tau <: \tau' \wedge . \models \Delta\iota \implies [\![\tau\ \sigma\iota]\!] \subseteq [\![\tau'\ \sigma\iota]\!]$

6.2 SEMANTIC MODEL OF TYPES

We only describe the value relation for the sub-exponential here as the remaining cases of the value relation are exactly the same as before. $(p, T, !e)$ is in the value interpretation at type $!_{a<I}\tau$ iff the potential p suffices for all I copies of e at the *instantiated* types $\tau[i/a]$ for $0 \leqslant i < I$. The other change to the model is in the interpretation of Ω. This time we have (p, δ) instead of $(0, \delta)$ in the interpretation of Ω s.t. p is sufficient for all copies of all variables in the context. The changes to the model are described in Fig. 6.4.

$$
\begin{array}{lcl}
[\![!_{a<I}\tau]\!] & \triangleq & \{(p, T, !e) \mid \exists p_0, \ldots, p_{I-1}. p_0 + \ldots + p_{I-1} \leqslant p \wedge \forall 0 \leqslant i < I. (p_i, T, e) \in [\![\tau[i/a]]\!]_{\mathcal{E}}\} \\
[\![\Omega]\!]_{\mathcal{E}} & = & \{(p, T, \delta) \mid \exists f : \mathcal{V}ars \to \mathcal{I}ndices \to \mathcal{P}ots. \\
& & (\forall (x :_{a<I} \tau) \in \Omega. \forall 0 \leqslant i < I. (f\ x\ i, T, \delta(x)) \in [\![\tau[i/a]]\!]_{\mathcal{E}}) \wedge \\
& & (\sum_{x :_{a<I}\tau \in \Omega} \sum_{0 \leqslant i < I} f\ x\ i) \leqslant p\}
\end{array}
$$

Figure 6.4: Changes to the model

We prove the soundness of the model by proving a slightly different fundamental theorem (Theorem 11). There is an additional potential (p_m) coming from the interpretation of Ω (which was 0 earlier).

Theorem 11 (Fundamental theorem of λ^{amor}). $\forall \Psi, \Theta, \Delta, \Omega, \Gamma, e, T, \tau \in \mathsf{Type}, p_l, p_m, \gamma, \delta, \sigma, \iota.$
$\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \wedge$
$(p_l, T, \gamma) \in [\![\Gamma\ \sigma\iota]\!]_{\mathcal{E}} \wedge (p_m, T, \delta) \in [\![\Omega\ \sigma\iota]\!]_{\mathcal{E}} \wedge . \models \Delta\ \iota \implies$
$(p_l + p_m, T, e\ \gamma\delta) \in [\![\tau\ \sigma\iota]\!]_{\mathcal{E}}.$

The proof of the theorem proceeds in a manner similar to that of Theorem 3, i.e., by induction on the typing derivation. Now, in the fix case, we additionally induct on the recursion tree (this also involves generalizing the induction hypothesis to account for the potential of the children of a node in the recursion tree). The technical report [52] has the entire proof.

7

EMBEDDING dℓPCF

In this chapter we show that λ^{amor} (full), as described in the previous chapter, is relatively complete for PCF programs. We prove this by showing a type, semantics and cost preserving embedding of dℓPCF into λ^{amor}.

7.1 BRIEF PRIMER ON dℓPCF

dℓPCF [39] is a call-by-name calculus with an affine type system for doing cost analysis of PCF programs. Terms and types of dℓPCF are described in Fig. 7.1. dℓPCF works with the standard PCF terms but refines the standard types of PCF a bit to perform cost analysis. The type of natural numbers is refined with two indices $\mathsf{Nat}[I, J]$ to capture types for natural numbers in the range $[I, J]$ specified by the indices. Function types are refined with index constraints in the negative position. For instance, $[a < I]\tau_1 \multimap \tau_2$ is the type of a function which when given I copies of an expression (since dℓPCF is call-by-name) of type τ_1 will produce a value of type τ_2. The $[a < I]$ acts both as a constraint on what values a can take and also as a binder for free occurrence of a in τ_1 (but not in τ_2). $[a < I]\tau_1 \multimap \tau_2$ is morally equivalent to $(\tau_1[0/a] \otimes \ldots \otimes \tau_1[I-1/a]) \multimap \tau_2$.

$$\begin{array}{llll}
\text{d}\ell\text{PCF terms} & t & ::= & n \mid s(t) \mid p(t) \mid \mathsf{ifz}\ t\ \mathsf{then}\ u\ \mathsf{else}\ v \mid \lambda x.t \mid tu \mid \mathsf{fix}\ \ x.t \\
\text{d}\ell\text{PCF types} & \sigma & ::= & \mathsf{Nat}[I, J] \mid A \multimap \sigma \\
 & A & ::= & [a < I]\sigma
\end{array}$$

Figure 7.1: dℓPCF's syntax of terms and types from [39]

The typing judgment of dℓPCF is given by $\Theta; \Delta; \Gamma \vdash^{\xi}_{C} e_d : \tau$. Θ denotes a context of index variables, Δ denotes a context for index constraints, Γ denotes a context of term variables and C denotes the cost of evaluation of e_d. This cost C is the number of variable lookups in a full execution of e_d. ξ on the turnstile denotes an equational program used for interpreting the function symbols of the index language. Like in the negative position of the function type, multiplicities also show up with the types of the variables in the typing context. The typing rules are designed to track these multiplicities (which is a coeffect in the system). For illustration, we only show the typing rule for function application in Fig. 7.2. Notice how the cost in the conclusion is lower bounded by the sum of: a) the number of times the argument of e_1 can be used by the body, i.e., I, b) the cost of e_1, i.e., J and c) the cost of I copies of e_2, i.e., $\sum_{a<I} K$. The authors of [39] show that this kind of coeffect tracking in the type system actually suffices to give an upper-bound on the cost of execution on a K_{PCF} machine, a Krivine-style machine [38] for PCF.

$$\frac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash_J e_1 : ([a < I].\tau_1) \multimap \tau_2 \\ \Theta, a;\Delta, a < I;\Delta \vdash_K e_2 : \tau_1 \qquad \Gamma' \sqsupseteq \Gamma \oplus \sum_{a<I} \Delta \qquad H \geqslant I + J + \sum_{a<I} K\end{array}}{\Theta;\Delta;\Gamma' \vdash_H e_1\ e_2 : \tau_2}\ \text{app}$$

Figure 7.2: Typing rule for function application from [39]

States of the K_{PCF} machine consist of triples of the form (t, ρ, θ) where t is a dℓPCF term, ρ is an environment for variable binding and θ is stack of closures. A closure (denoted by C) is simply a pair consisting of a term and an environment. The left side of Fig. 7.3 describes some evaluation rules of the K_{PCF} machine from [39]. For instance, the application triple $(e_1 e_2, \rho, \theta)$ reduces in one step to e_1, and e_2 along with the current closure is pushed on top of the stack for later evaluation. This is how one would expect an evaluation to happen in a call-by-name scheme. One final ingredient that we need to describe for the soundness of dℓPCF is a notion of the size of a term, denoted by $|t|$. The size of a dℓPCF term is defined in [39] (we describe some of the clauses on the right side of Fig. 7.3).

$$\begin{array}{lcl|lcl}
 & & & |x| & = & 1 \\
(e_1\ e_2, \rho, \theta) & \rightarrow & (e_1, \rho, (e_2, \rho).\theta) & |c| & = & 1 \\
(\lambda x.e, \rho, C.\theta) & \rightarrow & (e_1, C.\rho, \theta) & |\lambda x.e| & = & |e| + 1 \\
(x, (t_0, \rho_0) \ldots (t_n, \rho_n), \theta) & \rightarrow & (t_x, \rho_x, \theta) & |e_1\ e_2| & = & |e_1| + |e_2| + 1 \\
(\text{fix } x.e, \rho, \theta) & \rightarrow & (e, (x, (\text{fix } x.e, C).\rho, \theta) & |\text{fix } x.e| & = & |e| + 1
\end{array}$$

Figure 7.3: K_{PCF} reduction rules (left) and size function (right) from [39]

Finally dℓPCF soundness (Theorem 12) states that the execution cost (denoted by n) is upper-bounded by the product of the size of the initial term, t and $(I + 1)$. dℓPCF states the soundness result for base (bounded naturals) types only and soundness for functions is derived as a corollary. $\Downarrow^n$ is a shorthand for $\xrightarrow{n}$ (n-step closure under the K_{PCF} reduction relation).

Theorem 12 (dℓPCF's soundness from [39])**.** $\forall t, I, J, K.$
$\vdash_I t : \mathsf{Nat}[J,K] \wedge t \Downarrow^n m \implies n \leqslant |t| * (I+1)$

7.2 TRANSLATING dℓPCF TO λ^{AMOR}

Without loss of generality, as in RAML's embedding, we abstract the type of naturals and treat them as a general abstract base type b. Like RAML, dℓPCF's embedding is also type directed. The type translation function is described in Fig. 7.4. dℓPCF's base type is translated into the base type of λ^{amor}. The function type $([a < I]\tau_1 \multimap \tau_2)$ translates to a function which takes I copies of the monadic translation of τ_1 (following Moggi [44]) and I units of potential (to account for I substitutions during application) as a modal unit type, and returns a monadic type of translation of τ_2. The monad on the return type is essential as a function cannot consume (I units of) potential and still return a pure value. The translation of the typing context is defined pointwise for every variable in the context. Since all variables in the dℓPCF's typing context have comonadic types (carrying multiplicities), dℓPCF's typing context is translated into the non-linear typing context of λ^{amor}.

$$
\begin{array}{lcl|lcl}
(\!| b |\!) & = & b & (\!| . |\!) & = & . \\
(\!| [a < I]\tau_1 \multimap \tau_2 |\!) & = & !_{a<I}\, \mathbb{M}\, 0\, (\!|\tau_1|\!) \multimap [I]\, \mathbf{1} \multimap \mathbb{M}\, 0\, (\!|\tau_2|\!) & (\!| \Gamma, x : [a < I]\tau |\!) & = & (\!|\Gamma|\!), x :_{a<I} \mathbb{M}\, 0\, (\!|\tau|\!)
\end{array}
$$

Figure 7.4: Type and context translation for dℓPCF

The translation judgment is of the form $\Theta; \Delta; \Gamma \vdash_I e_d : \tau \quad \rightsquigarrow \quad e_a$ where e_a denotes the translated λ^{amor} term. ξ never changes in any of dℓPCF's typing rules, so for simplification we assume it to be present globally and thus we omit it from the translation judgment. The expression translation of dℓPCF terms is defined by induction on typing judgments (Fig. 7.5). Notice that in the variable rule (var) we place a deliberate tick construct which consumes one unit of potential. This is done to match the cost model of dℓPCF. Without this accounting our semantics and cost preservation theorem would not hold. The translation of function application and the fixpoint construct make use of a coercion function (coerce, which is written in λ^{amor} itself). It helps convert an application of exponentials into an exponential of application. The coercion function is described in the box along with the expression translation rules in Fig. 7.5.

$$\dfrac{\Theta;\Delta \models J \geqslant 0 \qquad \Theta;\Delta \models I \geqslant 1 \qquad \Theta;\Delta \vdash \sigma[0/a] <: \tau \qquad \Theta;\Delta \models [a < I]\sigma \Downarrow \qquad \Theta;\Delta \models \Gamma \Downarrow}{\Theta;\Delta;\Gamma, x : [a < I]\tau \vdash_J x : \tau[0/a] \rightsquigarrow \lambda p.\mathsf{release} - = p \text{ in } \mathsf{bind} - = \uparrow^1 \text{ in } x}\ \text{var}$$

$$\dfrac{\Theta;\Delta;\Gamma, x : [a < I]\tau_1 \vdash_J e : \tau_2 \rightsquigarrow e_t}{\begin{array}{c}\Theta;\Delta;\Gamma \vdash_J \lambda x.e : ([a < I].\tau_1) \multimap \tau_2 \rightsquigarrow \\ \lambda p_1.\,\mathsf{ret}\,\lambda y.\lambda p_2.\,\mathsf{let}\,!\,x = y \text{ in } \mathsf{release} - = p_1 \text{ in } \mathsf{release} - = p_2 \text{ in } \mathsf{bind}\, a = \mathsf{store}() \text{ in } e_t\ a\end{array}}\ \text{lam}$$

$$\dfrac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash_J e_1 : ([a < I].\tau_1) \multimap \tau_2 \rightsquigarrow e_{t1} \\ \Theta, a;\Delta, a < I;\Delta \vdash_K e_2 : \tau_1 \rightsquigarrow e_{t2} \qquad \Gamma' \sqsupseteq \Gamma \oplus \sum_{a<I} \Delta \qquad H \geqslant J + I + \sum_{a<I} K\end{array}}{\Theta;\Delta;\Gamma' \vdash_H e_1\ e_2 : \tau_2 \rightsquigarrow E_0}\ \text{app}$$

$E_0 = \lambda p.\mathsf{release} - = p \text{ in } \mathsf{bind}\ a = \mathsf{store}() \text{ in } E_1$

$E_1 = \mathsf{bind}\ b = e_{t1}\ a \text{ in } \mathsf{bind}\ c = \mathsf{store}!() \text{ in } \mathsf{bind}\ d = \mathsf{store}() \text{ in } b\ (\mathsf{coerce}\ !e_{t2}\ c)\ d$

$$\dfrac{\begin{array}{c}\Theta, b;\Delta, b < L;\Gamma, x : [a < I]\sigma \vdash_K e : \tau \rightsquigarrow e_t \\ \tau[0/a] <: \mu \qquad \Theta, a, b;\Delta, a < I, b < L;\Gamma \vdash \tau[(b + 1 + \bigcirc\!\!\!\!\!\triangle_b^{b+1,a} I)/b] <: \sigma \\ \Gamma' \sqsubseteq \sum_{b<L} \Gamma \qquad L, M \geqslant \bigcirc\!\!\!\!\!\triangle_b^{0,1} I \qquad N \geqslant M - 1 + \sum_{b<L} K\end{array}}{\Theta;\Delta;\Gamma' \vdash_N \mathsf{fix}\ x.e : \mu \rightsquigarrow E_0}\ \text{fix}$$

$E_0 = \mathsf{fix}\ Y.\lambda p.\mathsf{release} - = p \text{ in } E_1$

$E_1 = \mathsf{release} - = p \text{ in } E_2$

$E_2 = \mathsf{bind}\ A = \mathsf{store}() \text{ in } \mathsf{let}\,!\,x = (E_{2.1}\ E_{2.2}) \text{ in } \mathsf{bind}\ C = \mathsf{store}() \text{ in } e_t\ C$

$E_{2.1} = \mathsf{coerce}\ !Y$

$E_{2.2} = (\lambda u.!())\ A$

$$\boxed{\begin{array}{l}\mathsf{coerce} : !_{a<I}(\tau_1 \multimap \tau_2) \multimap !_{a<I}\tau_1 \multimap !_{a<I}\tau_2 \\ \mathsf{coerce}\ F\ X \triangleq \mathsf{let}\,!\,f = F \text{ in } \mathsf{let}\,!\,x = X \text{ in} !(f\ x)\end{array}}$$

Figure 7.5: Expression translation: dℓPCF to λ^{amor}

The translated terms have the type $[I + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$ where count is defined as $count(\Gamma, x : [a < I]\tau) = count(\Gamma) + I$ (with $count(.) = 0$ as the base case). Since dℓPCF counts cost for each variable lookup in a terminating K_{PCF} reduction, the translated term must have enough potential to make sure that all copies of free variables in the context can be used. This is ensured by having $(I + count(\Gamma))$ potential as input (in the argument position of the translated type): I accounts for the substitutions coming from function applications in the dℓPCF expression and $count(\Gamma)$ accounts for the total number of possible substitutions of context variables. All translated expressions release the input potential coming from the argument. This is later consumed using a tick as in the variable rule or stored with a unit value to be used up by the induction hypothesis. We show that the translated terms are well-typed in λ^{amor} (Theorem 13).

Theorem 13 (Type preservation: dℓPCF to λ^{amor})**.** If $\Theta; \Delta; \Gamma \vdash_I e : \tau$ in dℓPCF then there exists e' such that $\Theta; \Delta; \Gamma \vdash_I e : \tau \rightsquigarrow e'$ such that there is a derivation of $.; \Theta; \Delta; (\!|\Gamma|\!); . \vdash e' : [I + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$ in λ^{amor}.

We want to highlight another point about this translation. This is the second instance (the first one was embedding of Church numerals, Section 4.3) where embedding using just a cost monad (without potentials) does not seem to work. To understand this, let us try to translate dℓPCF's function type $([a < I]\tau_1 \multimap \tau_2)$ using only the cost monad and without the potentials. One possible translation of $[a < I]\tau_1 \multimap \tau_2$ is $(!_{a<I}(\!|\tau_1|\!)) \multimap \mathbb{M}\,I\,(\!|\tau_2|\!)$. The I in the monadic type is used to account for the cost of substitution of the I copies of the argument in dℓPCF. Now, in the rule for function abstraction we have to generate a translated term of the type $\mathbb{M}(J + count(\Gamma))\,(!_{a<I}(\!|\tau_1|\!) \multimap \mathbb{M}\,I\,(\!|\tau_2|\!))$. From the induction hypothesis, we have a term of type $\mathbb{M}(I + J + count(\Gamma))\,(\!|\tau_2|\!)$. A possible term translation can be $\mathsf{ret}\,\lambda y.\,\mathsf{let}\,!\,x = y\,\mathsf{in}\,e_t$. This would require us to type e_t with $\mathbb{M}\,I\,(\!|\tau_2|\!)$ under the given context with a free x. However e_t can only be typed with $\mathbb{M}(I + J + count(\Gamma))\,(\!|\tau_2|\!)$ (which cannot be coerced to the desired type). Hence, the translation with just cost monads does not work. We believe that such a translation can be made to work by adding appropriate coercion axioms for the cost monads.

