UiO : **Department of Informatics**
University of Oslo

Formalization of a Type and Effect System using Coq and Ott

Peter Brottveit Bock
Master's Thesis Autumn 2013

Formalization of a Type and Effect System using Coq and Ott

Peter Brottveit Bock

November 1, 2013

Abstract

Papers from the field of programming language theory, especially those related to concurrent programming languages, often contain languages and definitions with many rules, and proofs about these are therefore often long and tedious. The need for both formalization and automation is sought after, as indicated by the POPLMark-challenge.

This thesis presents a formalization of two type and effect systems, and a formalized proof relating one of the systems to the other. The formalization is realized with the tools Coq and Ott, which give a rigorous and machine checkable formalization.

Contents

Chapter 1

Introduction

Programming language theory often consists of complex definitions and proofs. They are usually written in English together with mathematical notation. This allows the authors to appeal to the intuition of the reader, skip trivialities, and let obvious things be implicit, which helps readability as well as being good from a pedagogical point of view.

The chance of errors increase with the size of a work. Errors can include trivialities like typographical errors, but also logical fallacies, omissions and misunderstandings.

To gain faith in the correctness of claims made in such papers, machine checkable formal methods are desirable. This desire is shown through the POPLMark challenge [ABF+05], which is a challenge for researchers to formalize the type system $F_{<:}$, where the formalization is evaluated using certain benchmarks.

Theorem prover assistants are tools to formalize theories and the proof of its theorems. The definitions, lemmas, theorems, and proofs are written in one or more formal, machine-readable languages, and the theorem prover assistant will check for errors and verify the correctness of the proofs.

1.1 Goal

The goal of this thesis is to formalize a non-trivial system, with the following sub-goals:

1. The formalization should be rigorous.
2. The formalization should be machine checkable.
3. The formalization should be readable.

The domain of the formalization will be static deadlock checking of programs. Creating concurrent programs is considered more difficult than creating single threaded programs, and one of the most common types of errors are *deadlocks*, which occur when several threads are waiting in a cyclic manner for resources that they collectively hold.

It is therefore desirable to statically analyze programs to check if they contain deadlocks. The systems which do this are not trivial; they usually contain many definitions and tedious proofs, and it is therefore of interest to formally check that the system has the properties it has been claimed to have.

Several papers were considered, and the article *Deadlock Checking by Data Race Condition* [PSS13] was chosen. It is a highly technical paper, with several type and effect systems, each with many rules; several auxiliary definitions; and several long proofs.

Two tools were used in the formalization work: Ott and Coq.

1.2 Coq

Coq [BC04] is an advanced proof assistant. Its foundation is type theory, a topic which is investigated in Chapter 2. It has support for higher-order functions, inductive and co-inductive types, and dependent types. The expressivity of Coq is very high; its logic corresponds to an intuitionistic higher-order predicate logic, and is compatible with classical axioms such as double negation elimination and the law of exclude middle.

The most famous use of Coq is the proof of the Four-Color theorem by Gonthier [Gon08], while in the field of programming language theory CompCert [Ler09] is the most well-known example. CompCert is a formally verified compiler for almost all of the C programming language, written and verified in Coq.

It is assumed the reader of this thesis has a basic knowledge of Coq, mainly inductive definitions, fixpoints, and dependent types. Relevant literature is the online book *Software Foundations*[1] by Benjamin C. Pierce et. al. which is a good, practical introduction to the use of Coq, while the book *Interactive Theorem Proving and Program Development* [BC04] is a thorough introduction to Coq, which also introduces the underlying system Calculus of Inductive Constructions. Finally, Adam Chlipala's *Certified Programming with Dependent Types* [Chl11] focuses on the use of dependent functions, and has an unusual technique for completing proofs in an automated way.

There are three major reasons for the choice of Coq. First of all, I wanted to investigate dependent type theory, which is the foundation of Coq. Coq is also one of three choices which can be used together with Ott. Finally, Coq is a mature project.

1.3 Ott

Ott [SNO+07] is a tool intended to help researchers in the field of programming language theory to formalize their work. It consists of a domain specific language for writing context free grammars and judgments. The syntax of the Ott-language mirrors quite closely how one would write grammars and judgments in scientific papers.

The Ott program can generate both a nicely typeset LaTeX-document and Coq-code from a given Ott-file. Generation of OCaml, HOL and Isabelle code is also supported, but it is not used for this thesis.

