



ON CONSTRUCTIVE TYPE THEORY AND LOGIC FOR STORABLE LOCKS

Alexandre Buisse

PHD DISSERTATION

August 2011

Abstract

This dissertation is about models for programming language semantics. In the first part, we make use of the slogan for dependent type theory that, used as a metalanguage, it can be a constructive counterpart to traditional set theory. We use its categorical model based on E-categories with families to interpret a version of itself.

In the second part, we develop a step-indexed model based on ultrametric spaces to provide an elegant solution to the circularity problem known as “knots in the store” which is created when concurrent programming languages are allowed to dynamically allocate locks, used as synchronisation resources. The issue comes from the fact that each lock, while stored in the heap, also possesses a resource invariant which is indexed over the same heaps. Finally, we also prove the model sound by relating it to standard operational semantics.

Contents

Abstract	iii
Acknowledgements	vii
Introduction	ix
1 The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories - an Intuitionistic Perspective	1
2 Concurrency and Separation Logic	13
2.1 A simple operational model	13
2.2 Concurrency and Race Free Programs	15
2.3 Critical Regions and Locks	16
2.4 Ownership	17
2.5 Modular Program Verification: Separation Logic	18
2.6 Concurrent Separation Logic	20
2.7 Knots in the Store	21
2.7.1 Circular Resource Invariants	21
2.7.2 Recursive Domain Equations	22
2.8 Related Work	23
2.8.1 A syntactic solution	23
2.8.2 Instrumenting the operational semantics	24
3 Our Setting	25
3.1 The Language	25
3.1.1 Feature choices	25
3.1.2 Grammar	26
3.2 Operational Semantics	27
3.3 The Assertion Language	28
3.3.1 Grammar and Predicates	28
3.3.2 Proof Rules	30
3.4 Example: Lock Coupling Lists	32

4	A Model for Storable Locks	35
4.1	Ultra-Metric Spaces	35
4.2	Expressing the Recursive Domain Equations	37
4.2.1	Worlds	37
4.2.2	Heaps	38
4.2.3	Predicates	39
4.3	Solving the Recursive Domain Equations	39
4.4	Building the Kripke Model	42
4.5	Locality Properties	43
5	Soundness	45
5.1	Logical View of the Machine State	45
5.2	Logical Transitions, Future Worlds	46
5.3	Safety, Consistency and Correctness	47
5.4	Soundness	48
5.5	Detailed Proof Cases	50
5.5.1	Technical lemmas	50
5.5.2	Sequential composition	51
5.5.3	Lock release	52
5.5.4	Lock creation	53
5.5.5	Thread creation	54
5.5.6	Thread synchronisation	55
	Conclusion	57

Acknowledgements

Throughout this PhD, I was lucky enough to be supervised by no less than three professors. My gratitude goes first and foremost to Lars Birkedal, Peter Dybjer and Hongseok Yang, without whom this dissertation would definitely never have existed.

Members of the PLS group at ITU, especially Jacob Thamsborg, Espen Højsgaard, Maxime Beauquier and Kasper Svendsen have been amazing in making the atmosphere of the department a fun and friendly one.

Friends from ENS Lyon were of great technological and psychological help. I believe it is fair to blame Daniel Hirschhoff for inspiring so many of us to pursue Programming Semantics. Jules Villard, Benoit Boissinot, Samuel Thibault and Jade Alglave have always provided their support and friendship in all of my various ventures. I am also happy to prove Grincheux, iderrick, mandos, Nimmy, stilgar, julio and Vinz wrong as to the vaporware-ness of my PhD.

Climbing was probably one of the main reason I (more or less) managed to keep my sanity, and I wish to thank everybody who let me dangle on their rope, in particular Rune, Karsten, Bernd, Anh, Sara, Will, Dave, Nick, Kristoffer, Susanne, Jonas, Birgitte, Eske, Kim, Stefan, Mads Peter, Alberto, Line, Mitchell, Daniel and Sune.

My family, as well as Ruthie, have always been supportive and are largely responsible for convincing me to finish this dissertation.

Finally, despite frequent and thorough complaints about the weather, the food and the lack of mountains, Denmark and its people have been incredibly welcoming, for which I am extremely grateful.

Introduction

Sadly, computer bugs are almost as ubiquitous as programs themselves, and can sometimes have dire consequences (the infamous Ariane V crash springs to mind). It has thus been for a long time a goal of computer scientists to be able to provide mathematical proofs that a program will behave according to its specifications.

There are a variety of reasons to explain why this is a difficult task. One of them is the sheer size of real-world programs, which can in some cases total several millions lines of code. Even in more modest cases, any real-world program will have a variety of different components and libraries which will increase exponentially the size of a proof. This makes the development of modular verification tools able to prove a sub-part of a program independently from the rest a priority. A particularly effective such method is *separation logic* [Rey02], where predicates can make explicit claims about which parts of the memory they are using. By using the special operator $*$, it is possible to join specifications which act on separate parts of the memory without any additional work, thus permitting to give localised proof of independent sub-programs.

Another issue comes from the evolution of programming languages to take advantages of the possibilities offered by advances in hardware development. In particular, concurrent programs competing to access shared resources in memory can lead to very complex race conditions or deadlock situations. Incidentally, the counter-intuitive nature of many of these situations is further encouragement to pursue formal proofs of concurrent programs.

Good progress in the domain has been provided by Peter O'Hearn's *Concurrent Separation Logic* [O'H04], which extends the separating conjunction to concurrent programs accessing shared resources. The interference between the different threads is handled via invariants associated with each resource and which must be respected by every thread, describing how they can interact with the resource.

This allows to reason about a single process of execution in isolation from all others, and leads to a form of implicit heap splitting, where every memory cell belongs to a particular thread at any given moment. Of course, such ownership can change over time, with sub-heaps being passed around between threads.

One particular limitation of Concurrent Separation Logic comes from its

treatment of locks and threads: they are fixed in number and pre-allocated. It is not possible to dynamically create and destroy new locks and new threads, which limits the type of programs that can be verified. To solve this problem, a way to reason about *storable locks* has to be found, where locks are dynamically allocated and have a "physical" existence as a particular type of memory cell. Similarly, new threads can be dynamically created via calls to a `fork` function and destroyed once they are done executing via a `join` call.

Having locks live in the heap is however very problematic, as it introduces a form of self-reference. Since there are resource invariants associated with each lock which describe the heap in which the lock lives, the invariants can contain themselves, leading to so-called "knots in the store", a challenge first recognised in a paper by Bornat et al.[BCOP05].

The model presented in this dissertation combines the best aspects of all existing solutions to this problem: it stays close to the operational semantics and leads to (relatively) straightforward proofs, yet has complete freedom for the resource invariants to be as generic as the program requires and gives a true semantic meaning to predicates.

To achieve this, we formulate the recursive domain equation expressing the problematic self-reference of locks and solve it directly using a model based the category of complete, bounded, ultrametric spaces (*CBUit*), a technique used in related works[BST10] and originally developed by America and Rutten [AR89]. The basic building brick of our model is the notion of downward closed *uniform predicate* (*UPred*), which is a way of introducing step-indexing methods to our proofs.

In order to prove soundness of our model with respect to the standard operational semantics of a small imperative language, we formulate a safety predicate expressing that the entire program is in a state which is safe to execute for a number of steps and that any thread which has finished execution will satisfy a given post-condition.

Narrative

This dissertation is split in two distinct parts, corresponding to the two main research directions pursued during the PhD.

First, we reproduce an article from the proceedings of the 2008 Mathematical Foundations of Programming Semantics conference, titled “The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories – an Intuitionistic Perspective” [BD08]. It was co-written with Peter Dybjer from Chalmers University of Technology and considers Type Theory (more specifically Martin-Löf Type Theory) as a constructive metalanguage in which to interpret a version of itself.

A natural shift to a slightly more applied research theme occurred in 2008 with work on Hoare Type Theory, which bridged our previous work on type theory with programming semantics and verification. An article was subsequently published in the Types in Language Design and Implementation workshop, in collaboration with Neelakantan Krishnaswami, Jonathan Aldrich, Lars Birkedal and Kasper Svendsen, titled “Design patterns in separation logic” [KAB⁺09]. Since we only contributed the formal verification part of it, however, we decided not to reproduce it in the present dissertation.

Starting in early 2009, we worked on building a model based on ultrametric spaces for a simple concurrent language supporting dynamically allocated storable locks, a final shift toward separation logic and program verification. This gave rise to a publication in the proceedings of the 2011 Mathematical Foundations of Programming Semantics, titled “A Step-Indexed Kripke Model of Separation Logic for Storable Locks” [BBS11] and co-written with Kristian Støvring and Lars Birkedal. The second part of the dissertation is an expanded version of this paper.

Outline

Chapter 1 is the in-extenso reproduction of the paper “The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories – an Intuitionistic Perspective” [BD08].

In **Chapter 2**, we discuss the general setting of our work and the main motivations for the domain of program verification. We present a simplified view of the sequential, imperative language which forms the basic building block of most works in the domain. We then discuss basic notions of concurrency, such as data races, locks and ownership. We present the main tool for reasoning about programs in a modular fashion, Separation Logic, and its extension to a concurrent setting, appropriately named Concurrent Separation Logic. Finally, we introduce the main issue we are striving to solve: adding locks as first class objects in the heap leads to a circularity in the model which is not trivial to solve. We also present two important related works on this issue: [GBC⁺07] and [Hob08].

In **Chapter 3**, we introduce all the necessary definitions and establish the settings for the rest of the dissertation. In particular, we present the imperative language we will aim to verify, along with its operational semantics and the assertion language used to express its specifications. Finally, we present a number of examples using various features of the language and provide their validity proofs.

In **Chapter 4**, we present our model, the main contribution of this work. We introduce necessary definitions and theorems relating to metric spaces and show how we use such spaces to solve a version of the recursive domain equations expressed in chapter 2. Finally, we show how the actual Kripke model is built from these bases.

Finally, in **Chapter 5**, we provide the necessary soundness proof of our model, with respect to the operational semantics defined in chapter 3. We introduce a logical abstraction of the machine state and show how to relate it to the concrete view and its transitions. We then present a *safety* predicate for logical states and use this to define what it means for a Hoare triple to be semantically valid. Finally, we present detailed proofs of soundness for most relevant cases.

Chapter 1

The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories - an Intuitionistic Perspective

The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories – an Intuitionistic Perspective

to Phil Scott on the occasion of his 60th birthday year

Alexandre Buisse¹

*Programming, Logic and Semantics Group
IT University of Copenhagen
Rued Langgaards Vej 7, 2300 København S*

Peter Dybjer²

*Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg*

Abstract

We give an intuitionistic view of Seely's interpretation of Martin-Löf's intuitionistic type theory in locally cartesian closed categories. The idea is to use Martin-Löf type theory itself as metalanguage, and E-categories, the appropriate notion of categories when working in this metalanguage. As an E-categorical substitute for the formal system of Martin-Löf type theory we use E-categories with families (E-cwfs). These come in two flavours: groupoid-style E-cwfs and proof-irrelevant E-cwfs. We then analyze Seely's interpretation as consisting of three parts. The first part is purely categorical: the interpretation of groupoid-style E-cwfs in E-locally cartesian closed categories. (The key part of this interpretation has been type-checked in the Coq system.) The second is a coherence problem which relates groupoid-style E-cwfs with proof-irrelevant ones. The third is a purely syntactic problem: that proof-irrelevant E-cwfs are equivalent to traditional lambda calculus based formulations of Martin-Löf type theory.

1 Introduction

In this lecture we draw together two of Phil Scott's interests: the relationship between type theory and category theory on the one hand and constructive category theory on the other. Let us begin with a quotation from the book *Introduction to higher order categorical logic* by Lambek and Scott [10]:

We also claim that intuitionistic type theories and toposes are closely related, in as much as there is a pair of adjoint functors between their respective categories. This is worked

¹ Email: abui@itu.dk

² Email: peterd@cs.chalmers.se

out in Part II. The relationship between Martin-Löf type theories and locally cartesian closed categories was established too recently (by Robert Seely) to be treated here.

We shall here discuss Seely's interpretation from an intuitionistic perspective. Our key idea is to work in a constructive metalanguage. In fact we shall use Martin-Löf type theory itself as a metalanguage! We shall use the same mindset as in the paper *Normalization and the Yoneda Embedding* by Čubrić, Dybjer, and Scott [4]. In that paper we used the constructive notion of a *P-category* where each hom-set is equipped with a partial equivalence relation. We showed how the decision problem for equality in cartesian closed categories follows more or less directly from a constructive reading of a few well-known facts about presheaf categories including the Yoneda lemma. This provides a categorical and constructive alternative to the traditional solution, where equality in cartesian closed categories is decided by using the normalization and Church-Rosser properties of the simply typed lambda calculus.

Note also that Martin-Löf called his theory "intuitionistic type theory", although it is quite different from the "usual" intuitionistic type theory of Lambek and Scott which is an intuitionistic version of type theory in the tradition of Russell and Church. Unlike the usual type theory Martin-Löf type theory is a *programming language*. It is based on the Curry-Howard *identification* of propositions and types, and the notion of *dependent type* is primitive. When we talk about "intuitionistic type theory" in this paper we henceforth always mean Martin-Löf's intuitionistic type theory.

Seely's interpretation is described in the paper *Locally cartesian closed categories and type theory* [15]. His main result states that the following two categories are equivalent:

- the category of "Martin-Löf theories" with types $\prod_{x \in A} B[x]$, $\sum_{x \in A} B[x]$, and $I(a, b)$, where the rules for the identity type I are those of the *extensional* intuitionistic type theory of Martin-Löf [12,13].
- the category of locally cartesian closed categories.

Close scrutiny of Seely's proof however reveals some issues in need of further clarification. These issues were discussed by Curien in his paper *Substitution up to isomorphism* [5]. Curien proposes a way

... to solve a difficulty arising from a mismatch between syntax and semantics: in locally cartesian closed categories, substitution is modelled by pullbacks (more generally pseudo-functors), that is, only up to isomorphism, unless split fibrational hypotheses are imposed. ... but not all semantics do satisfy them, and in particular not the general description of the interpretation in an arbitrary locally cartesian closed category. In the general case, we have to show that the isomorphisms between types arising from substitution are *coherent* in a sense familiar to category theorists.

Due to this coherence problem at the level of types, we are led to:

- switch to a syntax where substitutions are explicitly present (in traditional presentations substitution is a meta-operation, defined by induction);
- include type equality judgements in this modified syntax: we consider here only equalities describing the stepwise performance as substitution.

...

To our knowledge, the work presented here is the first solution to this problem, which, until very recently, had not even been clearly identified, mainly due to an emphasis on interesting mathematical models rather than on syntactic issues.

Curien proceeded to show that it is possible to interpret type equality as isomorphism in lcccs, and solve the coherence problem, for intuitionistic type theory with Π -types.

Somewhat later, Hofmann [7] showed how to construct a model of dependent type theory (category with attributes) with Π -types, Σ -types, and (extensional) identity types from a locally cartesian closed category by using a construction of Bénabou [2].

In this talk we shall revisit the Seely-Curien-Hofmann interpretation from an intuitionistic perspective. Seely, Curien and Hofmann of course all worked with the usual notion of category and with the usual classical (informal set-theoretic) metalanguage. We shall show how we get a new perspective on the problem if we work in an intuitionistic metalanguage. In particular, we shall explain why we are naturally led to interpret equality of types as isomorphism of objects in a category; why constructions of intuitionistic category theory nevertheless come with a choice (of pullbacks for example); and why the intuitionistic perspective helps us to understand the construction of a term models of intuitionistic type theory. Moreover, we point out that Seely's abstract fact about an equivalence of categories becomes a computer program. We write a "compiler" between two "programming languages": the language of intuitionistic type theory and the language of lcccs. Metamathematics has become metaprogramming!

Our approach will be based on the notion of an *E-category*. This is the standard notion of a category in the constructive sense. An E-category is just like a P-category, but hom-sets are equipped with equivalence relations rather than partial equivalence relations.

In the remainder of the paper we shall

- use intuitionistic type theory itself as metalanguage; in fact, we only need the very core of intuitionistic type theory, *the "logical framework"* with Π, Σ , and a universe;
- use the notion of an E-locally cartesian closed category (E-lccc);
- introduce the notion of an E-category with families (E-cwf) as a categorical substitute for the formal system of intuitionistic type theory;
- show two alternative definitions: groupoid-style and proof-irrelevant E-cwfs;
- outline the proof that *any E-lccc is a groupoid style E-cwf with Π, Σ , and I-types*
- show some code which suggests how this result is implemented in the Coq-system;
- introduce the coherence problem of relating groupoid-style and proof-irrelevant E-cwfs.

The key part of the proof that any E-lccc is a groupoid style E-cwf has been implemented in the Coq system by the first author. He has constructed the E-category of (groupoid style) E-families **EFam** and also shown how to construct an E-functor $T : C \rightarrow \mathbf{EFam}$ whenever C has finite limits. We plan to present the details of the Coq-implementation in a forthcoming publication.