However, there is an alternate way to make this translation work, using the modal type and that is what we use. The idea is to capture the I units as a *potential* using the modal type of λ^{amor} (in the negative position) instead of capturing it (in the positive position) as a cost on the monad. Concretely, this means that, instead of translating $[a < I]\tau_1 \multimap \tau_2$ to $(!_{a<I}(\!|\tau_1|\!)) \multimap \mathbb{M}\,I\,(\!|\tau_2|\!)$, we translate it to $(!_{a<I}\,\mathbb{M}\,0\,(\!|\tau_1|\!)) \multimap [I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!)$ (as described in Fig. 7.4 earlier). Likewise, the typing judgment is also translated using the same potential approach (as described in Theorem 13). Following this approach, we obtain a term of type $[(J + I + count(\Gamma))]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!)$

from the induction hypothesis and we are required to produce a term of type $[J + \mathit{count}(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0((!_{a<I}\,\mathbb{M}\,0\,(\!|\tau_1|\!)) \multimap [I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!))$ in the conclusion. By using the ghost constructs (namely store and release) to rearrange the given potential of $J + \mathit{count}(\Gamma)$ and I units into a potential of $(J + I + \mathit{count}(\Gamma))$ units, it is clear that we can obtain a term of the desired type from the induction hypothesis. The exact term is described in the lam rule of Fig. 7.5.

The semantic correctness of our translation is proved by defining a cross-language relation between dℓPCF and λ^{amor} terms (Fig. 7.6). As before, separate value and expression interpretations are given for source (in this case dℓPCF) types. The value relation for the function type makes use of an auxiliary relation, $\lfloor [a < I]\tau \rfloor_{NE}$. The key idea behind $\lfloor [a < I]\tau \rfloor_{NE}$ is the following two part observation: 1) any dℓPCF function (of type $[a < I]\tau_1 \multimap \tau_2$) and the translation both expect I copies for the argument and 2) translation of every well-typed dℓPCF expression is wrapped inside a function which expects a unit. As a consequence of 1) and 2), a source (dℓPCF) argument expression must be related to I copies of a related target (λ^{amor}) expression when applied to a unit. Specializing the relation with the coercion function is necessary because of call-by-name semantics, in a call-by-value scheme we could have just used $\exists e'_t.e_t = !(e'_t\ ())$.

$$
\begin{array}{lcl}
\lfloor \mathsf{b} \rfloor_V & \triangleq & \{({}^s v, {}^t v) \mid {}^s v \in [\![\mathsf{b}]\!] \wedge {}^t v \in [\![\mathsf{b}]\!] \wedge {}^s v = {}^t v\} \\
\lfloor [a < I]\tau_1 \multimap \tau_2 \rfloor_V & \triangleq & \{(\lambda x.e_s, \lambda x.\lambda p.\,\mathsf{let}\,!\,x = y\ \mathsf{in}\ e_t) \mid \forall e'_s, e'_t. \\
 & & (e'_s, e'_t) \in \lfloor [a < I]\tau_1 \rfloor_{NE} \implies (e_s[e'_s/x], e_t[e'_t/y][()/p]) \in \lfloor \tau_2 \rfloor_E\} \\
\lfloor \tau \rfloor_E & \triangleq & \{(e_s, e_t) \mid \forall {}^s v.e_s \Downarrow {}^s v \implies \exists {}^t v_t, {}^t v_f, J.e_t \Downarrow {}^t v_t \Downarrow^J {}^t v_f \wedge ({}^s v, {}^t v_f) \in \lfloor \tau \rfloor_V\} \\
\lfloor [a < I]\tau \rfloor_{NE} & \triangleq & \{(e_s, e_t) \mid \exists e'_t.e_t = \mathsf{coerce}\ !e'_t\ !() \wedge \forall 0 \leqslant i < I.(e_s, e'_t()) \in \lfloor \tau[i/a] \rfloor_E\} \\
\lfloor \Gamma \rfloor_E & \triangleq & \{(\delta_s, \delta_t) \mid \\
 & & (\forall x : [a < J]\tau \in \mathit{dom}(\Gamma).\forall 0 \leqslant j < J.(\delta_s(x), \delta_t(x)) \in \lfloor \tau[j/a] \rfloor_E)
\end{array}
$$

Figure 7.6: Cross-language model dℓPCF to λ^{amor}

We prove the correctness of this relation (Theorem 14) by proving that every well-typed source term e_s of type τ is related to the unit application of the translation at the source type.

Theorem 14 (Fundamental theorem). $\forall \Theta, \Delta, \Gamma, \tau, e_s, e_t, I, \delta_s, \delta_t.$
$\Theta; \Delta; \Gamma \vdash_I e_s : \tau \rightsquigarrow e_t \wedge (\delta_s, \delta_t) \in \lfloor \Gamma\ \iota \rfloor_E \wedge . \models \Delta\ \iota \implies (e_s\delta_s, e_t\ ()\ \delta_t) \in \lfloor \tau\ \iota \rfloor_E$

To show that the meaning of the cost annotation is not lost during this translation, we want to re-derive dℓPCF's soundness in λ^{amor} using the properties of the translation only. But dℓPCF's soundness is defined wrt reduction on a K_{PCF} machine [38], as described earlier. So, we would like to rederive a proof of Theorem 15. This is a generalized version of dℓPCF's soundness (Theorem 12), where we prove the cost bound for terms of arbitrary types.

Theorem 15 (Generalized dℓPCF's soundness)**.** $\forall t, I, \tau, \rho.$
$\vdash_I (t, \epsilon, \epsilon) : \tau \wedge (t, \epsilon, \epsilon) \xrightarrow{n} (v, \rho, \epsilon) \implies n \leqslant |t| * (I + 1)$

To prove this, we need to find a way to relate K_{PCF} triples to λ^{amor} terms. For that, we come up with an approach for decompiling K_{PCF} triples into dℓPCF terms (which we can then transitively relate to λ^{amor} terms via our translation). We describe this decompilation next.

7.3 DECOMPILING K_{PCF} TRIPLES TO dℓPCF TERMS

The required decompilation procedure is defined as a function (denoted by $\llparenthesis . \rrparenthesis$) from K_{PCF} triples to dℓPCF terms. We first define decompilation for closures (the notation $\llparenthesis . \rrparenthesis$ is overloaded), by induction on the environment. For an empty environment, the decompilation is simply an identity on the given term. For an environment of the form $C_1, \dots, C_n$, the decompilation is given by closing off the open parts of the given term. Direct substitution of closures in e would not work, as this will take away all the free variables in e. As a result, the decompiled term would not have any cost due to variable lookups, something which dℓPCF's type system explicitly tracks. So, the decompilation would not remain cost-preserving. So, instead, we decompile it using lambda abstraction and application as described on the left side of Fig. 7.7. Using this closure decompilation, we define decompilation for the full K_{PCF} triples. When the stack is empty, it is just the decompilation of the underlying closure. When stack is non-empty, the closures on the stack are applied one after the other on the closed term obtained via the translation of the closure. This is described on the right side of Fig. 7.7.

$$
\begin{array}{lcl|lcl}
\llparenthesis (e, \square) \rrparenthesis & \triangleq & e & \llparenthesis (e, \rho, \square) \rrparenthesis & \triangleq & \llparenthesis (e, \rho) \rrparenthesis \\
\llparenthesis (e, C_1, \dots, C_n) \rrparenthesis & \triangleq & (\lambda x_1 \dots x_n . e) \ \llparenthesis C_1 \rrparenthesis \ \dots \llparenthesis C_n \rrparenthesis & \llparenthesis (e, \rho, C.\theta) \rrparenthesis & \triangleq & \llparenthesis \llparenthesis (e, \rho) \rrparenthesis \ \llparenthesis C \rrparenthesis, \square, \theta \rrparenthesis
\end{array}
$$

Figure 7.7: Decompilation of closure (left) and K_{PCF} triple (right)

We prove that the decompilation preserves type, cost and semantics of the K_{PCF} triple. For type and cost preservation, we prove Theorem 16. The proof is by induction on the typing derivation of the given K_{PCF} triple. The typing judgment for a K_{PCF}

triple is given by $\Theta; \Delta \vdash_I (e, \rho, \theta) : \tau$ where Θ and Δ represent contexts of index variables and constraints, respectively; and I represents the cost as in dℓPCF's typing judgment.

Theorem 16 (Type and cost preservation for decompilation)**.** $\forall \Theta, \Delta, e, \rho, \theta, \tau.$
$\Theta; \Delta \vdash_I (e, \rho, \theta) : \tau \implies \Theta; \Delta; . \vdash_I (\!|(e, \rho, \theta)|\!) : \tau$

For semantics preservation (Theorem 17) we prove that a K_{PCF} triple and the decompilation are logically related (Fig. 7.8). The $\sim_e$ relation relates a K_{PCF} triple to the translation iff reduction on the K_{PCF} machine can be matched by reduction using dℓPCF's abstract semantics, resulting in related values (which is basically equality under the closing environment).

$$(v_k, \rho, \epsilon) \sim_v v_d \triangleq v_d = v_k\ \rho$$

$$(e_k, \rho, \theta) \sim_e e_d \triangleq \forall v_k, \rho'.(e_k, \rho, \theta) \xrightarrow{*} (v_k, \rho', \epsilon) \implies \exists v_d.e_d \xrightarrow{*} v_d \wedge (v_k, \rho', \epsilon) \sim_v v_d$$

Figure 7.8: Relating K_{PCF} triple with dℓPCF terms

Theorem 17 (Semantics preservation for decompilation)**.** $\forall e_k, \rho, \theta.\ (e_k, \rho, \theta) \sim_e (\!|(e_k, \rho, \theta)|\!)$

7.4 RE-DERIVING dℓPCF'S SOUNDNESS

We compose the decompilation of K_{PCF} triples to dℓPCF terms with the translation of dℓPCF to λ^{amor} terms to obtain a composite translation (represented by $\overline{(\!|.|\!)}$) from K_{PCF} triples to λ^{amor}. We then prove that this translation preserves the meaning of cost annotations wrt to the intensional soundness criteria stated in Theorem 15. The main idea of the proof lies in proving a key invariant (captured formally in Lemma 18[1]) about every K_{PCF} reduction step of the form $D_s \to E_s$: in going from D_s to E_s either 1) the cost of execution of the translation of the decompiled term reduces by one and the size increases by at most $|e_s|$, the size of initial term or 2) the cost remains the same and the size reduces. The intuition behind this result is the following: when the evaluation step involves variable substitution, then the size of the term will increase by the size of the substituted term (which cannot be greater than the size of the initial term, since we start from a closed expression e_s) and the cost will go down by one as dℓPCF only counts variable substitutions. Or, the size of the term will reduce while the cost will remain the same. This will happen in all steps not involving substitution.

1 dℓPCF uses a similar invariant in the "weighted subject reduction" lemma [39]

Lemma 18 (Cost and size lemma). $\forall e_s, D_s, E_s, e_t, v_a, j, v_1.$
$(e_s, \epsilon, \epsilon) \xrightarrow{*} D_s \to E_s \wedge$
D_s is well-typed $\wedge$ E_s is well-typed $\wedge$
$e_t = \overline{(\!|D_s|\!)} \wedge e_t\ () \Downarrow v_a \Downarrow^j v_1 \implies$
$\exists e'_t, v_b, v_2, j'.\ e'_t = \overline{(\!|E_s|\!)} \wedge e'_t\ () \Downarrow v_b \Downarrow^{j'} v_2 \wedge \forall s.\ v_1 \overset{s}{\approx}_{aE} v_2 \wedge$

1. $j' = j - 1 \wedge |E_s| < |D_s| + |e_s|$ or
2. $j' = j \wedge |D_s| > |E_s|$

The proof of this theorem works by induction on the reduction step $D_s \to E_s$ followed by a nested induction on the K_{PCF} stack in D_s. Note that we prove a relation ($\forall s. v_1 \overset{s}{\approx}_{aV} v_2$) between the values obtained by the full execution (pure application followed by forcing) of the translations obtained by applying the composite translation to D_s and E_s. This relation, although not required by the top-level soundness theorem (Theorem 15), is critical for finishing the proof of cost and size lemma inductively. Also this relation cannot be an exact equality, as the decompilation introduces some administrative applications, which in a call-by-name setting, do not coincide with exact equality. For an intuition of this, consider the K_{PCF} application $(e_1\ e_2, C, .) \to (e_1, C, (e_2, C))$. Now, the decompilation of $(e_1\ e_2, C, .)$ will give $(\lambda x.e_1\ e_2)\ (\!|C|\!)$ (call this D_1). Similarly, the decompilation of $(e_1, C, (e_2, C))$ will result in $(\lambda x.e_1)\ (\!|C|\!)\ ((\lambda x.e_2)\ (\!|C|\!))$ (call this D_2). D_1, when executed, will have $e_2[(\!|C|\!)/x]$ applied to $e_1[(\!|C|\!)/x]$ while D_2 will have $((\lambda x.e_2)\ (\!|C|\!))$ applied to $e_1[(\!|C|\!)/x]$. $e_2[(\!|C|\!)/x]$ and $((\lambda x.e_2)\ (\!|C|\!))$ are similar but unequal terms. This kind of inequality (but similarity) also shows up in the translations of the decompiled terms, i.e., at the level of λ^{amor}. So, we develop a similarity relation between λ^{amor} terms. The relation merely captures administrative bureaucracy related to our decompilation. There is nothing surprising and nothing related to costs or potentials. We defer the details of this relation to the technical report [52].

Finally, we re-derive dℓPCF's soundness (Theorem 15) by applying Lemma 18 for every step of the reduction starting from (t, ϵ, ϵ).

8 RELATED WORK FOR λ^{AMOR}

Cost analysis of lazy programs. [20] is a type system for amortized analysis of lazy functional data structures following Okasaki [49]. The type system uses only a type-level monad to represent cost but has no concrete type-level representation for potentials. Amortization is introduced via a term-level construct which is used pay for the cost partially or in full. Amortization in λ^{amor} on the other hand is type-theoretic. Our novel type constructor ($[p]\,\tau$) gives a type-theoretic representation to potentials and builds an affine type theory around it. We demonstrate that doing so yields an extremely expressive and very general approach for doing amortized resource analysis. Also, [20] works in a call-by-need setting where linearity/affineness is not required for soundness, thereby leaving the question of integration between amortization and affineness completely open. λ^{amor} bridges this gap by working in a call-by-name setting (which would be unsound without affineness) and showing that amortization and affineness can indeed work together in a fully general way.

[41] provides a tool based approach for verification of resource bounds for Scala programs with laziness and memoization. Bounds are specified as templates containing holes in them. The tool tries to infer these holes using inductive assume-guarantee reasoning. They experimentally evaluate the efficacy of their inference. This is clearly very different from λ^{amor}: While their focus is on building a tool for resource verification of lazy programs in Scala, we are after a general type theory for verification of amortized bounds.

[37] works with a call-by-push-value language to develop a framework for automatically extracting recurrences which represent the running time of a program in terms of the size of the input. The approach does not handle amortization and hence is very different from λ^{amor}.

Type and effect systems. Several type and effect system have been proposed for doing amortized analysis using the method of potentials. Approaches like [30, 34] can only handle linear resource bounds. Univariate RAML [29] generalizes linear potentials to univariate polynomial potentials. Multivariate RAML [27] further generalizes the potential to multivariate polynomials. AARA approaches like [28, 33] extend RAML with limited support for closures and higher-order functions. For instance, [33] can handle closures where potential is provided with only the last argument. [28] cannot handle Curry-style functions at all. The main limitation of all these approaches is limited or no support for higher-order functions and closures. λ^{amor} gets rid of this limitation. λ^{amor} can handle closures in their full generality with no restriction on which argument(s) have potential. [35] extends the RAML-style of amortized analysis to lazy functional programs. It does not provide a general type-theoretic construct for representing potentials, which λ^{amor} does. Also, the authors acknowledge that their approach only works for monomorphic types. λ^{amor}, on the other hand, scales effortlessly to polymorphic types as well.

The unary fragment of Relcost [16] is another type-and-effect system which establishes lower and upper bounds on the cost of execution. However, it is not an amortized analysis framework and works with only cost but not potentials.

Sized types. The key idea of [9] is to transform the source program into a program with explicit cost passing. Cost is denoted using unary counters. After this transformation, a type system for size analysis is used to actually obtain time complexity guarantees. [19] uses a notion of virtual clock in the type system, which winds down as the program executes (they only count function application as a winding step). Clock counters are associated with the argument and the result types of a function; polymorphism is allowed on such counters for expressivity. To make the bounds precise they use dependent types and build a type system for a language like F_ω. Use of sized-types is common to [9, 19] and λ^{amor} (we use sized types for list). But sized types in λ^{amor} are only required for expressiveness and not for resource analysis per-se. Also, resource analysis in λ^{amor} is performed using the potential capturing modal type and the cost monad, both of which are missing from [9, 19]. Finally [9, 19] do not have a semantic model for types and only have syntactic proofs of soundness. λ^{amor}, on the other hand, provides both a semantic interpretation of types and a semantic proof of soundness.

Resource analysis using program logics. [14] describes an amortized analysis for first-order imperative programs using quantitative Hoare-logic. Essentially, the idea is to track propositions about the potential before and after the program execution as pre- and post-conditions. This helps obtain compositional analysis of resource bounds. The tracking of quantitative bounds in pre- and post-conditions is in principle similar to how bounds are tracked on the typing judgment in RAML. The approach is still limited to first-order programs, while λ^{amor} scales to full higher-order programs.

[15] uses a separation logic based framework extended with time-credits to verify the amortized complexity of the union-find algorithm. Similarly, [46] uses a notion of

time credits and time receipts in the Iris program logic [36] for verification of upper and lower bounds of programs, respectively. Time credits are like potentials and are used to pay for the cost of execution. As a result, they are assumed as preconditions. Time receipts are a dual concept. They specify how many units of resources were consumed by an execution. As a result, they are specified in the postcondition. Cost analysis in these frameworks is only effect based. λ^{amor} on the other hand can work for both effect and coeffect based cost analysis. Also we show that cost analysis in λ^{amor} is relatively complete for PCF. Such a completeness result is not shown in any of the frameworks mentioned in this paragraph.