[1] `http://www.cis.upenn.edu/~bcpierce/sf/`

1.4 Results

The main results in this thesis is: the formalization in OTT of two type systems, a specification and an algorithm, together with all syntactical constructions and auxiliary judgments needed by these two type systems; and the formalization in COQ of the proof given in [PSS13] for a "soundness" property between the algorithm and the specification. The result includes several technical proofs that are outside the scope of [PSS13]. The formalization is rigorous and machine checkable.

In addition to this, the semantics and another type system is formalized in OTT. These are used for proofs in [PSS13], but those proofs have not been formalized. The formalization in OTT gives rise to fourteen pages of typeset grammars and relations. The typeset version of the formalization turned out to be helpful when discussing the formalization with others.

As a result of the formalization, several typographic errors in the original type systems were found, some insignificant, while other were more serious, such as wrongly indexed variables. In addition, an error where a rule was applied wrongly was found in the proof of soundness.

1.5 Overview of the Thesis

The first chapter gives an introduction to type theory. Then OTT is introduced in Chapter 3; deep and shallow embedding, which are categorizations of embedding of logics, is described in Chapter 4; the work to be formalized is presented in Chapter 5, and Chapter 6 describes the formalization process. The major shortcoming of the formalization and the experience of using OTT is described in Chapter 7. Finally, the conclusion is in Chapter 8. In Appendix A, the typeset version of the formalization, as generated by OTT, is given.

The source code can be found at `http://heim.ifi.uio.no/peterbb/master/`.

1.6 Acknowledgments

I would like to thank my supervisor Martin Steffen for all of his great feedback; and my girlfriend Julia Batkiewicz for her support and for proofreading my thesis.

Chapter 2

Type Theory

Type theory is the foundation of the theorem prover Coq, and it is used in the work which has been formalized in this thesis. We therefore have two good reasons to investigate type theory.

2.1 Type Theory and Coq

A theorem prover assistant is a program where one can express propositions and their proofs and check if a proof is correct. Examples of theorem provers are Coq [BC04], Agda [Nor07, CC99], Isabelle [Pau88], and ACL2 [BM79]. Some theorem provers, such as Coq, use type theory as its foundation, but users usually interact with Coq using notation from logic, even though Coq's "internal" language is that of types and programs. The core of Coq is a type checker, which can check if a given program has a given type.

It is the Curry-Howard correspondence [Cur34], bridging logic with type theory, that makes it possible to use type theory for proving propositions. In the correspondence, a type corresponds to a propositions, a program corresponds to a proof, and type checking corresponds to proof verification.

It is possible to express complex propositions in Coq, and the claim that they correspond to types may seem far-fetched for programmers. For which type, in for instance ML, would correspond to the proposition "f is a function which sorts lists of numbers"? While ML's and Coq's type systems share a common foundation, they differ in that Coq's type system can express more, but on the other hand, ML's can infer the type of a program without type annotations.

One of the goals of this chapter is to show what kind of type system one can use to express propositions. Another more practical goal is to get more knowledge of how Coq and similar systems work, so that one can understand practical problems which come up when working with them.

The syntax and inference rules are taken from Barendregt's Handbook [Bar92].

2.2 Simply-Typed Lambda Calculus

The *simply-typed lambda calculus* ($\lambda\!\!\rightarrow$) will be the basis for all our type systems. $\lambda\!\!\rightarrow$ can be viewed as a minimal functional programming language, for instance a very restricted version of ML. To make the system a bit more interesting, product and sum types has been added in the same way as in the book *Proofs and Types* [GLT89].

For the definition of the types, assume that there is a set B of base types, which for instance could be {`bool`, `int`} for boolean and integer values, and let $\mathbb{V} = \{\alpha, \beta, \ldots\}$ be a set of type variables. Given this, the set $\mathbb{T}$ of all the types of $\lambda{\rightarrow}$ is defined as follows:

$$\mathbb{T} ::= B \mid \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \mathbb{T} \times \mathbb{T} \mid \mathbb{T} + \mathbb{T}$$

Throughout this section, τ and σ will range over $\mathbb{T}$. The meaning of each construct is as follows: $\tau \to \sigma$ is read as "function from τ to σ"; and $\tau \times \sigma$ is read as "product of τ and σ", which is a pair where the first element is of type τ, and the second of type σ; and finally $\tau + \sigma$ is read as "disjoint sum of τ and σ" (or "tagged union"), which is a value of τ or σ tagged with which of the types it has.

The terms of $\lambda{\rightarrow}$ are defined as

$$\begin{aligned} M, N ::= {} & c^\tau \mid x \mid \lambda x{:}\tau.\ M \mid M\ N \mid \\ & \langle M,\ N\rangle \mid \pi^1 M \mid \pi^2 M \mid \\ & \iota^1 M \mid \iota^2 M \mid \mathit{case}(M, N, N') \end{aligned}$$

where c^τ is a constant c of type τ, x is a variable, $\lambda x{:}\tau.\ M$ is a lambda abstraction (i.e. an anonymous function), and $M\ N$ is a function application, where M is the function and N is the argument.