Acknowledgements

These notes are based on a lecture given by the second author in the Special Session to honour Phil Scott on the occasion of his 60th birthday year. It was held in conjunction with MFPS XXIV in Philadelphia in May 2008. The second author is grateful for the many things Phil Scott has taught him about the connections between logic, types, and categories, and for very enjoyable collaboration. He would also like to thank Rick Blute and Andre Scedrov for organizing the Special Session and for inviting him to contribute to it.

2 E-locally cartesian closed categories

In Martin-Löf type theory a *set* is the same as a *data type* in a programming language. However, many sets (the real numbers, the rationals, the carrier of the free group, etc) come equipped with a notion of "equality" which is not the intrinsic identity of objects of the data type. Since quotient formation is not a constructive operation on sets, constructive mathematicians instead work with sets which are equipped with an equivalence relation. We shall follow the terminology of Čubrić, Dybjer, and Scott [4] and call them "E-sets", although they are usually called "setoids" in the type-theoretic community. Martin-Löf has proposed to call them "extensional sets". Bishop simply called them "sets", and used the term "preset" for the underlying representing data type.

E-sets and E-functions

An *E-set* (setoid, Bishop set, extensional set) is a set with an equivalence relation. Type-theoretically, an equivalence relation is a quadruple: a relation together with proofs of reflexivity, transitivity, and symmetry. Hence an E-set is a quintuple, a set together with the four components of the equivalence relation. In Coq we use records to represent tuples and the type of E-sets is thus defined as follows, bearing in mind that "sets" in the sense of Martin-Löf are implemented as "types" in Coq.

```
Record ESet : Type := {
  carrier :> Type;
  eq       : carrier → carrier → Type;
  refl    : ∀x      : carrier, eq x x;
  trans   : ∀x y z : carrier, eq x y → eq y z → eq x z;
  sym     : ∀x y   : carrier, eq x y → eq y x
}.
Notation "x ≡ y" := (eq x y) (at level 70).
```

We would like to remark that this definition uses a form of universe polymorphism. The level of the two instances of `Type` is implicit and will be determined by the context in which the E-set is used. By choosing the levels appropriately we can both get a notion of "small" E-set and various levels of "large" E-sets. Note that the type of `ESet` must be one level higher in the universe hierarchy than the type of `carrier`.

Moreover, the sign `:>` signifies a coercion which allows us to use the same name for an E-set and its carrier set.

An *E-function* (setoid map, Bishop function, extensional function) preserves the equivalence relation:

```
Record EFun (A B : ESet) : Type := {
  func :> A → B;
  pres : ∀x y : A, x ≡ y → func x ≡ func y
}.
```

E-categories

As already mentioned, the constructive notion of category has E-homsets. However, we do not include a notion of equality of objects as part of the structure of an E-category.

As category-theorists often point out, the moral notion of equality of objects is isomorphism; we do not need another distinct, primitive notion of equality of objects. Our Coq implementation is as follows:

```
Record ECat : Type := {
  ob      : Type;
  hom     :> ob → ob → ESet;
  id      : ∀A:ob, hom A A;
  comp    : ∀A B C:ob, hom B C ⇒ hom A B ⇒ hom A C;
  idL     : ∀(A B:ob) (f:hom A B), comp ___ (id _) f ≡ f;
  idR     : ∀(A B:ob) (f:hom A B), comp ___ f (id _) ≡ f;
  assoc   : ∀(A B C D:ob) (f:hom C D) (g:hom B C) (h:hom A B),
            comp ___ (comp ___ f g) h ≡
            comp ___ f (comp ___ g h)
}.

```

The \Rightarrow -notation expresses that composition is a binary E-function on E-homsets.

We have used a coercion which allows us to use the notation $C A B$ for the E-set of arrows between A and B in the E-category C.

As for the definition of E-set, the definition of E-category can be instantiated to yield various notions of small and large category.

E-pullbacks

It is quite straightforward to formalize the basic E-categorical notions, see Huet and Saibi [9]. For example, the constructive notion of E-pullback is implemented by the following Coq code which states that an E-pullback is a function which maps a triple of objects A, B, C and pair of arrows f, g (with appropriate sources and target) to a quintuple consisting of the object D (the apex), the projection arrows h, k, and the proofs sq and un (of the commutativity of the pullback square and of the universal property, respectively):

```
Record EPullback (C : ECat) (A B C : ob C)
(f : C B A) (g : C C A) : Type := {
  D      : ob C;
  h      : C D B;
  k      : C D C;
  sq     : f ∘ h ≡ g ∘ k;
  un     : ...
}.

```

Since it is a constructive function an E-pullback always comes with a computable choice. We essentially have an instance of the type-theoretic axiom of choice, which expresses (in Coq-notation) how to construct a choice function f from a set A to an A-indexed family of sets B:

$$\begin{aligned}
 &(\forall x : A, \exists y : B x, C x y) \rightarrow \\
 &(\exists f : (\forall x : A, B x), \forall x : A, C x (f x))
 \end{aligned}$$

The validity of this axiom is a direct consequence of the constructive meaning of the logical constants following the Brouwer-Heyting-Kolmogorov (and Curry-Howard-Martin-Löf) interpretation.

Note however that the "extensional" axiom of choice [14] is not valid. If A and B are E-sets, there is no reason why the choice function f should preserve the equivalence relation. But does the E-pullback come with an extensional choice? This is only a meaningful question if we equip the set of objects of the category with an equivalence relation. If this is isomorphism, then the answer is yes, E-pullbacks map equal arrows to isomorphic objects.

E-locally cartesian closed categories

We can now define the notion of an E-locally cartesian closed category as an E-category C such that all E-slice categories C/A are E-ccc for all objects A . The objects of the E-slice category C/A are arrows of C with target A . The arrows of C/A are commuting triangles, formalized type-theoretically as pairs of arrows and proofs that the triangle commutes.

Note that since there is no primitive notion of equality of objects in C/A , the equality of arrows in C is not passed on to these objects. However, we can prove that equal arrows of C become isomorphic objects of C/A .

Since it is straightforward to define the notion of E-cartesian closure, we can define the notion of E-lccc as follows:

```
Record ELCCC : Type := {
  C   :> ECat;
  ccc : ∀ A : ob C, ECCC (C/A)
}.
```

3 E-categories with families

What is Martin-Löf's intuitionistic type theory?

Having completed the E-categorical definition of lccc, we now ask ourselves how to formalize Martin-Löf type theory. And since Martin-Löf type theory is also our metalanguage, the question actually is how to formalize it in itself!

Before we address this question we need to ask ourselves exactly where to find a precise definition of Martin-Löf type theory. Looking through the literature it becomes apparent that it is not clear that there is a canonical definition. When writing down the syntax and inference rules for intuitionistic type theory, we have to make some choices. Should we use typed lambda calculus à la Church or à la Curry? Is the rule of substitution primitive or derived? Is the substitution operation explicit (a constructor of syntax) or implicit (an operation on the metalevel)? How are variables represented, with names or de Bruijn indices or de Bruijn levels? Are universes formulated à la Russell or à la Tarski? Etc. Of course, we believe that there is a number of equivalent formulations, but it may not be so easy to prove this rigorously. And how do we make sure that we do not forget any inference rules? The lack of a canonical definition is somewhat disturbing.

We here propose to use an abstract algebraic characterization of intuitionistic type theory as the initial category with families (cwfs) with extra structure [6,8,3,1]. This is a notion defined up to isomorphism. Cwfs provide the "minimal algebraization" of intuitionistic type theory: substitution is made explicit and variables are replaced by projections. However, dependent types are not modelled by fibrations as in lccc and many other categorical notions of model of dependent types.

Note that cwfs are similar to indexed categories. However, cwfs match the syntac-

tic structure of dependent type theory better, whereas indexed categories are closer to the syntactic structure of predicate logic.

Categories with families (cwfs)

A category with families consists of

- C , a *category of contexts*. Its objects are called *contexts* and its morphisms are called *substitutions*.
- $T : C^{op} \rightarrow \mathbf{Fam}$, a functor where the
 - object part** maps a context Γ to the family of sets of *terms* $\{a \mid \Gamma \vdash a : A\}$ indexed by the set of *types* $\{A \mid \Gamma \vdash A \text{ type}\}$ in Γ .
 - arrow part** maps a substitution γ to a pair of functions which perform substitution of γ in types and terms respectively. We write $A[\gamma]$ for *substitution* of γ in a type A and $a[\gamma]$ for *substitution* of γ in the term a .
- A *terminal object* $[\]$ of C called the *empty context*. The unique arrow into $[\]$ is the *empty substitution*.
- A *context comprehension* operation which to an object Γ of C and a type A in Γ associates four components
 - context extension:** an object $\Gamma;A$ of C ;
 - weakening:** a morphism $p_{\Gamma,A} : \Gamma;A \rightarrow \Gamma$ of C - the *first projection*
 - assumption:** a term $q_{\Gamma,A} \in \Gamma;A \vdash A[p_{\Gamma,A}]$ - the *second projection*
 - substitution extension:** for each object Δ in C , morphism $\gamma : \Delta \rightarrow \Gamma$, and term $a \in \Delta \vdash A[\gamma]$, there is a unique morphism $\theta = \langle \gamma, a \rangle : \Delta \rightarrow \Gamma;A$, such that $p_{\Gamma,A} \circ \theta = \gamma$ and $q_{\Gamma,A}[\theta] = a$. This is the *universal property* of context comprehension.

Context comprehension in categories with families is similar to Lawvere's comprehension schema in hyperdoctrines [11].

E-cwfs and the E-category of E-families

It is clear how to understand the above definition of cwf if we base it on the usual (set-theoretic) notions of category, functor, etc. But how are cwfs understood constructively as "E-cwfs"? It should consist of

- an E-category C .
- an E-functor $T : C \rightarrow \mathbf{EFam}$.
- an E-terminal object.
- an E-context comprehension.

The crucial question is how to define the E-category of E-families \mathbf{EFam} , since the other E-categorical notions are clear. It turns out that there are two interesting alternatives: *groupoid style* E-families and *proof-irrelevant* E-families.

The first alternative uses the analogy between E-sets and *groupoids*. As already mentioned an E-set is a quintuple consisting of a set, a relation, and proofs of reflexivity, transitivity, and symmetry. If we equate all proofs we get a groupoid, where the carrier becomes the set of objects, the proofs that two objects are related become arrows, the proofs of reflexivity become an identity arrows, transitivity proofs become composition arrows, and symmetry proofs become inverse arrows in a groupoid.

Hence an E-set indexed E-family should be analogous to a groupoid-indexed family of groupoids. These are isomorphism-preserving functors from a groupoid \mathcal{A} to the category of groupoids:

$$B : \mathcal{A} \rightarrow \mathbf{Groupoid}$$

We write down the resulting definition in ordinary mathematical notation, since the Coq-code is somewhat lengthy.

If A is an E-set, then an A -indexed family of E-sets consists of

- a family B of E-sets indexed by the carrier set of A ;
- a *reindexing* map $\iota(p) : B(x') \rightarrow B(x)$ whenever $p : x \equiv_A x'$.

such that

- $\iota(\text{refl}) \equiv_{\text{ext}} \text{id}$ (the identity map);
- $\iota(\text{trans}(p, p')) \equiv_{\text{ext}} \iota(p) \circ \iota(p')$ (composition of maps);
- $\iota(p)$ is an E-bijection with inverse $\iota(\text{sym}(p))$.

Here, \equiv_{ext} refers to extensional equality of E-functions, that is, functions which map equivalent elements of the domain to equivalent elements of the codomain.

Let B be an A -indexed family of E-sets and let B' be an A' -indexed family with reindexings ι and ι' , respectively. A *morphism* between these two families consists of

- an E-function $f : A \rightarrow A'$;
- an A -indexed family of E-functions $g(x) : B(x) \rightarrow B'(f(x))$ for $x : A$.
- which is natural in x :

$$\begin{array}{ccc} B(x') & \xrightarrow{g(x')} & B'(f(x')) \\ \downarrow \iota(p) & & \downarrow \iota'(f(p)) \\ B(x) & \xrightarrow{g(x)} & B'(f(x)) \end{array}$$

whenever $p : x \equiv_A x'$.

There is an obvious definition of equivalence of morphisms of E-families.

The first author has implemented the E-category EFam in Coq, but we have to postpone showing the details of this implementation to a forthcoming publication. Given this definition and definitions of E-functors, E-terminal objects, and E-context comprehension, the code for E-cwfs can be given as the following Coq-record:

```
Record ECwf : Type := {
  C  :> ECat;
  T  : EFunctor C EFam;
  te : ETerminal C;
  cc : EContextComprehension C T
}.
```

Cwfs only capture the most basic structure of dependent types, but it is easy to add extra structure for interpreting Π -types, Σ -types, and I-types [6,1]:

```

Record ECwfPiI : Type := {
  C    :> ECwf;
  pi   : EPi C;
  sigma : ESigma C
  i    : EI C
}.
    
```

We have ended up with an iterated record structure. For example, the first component of an E-cwf is an E-category which itself is a record, and the second component of an E-category, the family of E-homsets, is a binary record-valued function.

E-cwfs as a flat record

Our iterated record structure can however be flattened (in the sense of functional programming). If we reorder and rename the components we see that this flattened record bears a strong similarity with the structure of the inference rules for the judgements of Martin-Löf type theory. The first seven components of the flattened record codify the seven forms of judgement of a substitution calculus in the style of Martin-Löf:

```

Record FlatECwf = := {
  Ctxt : Type;
  Hom  : Ctxt -> Ctxt -> Type;
  EHom : ∀ G D : Ctxt, Hom G D → Hom G D → Type;
  Ty   : Ctxt -> Type;
  ETy  : ∀ G : Ctxt, Ty G → Ty G → Type;
  Tm   : Ctxt -> Ty -> Type;
  ETm  : ∀ G : Ctxt, ∀ A : Ty G, Tm G A → Tm G A → Type;
  ...
  inference rules
  ...
}.
    
```

Note that there is no judgement for equality of contexts, since our category of context does not have an equality of objects.

The remaining components of the flattened record correspond to the inference rules of a substitution calculus for dependent types. We only give one example of an inference rule: the type equality rule (conversion rule). It comes from the reindexing map of E-families:

$$\text{iota} : \forall G : \text{Ctxt}, \forall A A' : \text{Ty } G,$$

$$\text{ETy } G A A' \rightarrow \text{Tm } G A' \rightarrow \text{Tm } G A$$

E-cwfs as a flat record resembles Curien's [5] explicit substitution calculus for dependent types with explicit witnesses of type equalities. We can view it as a systematic reconstruction of Curien's syntactic calculus, where we have relied on E-categorical structures.

4 Seely's interpretation, intuitionistically

E-cwfs from E-categories with finite limits

We can now prove an E-categorical version of Seely's theorem. As in Seely [15] we get the E-cwf structure (with Π, Σ , and extensional identity types) from an E-lccc \mathcal{C} in the

following way:

- The base E-category is \mathcal{C} .
- A type in a context Γ is an object of the slice E-category \mathcal{C}/Γ . Equality of types is *isomorphism* in the slice E-category.
- A term of type A in context Γ is a section of A . Equality of terms is inherited from equality of arrows in the base E-category.
- Substitution in types is obtained from the E-pullback construction. We can here verify the laws of groupoid-style E-cwfs.
- Etc, essentially following Seely, but with explicit treatment of inference rules of (flattened) E-cwfs relating to the interpretation of type equalities as isomorphisms in \mathcal{C} .

We have implemented the key part in Coq, the construction of the groupoid style E-category EFam and the E-functor $\text{T} : \text{EFuncTor } \mathcal{C} \text{ EFam}$. To prove this result, and more generally to construct the groupoid style E-cwf structure it suffices for \mathcal{C} to be an E-category with finite limits. The details of this implementation are planned to appear in a forthcoming publication.

The coherence problem

Have we now finished our constructive version of Seely's theorem? No, although we have argued that the flattened version of the E-cwf record has a close correspondence to Curien's calculus of explicit substitution we need to relate this to the "usual" syntax. The usual syntax however corresponds to *proof irrelevant* E-cwfs, the second alternative mentioned above. This is because proofs of type-equalities do not matter in the usual inference system.

We define an E-cwf to be proof irrelevant iff the following principle holds.

$$\begin{aligned} \text{coh} : & \forall G : \text{CtxT}, \forall A A' : \text{Ty } G, \\ & \forall p p' : \text{ETy } G A A', \forall a : \text{Tm } G A, \\ & \text{ETm } (\text{iota } G A A' p a) (\text{iota } G A A' p' a) \end{aligned}$$

The coherence problem is to relate groupoid style E-cwfs and proof irrelevant ones. Curien solved a similar coherence problem by a process of cut-elimination. We expect that it is possible to provide an E-categorical version of Curien's proof, but have to leave this as a conjecture for future work. It is not clear to us whether Hofmann's use of the Bénabou construction [7] can be transferred to our constructive setting.