As a general note, although λ^{amor} has a type-theoretic take on amortization, the approach used in λ^{amor} is not fundamentally limited to type-based analysis only. We believe it should be possible to port these ideas into a program logic framework with all the desirable properties like completeness.

Coeffect based cost analysis. dℓPCF [39] and dℓPCF$_V$ [40] describe affine type systems for cost analysis of PCF programs in a call-by-name and call-by-value setting respectively. The key idea in both these papers is to use light-weight linear dependencies with dependent sub-exponentials to both count the number of occurrences of variables and also to express index dependencies in types. They represent the cost of execution of a term by the number of variable substitutions plus the number of substitution-free execution steps on a specific execution model based on the Krivine machine [38] in the case of dℓPCF and based on the CEK machine [21] in the case of dℓPCF$_V$. Although proved relatively complete wrt an oracle that can solve simple linear inequalities on index variables, both frameworks suffer from the limitation that the cost is expressed only in the typing derivation (but not in the type) making both frameworks non-compositional. λ^{amor} overcomes this limitation by providing a compositional way of doing cost analysis while still retaining relative completeness.

[6] presents Quantitative Type Theory (QTT), which is a dependent type theory with coeffects. QTT and λ^{amor} are very different in their goals. QTT is focused on studying the interaction between dependent types and coeffects, on the other hand, λ^{amor} studies coeffects from the perspective of cost analysis. Technically, QTT only considers non-dependent coeffects, as in $x :_n \tau$. In contrast, λ^{amor} studies coeffects with uniform linear dependencies coming from the dependent sub-exponential of dℓPCF [39], as in $x :_{a<n} \tau$.

Part II

Type theory for information flow control

9 APPLICATION TO INFORMATION FLOW CONTROL

We now show that ideas developed in λ^{amor} are quite general and can be applied to other domains. In particular, we show how to adapt these ideas for Information flow Control (IFC) by developing a very similar type theory for coarse-grained IFC (we explain shortly what we mean by "coarse-grained").

9.1 INFORMATION FLOW CONTROL AND GRANULARITY OF TRACKING

Information Flow Control (IFC) is a technique for tracking flows of information between different elements of a computer system. This is often used to prevent illicit flows wrt a security policy under consideration. In a language-based setting, IFC can be performed dynamically using runtime monitoring [7, 8] or statically using type systems [1, 11–13, 26, 43, 45, 50, 56], for instance. Here we focus only on the latter, i.e., on type-based approaches to IFC.

IFC type systems use confidentiality labels[1] as an abstraction for tracking and prohibiting undesired flows of information. In practice, these labels are used to indicate the level of secrecy associated with the underlying data. For instance, a label "high" could be used to indicate that the underlying data is secret while a label "low" could be used to indicate that the underlying data is public. Such labels are often drawn from a security lattice which is used to indicate a relative ordering amongst them. For instance, we can indicate that "low" labeled data is less confidential than "high" labeled data using a two element lattice, "low" $\sqsubseteq$ "high". Meet and join operations of such a lattice can then be used to combine labels. Join is of particular importance (as will become clear soon). Typing rules are then set up to combine and constrain these labels such that no secret information flows into public labels, either directly (by assignments, for instance) or indirectly (by branching over secret

1 We use the terms confidentiality label and security label synonymously.

data, for instance). This is often stated using a well-known *relational* criteria called *non-interference*.

There are two significant aspects of confidentiality labels that govern how many secure programs a type system can accept[2] (often referred to as the expressiveness of the type system). The first aspect, is the granularity at which labels are specified. For instance, a type system with fine-grained labels might be able to specify the precise variables or inputs on which a value depends. A coarse-grained type system might abstract those labels to specific points of a lattice like "low" and "high" (as explained above). The effect of varying such a granularity of labels on the expressiveness has been studied in prior work [31].

The second aspect, which is important here, is the granularity of labeling (which is different from the granularity of the label itself). It pertains to the extent to which labeling is used on the types. Under this classification, a fine-grained type system is one which labels every program value individually, denoted by label on the type. For instance, FlowCaml [50] is a fine-grained IFC type system for ML. Every type-constructor in FlowCaml is annotated with a confidentiality label. For example, $(A^{H} \times B^{L})^{L}$ is a type of low (public) pairs in FlowCaml whose first component is high (secret) and second component is low (public). H and L are standard labels used to denote high and low data respectively. Additionally, for correctness, a label of a value must represent an upper-bound on the labels of all the values that have flown into it. For instance, adding a low value to a high value must produce a high value. Therefore, a fine-grained type system must track such flows through all the operations in the language. These principles are embodied in several other type systems besides FlowCaml, some instances of those can be found in [12, 26, 45, 56].

Coarse-grained type systems, on the other hand, are very different. They do not assign labels to every individual value. Instead label(s) are associated with entire sub-computations. Values in the scope inherit the label of the sub-computation. This approach is used by type systems like [1, 11, 13, 43].

9.2 BRIEF SYNOPSIS: TYPE THEORY FOR COARSE-GRAINED IFC

In this part of the thesis, we use ideas from λ^{amor} to develop a new type theory (which we refer to as λ^{cg}) for coarse-grained IFC with higher-order state. λ^{cg} uses the modal and the monadic type like λ^{amor}, but it uses them to track confidentiality labels (which is the ghost state in the IFC setting) instead of potential and cost.

2 No IFC type system can be both sound and complete, i.e., accept exactly all secure programs as confidentiality (often specified using non-interference) is undecidable.

Unlike potential, a confidentiality label is not a non-duplicable resource and, hence, affineness has no role in λ^{cg}. This simplifies the type theory to some extent. But, there is an additional source of complexity in λ^{cg}, the relational nature of confidentiality labels. A confidentiality label is a relational ghost state that relates terms in two different executions (this is required to represent non-interference). Despite these glaring differences we show that λ^{cg} makes use of ghost constructs which have very similar typing to what was used in λ^{amor}. Besides demonstrating the generality of the type theoretic constructs, we also show that λ^{cg} is extremely expressive by showing an embedding of λ^{fg} (a variant of FlowCaml) in λ^{cg}. In fact, we also resolve a long standing confusion about the relative expressiveness of the two granularities of IFC, by showing that there is an embedding in the other direction i.e. from λ^{cg} to λ^{fg} as well. Thus, we give a constructive proof of the equi-expressiveness of λ^{fg} and λ^{cg}.

As an independent contribution, we show how to set up semantic, logical relations models of IFC types in both the fine-grained and the coarse-grained settings, over calculi with higher-order state. While models of IFC types have been considered before [1, 26, 42, 54], we do not know of any development that covers higher-order state. Our models are based on step-indexed Kripke logical relations [4]. Also, our models are relational. This is essential since we are interested in proving non-interference [25], the standard confidentiality property which says that public outputs of a program are not influenced by private inputs (i.e., there are no bad flows). This property is naturally defined using two runs. Using our models, we derive proofs of the soundness of both the fine-grained and the coarse-grained type systems.

We also use our logical relations to show that our translations are meaningful. Specifically, we set up cross-language logical relations to prove that our translations preserve program semantics, and from this, we derive a crucial result for each translation: Using the non-interference theorem of the target language as a lemma, we are able to re-prove the non-interference theorem for the source language directly. These results imply that our translations preserve label annotations meaningfully [10]. Like all logical relations models, we expect that our models can be used for other purposes as well.

To summarize, the contributions of this part of the thesis are:

- We describe how to use ideas from λ^{amor} to build a new type-theory (λ^{cg}) for a completely different domain, namely, IFC.

- We present typability- and meaning-preserving translations between a fine-grained and a coarse-grained IFC type system, showing that these type systems are equally expressive.

- And finally we present logical relations models of both type systems, covering both higher-order functions and higher-order state.

10

λ^{CG}: TYPE THEORY FOR COARSE-GRAINED IFC

We develop λ^{cg}, calculus similar to λ^{amor}, but additionally we also add higher-order state. The ghost operations namely, store and release which were only used to manipulate potential in λ^{amor} are of no use in λ^{cg}. So, we replace them with similar ghost operations (namely toLabeled and unlabel) which manipulate security labels.

λ^{cg} operates on a higher-order, eager, call-by-value language with state, but it separates pure expressions from impure (stateful) ones at the level of types like λ^{amor}. This is done by introducing a monad for state, and limiting all state-accessing operations (dereferencing, allocation, assignment) to the monad. We drop refinements, quantification and constraints as these are not needed in λ^{cg}. Values and types are not necessarily labeled individually in λ^{cg}. Instead, there is a confidentiality label on an entire monadic computation. This makes λ^{cg} coarse-grained. The type system of λ^{cg} is a variant of the static fragment of the hybrid IFC type system HLIO [13].[1]

10.1 TYPE SYSTEM

λ^{cg}'s syntax and type system are shown in Fig. 10.1. The types include all the usual types of the simply typed λ-calculus. There are two special types: the monadic type denoted by $\mathbb{C}\ \ell_1\ \ell_2\ \tau$[2] and the modal type denoted by $[\ell]\ \tau$. We assume that all labels are drawn from a given security lattice, denoted as $\mathcal{L}$.

The type $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ is the aforementioned monadic type of computations that may access the heap (expressions of other types cannot access the heap), eventually producing a value of type τ. The first label, ℓ_1, called the *pc-label*, is a lower bound on the write effects of the computation. It plays the role of the "program-counter label", often used in IFC type systems to prevent information leaks via effects [56]. The second label, ℓ_2, called the *taint label*, is an upper bound on the labels of all values

1 Differences between λ^{cg} and HLIO and their consequences are discussed in Chapter 14.
2 λ^{cg}'s monadic type is doubly graded. This is because unlike λ^{amor} which only handles one effect (which is cost) via the monad, λ^{cg}'s monad handle two effects (which are reading and writing of state).

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid \mathsf{fix}\ f(x).e \mid e\ e \mid (e,e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \\ & & & \mathsf{case}\ e, x.e, y.e \mid \mathsf{new}\ e \mid !e \mid e := e \mid () \mid \mathsf{Lb}(e) \mid \mathsf{unlabel}(e) \mid \\ & & & \mathsf{toLabeled}(e) \mid \mathsf{ret}(e) \mid \mathsf{bind}(e, x.e) \\ \text{Types} & \tau & ::= & \mathsf{b} \mid \mathbf{1} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}\ \ell\ \tau \mid [\ell]\,\tau \mid \mathbb{C}\ \ell_1\ \ell_2\ \tau \end{array}$$

Typing judgment: $\boxed{\Gamma \vdash e : \tau}$

(Typing rules for $\mathsf{b}, \tau \to \tau, \tau \times \tau, \tau + \tau$, and $\mathbf{1}$ are standard and included in λ^{cg})

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbb{C}\ \ell_1\ \ell_2\ \tau \\ \Gamma, x : \tau \vdash e_2 : \mathbb{C}\ \ell_3\ \ell_4\ \tau' \qquad \ell \sqsubseteq \ell_1 \qquad \ell \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_4\end{array}}{\Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \mathbb{C}\ \ell\ \ell_4\ \tau'}\ \text{CG-bind}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ret}(e) : \mathbb{C}\ \top\ \bot\ \tau}\ \text{CG-ret} \qquad \frac{\Gamma \vdash e : \tau' \qquad \mathcal{L} \vdash \tau' <: \tau}{\Gamma \vdash e : \tau}\ \text{CG-sub}$$

$$\frac{\Gamma \vdash e : [\ell]\,\tau}{\Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\ \top\ \ell\ \tau}\ \text{CG-unlabel} \qquad \frac{\Gamma \vdash e : [\ell]\,\tau}{\Gamma \vdash \mathsf{new}\ e : \mathbb{C}\ \ell\ \bot\ (\mathsf{ref}\ \ell\ \tau)}\ \text{CG-ref}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \ell'\ \tau}{\Gamma \vdash !e : \mathbb{C}\ \top\ \bot\ ([\ell']\,\tau)}\ \text{CG-deref} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \ell\ \tau \qquad \Gamma \vdash e_2 : [\ell]\,\tau}{\Gamma \vdash e_1 := e_2 : \mathbb{C}\ \ell\ \bot\ \mathbf{1}}\ \text{CG-assign}$$

$$\frac{\Gamma \vdash e : \mathbb{C}\ \ell\ \ell'\ \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell\ \bot\ ([\ell']\,\tau)}\ \text{CG-toLabeled}$$

Subtyping judgment: $\boxed{\mathcal{L} \vdash \tau <: \tau'}$

$$\frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell \sqsubseteq \ell'}{\mathcal{L} \vdash [\ell]\,\tau <: [\ell']\,\tau'}\ \text{CGsub-labeled}$$

$$\frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell_1' \sqsubseteq \ell_1 \qquad \mathcal{L} \vdash \ell_2 \sqsubseteq \ell_2'}{\mathcal{L} \vdash \mathbb{C}\ \ell_1\ \ell_2\ \tau <: \mathbb{C}\ \ell_1'\ \ell_2'\ \tau'}\ \text{CGsub-monad}$$

Figure 10.1: λ^{cg}: language syntax and type system (selected rules)

that the computation has analyzed so far. It is, for this reason, also an *implicit* label on the output type τ of the computation, and on any intermediate values within the computation.

The type $[\ell]\,\tau$ explicitly labels a value (of type τ) with label ℓ. The means labeling can be used *selectively* in λ^{cg}. Also, the reference type $\mathsf{ref}\ \ell\ \tau$ carries an explicit label ℓ in λ^{cg}. Such a reference stores a value of type $[\ell]\,\tau$. Labels on references are necessary to prevent implicit leaks via control dependencies—the type system relates the pc-label to the label of the written value at every assignment.

Typing rules. λ^{cg} uses the typing judgment $\Gamma \vdash e : \tau$. λ^{cg} uses the typing rules of the simply typed λ-calculus for the type constructs b, $\mathbf{1}$, $\times$, $+$ and $\to$. We do not re-iterate these standard rules, and focus here only on the new constructs (Fig. 10.1). The construct $\mathsf{ret}(e)$ is the monadic return that immediately returns e, without any heap access. Consequently, it can be given the type $\mathbb{C}\ \top\ \bot\ \tau$ (rule CG-ret). The pc-label is $\top$ since the computation has no writes, while the taint label is $\bot$ since the computation has not analyzed any value.

The monadic construct $\mathsf{bind}(e_1, x.e_2)$ sequences the computation e_2 after e_1, binding the return value of e_1 to x in e_2. The typing rule for this construct, CG-bind, is important and interesting. The rule says that $\mathsf{bind}(e_1, x.e_2)$ can be given the type $\mathbb{C}\ \ell\ \ell_4\ \tau'$ if $(e_1 : \mathbb{C}\ \ell_1\ \ell_2\ \tau)$, $(e_2 : \mathbb{C}\ \ell_3\ \ell_4\ \tau')$ and four conditions hold. The conditions $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_3$ check that the pc-label of $\mathsf{bind}(e_1, x.e_2)$, which is ℓ, is below the pc-label of e_1 and e_2, which are ℓ_1 and ℓ_3, respectively. This ensures that the write effects of $\mathsf{bind}(e_1, x.e_2)$ are indeed above the pc-label, ℓ. The conditions $\ell_2 \sqsubseteq \ell_3$ and $\ell_2 \sqsubseteq \ell_4$ prevent leaking the output of e_1 via the write effects and the output of e_2, respectively. Observe how these conditions together track labels at the level of entire subcomputations, i.e., coarsely.

Next we describe the typing rules for the ghost constructs of λ^{cg}, namely, $\mathsf{toLabeled}$ and $\mathsf{unlabel}$. $\mathsf{toLabeled}$ is an adaptation of the store construct of λ^{amor}. This construct transforms e of monadic type $\mathbb{C}\ \ell\ \ell'\ \tau$ to the type $\mathbb{C}\ \ell\ \bot\ ([\ell']\,\tau)$, as in the rule CG-toLabeled. This is perfectly safe since the only way to observe the output of a monad is by binding the result, and, that result is explicitly labeled in the final type. The purpose of using this construct is to reduce the taint label of a computation to $\bot$. This allows a subsequent computation, which will *not* analyze the output of the current computation, to not have a raised taint label of ℓ'. Hence, this construct limits the scope of the taint label to a single computation, and prevents overtainting subsequent computations. We make extensive use of this construct in our translation from λ^{fg} to λ^{cg} later. We note that HLIO's original typing rule for $\mathsf{toLabeled}$ is different, and does not always allow reducing the taint to $\bot$. We discuss the consequences of this difference in Chapter 14.

Similarly, unlabel is an adaptation of the release construct of λ^{amor}. This construct captures the effect of unlabeling a value of the labeled ($[\ell]\,\tau$) type, as captured in the rule CG-unlabel. If $e : [\ell]\,\tau$, then the construct $\mathsf{unlabel}(e)$ eliminates this label. This construct has the monadic type $\mathbb{C}\ \top\ \ell\ \tau$. The taint label ℓ indicates that the computation has (internally) analyzed something labeled ℓ (the pc-label is $\top$ since nothing has been written).

Rule CG-deref says that dereferencing (reading) a location of type ref $\ell'\ \tau$ produces a computation of type $\mathbb{C}\ \top\ \bot\ ([\ell']\,\tau)$. The type is monadic because dereferencing accesses the heap. The value the computation returns is explicitly labeled at ℓ'. The pc-label is $\top$ since the computation does not write, while the taint label is $\bot$ since the computation does not analyze the value it reads from the reference. (The taint label will change to ℓ' if the read value is subsequently unlabeled.) Dually, the rule CG-assign allows assigning a value labeled ℓ to a reference labeled ℓ. The result is a computation of type $\mathbb{C}\ \ell\ \bot\ \mathbf{1}$. The pc-label ℓ indicates a write effect at level ℓ.