The next operators: $\langle M,\ N\rangle$, $\pi^1 M$, and $\pi^2 M$, respectively construct pairs, fetch the first element of a pair, and fetch the second element of a pair. A variant of this pair construction, using labels instead of indexes, is found in languages like C, Algol, and ML under the name records or structs.

The final operators: $\iota^1 M$, $\iota^2 M$, and $\mathit{case}(M, N, N')$, respectively construct a left-hand sum object, construct a right-hand sum object, and does a case statement on the sum object (N and N' are functions of which one will be called with M, depending on M's type). A variant of this construct using labels (called constructors) instead of indexes is found in languages like ML and Haskell, where one uses pattern matching for better readability.

To check whether a given expression has a given type is called type checking. If an expression passes type checking, then, evaluating the expression will not result in a type error.

To perform type checking, a context will need to be maintained to remember the type of variables. Γ denotes a context, which is a finite sequence of pairs of variables and types, i.e. $\langle x_1 : \tau_1,\ \cdots,\ x_k : \tau_k\rangle$. Two operations are applied to contexts: $\Gamma(x) = \tau$ denotes that the right most pair in Γ which contains x is $x{:}\tau$, and $\Gamma[x \mapsto \tau]$ denotes the sequence where the pair $x{:}\tau$ is appended to the right of the sequence which Γ denotes. Thus a context is used as a kind of stack, with push and a specialized search.

Finally, $\Gamma \vdash M\ : \tau$ is the type checking relation, and it is true if in the context Γ, the expression M has type τ, and it is defined by the inference rules in Figure 2.1.[1] $\Gamma \vdash M\ : \tau$ is also called a type judgment. If one can construct

[1]We will not go into The Curry-Howard correspondence, but these rules are almost identical to the rules of Natural Deduction for propositional logic if one view a variable as an assumption, a function type as an implication, a sum type as a disjunction, and a product type as a conjunction. See for instance *Proof and Types* [GLT89].

$$\frac{}{\Gamma \vdash c^{\tau}\ :\tau}\ \textsc{Const}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x\ :\tau}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\tau \vdash M\ :\tau'}{\Gamma \vdash (\lambda x{:}\tau.\, M)\ :\tau \to \tau'}\ \to \textsc{Intro}$$

$$\frac{\Gamma \vdash M\ :\tau \to \tau' \qquad \Gamma \vdash N\ :\tau}{\Gamma \vdash M\ N\ :\tau'}\ \to \textsc{Elim}$$

$$\frac{\Gamma \vdash M\ :\tau \qquad \Gamma \vdash N\ :\tau'}{\Gamma \vdash \langle M, N\rangle\ :\tau \times \tau'}\ \times \textsc{Intro}$$

$$\frac{\Gamma \vdash M\ :\tau \qquad \tau' \in \mathbb{T}}{\Gamma \vdash \iota^1 M\ :\tau + \tau'}\ + \textsc{Intro-1}$$

$$\frac{\Gamma \vdash M\ :\tau \times \tau'}{\Gamma \vdash \pi^1 M\ :\tau}\ \times \textsc{Elim-1}$$

$$\frac{\tau \in \mathbb{T} \qquad \Gamma \vdash M\ :\tau'}{\Gamma \vdash \iota^2 M\ :\tau + \tau'}\ + \textsc{Intro-2}$$

$$\frac{\Gamma \vdash M\ :\tau \times \tau'}{\Gamma \vdash \pi^2 M\ :\tau'}\ \times \textsc{Elim-2}$$

$$\frac{\Gamma \vdash M\ :\tau + \tau' \qquad \Gamma \vdash f_l\ :\tau \to \sigma \qquad \Gamma \vdash f_r\ :\tau' \to \sigma}{\Gamma \vdash case(M, f_l, f_r)\ :\sigma}\ + \textsc{Elim}$$

Figure 2.1: Rules and axioms of the simply-typed lambda calculus

a derivation tree, and if all the leaves of the tree are axioms, then we call the program well-typed.

Three questions may be asked with regards to type judgments. Note that if $\Gamma = \langle x_1{:}\tau_1, \cdots, x_n{:}\tau_n\rangle$, then $\Gamma \vdash e\ :\sigma$ if and only if

$$\vdash \lambda x_1{:}\tau_1 \cdots x_n{:}\tau_n.e\ :\tau_1 \to \cdots \to \tau_n \to \sigma$$

Thus in all the questions, the context is assumed empty.