References

- [1] A. Abel, T. Coquand, and P. Dybjer. On the algebraic foundation of proof assistants for intuitionistic type theory. In *FLOPS*, pages 3–13, 2008.
- [2] J. Bénabou. Fibred categories and the foundation of naive category theory. *Journal of Symbolic Logic*, 50:10–37, 1985.
- [3] A. Buisse and P. Dybjer. Towards formalizing categorical models of type theory in type theory. *Electr. Notes Theor. Comput. Sci.*, 196:137–151, 2008.
- [4] D. Čubrić, P. Dybjer, and P. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.
- [5] P.-L. Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1,2):51–86, 1993.

- [6] P. Dybjer. Internal type theory. In *TYPES '95, Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 120–134. Springer, 1996.
- [7] M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In L. Pacholski and J. Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*. Springer, 1994.
- [8] M. Hofmann. Syntax and semantics of dependent types. In A. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1996.
- [9] G. Huet and A. Saibi. Constructive category theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg*, January 1995.
- [10] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.
- [11] F. W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In A. Heller, editor, *Applications of Categorical Algebra, Proceedings of Symposia in Pure Mathematics*. AMS, 1970.
- [12] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
- [13] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- [14] P. Martin-Löf. 100 years of Zermelo's axiom of choice: what was the problem with it? *The Computer Journal*, 49(3):343–350, 2006.
- [15] R. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc.*, 95(33), 1984.

Chapter 2

Concurrency and Separation Logic

2.1 A simple operational model

Though the ultimate goal of the field of software verification, as its name suggests, is to verify actual programs running on real world computers, the complexity of such machines is much too high to model directly. Indeed, there are many layers in a single machine, from the individual electric signal going through a transistor to the operating system executing a user program.

Though recent efforts have been made to model some of these layers in a way that matches as accurately as possible their physical counterparts [Alg10, OSS09, Ler06], it is far more practical to instead work with simplified, idealised versions of the machine. By removing needless details which are orthogonal to the issues at hand, we can more closely focus on what really interests us, instead of getting lost in a sea of irrelevant details.

To take a single example, we consider the memory to be infinite in our models. This is of course not true in the real world, but we can always assume that whenever needed, the operating system will *transparently* allocate more memory to the program. Though this is actually a complex operation requiring important modifications to the virtual space addressing, from the point of view of the program, nothing special happened: it tried to use more memory and got access to it. Since we are not interested in mechanisms used to cope with the entire machine running out of physical memory (a rare occurrence with modern computers and the type of programs we consider), we will simply consider it never happens.

We will thus consider and use a relatively simple and standard abstraction of what happens when a computer executes a program.

We use an *operational* model: a program is simply a sequence of instructions, executed one after the other. At any point of the execution, we can find a

program pointer to the instruction about to be run. In a sequential setting, there is only a single program pointer.

Memory is one of the most important resources available to a program and is used to store information. There are two different types of memory made available: the *heap* and the *stack*.

The **heap** is a long term store made of an infinite number of consecutive cells. Each cell has a unique address and can contain an object, usually an integer. If there are no safeguards in the model, any program can access any cell simply by using a pointer to its address (also an integer). It can then either read the value contained in the cell or write a new value. Especially when concurrency is involved, memory corruption can happen very easily if a program starts writing unexpected values in a specific location. A program can also request from the operating system to be given access to some unallocated memory cells, *fresh memory*.

A particular issue with the heap is what happens when a program is done using a particular piece of memory. Unused memory cells which are not reclaimed by the operating system are essentially lost until the operating system realises that it can reuse them next time a program requests some fresh memory.

There are two mechanisms to deal with this. Some low level programming languages, for instance C, require the program to keep track of all its memory and explicitly declare the memory as not needed anymore through the call to the procedure `free`. While a simple approach, this also demands from the programmer a thorough understanding of all the data structures he uses and, as attested by the well known SEGFAULT exception, often leads to addressing errors when a memory cell still in use is mistakenly deallocated.

The alternative solution is to let the machine try to guess which cells are safe to deallocate and which ones are still in use. This is done via the use of a program called the *garbage collector*, running in parallel with the user programs. GCs use a variety of strategies and heuristics to determine which cells are unused. By nature, they are conservative: a false negative slightly increases the memory footprint, while a false positive will almost certainly generate an addressing error and the sudden stop of the program execution. Most high level languages, such as C#, Haskell or OCaml use garbage collectors, as it greatly simplifies memory usage for the programmer.

The other type of memory is the **stack**. It is more volatile and less structured than the heap and is used to remember which values each variable used by the program contains. In our models, the stack is an association table from variable names to integer values. Each variable has a *scope*, the part of the program where it is defined, which is determined by the language. Unlike in the heap, knowing the name of the variable is not necessarily enough to access the contents of the stack - the variable also needs to be in scope. Since variable names are not significant in themselves, it is possible to modify all occurrences of a variable to a different name, a process known as *alpha conversion* which, despite its apparent simplicity, can lead to complex issues.

2.2 Concurrency and Race Free Programs

In this dissertation, we are interested in issues related to *concurrency*: running several programs in parallel and studying their interactions. There are many different ways that concurrency is used in the real world:

- Several machines can run programs and interact via a network connexion.
- A single machine can have multiple independent processor.
- A single processor can have multiple cores.
- A single processor core can run the programs along with a *scheduler* which will switch between different executions as required.

In all cases, however, the result is the same from the point of view of each of the programs (also known as *threads*).

Interesting things start happening when the threads interact with each other and compete for access to memory resources, either an address in the heap or a variable in the stack. While this is desirable since it allows programs to collaborate on a given task, it also creates a whole set of new problems.

A **race condition** occurs when two threads want to access the same resource in an unprotected way. If at least one of them tries to write to the memory, the outcome becomes uncertain, since it depends on which thread actually executes first, something over which the user has no control. Additionally, this non-determinism is extremely difficult to detect when debugging, since the race condition might be difficult to reproduce. For these reasons, programs exhibiting race conditions will generally be considered as erroneous. A large part of concurrency verification consists in making sure the considered programs are *race free*.

To illustrate this point, consider the program in figure 2.1.

```
1 inc (x) {  
2   local y;  
3   y = x + 1;  
4   x = y;  
5 }  
6 z = 0;  
7 inc (z) || inc(z);
```

Figure 2.1: A racy program

The operator `||` means we are executing both sides in parallel and the `local` keyword takes a fresh value in the stack, invisible outside of the scope of the procedure.

It would appear to a casual observer that this program will simply increment twice the value of z , resulting in a final value of 2. However, there is one possible interleaving which returns a different result: if there is a context switch after the first thread has executed line 3, then both threads will have a value of 1 in their respective y variables. Executing their line 4 will result in both thread assigning 1 to the shared memory cell at location z , thus generating a different final result.

2.3 Critical Regions and Locks

To make sure programs are race-free, one should make sure that no unwanted interleaving can happen when shared resources are manipulated. This can be achieved by ensuring that the granularity of the language is sufficient – for instance if instead of using the `inc` procedure from program 2.1, we have a built-in operator that we know is *atomic* (i.e. can't be interrupted), then the program can safely execute parallel incrementations of the same variable.

However, atomic operations are usually very low-level, if they exist at all in the considered programming language. An alternative solution which doesn't rely on the details of the language is to delimit explicitly *critical regions*. Concretely, portions of the program are allowed to execute only if they are in possession of a particular resource, or **lock**. If they are the first thread to execute (the lock is free), they acquire it and continue executing. The next thread, however, finds the lock currently unavailable and has to wait for the first thread to be done manipulating the shared resource, at which time it releases the lock.

A race-free version of program 2.1 would then be:

```

1 inc (x, 1) {
2   local y;
3   acquire (1);
4   y = x + 1;
5   x = y;
6   release (1);
7 }
8 z = 0;
9 m = makelock();
10 inc (z, m) || inc (z, m);

```

Figure 2.2: A race-free version of program 2.1

The lock operations at lines 3 and 6 delineate clearly the critical region, where it would be unacceptable for the program to be interrupted.

It is worth noting that critical regions are *labelled* by the lock they use to protect their resources, which allows a fine granularity: programs are allowed

to have multiple critical regions to protect different resources, or even the same resource from different threads.

This freedom, however, comes at a price. When multiple locks are used to nest critical regions into each other, an insidious type of error can appear: **deadlocks**, where every thread is waiting for somebody else to release a lock. In its simplest form, a deadlock appears in some interleavings of program 2.3.

```
1 f (l1, l2) {
2   acquire (l1);
3   acquire (l2);
4   skip
5   release (l2);
6   release (l1);
7 }
8 m1 = makelock();
9 m2 = makelock();
10 f (m1, m2) || f(m2, m1);
```

Figure 2.3: A deadlocking program

If execution switches from the left to the right thread after line 2, the left thread will possess lock `m1` and wait for lock `m2`, while the right one will be in the opposite situation: it holds `m2` and waits for `m1`. In this situation, progress is impossible, and both threads will wait forever, much to the despair of the programmer.

While deadlocks are often a major concern when working with concurrency, it is not a topic we will be overly interested in in this work, and no further mention will be made.

2.4 Ownership

Throughout this work, we will refer to a thread *owning* a particular piece of memory. This comes from the *ownership hypothesis* [O’H07], which is respected by non-racy programs:

Ownership hypothesis

Each thread only accesses parts of the memory that it owns.

All threads, or code fragments, have shared ownership of the portions of memory that they read from, and exclusive ownership of those they write to. Since ownership can evolve dynamically during the execution of the program, this is a very useful abstraction. In particular, acquiring or releasing a lock means that a thread gains or loses the right to access certain locations, which

translates to modifications of ownerships of those locations. In this sense, acquiring a lock can be seen as acquiring ownership of the piece of memory “protected” by that lock, while conversely, releasing it will give the memory back to the environment. This is a key idea of the model we will build in chapter 4.

It should be noted, however, that unlike locks, ownership is not a language mechanism but merely an abstraction, or a “useful way to think about this”. In particular, it is possible for incorrect programs to violate the ownership hypothesis and address memory locations they do not own, which concretely corresponds to using shared resources outside of critical regions.

2.5 Modular Program Verification: Separation Logic

Whenever we want to reason about heap manipulating programs, one of the most useful tools is that of the **Hoare Triple** [Hoa69]. Informally, a Hoare Triple $\{P\}c\{Q\}$ is satisfied if, whenever a state satisfies the predicate P (the *precondition*), we can execute the command c and the resulting state will satisfy the predicate Q (the *post-condition*).

This provides a simple and straightforward way to specify the desired properties of a program and to compose already verified fragments. However, there are also some severe limitations. One of them is that Hoare Logic necessitates to keep track of the whole memory - indeed, every fragment of the program needs to make sure that nobody else will modify the part of the heap that it relies on. It is not enough to ensure that variables are only used in this specific fragment, since *aliasing* (the addressing of the same memory location by two different variables) is a common occurrence.

To solve this problem within the framework of Hoare Logic, it is necessary to adjoin each predicate with an extra requirement saying essentially “*and nobody else interferes with us*”. This can be difficult to formulate precisely, and it gives rise to an explosion in complexity, exponential in the number of code fragments to be proved. For real world programs which can have hundreds or thousands of independent parts, this approach is clearly unpractical.

This is why *modularity* is a desired feature of any verification system: we want to be able to make independent proofs of independent parts of the programs: it should be enough to prove that a fragment satisfies its specification and that it doesn’t interfere with the rest of the program for us to be able to compose it with any other similarly well behaved components.

A solution was proposed in the form of **separation logic** [Rey02, O’H04]. Its basic idea is to add some operators to the predicate language to specify explicitly that different predicate describe independent parts of the heap.

Separating Conjunction

The separating conjunction of two heap fragments h_1 and h_2 is well defined

if and only if $h_1 \perp h_2$ (i.e. $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$). $h_1 * h_2$ is then defined as $h_1.h_2$ (i.e. $(h_1 * h_2)(l) = h_1(l)$ if $l \in \text{dom}(h_1)$ and $h_2(l)$ if $l \in \text{dom}(h_2)$).

The separating conjunction is naturally extended to predicates: $P * Q$ is satisfied by a heap h if there is a splitting $h = h_1 * h_2$ such that h_1 satisfies P and h_2 satisfies Q .

More formally, most separation logic models are used to provide proofs for heap manipulating programming languages:

$$\begin{aligned} \text{Heap} &= \text{Loc} \multimap^{\text{fin}} \text{Val} \\ \text{Stack} &= \text{Var} \multimap^{\text{fin}} \text{Val} \\ \text{State} &= \text{Heap} \times \text{Stack} \end{aligned}$$

The grammar of separation logic assertions is:

E, F	$::=$	<i>Logical Expressions</i>
	x	<i>Variables</i>
	v	<i>Values</i>
	$E + F$ $E - F$...	<i>Arithmetic Operations</i>
P, Q	$::=$	<i>Formulas</i>
	$E = F$	<i>Boolean Expressions</i>
	emp_s	<i>Empty Stack</i>
	emp_h	<i>Empty Heap</i>
	$E \mapsto F$	<i>Singleton Heap</i>
	$P \wedge Q$	<i>Classical Conjunction</i>
	$\neg P$	<i>Classical Negation</i>
	$P * Q$	<i>Separating Conjunction</i>
	$P \multimap Q$	<i>Magic Wand</i>

Figure 2.4: Separation logic assertions

The interpretation of these assertions is given by a satisfaction relation on states of the form $\sigma \models P$. The full semantics can be found in figure 2.5.

One of the most important features of separation logic is the soundness of the frame rule, which is key to proving modular programs.

The frame rule says that, whenever a command c satisfies its specification in a certain state, then it satisfies it in all independent extensions of this state.

Frame Rule

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}} \text{freevar}(R) \cap \text{freevar}(c) = \emptyset$$

$$\begin{aligned}
\llbracket \mathbf{x} \rrbracket_s &= s(\mathbf{x}) \\
\llbracket v \rrbracket_s &= v \\
\llbracket E + F \rrbracket_s &= \llbracket E \rrbracket_s + \llbracket F \rrbracket_s \\
&\dots
\end{aligned}$$

$(s, h) \models E = F$	<i>iff</i>	$\llbracket E \rrbracket_s = \llbracket F \rrbracket_s$
$(s, h) \models emp_s$	<i>iff</i>	$dom(s) = \emptyset$
$(s, h) \models emp_h$	<i>iff</i>	$dom(h) = \emptyset$
$(s, h) \models E \mapsto F$	<i>iff</i>	$dom(h) = \{\llbracket E \rrbracket_s\}$ and $h(\llbracket E \rrbracket_s) = \llbracket F \rrbracket_s$
$(s, h) \models P \wedge Q$	<i>iff</i>	$(s, h) \models P$ and $(s, h) \models Q$
$(s, h) \models \neg P$	<i>iff</i>	$(s, h) \not\models P$
$(s, h) \models P * Q$	<i>iff</i>	$\exists h_1, h_2$ such that $h = h_1 * h_2$, $(s, h_1) \models P$ and $(s, h_2) \models Q$
$(s, h) \models P \multimap Q$	<i>iff</i>	$\forall h',$ if $h \perp h'$ then $(h', s) \models P \Rightarrow (h * h', s) \models Q$

Figure 2.5: Semantics of Separation Logic assertions

The ability to split states, however, comes at a price: in the most general cases, there are many ways to split a heap h in two sub-heaps h_1 and h_2 which each satisfy predicates P and Q . Consider for instance the case where $P = 1 \mapsto _ * True$ and $Q = 2 \mapsto _ * True$. If we consider a heap h with domain $\{1, 2, 3\}$, then we could equally have $h_1 = h|_{1,3}$ and $h_2 = h|_2$ or $h_1 = h|_1$ and $h_2 = h|_{2,3}$. In both cases, $h = h_1 * h_2$ would be a valid splitting satisfying $P * Q$.

This non-determinism is problematic when we want to prove soundness of operations involving separating conjunctions, since we can't make any assumptions about which particular splitting will occur. To solve this issue, an additional requirement is often added to proof rules: that assertions are *precise*. In the previous example, it would be enough to remove the $*True$ part from P and Q to make them precise.

Precision

An assertion P is precise if, for any heap h , there is at most one sub-heap h' which satisfies P .

2.6 Concurrent Separation Logic

Separation logic as presented in the previous section cannot be used to reason about concurrency, as it doesn't support any of the synchronisation tools used

to deal with critical regions. Extending it to concurrency was first proposed by Peter O’Hearn [O’H07] with *Concurrent Separation Logic* (CSL in short).

CSL doesn’t use locks but *conditional critical regions* (CCR), with the following syntax: `resource r in c` defines a new resource `r` to be used in the continuation of the program `c`, while `with r c` uses that resource to define a conditional critical region. The command `c` will only be executed if the resource `r` is available. When it is finished, it will exit the critical region and release the resource for the next thread to use.

Crucially, each resource maintains an invariant which describes what portion of the state it protects in the critical region. Whenever a resource is first acquired, the invariant is assumed, and it will need to be re-established when the resource is released. The resource invariants are kept in a static context Γ over which all triples are indexed.