We briefly comment on subtyping for specific constructs in λ^{cg}. Subtyping of $[\ell]\,\tau$ is co-variant in ℓ, since it is always safe to increase a confidentiality label. Subtyping of $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ is contra-variant in the pc-label ℓ_1 and co-variant in the taint label ℓ_2 since the former is a lower bound while the latter is an upper bound.

We prove soundness for λ^{cg} by showing that every well-typed expression satisfies non-interference. Due to the presence of monadic types, the soundness theorem takes a specific form (shown below), and refers to a *forcing semantics* (described in Fig. 10.2). The forcing relation is defined using the judgment $(H, e) \Downarrow_i^f (H', v)$, which says that starting from a heap H, an expression e of monadic type gets forced to a final heap H' and a value v in i steps (the steps are needed only for the model, which we will describe soon). The forcing relation makes use of a pure evaluation relation represented by $e \Downarrow_i v$. The pure evaluation relation describes evaluation of pure terms and treats monadic terms as suspended values. The pure evaluation relation is standard, described in the technical report [52].

Theorem 19 (Non-interference for λ^{cg})**.** Suppose (1) $\ell_i \not\sqsubseteq \ell$, (2) $x : [\ell_i]\,\tau \vdash e : \mathbb{C}\ \_\ \ell$ bool, and (3) $v_1, v_2 : [\ell_i]\,\tau$. If both $e[v_1/x]$ and $e[v_2/x]$ terminate when forced, then they produce the same value (of type bool).

$$\boxed{\text{Forcing relation: } (H, e) \Downarrow^{f}_{i} (H', v)}$$

$$\frac{e \Downarrow_i v}{(H, \mathsf{ret}(e)) \Downarrow^{f}_{i+1} (H, v)} \text{ cg-ret}$$

$$\frac{e_1 \Downarrow_i v_1 \qquad (H, v_1) \Downarrow^{f}_{j} (H', v_1') \qquad e_2[v_1'/x] \Downarrow_k v_2 \qquad (H', v_2) \Downarrow^{f}_{l} (H'', v_2')}{(H, \mathsf{bind}(e_1, x.e_2)) \Downarrow^{f}_{i+j+k+l+1} (H'', v_2')} \text{ cg-bind}$$

$$\frac{e \Downarrow_i v}{(H, \mathsf{unlabel}(e)) \Downarrow^{f}_{i+1} (H, v)} \text{ cg-unlabel} \qquad \frac{e \Downarrow_i v \qquad (H, v) \Downarrow^{f}_{j} (H', v')}{(H, \mathsf{toLabeled}(e)) \Downarrow^{f}_{i+j+1} (H', v')} \text{ cg-toLabeled}$$

$$\frac{e \Downarrow_i v \qquad a \notin dom(H)}{(H, \mathsf{new}\ (e)) \Downarrow^{f}_{i+1} (H[a \mapsto v], a)} \text{ cg-ref} \qquad \frac{e \Downarrow_i a}{(H, !e) \Downarrow^{f}_{i+1} (H, H(a))} \text{ cg-deref}$$

$$\frac{e_1 \Downarrow_i a \qquad e_2 \Downarrow_j v}{(H, e_1 := e_2) \Downarrow^{f}_{i+j+1} (H[a \mapsto v], ())} \text{ cg-assign}$$

Figure 10.2: Forcing semantics of λ^{cg}

10.2 SEMANTIC MODEL OF λ^{CG}

We now describe our semantic model of λ^{cg}'s types. We use this model to show that the type system is sound (Theorem 19) and later to prove the soundness of our translations. Our semantic model uses step-indexed Kripke logical relations [4] and is related to the semantic model of λ^{amor}. In particular, our model captures all the invariants necessary to prove non-interference.

The central idea behind our model is to interpret each type in two different ways—as a set of values (unary interpretation), and as a set of pairs of values (binary interpretation). The binary interpretation is used to relate *low*-labeled types in the two runs mentioned in the non-interference theorem, while the unary interpretation is used to interpret *high*-labeled types independently in the two runs (since high-labeled values may be unrelated across the two runs). What is high and what is low is determined by the level of the observer (adversary), which is a parameter to our binary interpretation.

Remark. Readers familiar with earlier models of IFC type systems [1, 26, 54] may wonder why we need a unary relation, when prior work did not. The reason is that we handle an effect (mutable state) in our model, which prior work did not. In the absence of effects, the unary model is unnecessary. In the presence of effects, the unary relation captures what is often called the "confinement lemma" in proofs of non-interference — we need to know that while the two runs are executing high branches independently, neither will modify low-labeled locations.

10.2.1 *Unary interpretation*

The unary interpretation of types is shown in Fig. 10.3. The interpretation is actually a Kripke model. It uses *worlds*, written θ, which specify the type for each valid (allocated) location in the heap. For example, $\theta(a) = \text{bool}$ means that location a should hold a boolean. The world can grow as the program executes and allocates more locations. A second important component used in the interpretation is a *step-index*, written m or n [2]. Step-indices are natural numbers, and are merely a technical device to break a non-well-foundedness issue in Kripke models of higher-order state, like this one. Our use of step-indices is standard and readers may ignore them.

The interpretation itself consists of three mutually inductive relations—a *value relation* for types (labeled and unlabeled), written $\lfloor\tau\rfloor_V$; an *expression relation* for labeled types, written $\lfloor\tau\rfloor_E$; and a *heap conformance relation*, written $(n, H) \triangleright \theta$. These relations are well-founded by induction on the step indices n and types. This is the only role of step-indices in our model.

The value relation $\lfloor\tau\rfloor_V$ defines, for each type, which values (at which worlds and step-indices) lie in that type. For base types b, this is straightforward: All syntactic values of type b (written $[\![b]\!]$) lie in $\lfloor b\rfloor_V$ at any world and any step index. For pairs, the relation is the intuitive one: (v_1, v_2) is in $\lfloor\tau_1 \times \tau_2\rfloor_V$ iff v_1 is in $\lfloor\tau_1\rfloor_V$ and v_2 is in $\lfloor\tau_2\rfloor_V$. The function type $\tau_1 \to \tau_2$ contains a value $\text{fix } f(x).e$ at world θ if in any world θ' that extends θ, if v is in $\lfloor\tau_1\rfloor_V$, then $(\text{fix } f(x).e)\ v$ or, equivalently, $e[v/x][\text{fix } f(x).e/f]$, is in the *expression relation* $\lfloor\tau_2\rfloor_E$. We describe this expression relation below. Importantly, we allow for the world θ to be extended to θ' since between the time that the function $\lambda x.e$ was created and the time that the function is applied, new locations could be allocated. The interpretation of $\text{ref}\ \ell\ \tau$ contains all locations a whose type according to the world θ matches $[\ell]\,\tau$.

There are two things to note about the interpretation of $[\ell]\,\tau$: a) the security label ℓ is completely irrelevant in the unary interpretation (in contrast, labels play a significant role in the binary interpretation) and b) as in λ^{amor}, a value of $[\ell]\,\tau$ is a value of type τ, signifying the ghost nature of ℓ at the level of terms.

Finally, we consider $\mathbb{C}\ \ell_1\ \ell_2\ \tau$. The interpretation may look complex, but is relatively straightforward: e is in $\lfloor\mathbb{C}\ \ell_1\ \ell_2\ \tau\rfloor_V$ if for any heap H that conforms to the world θ_e (an extension of θ) such that forcing e starting from H results in a value v' and

$$
\begin{aligned}
\lfloor \mathsf{b} \rfloor_V &\triangleq \{(\theta, m, v) \mid v \in [\![\mathsf{b}]\!]\} \\
\lfloor \mathbf{1} \rfloor_V &\triangleq \{(\theta, m, v \mid v \in [\![\mathbf{1}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V &\triangleq \{(\theta, m, (v_1, v_2)) \mid (\theta, m, v_1) \in \lfloor \tau_1 \rfloor_V \wedge (\theta, m, v_2) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V &\triangleq \{(\theta, m, \mathsf{inl}(\ )v) \mid (\theta, m, v) \in \lfloor \tau_1 \rfloor_V\} \cup \{(\theta, m, \mathsf{inr}(\ )v) \mid (\theta, m, v) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 \to \tau_2 \rfloor_V &\triangleq \{(\theta, m, \mathsf{fix}\ f(x).e) \mid \forall \theta' \sqsupseteq \theta, v, j < m.(\theta', j, v) \in \lfloor \tau_1 \rfloor_V \implies \\
&\qquad (\theta', j, e[v/x][\mathsf{fix}\ f(x).e/f]) \in \lfloor \tau_2 \rfloor_E\} \\
\lfloor \mathsf{ref}\ \ell\ \tau \rfloor_V &\triangleq \{(\theta, m, a) \mid \theta(a) = [\ell]\,\tau\} \\
\lfloor [\ell]\,\tau \rfloor_V &\triangleq \{(\theta, m, v \mid (\theta, m, v) \in \lfloor \tau \rfloor_V\} \\
\lfloor \mathbb{C}\ \ell_1\ \ell_2\ \tau \rfloor_V &\triangleq \{(\theta, m, e) \mid \\
&\qquad \forall k \leqslant m, \theta_e \sqsupseteq \theta, H, j.(k, H) \triangleright \theta_e \wedge (H, v) \Downarrow^f_j (H', v') \wedge j < k \implies \\
&\qquad \exists \theta' \sqsupseteq \theta_e.(k - j, H') \triangleright \theta' \wedge (\theta', k - j, v') \in \lfloor \tau \rfloor_V \wedge \\
&\qquad (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta_e(a) = [\ell']\,\tau' \wedge \ell_1 \sqsubseteq \ell') \wedge \\
&\qquad (\forall a \in dom(\theta') \backslash dom(\theta_e).\theta'(a) \searrow \ell_1)\} \\
\\
\lfloor \tau \rfloor_E &\triangleq \{(\theta, n, e) \mid \forall i < n.e \Downarrow_i v \implies (\theta, n - i, v) \in \lfloor \tau \rfloor_V\} \\
\\
(n, H) \triangleright \theta &\triangleq dom(\theta) \subseteq dom(H) \wedge \forall a \in dom(\theta).(\theta, n - 1, H(a)) \in \lfloor \theta(a) \rfloor_V
\end{aligned}
$$

Figure 10.3: Unary interpretation for λ^{cg}

a heap H', there is some extension θ' of θ_e to which H' conforms and at which v' is in $\lfloor\tau\rfloor_V$. Additionally, all writes performed during the execution (defined as the locations at which H and H' differ) must have labels above the program counter, ℓ_1. In simpler words, the definition simply says that e lies in $\lfloor\mathbb{C}\ \ell_1\ \ell_2\ \tau\rfloor_V$ if the resulting value (obtained by forcing e) is in $\lfloor\tau\rfloor_V$, it preserves heap conformance with worlds and, importantly, the write effects are at labels above ℓ_1. (Readers familiar with proofs of non-interference should note that the condition on write effects is our model's analogue of the so-called "confinement lemma".)

The expression relation $\lfloor\tau\rfloor_E$ is extremely simple. It states that e is in $\lfloor\tau\rfloor_E^{pc}$ if the value obtained by pure reduction of e is in the value interpretation of τ. The heap conformance relation $(n, H) \rhd \theta$ defines when a heap H conforms to a world θ. The relation is simple; it holds when the heap H maps every location to a value in the semantic interpretation of the location's type given by the world θ.

10.2.2 *Binary interpretation*

The binary interpretation of types is shown in Fig. 10.4. This interpretation relates two executions of a program with different inputs. Like the unary interpretation, this interpretation is also a Kripke model. The worlds, written W, are different, though. Each world is a triple $W = (\theta_1, \theta_2, \hat{\beta})$. θ_1 and θ_2 are unary worlds that specify the types of locations allocated in the two executions. Since executions may proceed in sync on the two sides for a while, then diverge in a high-labeled branch, then possibly re-synchronize, and so on, some locations allocated on one side may have analogues on the other side, while other locations may be unique to either side. This is captured by $\hat{\beta}$, which is a *partial bijection* between the domains of θ_1 and θ_2. If $(a_1, a_2) \in \hat{\beta}$, then location a_1 in the first run corresponds to location a_2 in the second run. Any location not in $\hat{\beta}$ has no analogue on the other side.

As before, the interpretation itself consists of three mutually inductive relations—a *value relation* for types (labeled and unlabeled), written $\lceil\tau\rceil_V^{\mathcal{A}}$; an *expression relation* for labeled types, written $\lceil\tau\rceil_E^{\mathcal{A}}$; and a *heap conformance relation*, written $(n, H_1, H_2) \overset{\mathcal{A}}{\rhd} W$. These relations are all parameterized by the level of the observer (adversary), $\mathcal{A}$, which is also an element of $\mathcal{L}$.

The value relation $\lceil\tau\rceil_V^{\mathcal{A}}$ defines, for each type, which pairs of values from the two runs are related by that type (at each world, each step-index and each adversary). At base types, b, only identical values are related. For pairs, the relation is the intuitive one: (v_1, v_2) and (v_1', v_2') are related in $\lceil\tau_1 \times \tau_2\rceil_V^{\mathcal{A}}$ iff v_i and v_i' are related in $\lceil\tau_i\rceil_V^{\mathcal{A}}$ for $i \in \{1, 2\}$. Two values are related at a sum type only if they are both left injections or

$$
\begin{aligned}
\lceil \mathsf{b} \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, v_1, v_2) \mid v_1 = v_2 \wedge \{v_1, v_2\} \in [\![\mathsf{b}]\!]\} \\
\lceil \mathbf{1} \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, (), ()) \mid () \in [\![\mathbf{1}]\!]\} \\
\lceil \tau_1 \times \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, (v_1, v_2), (v_1', v_2')) \mid (W, n, v_1, v_1') \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V} \wedge (W, n, v_2, v_2') \in \lceil \tau_2 \rceil^{\mathcal{A}}_{V}\} \\
\lceil \tau_1 + \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, \mathsf{inl}(\ )v, \mathsf{inl}(\ )v') \mid (W, n, v, v') \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V}\} \cup \\
&\qquad \{(W, n, \mathsf{inr}(\ )v, \mathsf{inr}(\ )v') \mid (W, n, v, v') \in \lceil \tau_2 \rceil^{\mathcal{A}}_{V}\} \\
\lceil \tau_1 \to \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, \mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1, \mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2) \mid \\
&\qquad \forall W' \sqsupseteq W, j < n, v_1, v_2. \\
&\qquad ((W', j, v_1, v_2) \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V} \implies \\
&\qquad (W', j, \mathsf{e}_1[v_1/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1/\mathsf{f}], \mathsf{e}_2[v_2/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2/\mathsf{f}]) \in \lceil \tau_2 \rceil^{\mathcal{A}}_{E}) \wedge \\
&\qquad \forall \theta_l \sqsupseteq W.\theta_1, v_c, j. \\
&\qquad ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_V \implies (\theta_l, j, \mathsf{e}_1[v_c/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1/\mathsf{f}]) \in \lfloor \tau_2 \rfloor_E) \wedge \\
&\qquad \forall \theta_l \sqsupseteq W.\theta_2, v_c, j. \\
&\qquad ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_V \implies (\theta_l, j, \mathsf{e}_2[v_c/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2/\mathsf{f}]) \in \lfloor \tau_2 \rfloor_E)\} \\
\lceil \mathsf{ref}\ \ell\ \tau \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, a_1, a_2) \mid \\
&\qquad (a_1, a_2) \in W.\hat{\beta} \wedge W.\theta_1(a_1) = W.\theta_2(a_2) = [\ell]\,\tau\} \\
\lceil [\ell]\,\tau \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, v_1, v_2) \mid \mathit{ValEq}(\mathcal{A}, W, \ell, n, v_1, v_2, \tau)\} \\
\lceil \mathbb{C}\ \ell_1\ \ell_2\ \tau \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, v_1, v_2) \mid \\
&\qquad \Big(\forall k \leqslant n, W_e \sqsupseteq W, H_1, H_2.(k, H_1, H_2) \rhd W_e \wedge \\
&\qquad \forall v_1', v_2', j.(H_1, v_1) \Downarrow_j^{\mathsf{f}} (H_1', v_1') \wedge (H_2, v_2) \Downarrow^{\mathsf{f}} (H_2', v_2') \wedge j < k \implies \\
&\qquad \exists W' \sqsupseteq W_e.(k - j, H_1', H_2') \rhd W' \wedge \mathit{ValEq}(\mathcal{A}, W', k - j, \ell_2, v_1', v_2', \tau)\Big) \wedge \\
&\qquad \forall l \in \{1, 2\}.\Big(\forall k, \theta_e \sqsupseteq W.\theta_l, H, j.(k, H) \rhd \theta_e \wedge (H, v_l) \Downarrow_j^{\mathsf{f}} (H', v_l') \wedge j < k \implies \\
&\qquad \exists \theta' \sqsupseteq \theta_e.(k - j, H') \rhd \theta' \wedge (\theta', k - j, v_l') \in \lfloor \tau \rfloor_V \wedge \\
&\qquad (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta_e(a) = [\ell']\,\tau' \wedge \ell_1 \sqsubseteq \ell') \wedge \\
&\qquad (\forall a \in \mathit{dom}(\theta') \backslash \mathit{dom}(\theta_e).\theta'(a) \searrow \ell_1)\Big)\}
\end{aligned}
$$

$$
\lceil \tau \rceil^{\mathcal{A}}_{E} \triangleq \{(W, n, \mathsf{e}_1, \mathsf{e}_2) \mid \forall i < n.\mathsf{e}_1 \Downarrow_i v_1 \wedge \mathsf{e}_2 \Downarrow v_2 \implies (W, n - i, v_1, v_2) \in \lceil \tau \rceil^{\mathcal{A}}_{V}\}
$$

$$
\begin{aligned}
(n, H_1, H_2) \overset{\mathcal{A}}{\rhd} W &\triangleq \mathit{dom}(W.\theta_1) \subseteq \mathit{dom}(H_1) \wedge \mathit{dom}(W.\theta_2) \subseteq \mathit{dom}(H_2) \wedge \\
&\qquad (W.\hat{\beta}) \subseteq (\mathit{dom}(W.\theta_1) \times \mathit{dom}(W.\theta_2)) \wedge \\
&\qquad \forall (a_1, a_2) \in (W.\hat{\beta}).(W.\theta_1(a_1) = W.\theta_2(a_2) \wedge \\
&\qquad (W, n - 1, H_1(a_1), H_2(a_2)) \in \lceil W.\theta_1(a_1) \rceil^{\mathcal{A}}_{V}) \wedge \\
&\qquad \forall i \in \{1, 2\}.\forall m.\forall a_i \in \mathit{dom}(W.\theta_i).(W.\theta_i, m, H_i(a_i)) \in \lfloor W.\theta_i(a_i) \rfloor_V
\end{aligned}
$$

Figure 10.4: Binary interpretation for λ^{cg}

both right injections. At the function type $\tau_1 \to \tau_2$, two functions are related if they map values related at the argument type τ_1 to expressions related at the result type τ_2. For technical reasons, we also need both the functions to satisfy the conditions of the *unary* relation. At a reference type $\mathsf{ref}\ \ell\ \tau$, two locations a_1 and a_2 are related at world $W = (\theta_1, \theta_2, \hat{\beta})$ only if they are related by $\hat{\beta}$ (i.e., they are correspondingly allocated locations) and their types as specified by θ_1 and θ_2 are equal to $[\ell]\ \tau$.