Type checking Does it exist a derivation with $\vdash M\ :\tau$ as its root?

Typability Given an expression M, does it exist a type τ such that $\vdash M\ :\tau$ is derivable?

Inhabitation Given a type τ, does there exists an expression M such that $\vdash M\ :\tau$ is derivable?

An example of a derivation is:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma(x) = \alpha}{\Gamma \vdash x\ :\alpha}\ \textsc{Var} \qquad \cfrac{\Gamma(y) = \beta}{\Gamma \vdash y\ :\beta}\ \textsc{Var}}{\Gamma = \langle x : \alpha,\ y : \beta\rangle \vdash \langle x, y\rangle\ :\alpha \times \beta}\ \times \textsc{Intro}}{\langle x : \alpha\rangle \vdash \lambda y{:}\beta.\ \langle x, y\rangle\ :\beta \to \alpha \times \beta}\ \to \textsc{Intro}}{\vdash \lambda x{:}\alpha.\ \lambda y{:}\beta.\ \langle x, y\rangle\ :\alpha \to \beta \to \alpha \times \beta}\ \to \textsc{Intro}$$

2.3 Polymorphic Types

Our first extensions to the simply-typed lambda calculus will be polymorphic types, which is also known as (polymorphic) second order types, giving the system $\lambda 2$, equivalent to System F [Rey74][Gir72]. The goal is to allow functions which work independent of the types of some part of its input. For instance, the map-function working on lists: given a list with elements of type α, and a function from α to β, then the map function has the type $\texttt{list}\,\alpha \to (\alpha \to \beta) \to \texttt{list}\,\beta$, and it does not depend on the type of α and β. So in $\lambda 2$ the type is then $\forall\alpha.\,\forall\beta\,.(\texttt{list}\,\alpha \to (\alpha \to \beta) \to \texttt{list}\,\beta)$ (assuming `list` is a primitive type operator). Formally, the set of types is extended with types on the form $\forall\,\mathbb{V}.\,\mathbb{T}$, and to eliminate and introduce the quantifier type, there are two extensions to the syntax with corresponding rules.

The first extension to the syntax is $\Lambda\alpha.\,M$, which says that the type variable α is universally quantified in the expression M. The rule for this is:

$$\dfrac{\Gamma \vdash M\ :\tau \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash (\Lambda\alpha.\,M)\ :(\forall\alpha\,.\,\tau)}\ \forall\ \textsc{Introduction}$$

where $FV(\Gamma)$ is the set of free variables from Γ.

The final extension is $M\,\tau$, which is the instantiating of a universally quantified variable. Assuming M is of type $\forall\alpha.\,\tau$, then M can be viewed as having the type $\tau[\sigma/\alpha]$. Formally:

$$\dfrac{\Gamma \vdash M\ :(\forall\alpha\,.\,A)}{\Gamma \vdash M\ \tau\ :A[\tau/\alpha]}\ \forall\ \textsc{Elimination}$$

Now follows two examples which shows how polymorphic types may be used.

Example of Identity Function in $\lambda 2$

A simple example is the identity function which in $\lambda\!\to$ needs to be implemented for each type, while in $\lambda 2$ is a single function $\Lambda\alpha.\,(\lambda x{:}\alpha.\,x)$ of type $\forall\alpha\,.\,(\alpha \to \alpha)$.

The following derivation tree shows that the identity function is well-typed in $\lambda 2$.

$$\dfrac{\dfrac{\dfrac{\langle x{:}\alpha\rangle(x) = \alpha}{\langle x{:}\alpha\rangle \vdash x\ :\alpha}\ \textsc{Var}}{\vdash \lambda x{:}\alpha.\,x\ :(\alpha \to \alpha)}\ \forall\ \textsc{Intro.}}{\vdash \Lambda\alpha.\,\lambda x{:}\alpha.\,x\ :\forall\alpha.(\alpha \to \alpha)}\ \forall\ \textsc{Intro.}$$

Example of Church Numerals in $\lambda 2$

Another example is Church numerals, which is a way to represent natural numbers as lambda terms. The intuition is that the Church numeral of 5 is an iterator, which given a function f and an element x, applies f five times to x.

In the untyped lambda calculus we have that $0 \equiv \lambda f\,x\,.\,x$, and $1 \equiv \lambda f\,x\,.\,fx$, and $2 \equiv \lambda f\,x\,.\,f(fx)$, and in general, $n \equiv \lambda f\,x\,.\,f^n x$. The successor function is $\lambda n\,f\,x\,.\,nf(fx)$, where n is the number to be increased. It works by doing one iteration on x “manually” by calling (fx), and then makes n do the rest of the iterations, thus one more iteration than just calling n is executed.