Triples are now of the form $\Gamma \vdash \{P\}c\{Q\}$ where Γ is an association list from resource names r to invariants P_r . All previous rules, having no concurrent effects, simply get a new Γ index without further modifications. We then add three new rules:

$$\frac{\Gamma \vdash \{P\}c\{Q\} \quad \Gamma \vdash \{P'\}c'\{Q'\}}{\Gamma \vdash \{P * P'\}c || c' \{Q * Q'\}}$$

$$\frac{\Gamma, r : P_r \vdash \{P\}c\{Q\}}{\Gamma \vdash \{P * P_r\}\text{resource } r \text{ in } c\{Q * P_r\}}$$

$$\frac{\Gamma \vdash \{P * P_r\}c\{Q * P_r\}}{\Gamma, r : P_r \vdash \{P\}\text{with } r \text{ } c\{Q\}}$$

The first allows us to perform parallel composition of two program fragments as long as they access disjoint parts of the state. The second declares a new resource `r` and adds its invariant P_r to the context. It also requires that the resource invariant holds at the time of the creation. Finally, the third declares a critical region protected by resource `r`. During the execution of `c`, *inside* the critical region, the invariant can be assumed to hold, and it is re-established when the program exits the critical region.

2.7 Knots in the Store

2.7.1 Circular Resource Invariants

Crucially, locks (or resources) in CSL are statically allocated at the beginning of the program. This means that the programmer needs to decide before execution

just how many locks he will need and which exact resources they will need to protect. While this is straightforward in many cases, there are also situations where it becomes an important limitation in language expressiveness.

Consider the canonical example of fine grained list control: several threads want to collaborate on a linked list, iterating through it and adding or deleting new elements. A naive approach is to use a global lock for the whole list, owned by whatever thread is currently operating. In effect, this disallows any concurrency (at most one thread can manipulate the whole data structure at any time) and presents a very poor performance. If the list is long enough, it makes no sense to forbid thread A to work on the head while thread B is 500 nodes down performing some operation on the tail.

An alternative solution is to use fine grain locks: instead of a global lock for the entire list, allocate a lock per node. Each thread then only needs to acquire the locks corresponding to the data it is actually working on, thus allowing multiple threads to operate on different parts of the list simultaneously. For a more precise example of this type of program, see the lock coupling example in section 3.4.

Such a solution, however, cannot be implemented in CSL, since locks need to be dynamically allocated at the same time than the elements in the list. More precisely, locks need to be *storable* in the heap, becoming first class objects, just like regular memory cells.

Syntactically, it is trivial to make such a modification - all that is required is a new instruction `makeLock` used to turn a normal memory cell into a “special” lock cell. This memory location would only be special in that it would be possible to use `acquire` and `release` commands on it. In every other respect, it is just be another bit of memory containing an integer value.

The issue arises when we consider the resource invariants associated with our new storable locks. They describe heap fragments in which live the locks to which they are associated, which means that they can potentially refer to themselves, a circularity which makes modelling them precisely very challenging. As described by Bornat et al. [BCOP05]:

The program must dynamically allocate semaphores as well as buffers, and the idea of semaphores in the heap makes theoreticians wince. The semaphore has to be available to a shared resource bundle: that means a bundle will contain a bundle which contains resource, a notion which makes everybody’s eyes water. None of it seems impossible, but it’s a significant problem, and solving it will be a small triumph.

2.7.2 Recursive Domain Equations

More formally, a natural approach is to store lock invariants in a *World*, which can be considered as a semantic layer above the heap: regular memory cells are

not mapped to anything, while cells of the lock “type” will be associated with their invariant. Incidentally, this also gives a precise definition of what a lock cell is: a cell which location is in the domain of the world. This underlines the fact that from a purely operational point of view, locks are pieces of memory like everything else, one has to go to a higher semantic level to really notice a difference.

$$W = Loc \multimap^{fin} Pred$$

In order for invariants to be expressive enough, however, they also need access to the world – for instance, a predicate might want to postulate that some other lock will be owned by the current thread, or that its invariant will have a certain form. Following the example from section 2.7.1, the lock invariant for a node N will want to say, among other thing, that if $N.next$ exists, then $N.next.lock$ will be a lock with the same invariant, which will make it possible to the active thread to acquire $N.next.lock$ and thus iterate through the list.

This means that predicates need to be indexed by the World, and will then be the usual partitions of heaps:

$$Pred = W \rightarrow \mathcal{P}(H)$$

We thus end up with a set of Recursive Domain Equations that need to be solved before a model of a language with storable locks can be given:

$$\begin{aligned} W &= Loc \multimap^{fin} Pred \\ Pred &= W \rightarrow \mathcal{P}(H) \end{aligned}$$

Figure 2.6: Key Recursive Domain Equations

2.8 Related Work

2.8.1 A syntactic solution

A solution to this issue has been presented by Gotsman et al. in [GBC⁺07]. The main idea is to use syntactic *handles* to “cut” the knot, replacing a generic $Exists(E, A)$ predicate, which would stipulate that a lock exists at location E with resource invariant A , by the actual handle $A(E, F)$, where F is a list of parameters to be given to the invariant.

Only a static number of handles, which has to be determined before execution, is permitted. This allows to build an ad-hoc logic, thus removing the circularity of the recursive domain equations. The use of parameters for invariants retains a lot of the language expressivity.

They finally show their model sound with respect to standard interleaving operational semantics.

While a very convincing model with the added advantage of relative simplicity, there are two main objections:

- On a practical level, there is a syntactic limitation in that it is not possible to have full genericity of the locks invariants - locks need a handle taken from a static set which has to be predetermined and built into the model itself.

It is however unclear just how limiting this really is. We are currently unaware of any program which really requires lock invariants to be so generic that Gotsman's model can't be used anymore.

- On a more abstract level, this solution arguably feels like more of a syntactic "hack" and avoids the circularity of the recursive domain equations rather than providing a real solution. In particular, none of the spaces considered have a strong semantic meaning in their model.

2.8.2 Instrumenting the operational semantics

An alternative solution was proposed by Hobor in his PhD dissertation [Hob08], as well as in an ESOP article [HAZN08].

Rather than using standard operational semantics for their language, they instrument it with resource invariant annotations. They then define a multi-layered model with erased and unerased variants, as well as an oracle semantics which models possible interleavings.

While this solution does not have the same syntactic limitation on lock invariants as Gotsman's model, it comes at the cost of a much more complex and less intuitive model as well as an instrumented operational semantics, which requires additional work and knowledge from the programmer.

Chapter 3

Our Setting

In this chapter, we will present the various definitions of the language we are studying, its small steps operational semantics, the assertion language we use to state properties that our programs need to satisfy, and finally three examples of increasing complexity, with their associated proof sketches.

3.1 The Language

3.1.1 Feature choices

Throughout this dissertation, all proofs and assertions will be made relative to programs written in a concurrent imperative language. It can be considered a subset of C and has been designed to be as simple as possible, supporting only those features directly relevant to our work, most notably *storable locks* and *dynamic threads*.

Parallelism is supported via calls to the `fork` command, which will create a new thread executing a given procedure. The parent thread (the one which executed the `fork` call) will share its local variables with the new thread and carry on its own sequential execution.

Two important design choices are *immutable stack variables* and *lack of explicit deallocation*.

By using continuation passing style commands for allocation and look-up (`let x = new in C`, `let x = [E] in C` and `let x = E in C`, respectively), we can forbid the modification of the variable `x` in the rest of the program. Any further assignment to `x` will actually be (implicitly) done on a fresh variable. This considerably simplifies some of the considerations relative to stack variables and their scope in the rest of this work.

A more comprehensive presentation of this language style can be found in [BTSY06].

There is no explicit deallocation of memory cells or locks (via calls to commands like `free` or `destroy_lock`). This allows us to forego the use of fractional permissions associated with each lock, which simplifies both our model and our already heavy notations.

In previous work on dynamically allocated locks (such as [GBC⁺07] and [Hob08]), the knowledge of the existence of a lock comes with a fractional permission comprised between 0 and 1. Every time that the thread shares that knowledge with another thread, it needs to split its permission into two and hand one to the new thread. Conversely, a thread done executing hands back all its lock permissions to its parent thread, which adds them to its own.

Any permission is enough to try to acquire and release a lock, but lock destruction requires the full permission of 1. This ensures that whichever thread performs the destruction is the only one which actually knows of the existence of the lock. Without such safeguards, a thread could try to acquire a lock which has been deallocated by somebody else, leading to a memory fault.

The main drawback of not allowing lock deallocation is that we can't prove one of the canonical programs found in the literature (e.g. in [GBC⁺07]): *last one cleans up*.

3.1.2 Grammar

The language obeys a relatively simple grammar, defined in figure 3.1. Its basic building bricks are variables (*Var*), expressions (*E*, *F*) which evaluate to integer values, boolean expressions (*B*) which evaluate to *True* or *False* and commands (*C*).

It is worth noting that memory locations are simply integer values. Trying to access an unallocated location will result in an execution fault.

<i>Var</i>	::= <i>x, y, z, ...</i>	
<i>E, F</i>	::= <i>nil</i> <i>Var</i> 0 <i>S(E)</i> <i>E + F</i> <i>E - F</i>	
<i>B</i>	::= <i>True</i> <i>E = F</i> [<i>E</i>] = <i>F</i> <i>E < F</i> not <i>B</i>	
<i>C</i>	::= let <i>x = new</i> in <i>C</i>	<i>allocation</i>
	let <i>x = E</i> in <i>C</i>	<i>stack lookup</i>
	let <i>x = [E]</i> in <i>C</i>	<i>heap lookup</i>
	[<i>E</i>] = <i>F</i>	<i>assignment</i>
	if <i>B</i> then <i>C</i> else <i>C'</i>	<i>branching</i>
	while <i>B</i> do <i>C</i>	<i>looping</i>
	skip	<i>inaction</i>
	<i>C; C'</i>	<i>sequential composition</i>
	makelock_P(<i>E</i>)	<i>lock creation</i>
	acquire(<i>E</i>)	<i>lock acquisition</i>
	release(<i>E</i>)	<i>lock release</i>
	let <i>x = fork(f)</i> in <i>C</i>	<i>thread creation</i>
	join(<i>E</i>)	<i>thread synchronisation</i>

Figure 3.1: Language grammar

Since our language supports dynamic thread creation, we need to define the code to execute whenever a fork call is executed, and do so outside of the main procedure. This is done statically before the code of the main command is called, which gives programs the general form of figure 3.2.

$$\begin{array}{l} \text{let } f_0 = C_0 \text{ and} \\ \quad f_1 = C_1 \text{ and} \\ \quad f_2 = C_2 \text{ and} \\ \quad \dots \\ \quad f_n = C_n \text{ in} \\ \quad C \end{array}$$

Figure 3.2: *General form of a program*

3.2 Operational Semantics

We will use a simple non-deterministic operational semantics for the language, close to what would actually be executed by a real machine. Our abstract locks behave similarly to spinlocks in that an `acquire` instruction on a lock that is not available will simply “hang” and will be allowed to try to acquire the lock again after another thread took an execution step.

The semantic is non-deterministic in that from any state, all of the active threads have an equal right to execution. However, with the exception of memory allocation (which chooses a random free memory cell), the execution *in one particular thread* is deterministic.

A state in our semantics is a triple (h, s, tp) where h is a heap in $H = Loc \rightarrow_{fin} Val$, s a stack in $Var \rightarrow Val$ and tp a thread pool mapping each thread identifier to the code it should execute, i.e. $tp : \{1, \dots, n\} \rightarrow Cmd$ for some n .

It is important to note that this operational semantics is very low-level and doesn’t deal with some of the more “semantic” features. In particular, neither the heap nor the stack are split between threads and there is no notion of private or shared space. Similarly, locks are implemented in a very simple fashion: as an integer which contains 0 if the lock is available, and k if it is currently owned by the thread with identifier k . Since there is no notion of world, nothing really distinguishes a lock or a thread handle from a regular memory cell, and an incorrect program could easily overwrite lock values.

The operational semantics only safeguards against the most elementary misuses, mostly addressing unallocated memory cells. This reflects the way code is really executed on a machine: a memory fault will usually raise an error and stop the execution of the program (the famous `SEGFAULT` in C) while misusing locks, by for instance writing in their location as if it was a regular memory cell, will lead to a much harder to detect memory corruption.

Operational Semantics

The general form of a reduction in our operational semantics is $(h, s, t) \rightarrow_j (h', s', t')$, meaning that in one reduction step, the thread j takes the state (h, s, t) to the new state (h', s', t') . However, to improve readability, we will write $(c, h, s, t) \rightarrow_j (c', h', s', t')$ as a shorthand notation for “ $(h, s, t) \rightarrow_j (h', s', t')$ and $t(j) = c$ and $t'(j) = c'$ ”.

Furthermore, a state (h, s, t) can also be *stuck* with respect to thread j , written $(h, s, t) \not\rightarrow_j$, or lead to an execution fault, written $(h, s, t) \rightarrow_j \text{abort}$.

The full definition can be found in figure 3.3.

3.3 The Assertion Language

In order to state properties of the programs we study, we need a number of predicates describing the states of locks, threads and memory cells, and ways to organise and combine them. This is the assertion language.

3.3.1 Grammar and Predicates

Our assertion language is built upon separation logic, with extra assertions dealing with locks and dynamic thread creation and synchronisation. In addition, the intrinsic circularity of storable locks is expressed via a fixed point operator μ .

$$\begin{aligned}
 \text{PredVar} & ::= p, q, r, \dots \\
 P & ::= E_1 = E_2 \mid E_1 \mapsto E_2 \mid P \wedge P \mid P \vee P \mid P * P \mid P \multimap P \mid \\
 & \quad \text{Locked}(E, P) \mid \text{Ex}(E, P) \mid \text{tid}(f, E) \mid \\
 & \quad r(E) \mid (\mu r. \lambda x. P)(E) (*) \\
 \text{Val} & = \text{Loc} \uplus \mathbb{Z} \uplus (\text{Val} \times \text{Val})
 \end{aligned}$$

In order to express lock properties, we use two predicates: $\text{Ex}(E, P)$ and $\text{Locked}(E, P)$. The first affirms that the interpretation of expression E points to a lock in the heap, with resource invariant P . No hypothesis is made as to whether the lock is available or not. $\text{Locked}(E, P)$ is similar, with the additional affirmation that the current thread owns the lock.

It is necessary for a thread to know about the existence of a lock before it can try to acquire it, but not to *know* that the lock is available. Similarly, the post-condition of a lock release simply says that the lock still exists, but not that it is available, in order to protect against interferences from other threads, which can now successfully acquire the lock. If we used an *Unlocked* predicate instead of Ex , we would face a serious issue: how can we know for certain that the lock which was indeed available in the past (at the moment it was released by whoever owned it) hasn't been acquired by another thread since?

The predicate $\text{tid}(f, E)$ expresses that the location E contains a thread handle, created when the active thread made a `fork` call. Furthermore, we wish

$$\begin{aligned}
& (\text{let } x = \text{new in } c, h, s, t) \longrightarrow_j (c, h[v \mapsto 0], s[x \mapsto v], t[j \mapsto c]) \\
& \quad \text{where } v \notin \text{dom}(h) \\
& (\text{let } x = [E] \text{ in } c, h, s, t) \longrightarrow_j (c, h, s[x \mapsto v], t[j \mapsto c]) \\
& \quad \text{where } h(\llbracket E \rrbracket_s) = v \\
& (\text{let } x = E \text{ in } c, h, s, t) \longrightarrow_j (c, h, s[x \mapsto \llbracket E \rrbracket_s], t[j \mapsto c]) \\
& \quad ([E] = F, h, s, t) \longrightarrow_j (\text{skip}, h[\llbracket E \rrbracket_s \mapsto \llbracket F \rrbracket_s], s, t[j \mapsto \text{skip}]) \\
& (\text{if } E \text{ then } c \text{ else } c', h, s, t) \longrightarrow_j (c, h, s, t[j \mapsto c]) \\
& \quad \text{if } \llbracket E \rrbracket_s = \text{True} \\
& (\text{if } E \text{ then } c \text{ else } c', h, s, t) \longrightarrow_j (c', h, s, t[j \mapsto c']) \\
& \quad \text{if } \llbracket E \rrbracket_s = \text{False} \\
& (\text{while } E \text{ do } c, h, s, t) \longrightarrow_j (\text{skip}, h, s, t[j \mapsto \text{skip}]) \\
& \quad \text{if } \llbracket E \rrbracket_s = \text{False} \\
& (\text{while } E \text{ do } c, h, s, t) \longrightarrow_j (c; \text{while } E \text{ do } c, h, s, t') \\
& \quad \text{if } \llbracket E \rrbracket_s = \text{True} \\
& (\text{skip}; c, h, s, t) \longrightarrow_j (c, h, s, t[j \mapsto c]) \\
& (c; c', h, s, t) \longrightarrow_j (c'; c', h', s', t'[j \mapsto c'; c']), \\
& \quad \text{if } (c, h, s, t) \longrightarrow_j (c', h', s', t') \\
& (\text{skip}, h, s, t) \not\rightarrow_j \\
& (\text{make_lock_P}(E), h, s, t) \longrightarrow_j (\text{skip}, h[\llbracket E \rrbracket_s \mapsto j], s, t[j \mapsto \text{skip}]) \\
& \quad \text{if } h(\llbracket E \rrbracket_s) = _ \\
& (\text{acquire}(E), h, s, t) \longrightarrow_j (\text{skip}, h[\llbracket E \rrbracket_s \mapsto j], s, t[j \mapsto \text{skip}]) \\
& \quad \text{if } h(\llbracket E \rrbracket_s) = 0 \\
& (\text{acquire}(E), h, s, t) \not\rightarrow_j \\
& \quad \text{if } h(\llbracket E \rrbracket_s) > 0 \\
& (\text{release}(E), h, s, t) \longrightarrow_j (\text{skip}, h[\llbracket E \rrbracket_s \mapsto 0], s, t[j \mapsto \text{skip}]) \\
& \quad \text{if } h(\llbracket E \rrbracket_s) = j \\
& (\text{let } x = \text{fork}(f) \text{ in } c, h, s, t) \longrightarrow_j (c, h[v \mapsto k], s[x \mapsto v], t[k \mapsto c', j \mapsto c]) \\
& \quad \text{if } v \notin \text{dom}(h), k \notin \text{dom}(t) \text{ and } f : c' \in \Gamma \\
& (\text{join}(E), h, s, t) \longrightarrow_j (\text{skip}, h, s, t[j \mapsto \text{skip}]|_{\text{dom}(t) - \{k\}}) \\
& \quad \text{if } h(\llbracket E \rrbracket_s) = k \text{ and } t(k) = \text{skip} \\
& (\text{join}(E), h, s, t) \not\rightarrow_j \\
& \quad \text{if } h(\llbracket E \rrbracket_s) = k \text{ and } t(k) \neq \text{skip} \\
& (h, s, t) \longrightarrow_j \text{abort in all other cases}
\end{aligned}$$

Figure 3.3: Operational Semantics

to remember the name of the procedure being executed by the child thread (so that we can look up its post-condition and check it when execution is finished). Whenever the active thread wants to synchronise with its child with a call to `join`, it will need to provide the thread handle.