The interpretation of the labeled type $[\ell]\ \tau$, $\lceil [\ell]\ \tau \rceil_V^{\mathcal{A}}$, relates values depending on the ordering between ℓ and the adversary $\mathcal{A}$. When $\ell \sqsubseteq \mathcal{A}$, the adversary can see values labeled ℓ, so $\lceil [\ell]\ \tau \rceil_V^{\mathcal{A}}$ contains exactly the values related in $\lceil \tau \rceil_V^{\mathcal{A}}$. When $\ell \not\sqsubseteq \mathcal{A}$, values labeled ℓ are opaque to the adversary (in colloquial terms, they are "high"), so they can be arbitrary. In this case, $\lceil [\ell]\ \tau \rceil_V^{\mathcal{A}}$ is the cross product of the *unary* interpretation of τ with itself. This is the only place in our model where the binary and unary interpretations interact. This is all internalized in the definition of *ValEq*, described in the technical report [52]. Finally we have the monadic type $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ which relates pairs of values (at each world W, each step index n and each adversary $\mathcal{A}$). The definition is similar to that in the unary case: v_1 and v_2 lie in $\lceil \mathbb{C}\ \ell_1\ \ell_2\ \tau \rceil_V^{\mathcal{A}}$ if the values obtained by forcing are related in the value relation $\lceil \tau \rceil_V^{\mathcal{A}}$, and the expressions preserve heap conformance. The expression relation $\lceil \tau \rceil_E^{\mathcal{A}}$ is again extremely simple as in the unary case.

The heap conformance relation $(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W$ defines when a pair of heaps H_1, H_2 conforms to a world $W = (\theta_1, \theta_2, \hat{\beta})$. The relation requires that any pair of locations related by $\hat{\beta}$ have the same types (according to θ_1 and θ_2), and that the values stored in H_1 and H_2 at these locations lie in the binary value relation of that common type.

10.2.3 *Meta-theoretic properties*

The primary meta-theoretic property of a logical relations model like ours is the so-called *fundamental theorem*. This theorem says that any expression syntactically in a type (as established via the type system) also lies in the semantic interpretation (the expression relation) of that type. Here, we have two such theorems—one for the unary interpretation and one for the binary interpretation.

To write these theorems, we define unary and binary interpretations of contexts, $\lfloor \Gamma \rfloor_V$ and $\lceil \Gamma \rceil_V^{\mathcal{A}}$, respectively. These interpretations specify when unary and binary substitutions conform to Γ. A unary substitution δ maps each variable to a value whereas a binary substitution γ maps each variable to two values, one for each run.

$$\lfloor \Gamma \rfloor_V \triangleq \{(\theta, n, \delta) \mid \mathit{dom}(\Gamma) \subseteq \mathit{dom}(\delta) \wedge \forall x \in \mathit{dom}(\Gamma).(\theta, n, \delta(x)) \in \lfloor \Gamma(x) \rfloor_V\}$$

$$\lceil \Gamma \rceil_V^{\mathcal{A}} \triangleq \{(W, n, \gamma) \mid \mathit{dom}(\Gamma) \subseteq \mathit{dom}(\gamma) \wedge \forall x \in \mathit{dom}(\Gamma).(W, n, \pi_1(\gamma(x)), \pi_2(\gamma(x))) \in \lceil \Gamma(x) \rceil_V^{\mathcal{A}}\}$$

Theorem 20 (Unary fundamental theorem)**.** $\forall \Gamma, \theta, e, \tau, \delta, n.$
$\Gamma \vdash e : \tau \wedge$
$(\theta, n, \delta) \in \lfloor \Gamma \rfloor_V \implies$
$(\theta, n, e\ \delta) \in \lfloor \tau \rfloor_E$

Theorem 21 (Binary fundamental theorem)**.** $\forall \Gamma, pc, W, \mathcal{A}, e, \tau, , \gamma, n.$
$\Gamma \vdash e : \tau \wedge$
$(W, n, \gamma) \in \lceil \Gamma \rceil_V^{\mathcal{A}} \implies$
$(W, n, e\ (\gamma{\downarrow}_1), e\ (\gamma{\downarrow}_2)) \in \lceil \tau \rceil_E^{\mathcal{A}}$

The proofs of these theorems proceed by induction on the given derivations of $\Gamma \vdash e : \tau$. The proofs are tedious, but not difficult or surprising. The primary difficulty, as with all logical relations models, is in setting up the model correctly, not in proving the fundamental theorems.

λ^{cg}'s non-interference theorem (Theorem 19) is a simple corollary of these two theorems.

11

λ^{FG}: TYPE THEORY FOR FINE-GRAINED IFC

In order to show that λ^{cg} can express everything that a standard fine-grained IFC type system can, we would like to show an embedding from such a fine-grained type system into λ^{cg}. To achieve that, we first have to introduce such a type system. We call this type system λ^{fg}. λ^{fg} is not new (it is essentially a close variant of the SLam calculus [26] or the exception free fragment of FlowCaml [50]), but its meta-theory is new. Prior presentations of λ^{fg} either relied on syntactic proofs of soundness (as in the case of FlowCaml [50]) or did not handle higher-order state (as in the case of SLAM [26]).

λ^{fg} works on a call-by-value, eager language, which is a simplification of ML. The language has all the usual expected constructs: Functions, pairs, sums, and mutable references (heap locations). The expression $!e$ dereferences the location that e evaluates to, while $e_1 := e_2$ assigns the value that e_2 evaluates to, to the location that e_1 evaluates to. The dynamic semantics of the language are defined by a "big-step" judgment $(H, e) \Downarrow_j (H', v)$, which means that starting from heap H, expression e evaluates to value v, ending with heap H'. This evaluation takes j steps. The number of steps is important only for our logical relations models. The rules for the big-step judgment are standard, hence omitted here.

11.1 TYPE SYSTEM

Unlike λ^{cg}, every type τ in λ^{fg}, including a type nested inside another, carries a security label. The security label represents the confidentiality level of the values the type ascribes. Like in the case of λ^{cg}, here also we assume that all labels are drawn from a given security lattice, denoted as $\mathcal{L}$. It is also convenient to define unlabeled types, denoted A, as shown in Fig. 11.1.

Typing rules. λ^{fg} uses the typing judgment $\Gamma \vdash_{pc} e : \tau$. As usual, Γ maps free variables of e to their types. The judgment means that, given the types for free variables as in Γ,

$$\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \mathsf{fix}\ f(x).e \mid e\ e \mid (e,e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \\
 & & & \mathsf{case}\ e, x.e, y.e) \mid \mathsf{new}\ e \mid !e \mid e := e \\
\text{(Labeled) Types} & \tau & ::= & A^{\ell} \\
\text{Unlabeled types} & A & ::= & b \mid \mathbf{1} \mid \tau \xrightarrow{\ell_e} \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}\ \tau \quad \text{(b denotes a base type)}
\end{array}$$

Typing judgment: $\boxed{\Gamma \vdash_{pc} e : \tau}$

$$\dfrac{}{\Gamma, x : \tau \vdash_{pc} x : \tau}\ \text{FG-var} \qquad \dfrac{\Gamma, f : (\tau_1 \multimap \tau_2)^{\perp}, x : \tau_1 \vdash_{\ell_e} e : \tau_2}{\Gamma \vdash_{pc} \mathsf{fix}\ f(x).e : (\tau_1 \multimap \tau_2)^{\perp}}\ \text{FG-fix}$$

$$\dfrac{\Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell} \qquad \Gamma \vdash_{pc} e_2 : \tau_1 \qquad \mathcal{L} \vdash \tau_2 \searrow \ell \qquad \mathcal{L} \vdash pc \sqcup \ell \sqsubseteq \ell_e}{\Gamma \vdash_{pc} e_1\ e_2 : \tau_2}\ \text{FG-app}$$

$$\dfrac{\Gamma \vdash_{pc} e_1 : \tau_1 \qquad \Gamma \vdash_{pc} e_2 : \tau_2}{\Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\perp}}\ \text{FG-prod} \qquad \dfrac{\Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \qquad \mathcal{L} \vdash \tau_1 \searrow \ell}{\Gamma \vdash_{pc} \mathsf{fst}(e) : \tau_1}\ \text{FG-fst}$$

$$\dfrac{\Gamma \vdash_{pc} e : \tau_1}{\Gamma \vdash_{pc} \mathsf{inl}(e) : (\tau_1 + \tau_2)^{\perp}}\ \text{FG-inl}$$

$$\dfrac{\Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \qquad \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash_{pc \sqcup \ell} e_2 : \tau \qquad \mathcal{L} \vdash \tau \searrow \ell}{\Gamma \vdash_{pc} \mathsf{case}\ e, x.e_1, y.e_2 : \tau}\ \text{FG-case}$$

$$\dfrac{\Gamma \vdash_{pc'} e : \tau' \qquad \mathcal{L} \vdash pc \sqsubseteq pc' \qquad \mathcal{L} \vdash \tau' <: \tau}{\Gamma \vdash_{pc} e : \tau}\ \text{FG-sub} \qquad \dfrac{\Gamma \vdash_{pc} e : \tau \qquad \mathcal{L} \vdash \tau \searrow pc}{\Gamma \vdash_{pc} \mathsf{new}\ e : (\mathsf{ref}\ \tau)^{\perp}}\ \text{FG-ref}$$

$$\dfrac{\Gamma \vdash_{pc} e : (\mathsf{ref}\ \tau)^{\ell} \qquad \mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \tau' \searrow \ell}{\Gamma \vdash_{pc} !e : \tau'}\ \text{FG-deref}$$

$$\dfrac{\Gamma \vdash_{pc} e_1 : (\mathsf{ref}\ \tau)^{\ell} \qquad \Gamma \vdash_{pc} e_2 : \tau \qquad \mathcal{L} \vdash \tau \searrow (pc \sqcup \ell)}{\Gamma \vdash_{pc} e_1 := e_2 : \mathbf{1}}\ \text{FG-assign} \qquad \dfrac{}{\Gamma \vdash_{pc} () : \mathbf{1}^{\perp}}\ \text{FG-unitI}$$

Figure 11.1: λ^{fg}'s language syntax and type system (selected rules)

e has type τ. The annotation *pc* is also a security label referred to as the "program counter" label. This label is a *lower bound* on the write effects of e. The type system ensures that any reference that e writes to is at a level *pc* or higher. This is necessary to prevent information leaks via the heap. A similar annotation, ℓ_e, appears in the function type $\tau_1 \xrightarrow{\ell_e} \tau_2$. Here, ℓ_e is a lower bound on the write effects of the body of the function.

λ^{fg}'s typing rules are shown in Fig. 11.1. We describe some of the important rules. In the rule for case analysis (FG-case), if the case analyzed expression e has label ℓ, then both the case branches are typed in a *pc* that is *joined* with ℓ. This ensures that the branches do not have write effects below ℓ, which is necessary for IFC since the execution of the branches is control dependent on a value (the case condition) of confidentiality ℓ. Similarly, the type of the result of the case branches, τ, must have a top-level label at least ℓ. This is indicated by the premise $\tau \searrow \ell$ and prevents implicit leaks via the result. The relation $\tau \searrow \ell$, read "τ protected at ℓ" [1], means that if $\tau = \mathsf{A}^{\ell'}$, then $\ell \sqsubseteq \ell'$.

The rule for function application (FG-app) follows similar principles. If the function expression e_1 being applied has type $(\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell}$, then ℓ must be below ℓ_e and the result τ_2 must be protected at ℓ to prevent implicit leaks arising from the identity of the function that e_1 evaluates to.

In the rule for assignment (FG-assign), if the expression e_1 being assigned has type $(\mathsf{ref}\ \tau)^{\ell}$, then τ must be protected at $pc \sqcup \ell$ to ensure that the written value (of type τ) has a label above *pc* and ℓ. The former enforces the meaning of the judgment's *pc*, while the latter protects the identity of the reference that e_1 evaluates to.

All introduction rules such as those for functions, pairs and sums produce expressions labeled $\perp$. This label can be weakened (increased) freely with the subtyping rule FGsub-label. The other subtyping rules are the expected ones, e.g., subtyping for unlabeled function types $\tau_1 \xrightarrow{\ell_e} \tau_2$ is co-variant in τ_2 and contra-variant in τ_1 and ℓ_e (contra-variance in ℓ_e is required since ℓ_e is a *lower* bound on an effect). Subtyping for $\mathsf{ref}\ \tau$ is invariant in τ, as usual. Selected subtyping rules are described in Fig. 11.2.

The main meta-theorem of interest to us is soundness. This theorem says that every well-typed expression is non-interferent, i.e., the result of running an expression of a type labeled low is independent of substitutions used for the high-labeled free variables. This theorem is formalized below.

Theorem 22 (Non-interference for λ^{fg})**.** Suppose (1) $\ell_i \not\sqsubseteq \ell$, (2) $x : \mathsf{A}^{\ell_i} \vdash_{pc} e : \mathsf{bool}^{\ell}$, and (3) $v_1, v_2 : \mathsf{A}^{\ell_i}$. If both $e[v_1/x]$ and $e[v_2/x]$ terminate, then they produce the same value (of type bool).

Subtyping judgments: $\boxed{\mathcal{L} \vdash A <: A'}$ and $\boxed{\mathcal{L} \vdash \tau <: \tau'}$

$$\frac{\mathcal{L} \vdash \ell \sqsubseteq \ell' \qquad \mathcal{L} \vdash A <: A'}{\mathcal{L} \vdash A^{\ell} <: A'^{\ell'}}\ \text{FGsub-label} \qquad \frac{}{\mathcal{L} \vdash \mathsf{ref}\ \tau <: \mathsf{ref}\ \tau}\ \text{FGsub-ref}$$

$$\frac{\mathcal{L} \vdash \tau_1' <: \tau_1 \qquad \mathcal{L} \vdash \tau_2 <: \tau_2' \qquad \mathcal{L} \vdash \ell_e' \sqsubseteq \ell_e}{\mathcal{L} \vdash \tau_1 \xrightarrow{\ell_e} \tau_2 <: \tau_1' \xrightarrow{\ell_e'} \tau_2'}\ \text{FGsub-arrow}$$

Figure 11.2: λ^{fg}'s subtyping relation (selected rules)

By definition, non-interference, as stated above is a relational (binary) property, i.e., it relates two runs of a program. Next, we show how to build a semantic model of λ^{fg}'s types that allows proving this property.

11.2 SEMANTIC MODEL OF λ^{FG}

We now describe our semantic model of λ^{fg}'s types. As in the λ^{cg} case, we set up a unary interpretation and a binary interpretation for the types.

11.2.1 *Unary interpretation*

As before, the value relation $\lfloor \tau \rfloor_V$ in Fig. 11.3 defines, for each type, which values (at which worlds and step-indices) lie in that type. Interpretation for base, pair and sum type is exactly as we saw in the λ^{cg} case. The interpretation of function type $\tau_1 \xrightarrow{\ell_e} \tau_2$ changes because of the effect label ℓ_e on the function type. In the interpretation this is reflected by indexing the expression relation with the extra *pc* label. $\mathsf{fix}\ f(x).e$ is in the interpretation of $\tau_1 \xrightarrow{\ell_e} \tau_2$ at world θ if in any world θ' that extends θ, if v is in $\lfloor \tau_1 \rfloor_V$, then $e[v/x]$, is in the *expression relation* $\lfloor \tau_2 \rfloor_E^{\ell_e}$. The type $\mathsf{ref}\ \tau$ contains all locations a whose type according to the world θ matches τ. Unlike λ^{cg}, we do not have an explicit label on the reference. This is because the value contained in the reference is implicitly labeled. As before, security labels play no role in the unary interpretation, i.e. $\lfloor A^{\ell} \rfloor_V = \lfloor A \rfloor_V$.