The side condition (*) on the fixpoint operator stipulates that all occurrences of the predicate variable r in P are *guarded*, i.e. that they only appear under a *Locked* or *Ex* predicate. More formally, using the notation that $P[\]$ is a formula with a hole which can be filled by a predicate Q , written $P[Q]$:

Guardedness

A predicate variable r is guarded in P if there exists some P_1, E, Q such that $P = P_1[Ex(E, Q)]$ or $P = P_1[Locked(E, Q)]$ and $r \notin fv(P_1)$.

Additionally, we also have the usual intuitionistic logic rules and some additional rules for the assertion logic, such as

$$\begin{aligned} Locked(E, P) &\Rightarrow Locked(E, P) * Ex(E, P) \\ (\mu r. \lambda x. P)(E) &\Leftrightarrow P[(\mu r. \lambda x. P)/r, E/x] \end{aligned}$$

3.3.2 Proof Rules

We now present some of the proof rules for the specification logic, choosing to omit most of the structural rules. We need the following definitions: a predicate P is *thread-independent* if it does not contain any sub-predicates of the form $tid(f, E)$ or $Locked(E, P')$. Intuitively, the meaning of such a predicate will be the same for all threads in a given configuration, whereas, e.g., a $Locked(E, P')$ predicate is true for at most one of the threads. A predicate is *mobile* if it is thread-independent, precise, and does not contain separating implication \multimap .

The rules marked (*) have the following general provisos:

- All predicates P occurring in $Locked(E, P)$ or $Ex(E, P)$ in the rules are mobile.
- For every triple $\{R\}f\{S\}$ in the assumptions (i.e., on the left of “ \vdash ”), R is mobile while S is thread-independent and does not contain separating implication.

The precision requirements for the locks’ resource invariants and the preconditions of forkable procedures are necessary to prove soundness of, respectively, the `release` case and the frame rule, since there are two operations, `release` and `fork`, which perform heap splitting of the private space of the active thread, about which we need to be able to reason.

The predicates which are required to be thread-independent are precisely those describing parts of the heap that, conceptually speaking, is transferred between different threads. The meaning of these predicates needs to be the same before and after such a transfer.

The restrictions about uses of separating implication is more technical, and is required by our soundness proof (see Proposition 4.5.2 below).

$$\begin{array}{c}
\frac{\{P * x \mapsto 0\}c\{Q\}}{\{P\}\text{let } x = \text{new in } c\{Q\}} \quad x \notin fv(P) \cup fv(Q) \\
\\
\frac{\{P * E \mapsto x\}c\{Q\}}{\{\exists x, P * E \mapsto x\}\text{let } x = [E] \text{ in } c\{Q\}} \quad x \notin fv(Q) \\
\\
\frac{\{P \wedge x = E\}c\{Q\}}{\{P\}\text{let } x = E \text{ in } c\{Q\}} \quad x \notin fv(Q) \qquad \frac{}{\{[E \mapsto _]\}[E] = F\{[E \mapsto F]\}} \\
\\
\frac{\{E = \text{true} \wedge P\}c\{Q\} \quad \{E = \text{false} \wedge P\}c'\{Q\}}{\{P\}\text{if } E \text{ then } c \text{ else } c'\{Q\}} \\
\\
\frac{\{E = \text{true} \wedge P\}c\{P\}}{\{P\}\text{while } E \text{ do } c\{P \wedge E = \text{false}\}} \qquad \frac{}{\{P\}\text{skip}\{P\}} \\
\\
\frac{\{P\}c\{Q\} \quad \{Q\}c'\{R\}}{\{P\}c; c'\{R\}} \qquad \frac{}{\{E \mapsto _ \}\text{makelock_P}(E)\{Locked(E, P)\}} \quad (*) \\
\\
\frac{}{\{Ex(E, P)\}\text{acquire}(E)\{Locked(E, P) * P\}} \quad (*) \\
\\
\frac{}{\{Locked(E, P) * P\}\text{release}(E)\{Ex(E, P)\}} \quad (*) \\
\\
\frac{\Gamma, \{R\}f\{S\} \vdash \{P * tid(f, x)\}c\{Q\}}{\Gamma, \{R\}f\{S\} \vdash \{P * R\}\text{let } x = \text{fork}(f) \text{ in } c\{Q\}} \quad (*) \\
\\
\frac{}{\Gamma, \{R\}f\{S\} \vdash \{tid(f, E)\}\text{join}(E)\{S\}} \quad (*) \\
\\
\frac{f : \{R\}c'\{S\} \in \Gamma \quad \Gamma \vdash \{R\}c'\{S\} \quad \Gamma \vdash \{P\}c\{Q\}}{\vdash \text{let } f : \{R\}c'\{S\} \text{ in } \{P\}c\{Q\}} \quad (*)
\end{array}$$

Figure 3.4: Proof Rules

3.4 Example: Lock Coupling Lists

The lock coupling algorithm can be considered as a seminal example for storable locks. It is for instance treated in a very similar fashion by Gotsman et al. [GBC⁺07]. The idea is to allow multiple threads to simultaneously add and remove elements from an ordered singly linked list. Instead of having a single general lock for the entire list, which would require all threads to wait for the active one to finish its operation, there is one lock per node. By requiring threads to hold the lock for both the current node and the previous one whenever they search through the list or try to modify it, we can ensure that the data structures are never in an incoherent state, while the fine-grained locks allow for a great gain in efficiency.

Each node contains three fields: a lock protecting the entire node, with resource invariant P , a data field `val` containing an integer and a pointer to the next node or `nil`. We also use the convention that the head of the list always contains the value $-\infty$ and the tail $+\infty$. We present here code for initialising the list and for adding a new node with value e in the correct location in the heap. A procedure to remove a node with value e would be very similar.

Since the only way to know about the existence of a particular lock is by acquiring its predecessor in the list, the invariant itself enforces the particular locking procedure for iterating through a list. We present below a simplified version of the annotated proof for `add`.

The resource invariant is a recursively defined predicate:

$$P(n, e) = (\mu r. \lambda(n, e). [(n.val \mapsto e \wedge e = \infty) \vee (\exists x, e', n.val \mapsto e * n.next \mapsto x * Ex(x.lock, r(x, e')) \wedge e < e')])](n, e)$$

```

initialise() {
  let tail = new NODE in
  [tail.val] = +∞;
  [tail.next] = NULL;
  makelockP(tail, +∞)(tail.lock);
  release(tail.lock);
  let head = new NODE in
  [head.val] = -∞;
  [head.next] = tail;
  makelockP(head, -∞)(head.lock);
  release(head.lock);
}

add(e) {
  Ex(head.lock, P(head, -∞))
  let prev = head in
  acquire(prev.lock);
  Locked(prev.lock, P(prev, -∞)) * P(prev, -∞)
  let curr = prev.next in

```

```

acquire (curr.lock);
 $\exists v', \text{Locked}(\text{prev.lock}, P(\text{prev}, -\infty)) *$ 
 $\text{prev.val} \mapsto -\infty * \text{prev.next} \mapsto \text{curr} *$ 
 $\text{Locked}(\text{curr.lock}, P(\text{curr}, v')) * P(\text{curr}, v') \wedge$ 
 $-\infty < v'$ 
  while (curr.val < e) do {
 $\exists x, v, v', v'', \text{Locked}(\text{prev.lock}, P(\text{prev}, v)) *$ 
 $\text{Locked}(\text{curr.lock}, P(\text{curr}, v')) * \text{prev.val} \mapsto v *$ 
 $\text{prev.next} \mapsto \text{curr} * \text{curr.val} \mapsto v' * (v' = \infty \vee$ 
 $(\text{curr.next} \mapsto x * \text{Ex}(x.\text{lock}, P(x, v'')) \wedge v' < v''))$ 
 $\wedge v' < e \wedge v < v'$ 
    release(prev.lock);
    let prev = curr in
    let curr = curr.next in
    acquire(curr.lock);
  }
 $\exists x, v, v', v'', \text{Locked}(\text{prev.lock}, P(\text{prev}, v)) *$ 
 $\text{Locked}(\text{curr.lock}, P(\text{curr}, v')) * \text{prev.val} \mapsto v *$ 
 $\text{prev.next} \mapsto \text{curr} * \text{curr.val} \mapsto v' * (v' = \infty \vee$ 
 $(\text{curr.next} \mapsto x * \text{Ex}(x.\text{lock}, P(x, v'')) \wedge v' < v''))$ 
 $\wedge v < e \leq v'$ 
  if (curr.val != e) {
    let n = new NODE in
      [n.val] = e;
      [n.next] = curr;
      makelock $P(n, e)$ (n.lock);
 $\exists x, v, v', \text{Locked}(\text{prev.lock}, P(\text{prev}, v)) *$ 
 $\text{Locked}(n.\text{lock}, P(n, e)) *$ 
 $\text{Locked}(\text{curr.lock}, P(\text{curr}, v')) *$ 
 $\text{prev.val} \mapsto v * n.\text{val} \mapsto e * n.\text{next} \mapsto \text{curr} *$ 
 $\text{curr.val} \mapsto v' * \wedge v < e < v'$ 
      release(n.lock);
      [prev.next] = n;
  }
  release(prev.lock);
  release(curr.lock);
 $\exists v, v', \text{Ex}(\text{prev.lock}, P(\text{prev}, v)) *$ 
 $\text{Ex}(\text{curr.lock}, P(\text{curr}, v')) * ((v' = e \wedge v < v')$ 
 $\vee (\exists n, \text{Ex}(n.\text{lock}, P(n, e)) \wedge v < e < v'))$ 
}

```


Chapter 4

A Model for Storable Locks

With the “knots in the store”-generated recursive domain equations explicit and our setting defined, we are now ready to provide a solution to the equations and build a model around them.

This chapter is divided in two parts: first, we present important notions related to ultra-metric spaces and uniform predicates, which will provide a solution to the recursive domain equations. Using this step-indexed solution, we then build a Kripke model for interpreting our assertion language.

One way to see the ultra-metric spaces approach is as a very traditional step-indexing way of solving recursive equations, but with the actual indices and induction abstracted away – in a way, we are simply hiding the low-level machinery.

4.1 Ultra-Metric Spaces

In this section, we recall some basic definitions and useful theorems about metric spaces.

Metric Space

A *metric space* is a set X and a function $d : X \times X \rightarrow \mathbb{R}^+$. In order to be called a *distance*, d needs to satisfy three properties:

- (i) : $\forall x, y : X, d(x, y) = 0 \Leftrightarrow x = y$
- (ii) : $\forall x, y : X, d(x, y) = d(y, x)$
- (iii) : $\forall x, y, z : X, d(x, z) \leq d(x, y) + d(y, z)$

This corresponds to the intuitive notion of Euclidian distance and is straightforward to visualise.

Ultrametricity

A metric space is *ultrametric* if it satisfies the stronger ultrametric inequality (iii') rather than the triangle inequality (iii):

$$(iii') : \forall x, y, z : X, d(x, z) \leq \max(d(x, y), d(y, z))$$

This notion of ultrametric distance is much more difficult to visualise. It can be helpful to think of it not as a distance between points in a space, but rather as the *degree of similarity* between two elements.

1-boundedness

A metric space is *1-bounded* if $\forall x, y : X, d(x, y) \leq 1$. A distance of 0 then means equality and a distance of 1 total dissimilarity.

Bisectedness

A metric space is *bisected* if $\forall x, y : X, d(x, y) \neq 0 \Rightarrow \exists n \in \mathbb{N}, d(x, y) = 2^{-n}$.

It follows naturally that all bisected metric spaces are 1-bounded.

A useful notation on bisected spaces is $x =^n y$, meaning that $d(x, y) \leq 2^{-n}$. On ultrametric spaces, the intuitive meaning is that x and y are similar to at least the n -th degree.

Cauchy Sequence

A Cauchy sequence in a metric space (X, d) is a sequence $(x_n)_{n \in \mathbb{N}}$ such that for any $\varepsilon > 0$, there exists some $N \in \mathbb{N}$ such that $d(x_m, x_n) < \varepsilon$ for any $m, n \geq N$.

Limit

The limit of a sequence $(x_n)_{n \in \mathbb{N}}$ is an element $x \in X$ such that for any $\varepsilon > 0$, there exists $N \in \mathbb{N}$ such that $d(x_n, x) < \varepsilon$ for any $n \geq N$.

Completeness

A metric space (X, d) is said to be complete if every Cauchy sequence has a limit.

Non-expansive and contractive functions

A function f between metric spaces (X_1, d_1) and (X_2, d_2) is *non-expansive* if $\forall x, y : X, d_2(f(x), f(y)) \leq d_1(x, y)$.

Furthermore, f is *contractive* if there exists some $c < 1$ such that $\forall x, y : X, d_2(f(x), f(y)) \leq c \cdot d_1(x, y)$.

One of the central result in metric spaces is *Banach's Fixed Point Theorem* [Ban22]:

Theorem 4.1.1 *In a complete, non-empty metric space (X, d) , any contractive function $f : X \rightarrow X$ has a unique fixpoint $x = f(x)$.*

The next step is to define a categorical structure for complete ultra-metric spaces.

CBUlt

The category *CBUlt* (**C**omplete **B**ounded **U**ltra-metric spaces) has bounded, complete, ultra-metric spaces as objects and non-expansive functions as morphisms. Furthermore, it is cartesian closed [WBR94].

Locally contractive functors

A functor $F : \text{CBUlt}^{op} \times \text{CBUlt} \rightarrow \text{CBUlt}$ is *locally contractive* if there exists $c < 1$ such that

$$d(F(f, g), F(f', g')) < c \cdot \max(d(f, f'), d(g, g'))$$

for any non-expansive functions f, f', g, g' .

Finally, we can formulate the main theorem we use to solve our recursive domain equation. A detailed proof of the original domain theoretical version can be found in [AR89], while translations to ultra-metric spaces are found in e.g. [BST10].

Theorem 4.1.2 *Let $F : \text{CBUlt}^{op} \times \text{CBUlt} \rightarrow \text{CBUlt}$ be a locally contractive functor satisfying that $F(1, 1) \neq \emptyset$. There exists a unique (up to isomorphism) non-empty $(X, d) \in \text{CBUlt}$ such that $F((X, d), (X, d)) \cong (X, d)$.*

4.2 Expressing the Recursive Domain Equations

4.2.1 Worlds

As described in Chapter 2, the main issue with storable locks originates from the fact that the resource invariants depend on “worlds” which contain locks and resource invariants. The worlds can be seen as a semantic layer above the heap, containing additional information about the state of the machine. In our case, in order to be able to formulate soundness of the logic, we want to keep track of three different kinds of information:

- Whenever a memory cell is a lock, we want access to its resource invariant.
- Whenever a memory cell is a thread handle (the value returned by a call to `fork`), we want to remember the name of the procedure we are executing, so that we can look up its pre and post-conditions in the context. This is useful when, for instance, we want to check that a thread which is done executing really satisfies its post-condition.

- If a memory cell is “regular” (i.e. not a lock or a thread handle), we don’t need to remember any other information than the fact that it does exist.

This is formulated in the following equation:

$$W = Loc \multimap^{fin} (Fun + Cell + Pred) \quad (4.1)$$

Here Loc represents locations in the heap and is modelled simply by \mathbb{N} , Fun is a static set of procedure names, and $Cell$ is a singleton set.

For clarity, we will use the constructor-style notations $T(f)$ and $Lock(Q)$ to refer to the first and third components of the $Fun + Cell + Pred$ sum type.