There is no monadic type in λ^{fg}. As a result, all the complexity pertaining to preventing leaks via state effects goes into the expression relation. The expression relation $\lfloor \tau \rfloor_E^{pc}$ basically states that an expression e is in $\lfloor \tau \rfloor_E^{pc}$ if for any heap H that conforms to the world θ such that running e starting from H results in a value v' and

$$
\begin{array}{lcl}
\lfloor \mathsf{b} \rfloor_V & \triangleq & \{(\theta, m, v) \mid v \in [\![\mathsf{b}]\!]\} \\
\lfloor \mathbf{1} \rfloor_V & \triangleq & \{(\theta, m, v) \mid v \in [\![\mathbf{1}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V & \triangleq & \{(\theta, m, (v_1, v_2)) \mid (\theta, m, v_1) \in \lfloor \tau_1 \rfloor_V \wedge (\theta, m, v_2) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V & \triangleq & \{(\theta, m, \mathsf{inl}(\ )v) \mid (\theta, m, v) \in \lfloor \tau_1 \rfloor_V\} \cup \{(\theta, m, \mathsf{inr}(\ )v) \mid (\theta, m, v) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 \xrightarrow{\ell_e} \tau_2 \rfloor_V & \triangleq & \{(\theta, m, \mathsf{fix}\ f(x).e) \mid \forall \theta'.\theta \sqsubseteq \theta' \wedge \forall j < m. \forall v.(\theta', j, v) \in \lfloor \tau_1 \rfloor_V \implies \\
& & (\theta', j, e[v/x][\mathsf{fix}\ f(x).e/f]) \in \lfloor \tau_2 \rfloor_E^{\ell_e}\} \\
\lfloor \mathsf{ref}\ \tau \rfloor_V & \triangleq & \{(\theta, m, a) \mid \theta(a) = \tau\} \\
\\
\lfloor \mathsf{A}^{\ell} \rfloor_V & \triangleq & \lfloor \mathsf{A} \rfloor_V \\
\\
\lfloor \tau \rfloor_E^{pc} & \triangleq & \{(\theta, n, e) \mid \forall H.(n, H) \rhd \theta \wedge \forall j < n.(H, e) \Downarrow_j (H', v') \implies \\
& & \quad \exists \theta'.\theta \sqsubseteq \theta' \wedge (n - j, H') \rhd \theta' \wedge (\theta', n - j, v') \in \lfloor \tau \rfloor_V \wedge \\
& & \quad\quad (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta(a) = \mathsf{A}^{\ell'} \wedge pc \sqsubseteq \ell') \wedge \\
& & \quad\quad (\forall a \in dom(\theta') \backslash dom(\theta).\theta'(a) \searrow pc)\} \\
\\
(n, H) \rhd \theta & \triangleq & dom(\theta) \subseteq dom(H) \wedge \forall a \in dom(\theta).(\theta, n - 1, H(a)) \in \lfloor \theta(a) \rfloor_V
\end{array}
$$

Figure 11.3: Unary value, expression, and heap conformance relations for λ^{fg}

a heap H', there is a some extension θ' of θ to which H' conforms and at which v' is in $\lfloor\tau\rfloor_V$. Additionally, all writes performed during the execution (defined as the locations at which H and H' differ) must have labels above the program counter, *pc*. In other words, the definition simply says that e lies in $\lfloor\tau\rfloor_E^{pc}$ if the resulting value (obtained by executing e) is in $\lfloor\tau\rfloor_V$, it preserves heap conformance with worlds and, importantly, the write effects (produced via the execution of e) are at labels above *pc*.

The heap conformance relation $(n, H) \triangleright \theta$ defines when a heap H conforms to a world θ. The formal definition of $(n, H) \triangleright \theta$ is similar to what we saw in the λ^{cg} case.

11.2.2 *Binary interpretation*

The binary interpretation of types is shown in Fig. 11.4. This interpretation relates two executions of a program with different inputs.

The value relation $\lceil\tau\rceil_V^{\mathcal{A}}$ defines, for each type, which pairs of values from the two runs are related by that type (at each world, each step-index and each adversary). Again, interpretation at base, pair and sum type is similar to λ^{cg}. Interpretation for the function type, $\tau_1 \xrightarrow{\ell_e} \tau_2$, also follows the same intuition of mapping related input values to related expression with substitutions. The conditions of the *unary* relation are kept again for technical reasons as in the λ^{cg} case. At a reference type $\mathsf{ref}\ \tau$, two locations a_1 and a_2 are related at world $W = (\theta_1, \theta_2, \hat{\beta})$ only if they are related by $\hat{\beta}$ (i.e., they are correspondingly allocated locations) and their types as specified by θ_1 and θ_2 are equal to τ. For a labeled type A^{ℓ}, $\lceil\mathsf{A}^{\ell}\rceil_V^{\mathcal{A}}$ relates values depending on the ordering between ℓ and the adversary $\mathcal{A}$ as for the type $[\ell]\,\tau$ in λ^{cg}.

Expressions are related via the $\lceil\tau\rceil_E^{\mathcal{A}}$ relation. The definition is similar to the definition of the monadic type in the λ^{cg} case. The heap conformance relation $(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W$ is defined in a way similar to λ^{cg}.

11.2.3 *Meta-theoretic properties*

As before, the meta-theoretic properties we prove about the model are the unary and binary fundamental theorems. To state these we define unary and binary interpretations of contexts, $\lfloor\Gamma\rfloor_V$ and $\lceil\Gamma\rceil_V^{\mathcal{A}}$.

$$
\begin{aligned}
\lceil \mathsf{b} \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, v_1, v_2) \mid v_1 = v_2 \wedge \{v_1, v_2\} \in [\![\mathsf{b}]\!]\} \\
\lceil \mathbf{1} \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, (), ()) \mid () \in [\![\mathbf{1}]\!]\} \\
\lceil \tau_1 \times \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, (v_1, v_2), (v_1', v_2')) \mid (W, n, v_1, v_1') \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V} \wedge (W, n, v_2, v_2') \in \lceil \tau_2 \rceil^{\mathcal{A}}_{V}\} \\
\lceil \tau_1 + \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, \mathsf{inl}(\ )v, \mathsf{inl}(\ )v') \mid (W, n, v, v') \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V}\} \cup \\
& \{(W, n, \mathsf{inr}(\ )v, \mathsf{inr}(\ )v') \mid (W, n, v, v') \in \lceil \tau_2 \rceil^{\mathcal{A}}_{V}\} \\
\lceil \tau_1 \xrightarrow{\ell_e} \tau_2 \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, \mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1, \mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2) \mid \\
& \forall W' \sqsupseteq W, j < n, v_1, v_2.((W', j, v_1, v_2) \in \lceil \tau_1 \rceil^{\mathcal{A}}_{V} \implies \\
& (W', j, \mathsf{e}_1[v_1/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1/\mathsf{f}], \mathsf{e}_2[v_2/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2/\mathsf{f}]) \in \lceil \tau_2 \rceil^{\mathcal{A}}_{E}) \wedge \\
& \forall \theta_l \sqsupseteq W.\theta_1, j, v_c. \\
& ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_{V} \implies (\theta_l, j, \mathsf{e}_1[v_c/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_1/\mathsf{f}]) \in \lfloor \tau_2 \rfloor^{\ell_e}_{E}) \wedge \\
& \forall \theta_l \sqsupseteq W.\theta_2, j, v_c. \\
& ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_{V} \implies (\theta_l, j, \mathsf{e}_2[v_c/\mathsf{x}][\mathsf{fix}\ \mathsf{f}(\mathsf{x}).\mathsf{e}_2/\mathsf{f}]) \in \lfloor \tau_2 \rfloor^{\ell_e}_{E})\} \\
\lceil \mathsf{ref}\ \tau \rceil^{\mathcal{A}}_{V} &\triangleq \{(W, n, a_1, a_2) \mid (a_1, a_2) \in W.\hat{\beta} \wedge W.\theta_1(a_1) = W.\theta_2(a_2) = \tau\}
\end{aligned}
$$

$$
\lceil \mathsf{A}^{\ell} \rceil^{\mathcal{A}}_{V} \triangleq \begin{cases} \{(W, n, v_1, v_2) \mid (W, n, v_1, v_2) \in \lceil \mathsf{A} \rceil^{\mathcal{A}}_{V}\} & \ell \sqsubseteq \mathcal{A} \\ \{(W, n, v_1, v_2) \mid \forall i \in \{1, 2\}. \forall m. (W.\theta_i, m, v_i) \in \lfloor \mathsf{A} \rfloor_{V}\} & \ell \not\sqsubseteq \mathcal{A} \end{cases}
$$

$$
\begin{aligned}
\lceil \tau \rceil^{\mathcal{A}}_{E} &\triangleq \{(W, n, \mathsf{e}_1, \mathsf{e}_2) \mid \forall H_1, H_2, j < n. \\
& (n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W \wedge (H_1, \mathsf{e}_1) \Downarrow_j (H_1', v_1') \wedge (H_2, \mathsf{e}_2) \Downarrow (H_2', v_2') \implies \\
& \exists W' \sqsupseteq W.(n - j, H_1', H_2') \overset{\mathcal{A}}{\triangleright} W' \wedge (W', n - j, v_1', v_2') \in \lceil \tau \rceil^{\mathcal{A}}_{V}\}
\end{aligned}
$$

$$
\begin{aligned}
(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W &\triangleq \mathit{dom}(W.\theta_1) \subseteq \mathit{dom}(H_1) \wedge \mathit{dom}(W.\theta_2) \subseteq \mathit{dom}(H_2) \wedge \\
& (W.\hat{\beta}) \subseteq (\mathit{dom}(W.\theta_1) \times \mathit{dom}(W.\theta_2)) \wedge \\
& \forall (a_1, a_2) \in (W.\hat{\beta}).(W.\theta_1(a_1) = W.\theta_2(a_2) \wedge \\
& (W, n - 1, H_1(a_1), H_2(a_2)) \in \lceil W.\theta_1(a_1) \rceil^{\mathcal{A}}_{V}) \wedge \\
& \forall i \in \{1, 2\}. \forall m. \forall a_i \in \mathit{dom}(W.\theta_i).(W.\theta_i, m, H_i(a_i)) \in \lfloor W.\theta_i(a_i) \rfloor_{V}
\end{aligned}
$$

Figure 11.4: Binary value, expression and heap conformance relations for λ^{fg}

$$\begin{aligned}
\lfloor \Gamma \rfloor_{V} &\triangleq \{(\theta, n, \delta) \mid \mathit{dom}(\Gamma) \subseteq \mathit{dom}(\delta) \wedge \forall x \in \mathit{dom}(\Gamma).\\
&\qquad (\theta, n, \delta(x)) \in \lfloor \Gamma(x) \rfloor_{V}\}\\
\lceil \Gamma \rceil_{V}^{\mathcal{A}} &\triangleq \{(W, n, \gamma) \mid \mathit{dom}(\Gamma) \subseteq \mathit{dom}(\gamma) \wedge \forall x \in \mathit{dom}(\Gamma).\\
&\qquad (W, n, \pi_1(\gamma(x)), \pi_2(\gamma(x))) \in \lceil \Gamma(x) \rceil_{V}^{\mathcal{A}}\}
\end{aligned}$$

The respective fundamental theorems are as follows.

Theorem 23 (Unary fundamental theorem)**.** If $\Gamma \vdash_{pc} e : \tau$ and $(\theta, n, \delta) \in \lfloor \Gamma \rfloor_{V}$, then $(\theta, n, e\ \delta) \in \lfloor \tau \rfloor_{E}^{pc}$.

Theorem 24 (Binary fundamental theorem)**.** If $\Gamma \vdash_{pc} e : \tau$ and $(W, n, \gamma) \in \lceil \Gamma \rceil_{V}^{\mathcal{A}}$, then $(W, n, e\ (\gamma{\downarrow_1}), e\ (\gamma{\downarrow_2})) \in \lceil \tau \rceil_{E}^{\mathcal{A}}$, where $\gamma{\downarrow_1}$ and $\gamma{\downarrow_2}$ are the left and right projections of γ.

The proofs of these theorems proceed by induction on the given derivations of $\Gamma \vdash_{pc} e : \tau$. The proofs are tedious, but not difficult or surprising. The primary difficulty, as with all logical relations models, is in setting up the model correctly, not in proving the fundamental theorems.

λ^{fg}'s non-interference theorem (Theorem 22) is a simple corollary of these two theorems.

12

TRANSLATING λ^{FG} TO λ^{CG}

Our goal in translating λ^{fg} to λ^{cg} is to show how a fine-grained IFC type system can be simulated in a coarse-grained one. This shows that λ^{cg} which follows the principles of λ^{amor} actually yields a very expressive type theory for IFC. We describe the translation below, followed by formal properties of the translation. As a convention, we use the subscript or superscript s to indicate source (λ^{fg}) elements, and t to indicate target (λ^{cg}) elements. Thus, e_s denotes a source expression, whereas e_t denotes a target expression.

12.1 TYPE TRANSLATION

The key idea of our translation is to map a source expression e_s satisfying $\vdash_{pc} e_s : \tau$ to a monadic target expression e_t satisfying $\vdash e_t : \mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$. The pc used to type the source expression is mapped as-is to the pc-label of the monadic computation. The type of the source expression, τ, is translated by the function $(\!|\cdot|\!)$ that is described below. However—and this is the crucial bit—the taint label on the translated monadic computation is $\bot$. To get this $\bot$ taint we use the toLabeled construct judiciously. Not setting the taint to $\bot$ can cause a taint explosion in translated expressions, which would make it impossible to simulate the fine-grained dependence tracking of λ^{fg}.

The function $(\!|\cdot|\!)$ defines how the types of source values are translated. This function is defined by induction on labeled and unlabeled source types.

The translation should be self-explanatory. The only nontrivial case is the translation of the function type $\tau_1 \xrightarrow{\ell_e} \tau_2$. A source function of this type is mapped to a target function that takes an argument of type $(\!|\tau_1|\!)$ and returns a monadic computation (the translation of the body of the source function) that has pc-label ℓ_e and eventually returns a value of type $(\!|\tau_2|\!)$.

$$
\begin{array}{lcll}
(\!|\mathsf{b}|\!) & = & \mathsf{b} & \\
(\!|\mathbf{1}|\!) & = & \mathbf{1} & \\
(\!|\tau_1 \xrightarrow{\ell_e} \tau_2|\!) & = & (\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!) & \\
(\!|\tau_1 \times \tau_2|\!) & = & (\!|\tau_1|\!) \times (\!|\tau_2|\!) & \\
(\!|\tau_1 + \tau_2|\!) & = & (\!|\tau_1|\!) + (\!|\tau_2|\!) & \\
(\!|\mathsf{ref}\ \tau|\!) & = & \mathsf{ref}\ \ell\ (\!|\mathsf{A}|\!) & \text{when } \tau = \mathsf{A}^{\ell} \\
(\!|\mathsf{A}^{\ell}|\!) & = & [\ell]\ (\!|\mathsf{A}|\!) &
\end{array}
$$

Figure 12.1: Type translation function for λ^{fg} to λ^{cg} translation

12.2 TYPE-DIRECTED TERM TRANSLATION

Given this translation of types, we next define a type derivation-directed translation of expressions. This translation is formalized by the judgment $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$. The judgment means that translating the source expression e_s, which has the typing derivation $\Gamma \vdash_{pc} e_s : \tau$, yields the target expression e_t. This judgment is *functional*: For each type derivation $\Gamma \vdash_{pc} e_s : \tau$, it yields exactly one e_t. It is also easily implemented by induction on typing derivations. The rules for the judgment are shown in Fig. 12.2. The thing to keep in mind while reading the rules is that e_t should have the type $\mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$.

We illustrate how the translation works using one rule, FC-app. In this rule, we know inductively that the translation of e_1, i.e., e_{c1}, has type $\mathbb{C}\ pc\ \bot\ (\!|(\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell}|\!)$, which is equal to $\mathbb{C}\ pc\ \bot\ ([\ell]\ ((\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)))$. The translation of e_2, i.e., e_{c2} has type $\mathbb{C}\ pc\ \bot\ (\!|\tau_1|\!)$. We wish to construct something of type $\mathbb{C}\ pc\ \bot\ (\!|\tau_2|\!)$.

To do this, we bind e_{c1} to the variable a, which has the type $[\ell]\ ((\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!))$. Similarly, we bind e_{c2} to the variable b, which has the type $(\!|\tau_1|\!)$. Next, we unlabel a and bind the result to variable c, which has the type $(\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)$. However, due to the unlabeling, the *taint label on whatever computation we sequence after this bind must be at least* ℓ. Next, we apply b to c, which yields a value of type $\mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)$. Via subtyping, using the assumption $pc \sqsubseteq \ell_e$, we can weaken this to $\mathbb{C}\ pc\ \ell\ (\!|\tau_2|\!)$. This satisfies the constraint that the taint label be at least ℓ and is *almost* what we need, except that we need the taint $\bot$ in place of ℓ.

To reduce the taint back to $\bot$, we use the *defined* λ^{cg} function `coerce_taint`, which has the type $\mathbb{C}\ pc\ \ell\ \tau \to \mathbb{C}\ pc\ \bot\ \tau$, when τ has the form $[\ell']\ \tau'$ with $\ell \sqsubseteq \ell'$. This last

$$\frac{}{\Gamma, x : \tau \vdash_{pc} x : \tau \rightsquigarrow \mathsf{ret}\ x}\ \text{FC-var}$$

$$\frac{\Gamma, f : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\perp}, x : \tau_1 \vdash_{\ell_e} e : \tau_2 \rightsquigarrow e_{c1}}{\Gamma \vdash_{pc} \mathsf{fix}\ f(x).e : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\perp} \rightsquigarrow \mathsf{toLabeled}(\mathsf{ret}(\mathsf{fix}\ f(x).\mathsf{bind}(\mathsf{toLabeled}(\mathsf{ret} f), f'.e_c[f/f'])))}\ \text{FC-fix}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell} \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau_1 \rightsquigarrow e_{c2} \qquad \mathcal{L} \vdash \ell \sqcup pc \sqsubseteq \ell_e \qquad \mathcal{L} \vdash \tau_2 \searrow \ell}{\Gamma \vdash_{pc} e_1\ e_2 : \tau_2 \rightsquigarrow \mathtt{coerce\_taint}(\mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{bind}(\mathsf{unlabel}\ a, c.(c\ b)))))}\ \text{FC-app}$$

$$\frac{\Gamma \vdash_{pc} e_1 : \tau_1 \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau_2 \rightsquigarrow e_{c2}}{\Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\perp} \rightsquigarrow \mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{toLabeled}(\mathsf{ret}(a, b))))}\ \text{FC-prod}$$

$$\frac{\Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \rightsquigarrow e_c \qquad \mathcal{L} \vdash \tau_1 \searrow \ell}{\Gamma \vdash_{pc} \mathsf{fst}(()e) : \tau_1 \rightsquigarrow \mathtt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.\mathsf{ret}(\mathsf{fst}(()b)))))}\ \text{FC-fst}$$

$$\frac{\Gamma \vdash_{pc} e : \tau_1 \rightsquigarrow e_c}{\Gamma \vdash_{pc} \mathsf{inl}(()e) : (\tau_1 + \tau_2)^{\perp} \rightsquigarrow \mathsf{bind}(e_c, a.\mathsf{toLabeled}(\mathsf{ret}\ (\mathsf{inl}(()a))))}\ \text{FC-inl}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \rightsquigarrow e_c \\ \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \rightsquigarrow e_{c1} \qquad \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_2 : \tau \rightsquigarrow e_{c2} \qquad \mathcal{L} \vdash \tau \searrow \ell\end{array}}{\begin{array}{c}\Gamma \vdash_{pc} \mathsf{case}\ (, x.e, y., x.e_1, y.e_2) : \tau \rightsquigarrow \\ \mathtt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.\mathsf{case}\ (, x.b, y., x.e_{c1}, y.e_{c2}))))\end{array}}\ \text{FC-case}$$

$$\frac{\Gamma \vdash_{pc} e : (\mathsf{ref}\ \tau)^{\ell} \rightsquigarrow e_c \qquad \mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \tau' \searrow \ell}{\Gamma \vdash_{pc} !e : \tau \rightsquigarrow \mathtt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.!b)))}\ \text{FC-deref}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\mathsf{ref}\ \tau)^{\ell} \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau \rightsquigarrow e_{c2} \qquad \tau \searrow (pc \sqcup \ell)}{\begin{array}{c}\Gamma \vdash_{pc} e_1 := e_2 : \mathbf{1} \rightsquigarrow \\ \mathsf{bind}(\mathsf{toLabeled}(\mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{bind}(\mathsf{unlabel}\ a, c.c := b)))), d.\mathsf{ret}())\end{array}}\ \text{FC-assign}$$

where, $$\boxed{\begin{array}{ll}\mathtt{coerce\_taint}\ :\ \mathbb{C}\ pc\ \ell\ \tau \to \mathbb{C}\ pc\ \perp\ \tau & \text{when } \tau = [\ell']\,\tau' \text{ and } \ell \sqsubseteq \ell' \\ \mathtt{coerce\_taint} \triangleq \lambda x.\mathsf{toLabeled}(\mathsf{bind}(x, y.\mathsf{unlabel}\ y)) & \end{array}}$$

Figure 12.2: Expression translation λ^{fg} to λ^{cg} (selected rules only)

constraint is satisfied here since we know that $\tau_2 \searrow \ell$. The function `coerce_taint` uses toLabeled internally and is defined in the figure.

This pattern of using `coerce_taint`, which internally contains toLabeled, to restrict the taint to $\bot$ is used to translate all elimination forms (application, projection, case, etc.). Overall, our translation uses toLabeled judiciously to prevent taint from exploding in the translated expressions.

Remark. Readers familiar with monads may note that our translation from λ^{fg} to λ^{cg} is based on the standard interpretation of the call-by-value λ-calculus in the computational λ-calculus [44]. Our translation additionally accounts for the pc and security labels, but is structurally the same.

12.3 PROPERTIES OF THE TRANSLATION

Our translation preserves typing by construction. This is formalized in the following theorem. The context translation $(\!|\Gamma|\!)$ is defined pointwise on all types in Γ.

Theorem 25 (Typing preservation)**.** If $\Gamma \vdash_{pc} e_s : \tau$ in λ^{fg}, then there is a unique e_t such that $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$ and that e_t satisfies $(\!|\Gamma|\!) \vdash e_t : \mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$ in λ^{cg}.

An immediate corollary of this theorem is that well-typed source programs translate to non-interfering target programs (since target typing implies non-interference in the target).

Next, we show that our translation preserves the meaning of programs, i.e., it is semantically "correct". For this, we define a cross-language logical relation, which relates source values (expressions) to target values (expressions) at each source type. This relation has three key properties: (A) A source expression and the translation (of the source) are always in the relation (Theorem 26), (B) Related expressions reduce to related values, and (C) At base types, the relation is the identity. Together, these imply that our translation preserves the meanings of programs in the sense that a function from base types to base types maps to a target function with the same extension.

An excerpt of the relation is shown in Fig. 12.3. The relation is defined over source (λ^{fg}) types, and is divided (like our earlier relations) into a value relation $\lfloor\cdot\rfloor_V^{\hat{\beta}}$, an expression relation $\lfloor\cdot\rfloor_E^{\hat{\beta}}$, and a heap relation $(n, H_s, H_t) \overset{\hat{\beta}}{\triangleright} {}^s\theta$, which we omit here. The relations specify when a source value (resp. expression, heap) is related to a target value (resp. expression, heap) at a source unary world ${}^s\theta$, a step index n and a partial bijection $\hat{\beta}$ that relates source locations to corresponding target locations. The relation actually mirrors the unary logical relation for λ^{fg}. The definition of the

$$
\begin{aligned}
\lfloor \mathsf{b} \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, {}^sv, {}^tv) \mid {}^sv \in [\![\mathsf{b}]\!] \wedge {}^tv \in [\![\mathsf{b}]\!] \wedge {}^sv = {}^tv\} \\
\lfloor \mathbf{1} \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, {}^sv, {}^tv) \mid {}^sv \in [\![\mathbf{1}]\!] \wedge {}^tv \in [\![\mathbf{1}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, ({}^sv_1, {}^sv_2), ({}^tv_1, {}^tv_2)) \mid \\
&\qquad ({}^s\theta, m, {}^sv_1, {}^tv_1) \in \lfloor \tau_1 \rfloor_V^{\hat\beta} \wedge ({}^s\theta, m, {}^sv_2, {}^tv_2) \in \lfloor \tau_2 \rfloor_V^{\hat\beta}\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, \mathsf{inl}(\ )\,{}^sv, \mathsf{inl}(\ )\,{}^tv) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \tau_1 \rfloor_V^{\hat\beta}\} \cup \\
&\qquad \{({}^s\theta, m, \mathsf{inr}(\ )\,{}^sv, \mathsf{inr}(\ )\,{}^tv) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \tau_2 \rfloor_V^{\hat\beta}\} \\
\lfloor \tau_1 \xrightarrow{\ell_e} \tau_2 \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, \mathsf{fix}\ f(x).e_s, \mathsf{fix}\ f(x).e_t) \mid \\
&\qquad \forall {}^s\theta' \sqsupseteq {}^s\theta, {}^sv, {}^tv, j < m, \hat\beta \sqsubseteq \hat\beta'.({}^s\theta', j, {}^sv, {}^tv) \in \lfloor \tau_1 \rfloor_V^{\hat\beta'} \implies \\
&\qquad ({}^s\theta', j, e_s[{}^sv/x][\mathsf{fix}\ f(x).e_s/f], e_t[{}^tv/x][\mathsf{fix}\ f(x).e_t/f]) \in \lfloor \tau_2 \rfloor_E^{\hat\beta'}\} \\
\lfloor \mathsf{ref}\ \tau \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, a_s, a_t) \mid {}^s\theta(a_s) = \tau \wedge ({}^sa, {}^ta) \in \hat\beta\} \\
\lfloor \mathsf{A}^{\ell'} \rfloor_V^{\hat\beta} &\triangleq \{({}^s\theta, m, {}^sv, {}^tv) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \mathsf{A} \rfloor_V^{\hat\beta}\} \\
\\
\lfloor \tau \rfloor_E^{\hat\beta} &\triangleq \{({}^s\theta, n, e_s, e_t) \mid \\
&\qquad \forall H_s, H_t.(n, H_s, H_t) \overset{\hat\beta}{\triangleright} {}^s\theta \wedge \forall i < n, {}^sv.(H_s, e_s) \Downarrow_i (H_s', {}^sv) \implies \\
&\qquad \exists H_t', {}^tv.(H_t, e_t) \Downarrow^f (H_t', {}^tv) \wedge \exists {}^s\theta' \sqsupseteq {}^s\theta, \hat\beta' \sqsupseteq \hat\beta.(n-i, H_s', H_t') \overset{\hat\beta'}{\triangleright} {}^s\theta' \\
&\qquad \wedge ({}^s\theta', n-i, {}^sv, {}^tv) \in \lfloor \tau \rfloor_V^{\hat\beta'}\} \\
\\
(n, H_s, H_t) \overset{\hat\beta}{\triangleright} {}^s\theta &\triangleq \mathit{dom}({}^s\theta) \subseteq \mathit{dom}(H_s) \wedge \\
&\qquad \hat\beta \subseteq (\mathit{dom}({}^s\theta) \times \mathit{dom}(H_t)) \wedge \\
&\qquad \forall (a_1, a_2) \in \hat\beta.({}^s\theta, n-1, H_s(a_1), H_t(a_2)) \in \lfloor {}^s\theta(a_1) \rfloor_V^{\hat\beta}
\end{aligned}
$$

Figure 12.3: Cross-language value and expression relations for the λ^{fg} to λ^{cg} translation

expression relation forces property (B) above, while the value relation at base types forces property (C).

Our main result is again a fundamental theorem, shown below. The symbols δ^s and δ^t denote unary substitutions in the source and target, respectively. The relation $\lfloor\Gamma\rfloor_V^{\hat{\beta}}$ (described in the technical report [52]) is the obvious one, obtained by pointwise lifting of the value relation.

Theorem 26 (Fundamental theorem)**.** If $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$ and $({}^s\theta, n, \delta^s, \delta^t) \in \lfloor\Gamma\rfloor_V^{\hat{\beta}}$, then $({}^s\theta, n, e_s\ \delta^s, e_t\ \delta^t) \in \lfloor\tau\rfloor_E^{\hat{\beta}}$.

The proof of this theorem is by induction on the derivation of $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$. This theorem has two important consequences. First, it immediately implies property (A) above and, hence, completes the argument that our translation is semantically correct. Second, the theorem, along with the binary fundamental theorem for λ^{cg}, allows us to re-derive the non-interference theorem for λ^{fg} (Theorem 22) directly. This re-derivation is important because it provides confidence that our translation preserves the meaning of security labels. As a simple counterexample, it is perfectly possible to translate λ^{fg} programs to λ^{cg} programs, preserving both typing and semantics, by mapping all source labels to the same target label (say, $\bot$). However, we would not be able to re-derive the source non-interference theorem using the target's properties if this were the case.

13

TRANSLATING λ^{CG} TO λ^{FG}

This chapter describes the translation in the other direction—from λ^{cg} to λ^{fg}. This translation coupled with the translation from λ^{fg} to λ^{cg} gives a constructive proof of the equi-expressiveness of the two styles of IFC type systems. The overall structure (but not the details!) of this translation are similar to that of the earlier λ^{fg} to λ^{cg} translation, so we skip some boilerplate material here. The superscript or subscript s (source) now marks elements of λ^{cg} and t (target) marks elements of λ^{fg}.

13.1 TYPE TRANSLATION

The key idea of the translation is to map a source (λ^{cg}) expression e_s satisfying $\vdash e_s : \tau$ to a target (λ^{fg}) expression e_t satisfying $\vdash_\top e_t : [\![\tau]\!]$. The type translation $[\![\tau]\!]$ is defined below. The *pc* for the translated expression is $\top$ because, in λ^{cg}, all effects are confined to a monad, so at the top-level, there are no effects. In particular, there are no write effects, so we can pick any *pc*; we pick the most informative *pc*, $\top$.

The type translation, $[\![\tau]\!]$, is defined by induction on τ.

$$
\begin{aligned}
[\![\mathsf{b}]\!] &= \mathsf{b}^\perp \\
[\![\tau_1 \to \tau_2]\!] &= ([\![\tau_1]\!] \xrightarrow{\top} [\![\tau_2]\!])^\perp \\
[\![\tau_1 \times \tau_2]\!] &= ([\![\tau_1]\!] \times [\![\tau_2]\!])^\perp \\
[\![\tau_1 + \tau_2]\!] &= ([\![\tau_1]\!] + [\![\tau_2]\!])^\perp \\
[\![\mathsf{ref}\ \ell\ \tau]\!] &= (\mathsf{ref}\ ([\![\tau]\!] + \mathbf{1})^\ell)^\perp \\
[\![\mathbb{C}\ \ell_1\ \ell_2\ \tau]\!] &= (\mathbf{1} \xrightarrow{\ell_1} ([\![\tau]\!] + \mathbf{1})^{\ell_2})^\perp \\
[\![[\ell]\,\tau]\!] &= ([\![\tau]\!] + \mathbf{1})^\ell
\end{aligned}
$$

The most interesting case of the translation is that for $\mathbb{C}\ \ell_1\ \ell_2\ \tau$. Since a λ^{cg} value of this type is a suspended computation, we map this type to a *thunk*—a suspended computation implemented as a function whose argument has type $\mathbf{1}$. The pc-label on the function matches the pc-label ℓ_1 of the source type. The taint label ℓ_2 is placed on the output type $[\![\tau]\!]$ using a coding trick: $([\![\tau]\!] + \mathbf{1})^{\ell_2}$. The expression translation of monadic expressions only ever produces values labeled $\mathsf{inl}()$, so the right type of the sum, $\mathbf{1}$, is never reached during the execution of a translated expression. The same coding trick is used to translate labeled and ref types[1].

13.2 TYPE-DIRECTED TERM TRANSLATION

The expression translation is directed by source typing derivations and is defined by the judgment $\Gamma \vdash e_s : \tau \rightsquigarrow e_t$, some of whose rules are shown in Fig. 13.1 (full translation can be found in the technical report [52]). The translation is fairly straightforward (given the type translation). The only noteworthy aspect is the use of the injection $\mathsf{inl}()$ wherever an expression of the type form $([\![\tau]\!] + \mathbf{1})^{\ell}$ needs to be constructed.

$$\frac{\Gamma \vdash e : [\ell]\,\tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\ \top\ \ell\ \tau \rightsquigarrow \mathsf{fix}\ \_\_.e_F}\ \text{unlabel}$$

$$\frac{\Gamma \vdash e : \mathbb{C}\ \ell_1\ \ell_2\ \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell_1\ \bot\ ([\ell_2]\,\tau) \rightsquigarrow \mathsf{fix}\ \_\_.\mathsf{inl}(e_F\ ())}\ \text{toLabeled}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{ret}(e) : \mathbb{C}\ \ell_1\ \ell_2\ \tau \rightsquigarrow \mathsf{fix}\ \_\_.\mathsf{inl}(e_F)}\ \text{ret}$$

Figure 13.1: Expression translation λ^{cg} to λ^{fg} (selected rules only)

13.3 PROPERTIES OF THE TRANSLATION

The translation preserves typing by construction, as formalized in the following theorem. The context translation $[\![\Gamma]\!]$ is defined pointwise on all types in Γ.

Theorem 27 (Typing preservation)**.** If $\Gamma \vdash e_s : \tau$ in λ^{cg}, then there is a unique e_t such that $\Gamma \vdash e_s : \tau \rightsquigarrow e_t$ and that e_t satisfies $[\![\Gamma]\!] \vdash_\top e_t : [\![\tau]\!]$ in λ^{fg}.

1 We could also have used a different coding in place of $([\![\tau]\!] + \mathbf{1})^{\ell_2}$. For example, $([\![\tau]\!] \times \mathbf{1})^{\ell_2}$ works equally well.

Again, a corollary of this theorem is that well-typed source programs translate to non-interfering target programs.

We further prove that the translation preserves the semantics of programs. Our approach is the same as that for the λ^{fg} to λ^{cg} translation—we set up a cross-language logical relation, this time indexed by λ^{cg} types, and show the fundamental theorem. From this, we derive that the translation preserves the meanings of programs. Additionally, we derive the non-interference theorem for λ^{cg} using the binary fundamental theorem of λ^{fg}, thus gaining confidence that our translation maps security labels properly. This development mirrors that for our earlier translation. We defer the details to the technical report [52].

14

RELATED WORK FOR λ^{CG}

We focus on related work directly connected to our contributions— coarse-grained IFC type system, logical relations for IFC type systems and language translations that care about IFC.

Coarse-grained IFC type systems. Besides λ^{amor}, the IFC type system that comes closest to λ^{cg} is the SLIO type system which is a static fragment of the hybrid HLIO system from [13]. However, there is a crucial difference in how the two type systems interpret the monadic type, $\mathbb{C}\ \ell_1\ \ell_2\ \tau$. λ^{cg} interprets the two indices on the monadic type as the pc-label and the taint label, respectively. However, SLIO's interpretation is very different. The SLIO monad is an instance of the Hoare state monad from [47]. As a result, SLIO interprets the two labels as the *starting taint* and the *ending taint* of the computation. Consequently, it is an invariant in SLIO that $\ell_1 \sqsubseteq \ell_2$. This makes SLIO more restrictive than λ^{cg}. The toLabeled construct in SLIO cannot always lower the final taint to $\perp$. SLIO's toLabeled rule is:

$$\frac{\Gamma \vdash e : \mathbb{C}\ \ell\ \ell'\ \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell\ \ell\ ([\ell']\ \tau)}\ \text{SLIO-toLabeled}$$

This restrictive rule makes it impossible to translate from λ^{fg} to SLIO in the way we translate from λ^{fg} to λ^{cg}. Our observation here is that SLIO's restriction, inherited from a prior system called LIO, is not important for statically enforced IFC and eliminating it allows a simple embedding of a fine-grained IFC type system.

Nonetheless, we did investigate further whether we can embed λ^{fg} into the static fragment of the unmodified SLIO. The answer is still affirmative, but the embedding is complex and requires nontrivial quantification over labels. Part II of the technical appendix of [53] contains a complete account of this embedding, which we do not repeat in this thesis.

HLIO also has two constructs, getLabel and labelOf, that allow reflection on labels. However, these constructs are meaningful only because HLIO uses hybrid (both static and dynamic) enforcement and carries labels at runtime. In a purely static enforcement, such as λ^{cg}'s, labels are not carried at runtime, so reflection on them is not meaningful.