4.2.2 Heaps

One particular issue arises when we try to model thread-local heaps: since threads are allowed to share parts of the memory – this is, after all, the very reason we need locks – it becomes impossible to use the traditional separating conjunction to combine several sub-heaps, since their domain will potentially have a non-empty intersection.

The usual solution, found for instance in CSL, is to separate the heaps in private parts, local to each thread and invisible from others, and a big shared heap that everyone is allowed to read and write in. We choose an alternative, more fine-grained solution: allow duplicate values of the shared memory cells in the relevant memory cells and modifying the separating conjunction operator to allow some overlapping on locations which correspond to locks.

This allows to limit the knowledge of the existence of a lock to only the threads which have acquired this knowledge explicitly. Interestingly, this follows exactly the scope of the lock variables.

Semantic Heaps

Heaps \hat{H} in our semantic layer will be identical to the concrete ones used in the operational semantics, with one notable exception: we will use an *unknown* value \mathcal{U} to denote that no assumption is made as to the exact state of a lock.

$$\hat{H} = Loc \rightarrow_{fin} Val \cup \{\mathcal{U}\}$$

We can then redefine the separating conjunction, first on individual memory cells:

Redefined separating conjunction

$$\begin{aligned} \mathcal{U} * \mathcal{U} &= \mathcal{U} \\ \mathcal{U} * j &= j \\ j * \mathcal{U} &= j \\ j * k &\text{ undefined} \end{aligned}$$

for any $j, k \in Val$.

On full heaps:

Redefined separating conjunction on full heaps

$h_1 * h_2$ is well defined if and only if for all $l \in dom(h_1) \cap dom(h_2)$, $h_1(l) * h_2(l)$ is well defined.

$\forall l \in dom(h_1) \cap dom(h_2), (h_1 * h_2)(l) = h_1(l) * h_2(l)$, while $(h_1 * h_2)(l)$ is defined as for regular separating conjunction when $l \notin dom(h_1) \cap dom(h_2)$.

We also use an operation to merge a tuple of heaps into a single one: $\overline{hp} = \bigstar_{i \in dom(hp)} hp(i)$.

Since, because of interferences, threads can only be certain of the state of a lock when they actually own it, there is a simple way to map full abstract heaps to concrete ones: simply replace all unknown values by 0, as the absence of any claims to hold the lock means that it is available.

Projection into concrete heaps

$$|\cdot| : \hat{H} \mapsto H$$

$$|h| = \lambda x. \begin{cases} 0 & \text{if } h(x) = \mathcal{U} \\ h(x) & \text{otherwise} \end{cases}$$

4.2.3 Predicates

The one remaining set to define is *Pred*, the type of lock resource invariants. A predicate should be a function that takes a world (containing semantic information about the current state) and returns a set of heaps, i.e., those heaps that “satisfy the predicate.”

Roughly, we want something like:

$$Pred = W \rightarrow \mathcal{P}(\hat{H}). \quad (4.2)$$

We thus obtain the following system of equations which require solving (note the similarity with the equations of figure 2.6).

$$W = Loc \xrightarrow{fin} (Fun + Cell + Pred)$$

$$Pred = W \rightarrow \mathcal{P}(\hat{H})$$

4.3 Solving the Recursive Domain Equations

The first step toward solving the equations is to define a proper distance on the various spaces we are manipulating, and in particular $\mathcal{P}(\hat{H})$, which we turn

into the set of *uniform predicates*, $UPred$, by combining heaps with downwards closed integer indices.

Uniform Predicates

$$UPred(\hat{H}) = \left\{ P \in \mathcal{P}(\hat{H} \times \mathbb{N}) \mid \forall h \in \hat{H}, \forall k \in \mathbb{N}, \forall j \leq k, P(h, k) \Rightarrow P(h, j) \right\}$$

We define the following distance on $UPred(\hat{H})$. It is well defined because of the downward closed character of the index parameter in $UPred$.

$$d(P, Q) = \begin{cases} 0 & \text{if } P = Q \\ \min\{2^{-n} \mid \forall k \leq n, \forall h \in \hat{H}, P(h, k) \Leftrightarrow Q(h, k)\} & \text{otherwise.} \end{cases}$$

Let T be a fixed set. We want to solve the following equation in $CBUlt$:

$$X \cong \frac{1}{2} \left((\mathbb{N} \multimap^{fin} (T + X)) \rightarrow_n UPred(\hat{H}) \right) \quad (4.3)$$

Here the function space \rightarrow_n consists of those functions p which are non-expansive and satisfy the following property: If $(h, n) \in p(w)$, then $\text{dom}(h) \subseteq \text{dom}(w)$. This last, non-standard requirement is useful for proving locality of the predicates, a crucial point for the soundness of the `fork` case.

The $\frac{1}{2}$ factor is there to make sure the functions are all contractive, a requirement to apply most of the theorems of section 4.1. It is a very common technique found in virtually all works using such methods.

In order to do solve this equation, we must first specify which metric is used on the right-hand side, assuming that (X, d_X) is a given object of $CBUlt$. This metric is obtained from the following building blocks. We equip the set T with the discrete metric, i.e., every pair of distinct elements has distance 1. The operators \times and \rightarrow_n are given by the cartesian closed structure of $CBUlt$ (here \rightarrow_n is a closed subspace of the exponential), while $+$ is the co-product (which gives elements from different summands the maximal distance, 1). More details about these operators can be found in, e.g., [BST10]. It only remains to explain the “finite partial function space” operator \multimap^{fin} . Concretely, the distance function d on $\mathbb{N} \multimap^{fin} (T + X)$ is given by

$$d(P, Q) = \begin{cases} 1 & \text{if } \text{dom}(P) \neq \text{dom}(Q) \\ \max_{i \in \text{dom}(P)} (d_{T+X}(P(i), Q(i))) & \text{otherwise.} \end{cases}$$

We then only have to use theorem 4.1.2 to find a solution X to equation 4.3.

If we rename \mathbb{N} to Loc , T to $T(Fun) + Cell$ and X to \widehat{Pred} , we obtain a solution to the following equation:

$$\begin{aligned} W &= Loc \multimap^{fin} (T(Fun) + Cell + Lock(\widehat{Pred})) \\ Pred &= W \rightarrow_n UPred(\hat{H}) \\ \widehat{Pred} &\cong \frac{1}{2}Pred \end{aligned}$$

Let App be the isomorphism above, and let Abs be its inverse:

$$\begin{aligned} App : \widehat{Pred} &\rightarrow \frac{1}{2}Pred \\ Abs : \frac{1}{2}Pred &\rightarrow \widehat{Pred}. \end{aligned}$$

World composition (of elements w and w' of W) is defined as follows: $w * w'$ is defined if the domains of w and w' do not overlap, and in this case $w * w'$ is the union of the two partial functions w and w' .

We can then prove that $Pred$ models the usual separation logic operations, which is summarised by saying it is equipped with a complete BI-algebra structure. First, for every $w \in W$ we define $UPred_w(\hat{H})$ to be the subset of $UPred(\hat{H})$ where all heaps have domain contained in $\text{dom}(w)$. This set is given a BI-algebra structure as follows. We define $h_1 \perp_w h_2$ to hold if $h_1 * h_2$ is defined, and if furthermore $w(l) = Lock(-)$ for all $l \in \text{dom}(h_1) \cap \text{dom}(h_2)$. Then we define $*_w$ on heaps in the same way as $*$, but with additional requirement that $h_1 *_w h_2$ is only defined if $h_1 \perp_w h_2$. (The intuition is that only values which correspond to locks can be shared between heaps.) Now the BI-algebra structure on $UPred_w(\hat{H})$ is given as follows: \emptyset is \perp , $\{h \mid \text{dom}(h) \subseteq \text{dom}(w)\} \times \mathbb{N}$ is \top , and set-theoretic union and intersection are, respectively, join and meet. The remaining three operations are defined by:

$$\begin{aligned} (h, n) \in P \rightarrow Q &= \forall k \leq n, (h, k) \in P \Rightarrow (h, k) \in Q \\ (h, n) \in P * Q &= \exists h_1, h_2, h = h_1 *_w h_2 \wedge (h_1, n) \in P \wedge (h_2, n) \in Q \\ (h, n) \in P \multimap Q &= \forall k \leq n, \forall h' \perp_w h, (h', k) \in P \Rightarrow (h *_w h', k) \in Q \end{aligned}$$

These operations are then extended pointwise to $Pred$, using the fact that, by construction, every element of $Pred$ maps any world w into $UPred_w(\hat{H})$ (and not just into $UPred(\hat{H})$).

Lemma 4.3.1 *$Pred$ equipped with these operations is a complete BI-algebra on Set [BBTS07].*

Proof We start by proving that $UPred(\hat{H})$ is a complete BI-algebra, using \emptyset as \perp , $\hat{H} \times \mathbb{N}$ as \top , and set-theoretic union and intersection as, respectively, join and meet.

The remaining three operations are defined by:

$$\begin{aligned} (h, n) \in P \rightarrow Q &= \forall k \leq n, (h, k) \in P \Rightarrow (h, k) \in Q \\ (h, n) \in P * Q &= \exists h_1, h_2, h = h_1 * h_2 \wedge (h_1, n) \in P \wedge (h_2, n) \in Q \\ (h, n) \in P \multimap Q &= \forall k \leq n, \forall h' \perp h, (h', k) \in P \Rightarrow (h * h', k) \in Q \end{aligned}$$

All the necessary properties on these operations are easily checked. We conclude by using a pointwise lifting of $UPred(\hat{H})$ to $W \rightarrow_n UPred(\hat{H})$.

4.4 Building the Kripke Model

Having solved the recursive domain equations, we can now define interpretations for our assertion language. The interpretation of a predicate P , written $\llbracket P \rrbracket_s^k$, takes as arguments a stack s and the identifier k of the current thread, returning an object in $Pred$.

Stacks can store either plain values or predicates with an optional argument:

$$Stack = (Var \rightarrow_{fin} Val) \uplus (PredVar \rightarrow_{fin} (Val \rightarrow Pred))$$

The interpretation of nearly all predicates can be found in figure 4.1.

$$\begin{aligned} \llbracket Ex(E, P) \rrbracket_s^k &= \lambda w. \{ (h, n) \mid \exists Q. w(\llbracket E \rrbracket_s) = Lock(Q) \wedge h = \llbracket E \rrbracket_s \mapsto \mathcal{U} \wedge \\ &\quad \forall w'. App(Q)(w') =^{n-1} \llbracket P \rrbracket_s^k(w') \} \\ \llbracket Locked(E, P) \rrbracket_s^k &= \lambda w. \{ (h, n) \mid \exists Q. w(\llbracket E \rrbracket_s) = Lock(Q) \wedge h = \llbracket E \rrbracket_s \mapsto k \wedge \\ &\quad \forall w'. App(Q)(w') =^{n-1} \llbracket P \rrbracket_s^k(w') \} \\ \llbracket tid(f, E) \rrbracket_s^k &= \lambda w. \{ (h, n) \mid w(\llbracket E \rrbracket_s) = T(f) \wedge \exists j \neq k. h = \llbracket E \rrbracket_s \mapsto j \} \\ \llbracket r(E) \rrbracket_s^k &= s(r)(\llbracket E \rrbracket_s) \\ \llbracket (\mu r. \lambda x. P)(E) \rrbracket_s^k &= fix \left(\lambda P_1^{Val \rightarrow Pred} \lambda v. \llbracket P \rrbracket_{s[r \mapsto P_1, x \mapsto v]}^k \right) (\llbracket E \rrbracket_s) \\ \llbracket P * Q \rrbracket_s^k &= \lambda w. \{ (h, n) \mid \exists h_1, h_2. h = h_1 *_w h_2 \wedge \\ &\quad (h_1, n) \in \llbracket P \rrbracket_s^k(w) \wedge (h_2, n) \in \llbracket Q \rrbracket_s^k(w) \} \\ \llbracket E_1 \mapsto E_2 \rrbracket_s^k &= \lambda w. \{ (h, n) \mid w(\llbracket E_1 \rrbracket_s) = Cell \wedge h = \llbracket E_1 \rrbracket_s \mapsto \llbracket E_2 \rrbracket_s \} \end{aligned}$$

Figure 4.1: *Predicate Interpretation*

The most interesting point to note is how the interpretation of the two lock predicates Ex and $Locked$ refers to $(n - 1)$ -equality of predicates instead of “true” equality. This is necessary by the non-expansiveness requirement of $W \rightarrow_n UPred(\hat{H})$ in the domain equation, while the -1 part simply reflects the $\frac{1}{2}$ factor in the final isomorphism.

4.5 Locality Properties

Theorem 4.5.1 *For all P , s , and k , the predicate $\llbracket P \rrbracket_s^k$ is well-defined.*

Proof Using the fact that recursion can only happen in guarded environments, the interpretation of $(\mu r. \lambda x. P)(E)$ uses the fixpoint of a contractive function on $Val \rightarrow Pred$, which is in $CBUIt$ if we view Val as a discrete metric space.

Semantic predicates which are definable by syntactic predicates not containing \rightarrow^* satisfy a certain property which is crucial for our soundness proof:

Local predicates

A predicate $p \in Pred$ is *local* if for all h , n , and w ,

$$(h, n) \in p(w) \iff (h, n) \in p(w|_{dom(h)}).$$

Proposition 4.5.2 *For every syntactic predicate P containing no free predicate variables and no occurrences of \rightarrow^* , the semantic predicate $\llbracket P \rrbracket_s^k$ is local.*

Proposition 4.5.3 *If p is local, then p is monotone: $(h, n) \in p(w)$ implies that $(h, n) \in p(w * w')$ whenever $w * w'$ is defined.*

Chapter 5

Soundness

5.1 Logical View of the Machine State

With the model presented in chapter 4, we gave a clear semantic meaning to assertions (section 4.4) and built a logical layer on top of the basic operational view of the machine (section 4.2).

In order to have a notion of ownership, allowing us to reason with resource invariants, let us now go one step further and define an explicit thread-local splitting of the logical machine state. We will now have a global world and stack, as well as command, heap and post-condition pools.

Semantic Configuration

A *configuration* is a tuple (w, hp, s, tp, qp) where for some $n \in \mathbb{N}$ we have:

$$\begin{aligned} w &\in W \\ hp &: \{1, \dots, n, \Omega\} \rightarrow \hat{H} \\ s &\in Stack \\ tp &: \{1, \dots, n\} \rightarrow Cmd \\ qp &: \{1, \dots, n\} \rightarrow Pred \end{aligned}$$

Each of the pools hp , tp and qp map thread identifiers to, respectively, sub-heaps, command continuation and post-condition. In addition, there is a space $hp(\Omega)$ which corresponds to the heaps described by the invariants of available locks. A successful lock acquisition will then correspond to the transfer of a sub-heap owned by Ω into the private space of the new owner thread, with release being the reverse operation. Similar heap swaps occur with fork and join operations.

Finally, the intended semantic properties of the total heap \overline{hp} are described by the world w , and each predicate $qp(j)$ is a post-condition that the private heap of thread j must satisfy when that thread is done executing.

As we discussed in the presentation of the language in figure 3.2, a complete program will first define a number of sub-procedures, then the code of a main procedure which can make `fork` calls to create new threads executing any of the previously defined procedures. The important thing to note here is that the forkable code is *static* and *pre-defined*. We store a number of important pieces of information in the context Γ , such as the actual code and the post-conditions corresponding to each procedure name.

However, because it does not change over time, and because the notations are already very heavy, we have chosen to omit Γ from semantic states, except when it is actually relevant.

5.2 Logical Transitions, Future Worlds

We next define a transition relation on semantic configurations. This relation follows the standard operational semantics of section 3.3 but also keeps track of heap transfers and world updates.

Transition relation

The relation

$$(w, hp, s, tp, qp) \xrightarrow{n}_j (w', hp', s', tp', qp')$$

holds iff there exists a reduction $(|\overline{hp}|, s, tp) \rightarrow_j (h', s', tp')$ such that $|\overline{hp}'| = h'$ and $hp'(k) = hp(k)$ for all $k \notin \{j, \Omega\}$. Furthermore, letting $c = tp(j)$:

- If c is not a reference allocation, lock operation, or thread operation, then $w' = w$, $qp' = qp$, and $hp'(\Omega) = hp(\Omega)$.
- If c is `let x = new in C'`, then $w' = w[v \mapsto Cell]$, $qp' = qp$, and $hp'(\Omega) = hp(\Omega)$.
- If c is `make_lock_P(E)`, then $w(\llbracket E \rrbracket_s) = Cell$, $qp' = qp$, $hp'(\Omega) = hp(\Omega)$ and $w' = w[\llbracket E \rrbracket_s \mapsto Lock(Abs(\llbracket P \rrbracket_s^j))]$.
- If c is `acquire(E)`, then there exists Q such that $w(\llbracket E \rrbracket_s) = Lock(Q)$ and $hp(j)(\llbracket E \rrbracket_s) = \mathcal{U}$. Furthermore, $w' = w$, $qp' = qp$, and there exists $(h_0, n) \in App(Q)(w)$ such that $hp(\Omega) = h_0 *_w h_1$ while $hp'(\Omega) = h_1$ and $hp'(j) = h_0 *_w (hp(j)(\llbracket E \rrbracket_s \mapsto j))$. (Notice that here the superscript “ n ” on the relation symbol is used.)
- If c is `release(E)`, then there exists Q such that $w(\llbracket E \rrbracket_s) = Lock(Q)$ and $hp(j)(\llbracket E \rrbracket_s) = j$. Furthermore, $w' = w$, $qp' = qp$, and there exists $(h_0, n) \in App(Q)(w)$ such that $hp(j) = h_0 *_w h_1$ while $hp'(j) = h_1[\llbracket E \rrbracket_s \mapsto \mathcal{U}]$ and $hp'(\Omega) = h_0 *_w hp(\Omega)$.

- If c is `let x = fork(f)` in M , then $w' = w[v \mapsto T(f)]$ where $s' = s[x \mapsto v]$. Furthermore, $qp' = qp[k \mapsto \llbracket Q \rrbracket^k]$ where $\{P\}f\{Q\} \in \Gamma$ and $k \in \text{dom}(tp') \setminus \text{dom}(tp)$. Also, $hp'(\Omega) = hp(\Omega)$, and there exist h_1, h_2 , such that $hp(j) = h_1 *_{\omega} h_2$ and $(h_2, n) \in \llbracket P \rrbracket^k(w)$, while $hp'(j) = h_1[v \mapsto k]$ and $hp'(k) = h_2$.
- Finally, if c is `join(E)`, then $w(\llbracket E \rrbracket_s) = T(f)$ for some f . Also, $hp(j) = h_1 *_{\omega} [\llbracket E \rrbracket_s \mapsto k]$ where $k \in \text{dom}(tp) \setminus \text{dom}(tp')$. Furthermore, $w' = w|_{\text{dom}(w) \setminus \{\llbracket E \rrbracket_s\}}$, $hp'(\Omega) = hp(\Omega)$, and $hp'(j) = h_1 *_{\omega} hp(k)$.

The extra index n over which this relation is indexed serves as a maximal step counter whenever a predicate is interpreted, since they are only valid “up to” some number of steps.

5.3 Safety, Consistency and Correctness

In order to prove soundness, we make use of a method first developed by Viktor Vafeiadis [Vaf11] which relies on the use of a *safety predicate* stipulating what a correct configuration should look like and being maintained by logical transitions.

In our case, the safety predicate stipulates that, up to a given number of steps, the configuration d it considers will be semantically *correct*. Intuitively, this means the following:

- d is consistent.
- d cannot reduce to `abort` in one step.
- The private heap of any thread which has finished execution (i.e. it only needs to execute `skip`) satisfies its post-condition.
- If d reduces to another configuration d' , then d' is safe up to a smaller number of steps. (“Preservation”)
- If the configuration corresponding to d reduces in the standard operational semantics then d reduces. (“Progress”)

The notion of *consistency* is a weaker form of correctness which only performs basic checks at a given stage, without considering further reductions.

Consistency

A configuration $d = (w, hp, s, tp, qp)$ is *consistent*, which is written $\text{cons}(d)$, if the following properties hold:

- $\text{dom}(w) = \text{dom}(\overline{hp})$.

- For every l and all $j \neq k$, if $l \in \text{dom}(hp(j)) \cap \text{dom}(hp(k))$, then $w(l) = \text{Lock}(-)$.
- For every j , the predicate $qp(j)$ is local.
- For every l , if $w(l) = \text{Cell}$, then there exists j such that $l \in \text{dom}(hp(j))$ and $hp(j)(l) \neq \mathcal{U}$.
- For every l , if $w(l) = \text{Lock}(p)$, then the predicate p is local.
- For every l , if $w(l) = T(f_i)$, then $\exists j, k$ such that $l \in \text{dom}(hp(j))$ and $hp(j)(l) = k$ and $qp(k) = \llbracket Q_i \rrbracket^k$. Furthermore, $j \neq k$. Here $\{P_i\}f_i\{Q_i\}$ comes from the global environment Γ .

Theorem 5.3.1 *If $\text{cons}(d)$ and $d \xrightarrow{r}_j d'$, then $\text{cons}(d')$.*

Proof By induction.

Finally, the formal definition of safety is the following:

Safety

Let $d = (w, hp, s, tp, qp)$. The predicate $\text{safe}_n(d)$ is defined by induction on n as follows. $\text{safe}_0(d)$ always holds, and $\text{safe}_{n+1}(d)$ holds iff:

- d is consistent, i.e., $\text{cons}(d)$ holds.
- $\forall j \in \text{dom}(tp), \forall h$ such that $h \perp \overline{hp}$, there is no reduction of the form $(\overline{hp} * h |, s, tp) \rightarrow_j \text{abort}$.
- $\forall j \in \text{dom}(tp). tp(j) = \text{skip} \implies (hp(j), n) \in qp(j)(w)$
- If $d \xrightarrow{m}_j d'$, then $\text{safe}_{\min(n,m)}(d')$.
- If $(\overline{hp} |, s, tp) \rightarrow_j (h', s', tp')$, then $d \xrightarrow{r}_j d'$ for some d' .

5.4 Soundness

It remains to define what semantic validity of a Hoare triple $\{P\}c\{Q\}$ really means. Intuitively, it could be defined as: if one executes c in a state satisfying $\llbracket P \rrbracket$, then execution will not abort and will eventually finish in a state satisfying $\llbracket Q \rrbracket$. This can be neatly factored into the following:

$$\forall n \in \mathbb{N}, \forall w, s, \forall (h, n) \in \llbracket P \rrbracket_s, \text{safe}_n(w, [1 \mapsto h], s, [1 \mapsto c], [1 \mapsto \llbracket Q \rrbracket_s])$$

This, however, is too limited a definition: it only permits execution in a non-concurrent setting where a single thread is executing. In order to allow

concurrent execution while keeping interference under control, a very elegant solution is to allow composition of this simple, unique thread state executing c with any safe state (up to some index).

This allows the complete abstraction of the interferences between different threads, where the non-deterministic character of the operational semantics reductions is confined to the safety predicate.

We can now define semantic validity of Hoare triples.

Semantic validity

$$\Gamma \models_j \{P\}c\{Q\}$$

is defined as

$$\forall m < j, \forall w, hp, s, tp, qp,$$

if $(\forall i \in \text{dom}(\Gamma), \Gamma \models_m \{P_i\}c_i\{Q_i\} \wedge \text{safe}_m(w, hp, s, tp, qp))$ then

$\forall k \notin \text{dom}(hp), \forall w'$ such that $w * w'$ is well defined,

$$\forall (h, n) \in \llbracket P \rrbracket_s^k(w * w'),$$

$$(h \perp \overline{hp} \wedge \text{cons}(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto \llbracket Q \rrbracket^k])) \Rightarrow$$

$$\text{safe}_{\min(m, n)}(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto \llbracket Q \rrbracket^k]).$$

This can be extended to full programs in a straightforward way:

Semantic validity on full programs

$$\models_j \text{let } f_i : \{P_i\}c_i\{Q_i\} \text{ in } \{P\}c\{Q\}$$

is defined as

$$\forall i, \Gamma \models_j \{P_i\}c_i\{Q_i\} \wedge \Gamma \models_j \{P\}c\{Q\}$$

Finally, soundness is formulated in the usual way, with the elimination of the step indices.

Theorem 5.4.1 (Soundness)

$$\begin{aligned} & \vdash \text{let } f_i : \{P_i\}c_i\{Q_i\} \text{ in } \{P\}c\{Q\} \Rightarrow \\ & \forall j, \models_j \text{let } f_i : \{P_i\}c_i\{Q_i\} \text{ in } \{P\}c\{Q\} \end{aligned}$$

Proof (sketch) We prove soundness separately for each proof rule, in each case using strong induction on the index of the safety predicate. The proofs are relatively straightforward, consisting mostly in unfolding various definitions, considering all reduction steps that can be taken from the initial state and then proving safety of the new state.

Using lemma 5.5.1, it is enough to consider reductions in the newly added thread k when proving safety. Here the case of the “fork” rule is the most challenging, since (informally speaking) after a “fork” reduction there are *two* threads that need to be added to the “base” configuration, while the definition of soundness only allows adding one at a time. By using the locality property (Proposition 4.5.2) of preconditions of forkable threads, one can add the newly forked thread first, and then the old thread. Using precision of the considered predicates, we can then prove that the heap splitting operated by the transition relation yields a new configuration suitable for using our induction hypotheses.

We present detailed proofs of the most relevant cases in section 5.5.

5.5 Detailed Proof Cases

5.5.1 Technical lemmas

The following result takes care of the cases where a reduction happens in a thread other than the one we have just added.

Lemma 5.5.1 *Assume that $\text{safe}_j(w, hp, s, tp, qp)$ and $\text{cons}(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto q])$ where $k \notin \text{dom}(tp)$, and*

$$(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto q]) \xrightarrow{n}_j d$$

for some $j \neq k$. Then there exist w_0, hp_0, s_0, tp_0 , and qp_0 such that

$$d = (w_0 * w', hp_0[k \mapsto h], s_0, tp_0[k \mapsto c], qp_0[k \mapsto q])$$

and

$$(w, hp, s, tp, qp) \xrightarrow{n}_j (w_0, hp_0, s_0, tp_0, qp_0).$$

Proof (sketch). By case analysis following the definition of the reduction relation. The locality requirements in the definition of consistent configurations are needed to obtain the reduction starting from (w, hp, s, tp, qp) ; intuitively, we need to ensure that the relevant predicates which hold in world $w * w'$ already hold in world w .

We also prove a few small results on the operational semantics:

Lemma 5.5.2 *If $k \neq j$ and (if $tp(j) = \text{join}(E)$ then $h(\|E\|_s) \neq k$), then $(h, s, tp) \longrightarrow_j \text{abort} \Leftrightarrow (h, s, tp[k \mapsto c]) \longrightarrow_j \text{abort}$*

Lemma 5.5.3 *If $(h * h', s, tp) \longrightarrow_j \text{abort}$, then $(h, s, tp) \longrightarrow_j \text{abort}$.*

Proofs are all by simple inductions on the operational semantics.

5.5.2 Sequential composition

Before proving soundness of sequential composition, we first need a technical lemma:

Lemma 5.5.4 $\forall j, \forall w, hp, s, tp, qp, k, Q, R, c1, c2$, noting $qp_Q = qp[k \mapsto \llbracket Q \rrbracket^k]$ and $qp_R = qp[k \mapsto \llbracket R \rrbracket^k]$, if

- $safe_j(w, hp, s, tp[k \mapsto c1], qp_Q)$ (1)
- $\models_j \{Q\} c2\{R\}$

then $safe_j(w, hp, s, tp[k \mapsto c1; c2], qp_R)$.

Proof We proceed by induction on j . The only truly interesting case is when $(w, hp, s, tp[k \mapsto c1; c2], qp_R) \xrightarrow{n}_k (w', hp', s', tp[k \mapsto c1'; c2'], qp_R)$, for which we want to prove $safe_{min(j-1, n)}(w', hp', s', tp[k \mapsto c1; c2], qp_R)$.

From the operational semantics, we get that $(w, hp, s, tp[k \mapsto c1], qp_Q) \xrightarrow{n}_k (w', hp', s', tp[k \mapsto c1'], qp_Q)$ which, in conjunction with (1), gives $safe_{min(n, j-1)}(w', hp', s', tp[k \mapsto c1'], qp_Q)$. Since $min(j-1, n) < j$, we can apply the induction hypothesis and conclude.

The proof rule for sequential composition is

$$\frac{\vdash \{P\} c1\{Q\} \quad \vdash \{Q\} c2\{R\}}{\vdash \{P\} c1; c2\{R\}}$$

Following the definition of soundness, we assume that $\forall j, \models_j \{P\} c1\{Q\}$ (1) and $\forall j, \models_j \{Q\} c2\{R\}$ (2) and aim to prove that $\forall j, \models_j \{P\} c1; c2\{R\}$.

Proof We proceed by induction on j . The case where $j = 0$ is trivial since the definition of \models relies on properties for some $m < j$.

Let's assume that $\forall k \leq j, \models_k \{P\} c1; c2\{R\}$ (3) and prove $\models_{j+1} \{P\} c1; c2\{R\}$.

Following the definition of \models , we need to prove safety for all $m < j + 1$, but any $m < j$ is obtained directly via (3), so we only need to consider the case where $m = j$.

Let w, hp, s, tp, qp be fixed such that $safe_j(w, hp, s, tp, qp)$ (4) holds. Let now w' and $k \notin dom(hp)$ be fixed. We choose $(h, n) \in \llbracket P \rrbracket_s^k(w * w')$ (5) such that $h \perp \overline{hp}$ and $cons(q * w', hp[k \mapsto h], s, tp', qp')$ (6). We note $tp' = tp[k \mapsto c1; c2]$ and $qp' = qp[k \mapsto \llbracket R \rrbracket^k]$. Our goal is now to prove $safe_{min(n, j)}(w * w', hp[k \mapsto h], s, tp', qp')$.

- $cons(w * w', hp[k \mapsto h], s, tp', qp')$ is exactly (6).
- Let's assume there is a thread l such that $(\overline{hp} * h | s, tp') \longrightarrow_l \text{abort}$. If $l \neq k$, we get from lemmas 5.5.2 and 5.5.3 that $(\overline{hp} | s, tp) \longrightarrow_l \text{abort}$, which is absurd by (4). If $l = k$, the operational semantics entails that $(\overline{hp} * h | s, tp[k \mapsto c1]) \longrightarrow_l \text{abort}$ which contradicts (1).

- For any $l \in \text{dom}(tp')$ such that $tp'(l) = \text{skip}$, we know from (4) that $(hp(l), j-1) \in qp(l)(w)$. Since $tp'(k) \neq \text{skip}$, we have that $k \neq l$ and thus, $qp'(l) = qp(l)$ and $hp[k \mapsto h](l) = hp(l)$, hence that $(hp[k \mapsto h](l), j-1) \in qp'(l)(w)$. Since $\min(n, j) \leq j$, we obtain $(hp[k \mapsto h](l), \min(n, j) - 1) \in qp'(l)(w)$. Finally, with proposition 4.5.3, we conclude that $(hp[k \mapsto h](l), \min(n, j) - 1) \in qp'(l)(w * w')$.
- If $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{n}_l (w_0, hp_0, s_0, tp_0, qp_0)$, we want to show $\text{safe}_{\min(j-1, n)}(w_0, hp_0, s_0, tp_0, qp_0)$. If $l \neq k$, we conclude from lemma 5.5.1. If on the other hand, $l = k$, from (1), we obtain $\text{safe}_{\min(n, j)}(w * w', hp[k \mapsto h], s, tp[k \mapsto c1], qp[k \mapsto \llbracket Q \rrbracket_s^k])$ (7). There are then two possible situations:
 - If $c1 = \text{skip}$, we get $(|\overline{hp} * h|, s, tp') \rightarrow_k (|\overline{hp} * h|, s, tp[k \mapsto c2])$. From (7) and the safety definition, $(h, \min(n, j) - 1) \in \llbracket Q \rrbracket^k(w * w')$. From (2) with index $j - 1$, we get $\text{safe}_{\min(j-1, \min(n, j)-1)}(w * w', hp[k \mapsto h], s, tp[k \mapsto c2], qp')$. Since $\min(j-1, \min(n, j) - 1) = \min(n-1, j-1)$, we can conclude.
 - If $c1 \neq \text{skip}$ and $(|\overline{hp} * h|, s, tp[k \mapsto c1]) \rightarrow_k (|\overline{hp}'|, s', tp[k \mapsto c1'])$, from (7) we get that for some $j' < j$ and w'' , $\text{safe}_{j'}(w'', hp', s', tp[k \mapsto c1'], qp[k \mapsto \llbracket Q \rrbracket_s^k])$. Since the thread local operational semantics is deterministic, we want to show that $\text{safe}_{j'}(w'', hp', s', tp[k \mapsto c1' ; c2], qp')$, which is obtained directly by applying lemma 5.5.4.
- If $(|\overline{hp} * h|, s, tp') \rightarrow_l (h_0, s_0, tp_0)$, the fact that $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{n}_l d'$ for some d' follows from (4) and the definition of \xrightarrow{n}_l .

5.5.3 Lock release

The proof rule for releasing a lock is

$$\overline{\vdash \{ \text{Locked}(E, P) * P \} \text{release}(E) \{ \text{Ex}(E, P) \}}$$

Let's assume that $\forall k \leq j, \models_j \{ \text{Locked}(E, P) * P \} \text{release}(E) \{ \text{Ex}(E, P) \}$ (1) and let's prove that this holds for $j + 1$.

We choose w, hp, s, tp, qp such that $\text{safe}_j(w, hp, s, tp, qp)$ (2) holds. We then choose $k \notin \text{dom}(hp), w'$ such that $w * w'$ is well defined. Let then $(h, n) \in \llbracket \text{Locked}(E, P) * P \rrbracket_s^k(w * w')$ (4), $h \perp \overline{hp}$ and $\text{cons}(w * w', hp[k \mapsto h], s, tp[k \mapsto \text{release}(E)], qp[k \mapsto \llbracket \text{Ex}(E, P) \rrbracket^k])$ (5).