Logical relations for IFC type systems. Logical relations for IFC type systems have been studied before to a limited extent. Sabelfeld and Sands develop a general theory of models of information flow types based on partial-equivalence relations (PERs), the mathematical foundation of logical relations [54]. However, they do not use these models for proving any specific type system or translation sound. The pure fragment of the SLam calculus was proven sound (in the sense of non-interference) using a logical relations argument [26, Appendix A]. However, to the best of our knowledge, the relation and the proof were not extended to mutable state.

The proof of non-interference for FlowCaml [50], which is very close to SLam, considers higher-order state (and exceptions), but the proof is syntactic, not based on logical relations. The dependency core calculus (DCC) [1] also has a logical relations model but, again, the calculus is pure. The DCC paper also includes a state-passing embedding from the IFC type system of Volpano, Irvine and Smith [56], but the state is first-order.

Mantel *et al.* use a security criterion based on an indistinguishability relation that is a PER to prove the soundness of a flow-sensitive type system for a concurrent language [42]. Their proof is also semantic, but the language is first-order.

In contrast to these prior pieces of work, our logical relations handle higher-order state, and this complicates the models substantially; we believe we are the first to do so in the context of IFC.

Our models are based on the now-standard step-indexed Kripke logical relations [4], which have been used extensively for showing the soundness of program verification logics. Our model for λ^{fg} is directly inspired by Cicek *et al.*'s model for a pure calculus of incremental programs [17]. That calculus does not include state, but the model is structurally very similar to our model of λ^{fg} in that it also uses a unary and a binary relation that interact at labeled types. Extending that model with state was a significant amount of work, since we had to introduce Kripke worlds. Our model for λ^{cg} has no direct predecessor; we developed it using ideas from our model of λ^{fg} and λ^{amor}. (DCC is also coarse-grained and uses a labeled monad to track dependencies, but the model of DCC is quite different from ours in the treatment of the monadic type.)

Language translations that care about IFC. Language translations that preserve information flow properties appear in the DCC paper. The translations start from SLam's pure fragment and the type system of Volpano, Irvine and Smith and go into DCC. The paper also shows how to recover the non-interference theorem of the source of a translation from properties of the target, a theorem we also prove for our translations. Barthe *et al.* [10] describe a compilation from a high-level imperative language to a low-level assembly-like language. They show that their compilation is type and semantics preserving. They also derive non-interference for the source from the non-interference of the target. Fournet and Rezk [22] describe a compilation from an IFC-typed language to a low-level language where confidentiality and integrity are enforced using cryptography. They prove that well-typed source programs compile to non-interfering target programs, where the target non-interference is defined in a computational sense. Algehed and Russo [5] define an embedding of DCC into Haskell. They also consider an extension of DCC with state but, to the best of our knowledge, they do not prove any formal properties of the translation.

Part III

Epilogue

15

ABSTRACTING THE GHOST STATE

The two type theories that we have seen so far operate on specific instances of ghost state: potential in the case of λ^{amor} and the confidentiality label in the case of λ^{cg}. In this chapter, we show how to unify the two type theories using an abstract monoidal structure which generalizes both the potential and the confidentiality label. We describe the changes needed for this generalization and prove them sound.

15.1 DIFFERENCE IN THE PROOF THEORIES OF λ^{AMOR} AND λ^{CG}

As mentioned earlier, both λ^{amor} and λ^{cg} are based on the use of similar ghost operations like store and toLabeled, and release and unlabel. However from a proof-theoretic perspective there are subtle differences in their typing rules which makes it hard to unify them. To highlight the differences, we isolate the relevant rules from λ^{amor} (T-store and T-release) and the corresponding rules from λ^{cg} (CG-toLabeled and CG-unlabel) in Fig. 15.1 (for simplification, here we write the typing judgment of λ^{amor} with the linear context, Γ, only).

Rules T-store and CG-toLabeled introduce the modal type of λ^{amor} and λ^{cg} respectively. However, the way this is achieved is a bit different. T-store, on one hand, obtains the potential p associated with $[p]\,\tau$ from the context and represents it as a cost (/resource requirement) on the monad in the conclusion. CG-toLabeled, on the other hand, obtains the corresponding label ℓ' associated with $[\ell']\,\tau$ from taint-label on the monad in the premise, while the resulting taint-label on the monad in the conclusion is $\perp$[1]. Similarly, T-release uses the given potential (p_1) with e_1 to fulfill the resource requirement of the continuation e_2 partially. CG-unlabel, on the other hand, moves the complete label from the labeled type in the premise to the taint-label on the monadic type in the conclusion.

1 The *pc*-label on the λ^{cg}'s monad is irrelevant in the context of this generalization. As modal type of λ^{cg} only interacts with the taint-label and not with the *pc*-label whose purpose is only to prevent leaks via write effects.

$$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{store}\, e : \mathbb{M}\, p\, ([p]\, \tau)}\ \text{T-store} \qquad \dfrac{\begin{array}{c}\Gamma_1 \vdash e_1 : [p_1]\, \tau_1 \\ \Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}(p_1 + p_2)\, \tau_2\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{release}\, x = e_1 \ \mathsf{in}\ e_2 : \mathbb{M}\, p_2\, \tau_2}\ \text{T-release}$$

$$\dfrac{\Gamma \vdash e : \mathbb{C}\, \ell\, \ell'\, \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\, \ell\, \bot\, ([\ell']\, \tau)}\ \text{CG-toLabeled} \qquad \dfrac{\Gamma \vdash e : [\ell]\, \tau}{\Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\, \top\, \ell\, \tau}\ \text{CG-unlabel}$$

Figure 15.1: Typing rules for ghost operations: λ^{amor} and λ^{cg}

15.2 RECONCILING THE DIFFERENCES

We attribute the above differences to the difference in the *polarities* of the ghost state in λ^{amor} and λ^{cg}. In λ^{amor} the polarity of the potential in the modal type is different from the polarity of the potential in the monad – the former represents the *available* potential while the later represents the *required* potential. On the other hand λ^{cg} associates the same polarity to the confidentiality label both in the modal type and in the monad – both represent the taint label.

For the purpose of unifying the two proof theories, we flip the polarity of the potential in the monad by representing it as a negative potential. This means that in the monadic type $\mathbb{M}(-p)\,\tau$, $-p$ now represents availability of $-p$ units of resources (which is still the same as the original interpretation of a resource requirement/cost of p units). Additionally we generalize the types of store and toLabeled to make their typing rules match as shown in Fig. 15.2. And finally for the unlabel construct, we represent it in the same let style as the release rule. The new let style of unlabel is merely a syntactic sugar for $\mathsf{bind}((\mathsf{unlabel}\ e_1), x.e_2)$.

$$\dfrac{\Gamma \vdash e : \mathbb{M}(-p_1)\, \tau}{\Gamma \vdash \mathsf{store}\, e : \mathbb{M}(-(p_1 + p_2))\, ([p_2]\, \tau)}\ \text{T-store} \qquad \dfrac{\begin{array}{c}\Gamma_1 \vdash e_1 : [p_1]\, \tau_1 \\ \Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}(-(p_1 + p_2))\, \tau_2\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{release}\, x = e_1 \ \mathsf{in}\ e_2 : \mathbb{M}(-p_2)\, \tau_2}\ \text{T-release}$$

$$\dfrac{\Gamma \vdash e : \mathbb{C}\, \ell\, (\ell_1 \sqcup \ell_2)\, \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\, \ell\, \ell_1\, ([\ell_2]\, \tau)}\ \text{CG-toLabeled} \qquad \dfrac{\Gamma \vdash e_1 : [\ell_1]\, \tau \qquad \Gamma, x : \tau \vdash e_2 : \mathbb{C}\, \ell_1\, \ell_2\, \tau'}{\Gamma \vdash \mathsf{unlabel}(e_1, x.e_2) : \mathbb{C}\, \ell_1\, (\ell_1 \sqcup \ell_2)\, \tau'}\ \text{CG-unlabel}$$

Figure 15.2: Modified typing rules for ghost operations: λ^{amor} and λ^{cg}

We have checked that the above changes do not break the soundness of λ^{amor}. The technical details of these changes along with the soundness proof for both λ^{amor} and λ^{cg} can be found in the technical report [52].

With the above modifications, we can now replace the ghost state with a commutative monoid: set m with an associative operation $\odot$. In the case of λ^{amor}, m is the set of real numbers and $\odot$ is the addition operation over them. Additionally, since every element has an inverse, the monoid is actually a group in the case of λ^{amor}. In the case of λ^{cg}, $\mathfrak{m}$ is the set of confidentiality labels drawn from a lattice and $\odot$ represents the least upper bound operation ($\sqcup$).

16 CONCLUSION AND FUTURE WORK

16.1 CONCLUDING REMARKS

In this thesis we presented λ^{amor}, the first fully general affine type theory for amortized resource analysis of higher-order programs. λ^{amor} shows how by using well-understood concepts from sub-structural and modal type systems along with a new modal type for representing potential, we can define a sound and compositional type theory for verification of amortized bounds. Besides this, we also show that λ^{amor} is highly expressive via encoding of several non-trivial examples from different domains. Further, we show that cost verification using λ^{amor} is relatively complete for PCF, which means that all terminating programs of PCF can be type checked in λ^{amor} with their precise cost up to a constant factor.

Next, we shown that ideas developed via λ^{amor} are quite general and can be applied to other domains. We showed this by building a similar type theory for the domain of Information Flow Control (IFC). In particular, we showed that by using similar type theoretic constructs and ghost operations, we can build a type theory for coarse-grained IFC, λ^{cg}. Via λ^{cg} we showed how to build Kripke models for IFC with full higher-order state, something which was not known prior to this work. Besides proving the soundness of λ^{cg}, we also show that λ^{cg} is as expressive to an existing fine-grained IFC type system (λ^{fg}).

Finally, we showed that the two ghost states, namely, potential and the confidentiality label used in λ^{amor} and λ^{cg} are special instances of a more generic ghost state. We showed how to unify the two type theories using an abstract monoidal structure.

16.2 SOME DIRECTIONS FOR FUTURE WORK

There are several directions for future work. We highlight some of these here.

16.2.1 *Future directions for λ^{amor}*

Lower-bound analysis. So far we have used λ^{amor} only for verification of resource upper bounds. We believe that by interpreting potential in a dual manner, i.e., as an obligation to burn resources, we can derive a calculus for establishing lower bounds.

Relational interpretation of potential. λ^{cg} studies both the unary and the relational interpretation of confidentiality labels. While we understand the unary interpretation of potentials, their relational interpretation remains to be understood. Such a development might be non-trivial as, to the best of our knowledge amortized analysis has not been explored in a relational setting.

16.2.2 *Future directions for λ^{cg}*

Full abstraction. Since our translations between λ^{cg} and λ^{fg} preserve typed-ness, they map well-typed source programs to non-interfering target programs. However, an open question is whether they preserve contextual equivalence, i.e., whether they are fully abstract. Establishing full abstraction will allow translated source expressions to be freely co-linked with target expressions. We have not attempted a proof of full abstraction yet, but it looks like an interesting next step. We note that since our dynamic semantics (big-step evaluation) are not cognizant of IFC (which is enforced completely statically), it may be sufficient to generalize our translations to simply-typed variants of λ^{fg} and λ^{cg}, and prove those fully abstract.

BIBLIOGRAPHY

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. “A Core Calculus of Dependency.” In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1999.

[2] Amal J. Ahmed. “Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types.” In: *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*. 2006.

[3] Amal Jamil Ahmed. “Semantics of types for mutable state.” PhD thesis. Princeton university, 2004.

[4] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. “State-dependent representation independence.” In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2009.

[5] Maximilian Algehed and Alejandro Russo. “Encoding DCC in Haskell.” In: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*. 2017.

[6] Robert Atkey. “Syntax and Semantics of Quantitative Type Theory.” In: *Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2018.

[7] Thomas H. Austin and Cormac Flanagan. “Efficient Purely-dynamic Information Flow Analysis.” In: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*. 2009.

[8] Thomas H. Austin and Cormac Flanagan. “Permissive dynamic information flow analysis.” In: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*. 2010.

[9] Martin Avanzini and Ugo Dal Lago. “Automating Sized-type Inference for Complexity Analysis.” In: *Proc. ACM Program. Lang.* 1.ICFP (2017).

[10] Gilles Barthe, Tamara Rezk, and Amitabh Basu. “Security types preserving compilation.” In: *Computer Languages, Systems & Structures (CLSS)* 33.2 (2007).

[11] Gérard Boudol. “Secure Information Flow as a Safety Property.” In: *International Workshop on Formal Aspects in Security and Trust (FAST)*. 2008.

[12] Niklas Broberg, Bart Delft, and David Sands. "Paragon for Practical Programming with Information-Flow Control." In: *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*. 2013.

[13] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2015.

[14] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. "Compositional certified resource bounds." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015.

[15] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits." In: *J. Autom. Reasoning* 62.3 (2019).

[16] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. "Relational cost analysis." In: *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL)*. 2017.

[17] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. "A type theory for incremental computational complexity with control flow changes." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2016.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. 2009.

[19] Karl Crary and Stephanie Weirich. "Resource Bound Certification." In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2000.

[20] Nils Anders Danielsson. "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures." In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2008.

[21] Matthias Felleisen and Daniel P. Friedman. "Control operators, the SECD-machine, and the λ-calculus." In: *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*. 1987.

[22] Cédric Fournet and Tamara Rezk. "Cryptographically Sound Implementations for Typed Information-flow Security." In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2008.

[23] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining Effects and Coeffects via Grading." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2016.

[24] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. "Bounded linear logic: a modular approach to polynomial-time computability." In: *Theoretical Computer Science* 97.1 (1992).

[25] Joseph A. Goguen and José Meseguer. "Security policies and security models." In: *Proceedings of the IEEE Symposium on Security and Privacy*. 1982.

[26] Nevin Heintze and Jon G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity." In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1998.

[27] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate Amortized Resource Analysis." In: *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2011.

[28] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. "Towards Automatic Resource Bound Analysis for OCaml." In: *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 2017.

[29] Jan Hoffmann and Martin Hofmann. "Amortized Resource Analysis with Polynomial Potential: A Static Inference of Polynomial Bounds for Functional Programs." In: *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP)*. 2010.

[30] Martin Hofmann and Steffen Jost. "Static Prediction of Heap Space Usage for First-order Functional Programs." In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2003.

[31] Sebastian Hunt and David Sands. "On flow-sensitive security types." In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2006.

[32] Hoffman Jan. "Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis." PhD thesis. Ludwig-Maximilians-Universität München, 2011.

[33] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. "Static Determination of Quantitative Resource Usage for Higher-order Programs." In: *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2010.

[34] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. “"Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis.” In: *Proceedings of Formal Methods (FM)*. 2009.

[35] Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. “Type-Based Cost Analysis for Lazy Functional Languages.” In: *J. Autom. Reason.* 59.1 (2017).

[36] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning.” In: *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*. 2015.

[37] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. “Recurrence extraction for functional programs through call-by-push-value.” In: *PACMPL* 4.POPL (2020).

[38] Jean-Louis Krivine. “A Call-by-name Lambda-calculus Machine.” In: *Higher Order Symbolic Computation* 20.3 (2007).

[39] Dal Lago and Marco Gaboardi. “Linear Dependent Types and Relative Completeness.” In: *Logical Methods in Computer Science* 8.4 (2011).

[40] Dal Lago and Barbara Petit. “Linear Dependent Types in a Call-by-value Scenario.” In: *Science of Computer Programming* 84 (2012).

[41] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. “Contract-based Resource Verification for Higher-order Functions with Memoization.” In: *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 2017.

[42] Heiko Mantel, David Sands, and Henning Sudbrock. “Assumptions and Guarantees for Compositional Noninterference.” In: *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*. 2011.

[43] Ana Almeida Matos and Gérard Boudol. “On declassification and the non-disclosure policy.” In: *Journal of Computer Security (JCS)* 17.5 (2009).

[44] Eugenio Moggi. “Notions of Computation and Monads.” In: *Information and Computation* 93.1 (1991).

[45] Andrew C. Myers and Barbara Liskov. “Protecting Privacy Using the Decentralized Label Model.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000).

[46] Glen Mével, Jacques-Henri Jourdan, and François Pottier. “Time credits and time receipts in Iris.” In: *European Symposium on Programming (ESOP)*. 2019.

[47] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Polymorphism and Separation in Hoare Type Theory." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2006.

[48] Georg Neis, Derek Dreyer, and Andreas Rossberg. "Non-parametric parametricity." In: *J. Funct. Program.* 21.4-5 (2011).

[49] Chris Okasaki. "Purely Functional Data Structures." PhD thesis. Carnegie Mellon University, 1996.

[50] François Pottier and Vincent Simonet. "Information Flow Inference for ML." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.1 (2003).

[51] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. "Possible worlds and resources: the semantics of BI." In: *Theoretical Computer Science* 315.1 (2004).

[52] Vineet Rajani. *A type-theory for higher-order amortized analysis*. Tech. rep. MPI-SWS-2020-001. Max Planck Institute for Software Systems, 2020. URL: https://www.mpi-sws.org/tr/2020-001.pdf.

[53] Vineet Rajani and Deepak Garg. "Types for Information Flow Control: Labeling Granularity and Semantic Models." In: *31st IEEE Computer Security Foundations Symposium (CSF)*. 2018.

[54] Andrei Sabelfeld and David Sands. "A PER Model of Secure Information Flow in Sequential Programs." In: *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*. 1999.

[55] RE Tarjan. "Amortized computational complexity." In: *SIAM Journal on Algebraic and Discrete Methods* 6.2 (1985).

[56] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis." In: *Journal of Computer Security (JCS)* 4.2/3 (1996).

[57] Hongwei Xi. "Dependent ML An approach to practical programming with dependent types." In: *J. Funct. Program.* 17.2 (2007).