We note $qp' = qp[k \mapsto \llbracket \text{Ex}(E, P) \rrbracket^k], tp' = tp[k \mapsto \text{release}(E)]$ and choose and want to prove $\text{safe}_{\min(n, j)}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof We unfold the safety definition:

- Consistency of the state is given by (5).
- If there was a reduction to `abort` in thread l , we could use the same argument than in the sequential composition proof to show that $l \neq k$ is absurd. If $l = k$, the operational semantics tells us that $h(\llbracket E \rrbracket_s) \neq k$. By (4) and the definition of $\llbracket \text{Locked}(E, P) \rrbracket_s^k$, however, we have that $h(\llbracket E \rrbracket_s) = k$, which is absurd.
- Since the command to execute in thread k is not `skip`, we can conclude by (2) and proposition 4.5.3 that all threads done executing satisfy their post-conditions in qp' .
- If there is a transition of the form $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{m}_l (w_0, hp_0, s_0, tp_0, qp_0)$, the case where $l \neq k$ is handled by lemma 5.5.1.

If $l = k$, we know from the operational semantics and the definition of $\xrightarrow{k}_{n'}$ that $w_0 = w * w'$, $s_0 = s$, $tp_0 = tp[k \mapsto \text{skip}]$, $qp_0 = qp$. Furthermore, from (4), we know that $(w * w')(\llbracket E \rrbracket_s) = \text{Lock}(Q)$ for some Q , and there exists $(h_1, n') \in \text{App}(Q)(w * w')$ (6) such that $h = h_1 *_{w * w'} h_2$ (7) and $hp_0 = hp[k \mapsto h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], \Omega \mapsto hp(\Omega) *_{w * w'} h_1]$ (7). We want to prove $\text{saf}_{\min(m, n, j-1)}(w * w', hp_0, s, tp_0, qp')$.

From (4), $\exists h_3, h_4$ such that $(h_3, n) \in \llbracket \text{Locked}(E, P) \rrbracket_s^k(w * w')$ (8) and $(h_4, n) \in \llbracket P \rrbracket_s^k(w * w')$ (9).

From (8) and the interpretation of $\text{Locked}(E, P)$, we get that $\text{App}(Q)(w * w') =^{n-1} \llbracket P \rrbracket_s^k(w * w')$. From (6), $(h_1, \min(m, n-1)) \in \llbracket P \rrbracket_s^k(w * w')$.

From (9), $(h_4, \min(m, n-1)) \in \llbracket P \rrbracket_s^k(w * w')$. Since the resource invariants are required to be precise, we can conclude that $h_4 = h_1$, and further than $h_3 = h_2$.

Since now $(h_2, \min(m, n-1)) \in \llbracket \text{Locked}(E, P) \rrbracket_s^k(w * w')$, we get that $(h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], \min(m, n-1)) \in \llbracket \text{Ex}(E, P) \rrbracket_s^k(w * w')$ (10).

We can then unfold the safety definition one last time. All cases are trivial, with one exception: we now have $tp_0(k) = \text{skip}$. From (10), we get that $(h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], \min(m, n-1, j-1)) \in \llbracket \text{Ex}(E, P) \rrbracket_s^k(w * w') = qp'(k)$. We also know that there will be no more reduction in thread k , so all reduction cases are handled by lemma 5.5.1, which concludes our proof.

5.5.4 Lock creation

The proof rule for creating a lock is

$$\frac{}{\vdash \{E \mapsto _ \} \text{make_lock_P}(E) \{ \text{Locked}(E, P) \}}$$

Let's assume $\forall k \leq j, \models_k \{E \mapsto _ \} \text{make_lock_P}(E) \{ \text{Locked}(E, P) \}$ (1) and let's prove this also holds for $j+1$.

Let's choose w, hp, s, tp, qp such that $safe_j(w, hp, s, tp, qp)$ (2) holds. We pick $k \notin \text{dom}(hp)$ and w' such that $w * w'$ is well defined. Let then $(h, n) \in \llbracket E \mapsto _ \rrbracket_s^k(w * w')$ (3), $h \perp \overline{hp}$ and $cons(w * w', hp[k \mapsto h], s, tp', qp')$. We note $qp' = qp[k \mapsto \llbracket Locked(E, P) \rrbracket_s^k]$ and $tp' = tp[k \mapsto \text{make_lock_P}(E)]$ and want to prove $safe_{\min(n, j) - 1}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof • If there was a reduction to abort, it would have to be in thread k by the same argument than in the sequential composition proof. This would however contradict the operational semantics and (3), which implies there is no execution fault.

- All threads finished executing satisfy their post-conditions follows directly from (2) and $tp'(k) \neq \text{skip}$.
- If $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{m}_k (w_0, hp_0, s_0, tp_0, qp_0)$, we have from the definition of the transition relation that $hp_0 = hp[k \mapsto h']$, $w_0 = (w * w')[\llbracket E \rrbracket_s \mapsto Lock(Abs(\llbracket P \rrbracket_s^k))]$, $s_0 = s$, $tp_0 = tp[k \mapsto \text{skip}]$ and $qp_0 = qp'$, with the notation $h' = h[\llbracket E \rrbracket_s \mapsto k]$. We now want to show $safe_{\min(n-1, j-1, m)}(w_0, hp[k \mapsto h'], s, tp[k \mapsto \text{skip}], qp')$.

The only interesting case when we unfold the safety definition comes from the fact that $tp[k \mapsto \text{skip}](k) = \text{skip}$, which means we need to prove $(hp[k \mapsto h'])(k), \min(n-2, j-2, m-1) = (h', \min(n-2, j-2, m-1)) \in qp'(k)(w_0) = \llbracket Locked(E, P) \rrbracket_s^k(w_0)$. $\llbracket E \rrbracket_s$ is in the domains of both h' and w_0 , and $h'(\llbracket E \rrbracket_s) = k$. Furthermore, for any $w'', p, App(Abs(\llbracket P \rrbracket_s^k))(w'') =^p \llbracket P \rrbracket_s^k(w'')$.

5.5.5 Thread creation

The proof rule for fork is

$$\frac{\Gamma, \{R\}f\{S\} \vdash \{P * tid(f, x)\}c\{Q\}}{\Gamma, \{R\}f\{S\} \vdash \{P * R\}\text{let } x = \text{fork}(f) \text{ in } c\{Q\}}$$

We assume $\forall j, \Gamma \vDash_j \{P * tid(f, x)\}c\{Q\}$ (1) and $\{R\}f : c\mathcal{F}\{Q\} \in \Gamma$ (2) (note that Γ here is not exactly the same as in the proof rule, as we “factor in” the triple for f).

We now assume that $\forall k \leq j, \Gamma \vDash_k \{P * R\}\text{let } x = \text{fork}(f) \text{ in } c\{Q\}$ (3) and want to show that $\Gamma \vDash_{j+1} \{P * R\}\text{let } x = \text{fork}(f) \text{ in } c\{Q\}$.

Let us choose w, hp, s, tp, qp such that $safe_j(w, hp, s, tp, qp)$ (4) and $\Gamma \vDash_j \{R\}f\{S\}$ (5). We now choose $k \notin \text{dom}(hp)$, w' such that $w * w'$ is well defined, $(h, n) \in \llbracket P * R \rrbracket_s^k(w * w')$ (6), $h \perp \overline{hp}$ and $cons(w * w', hp[k \mapsto h], s, tp', qp')$ (7). We note $qp' = qp[k \mapsto \llbracket Q \rrbracket_s^k]$ and $tp' = tp[k \mapsto \text{let } x = \text{fork}(f) \text{ in } c]$ and now want to show $safe_j(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof As in the other proof cases, most of the safety conditions follow directly from (4). The only difficult condition is the case where a reduction happens in thread k :

$(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{m}_k (w_0, hp_0, s_0, tp_0, qp_0)$ for some $i \notin \text{dom}(tp')$, where $s_0 = s[x \mapsto v]$, $w_0 = w[v \mapsto T(f)]$, $qp_0 = qp'[i \mapsto \llbracket S \rrbracket^i]$, $tp_0 = tp[k \mapsto c, i \mapsto cf]$. Furthermore, there exists h_1, h_2 such that $h = h_1 *_{w * w'} h_2$, $h_2 \in \llbracket R \rrbracket^i(w * w')$ and $hp_0 = hp[k \mapsto h_1[v \mapsto i], i \mapsto h_2]$.

By (6), $\exists h_3, h_4$ such that $h = h_3 *_{w * w'} h_4$, $(h_3, \min(m, n-1, j-1)) \in \llbracket P \rrbracket_s^k(w * w')$ (8) and $(h_4, \min(m, j-1, n-1)) \in \llbracket R \rrbracket_s^k(w * w')$ (9). Using precision of R , we can deduce that $h_1 = h_3$ and $h_2 = h_4$.

The key step is now to start from the “base state” (w, hp, s, tp, qp) and first add the thread i , which we can do thanks to (5), (4), (7) and (9), which gives us $\text{safe}_{\min(m, j-1, n-1)}(w_0, hp[i \mapsto h_2], s_0, tp[i \mapsto cf], qp[i \mapsto \llbracket S \rrbracket^i])$.

In order to be able to use (1) to add thread k and conclude, we first need to apply the locality property of proposition 4.5.3 to (8), which gives that $h_1[v \mapsto i] \in \llbracket P * \text{tid}(f, x) \rrbracket_{s_0}^k(w_0)$. From there, using the soundness definition, we obtain the desired $\text{safe}_{\min(m, n-1, j-1)}(w_0, hp_0, s_0, tp_0, qp_0)$.

5.5.6 Thread synchronisation

The proof rule for join is

$$\frac{}{\Gamma, \{P\}f\{Q\} \vdash \{\text{tid}(f, E)\} \text{join}(\mathbf{E})\{Q\}}$$

We assume $\{P\}f\{Q\} \in \Gamma$ (1) and $\forall k \leq j, \Gamma \vDash_j \{\text{tid}(f, E)\} \text{join}(\mathbf{E})\{Q\}$ (2).

Let w, hp, s, tp, qp be fixed such that $\text{safe}_j(w, hp, s, tp, qp)$ (3). We choose $k \notin \text{dom}(hp)$, w' such that $w * w'$ is well defined. Let $(h, n) \in \llbracket \text{tid}(f, E) \rrbracket_s^k(w * w')$ (4), $h \perp hp$ and $\text{cons}(w * w', hp[k \mapsto h], s, tp', qp')$ (5).

We note $qp' = qp[k \mapsto \llbracket Q \rrbracket_s^k]$, $tp' = tp[k \mapsto \text{join}(\mathbf{E})]$, and want to show $\text{safe}_{\min(j, n)}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof • Hypothesis (3) implies that if a thread reduces to abort, it has to be k . From (4) and (7), we know there is a i such that $h(\llbracket E \rrbracket_s) = i$ and $i \in \text{dom}(tp)$. The operational semantics allows us to conclude that thread k does not reduce into abort.

- We only show the proof for the first of the two kinds of “join” reductions. If $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{m}_k (w_0, hp_0, s_0, tp_0, qp_0)$, we know from the definition of the reduction relation and (7) that there exists some $i = h(\llbracket E \rrbracket_s)$ and that $w_0 = (w * w')|_{\text{dom}(w * w') \setminus \{\llbracket E \rrbracket_s\}}$, $s_0 = s$, $tp_0 = tp[k \mapsto \text{skip}]|_{\text{dom}(tp') \setminus \{i\}}$ and $qp_0 = qp'|_{\text{dom}qp' \setminus \{i\}}$. Furthermore, $h = h_1 *_{w * w'} [\llbracket E \rrbracket_s \mapsto i]$ for some h_1 and $hp_0 = hp[k \mapsto hp(k) *_{w * w'} h_1]|_{\text{dom}hp' \setminus \{i\}}$.

We prove the desired $\text{safe}_{\min(m, n-1, j-1)}(w_0, hp_0, s, tp_0, qp_0)$ directly by unfolding the safety definition. The only interesting case arises from $tp_0(k) = \text{skip}$. Since there was a reduction, we know that $tp(i) = \text{skip}$, which from (1) and (3) gives that $(hp(i), j-1) \in \llbracket Q \rrbracket_s^i(w)$.

From the side condition on the join proof rule, we get $(hp(i), j-1) \in \llbracket Q \rrbracket_s^k(w)$ (since the interpretation of *Locked* is the only one making use

of the thread identifier). Using the locality property (proposition 4.5.3), $(hp(i), j - 1) \in \llbracket Q \rrbracket_s^k(w_0)$. Finally, we can weaken this into the desired $(hp_0(k), \min(n, j) - 2) \in \llbracket Q \rrbracket_s^k(w_0)$.

Conclusion

Contributions

The contributions to the two topics discussed in this dissertation are as follow.

In the first chapter, we have taken two of the most important results of constructive type theory, Seely’s proof that Martin-Löf’s intuitionistic type theory can be interpreted in locally cartesian closed categories [See84], and that dependent type theory can be used as a constructive metalanguage instead of classical set theory [ML82] and combined them in the most natural way: we interpreted the locally cartesian closed categorical model of Martin-Löf’s intuitionistic type theory in a version of itself. To achieve this, we re-used the notion of *category with families* which had been introduced by Peter Dybjer a few years earlier [Dyb95], and interpret them as groupoid-style E-categories.

Furthermore, we provided a machine-checked proof of the key theorem that E-locally cartesian closed categories are also E-categories with families, using the proof assistant `coq`.

In the second part, chapters 2 through 5, we have presented an elegant and relatively simple solution to a well-known problem. We have done so by combining two methods already present in the literature. First, using topological methods based on applying a version of the Banach’s fixed point theorem to ultra-metric spaces to solve recursive domain equations. This is based on the domain theoretical work of American and Rutten [AR89] and has been used in similar contexts more recently, for instance by Birkedal et al. [BST10]. Second, we used a proof method from Vafeiadis [Vaf11] to prove soundness of our model with respect to a standard operational semantics. This method is based on the idea of a *safety predicate* which stipulates properties of well behaved programs and is inductively proved to be maintained through every possible interleaving of a concurrent program. We have then provided such a paper proof for the soundness of our ultra-metric based model.

One of the main obstacles in combining these methods and applying them to the new setting of a logic for storable locks has been to specify a comprehensive safety predicate which could then provide exactly the level of detail about the program state that the soundness proof needs.

Future work

There are a number of directions in which both works can be taken.

The coq script proposed in chapter 1, while proving the key theorem, is not a complete interpretation of constructive type theory in itself. The data structures manipulated were growing so complex that the limits of what could be written manually in coq had been reached. The existence of many patterns when building elaborate categorical structures seems to call for a greater level of automation and inference of many trivial details.

Another issue still open is that of the coherence problem and whether it is possible to relate our groupoid-style (type theory interpreted) E-cwf to proof-irrelevant versions.

For storable locks, the great level of repetitive details and the multi-layered induction proof cry for a machine-checked coq proof with some level of automation. This could also be based on Benton and Varming's work on formalising ultra-metric methods in coq [BBKV10].

An open problem, as described in section 3.4, is whether it is possible to find example programs generic enough that they can not be proved in a model with the syntactic limitation on lock sorts found in Gotsman et. al. [GBC⁺07].

Finally, another natural extension would be to allow not only storable locks, but also storable procedures and callbacks, which would lead to an even greater level of genericity and expressiveness. It is expected that the model we have proposed would scale fairly well to such an extension.

Bibliography

- [Alg10] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 and INRIA, 2010.
- [AR89] Pierre America and Jan J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989.
- [Ban22] Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, III, 1922.
- [BBKV10] Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages. July 2010.
- [BBS11] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. A step-indexed kripke model of separation logic for storable locks. In *MFPS*, 2011.
- [BBTS07] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 2007.
- [BCOP05] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40(1):259–270, 2005.
- [BD08] Alexandre Buisse and Peter Dybjer. The interpretation of intuitionistic type theory in locally cartesian closed categories - an intuitionistic perspective. *Electr. Notes Theor. Comput. Sci.*, 218:21–32, 2008.
- [BST10] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *Mathematical Structures in Computer Science*, 20(4):655–703, 2010.

- [BTSY06] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *CoRR*, 2006.
- [Dyb95] Peter Dybjer. Internal type theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg*, January 1995. Revised version to appear in TYPES '95, Springer Verlag LNCS.
- [GBC⁺07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, pages 19–37, 2007.
- [HAZN08] A. Hobor, A.W. Appel, and F. Zappa-Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [Hob08] Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.
- [KAB⁺09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *TLDI*, pages 105–116, 2009.
- [Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, page 42–54, 2006.
- [ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
- [O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004.
- [O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
- [OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *TPHOLS, volume 5674 of LNCS*, page 391–407, 2009.
- [Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
- [See84] Robert Seely. Locally cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc.*, 95(33), 1984.

- [Vaf11] Viktor Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.
- [WBR94] Kim Ritter Wagner, Stephen Brookes, and John C. Reynolds. Solving recursive domain equations with enriched categories. Technical report, 1994.