In $\lambda 2$ all Church numerals has type $\textsc{Nat} = \forall\alpha.\,((\alpha \to \alpha) \to \alpha \to \alpha)$, and zero and the successor function is defined as:

zero: $\Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,x$ of type $\textsc{Nat}$

succ: $\lambda n{:}\textsc{Nat}.\,\Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f\,(n\,\alpha\,f\,(f\,x))$ of type $\textsc{Nat} \to \textsc{Nat}$

It is not possible to define Church numerals in $\lambda\!\to$, so this shows that $\lambda 2$ is closer than $\lambda\!\to$ to the expressive power of untyped lambda calculus.

2.4 Predicative and Impredicative System

The type systems in this chapter are of the Church style, which means that the variables of lambda abstractions are annotated with types. Without these annotations, the terms of the language is the same as the untyped lambda calculus, but it is still possible to define a typing relation for them, and such a type system is said to be of the Curry style.

It can be argued that having to write out the type of every variable is cumbersome, so programming languages like Haskell and ML do type inference to reconstruct the types which the user have left out. But it has been proved that it is impossible to do type inference for $\lambda 2$ in a Curry style system [Wel94]. Thus the type systems of ML and Haskell have a limited form of polymorphism which is *predicative*, for which type inference is computable, in contrast to $\lambda 2$, which is *impredicative*.

In a type $\forall \alpha . \tau$, the type variable α quantifies over *all* types of $\mathbb{T}$, including itself: $\forall \alpha . \tau$. Because of this possibility of self-reference, $\lambda 2$ is called impredicative. In the type system of ML, it is only allowed to have quantifiers at the outermost position of types (called polytypes or type schema), and quantifiers only range over monotypes (types free of quantifiers).

Note that the type of the successor function in the implementation of natural numbers using Church encoding cannot be expressed in a predicative type system.

2.5 Type Operators

Another way to extend the simply-typed lambda calculus is to add *type operators*. The intuition is that type operators are functions from types to types. The function type, sum type, and product type of the simply-typed lambda calculus are type operators, but in a system with type operators one may add new type operators from within the system.

As an informal example, look at the untyped function $\lambda \alpha \, \beta \, . \, \alpha + \beta$. If it is applied to the types `int` and `bool`, the result is the type $\texttt{int} + \texttt{bool}$.

There are two questions which now arise: What are the types of α and β? And what is the type of the whole function? One way to answer this is by introducing *kinds*. The set of kinds is defined as $\mathbb{K} ::= * \mid \mathbb{K} \to \mathbb{K}$, where the kind $*$ denotes any type (such as `int`, $\texttt{int} \to \texttt{bool}$), while $\mathbb{K} \to \mathbb{K}$ denote a type operator from one kind to another. Thus in the informal example, it is the case that $\vdash \lambda \alpha{:}*, \; \beta{:}*. \; \alpha + \beta \; : * \to * \to *$. As with normal lambda abstractions, type operators are higher order.

The function above uses a mixture of types and terms, which is called pseudo-terms. The pseudo-terms of $\lambda \underline{\omega}$ are defined as: [2]

$$A, a, B, b ::= V \mid C \mid a\, b \mid \lambda V{:}A.\, b \mid A \to B$$

C is the set of constants, which contains at least $*$ and $\Box$, and also base types like `int` and its values. $*$ and $\Box$ are called *sorts*, and if one views types as sets, then $* = \mathbb{T}$ and $\Box = \mathbb{K}$. More formally we have the axioms $\vdash * : \Box$ and $\vdash \texttt{int} : *$ (and similar for all other base types), and two rules for forming new elements of a sort:

[2] The sum and product types has been dropped to focus on the essence of type operators.

$$\frac{\Gamma \vdash A\ : * \qquad \Gamma \vdash B\ : *}{\Gamma \vdash A \to B\ : *}\ \text{Type}{\to}\ \text{formation} \qquad \frac{\Gamma \vdash A\ : \square \qquad \Gamma \vdash B\ : \square}{\Gamma \vdash A \to B\ : \square}\ \text{Kind}{\to}\ \text{formation}$$

V is the set of variables for both values, types and type operators.

$\lambda V : A.\ B$ is the abstraction of a variable of any sort, and the rule for constructing a well sorted abstraction is almost the same as before, but there is an extra requirement: The resulting "type" $A \to B$ must either be a kind or a type. Throughout this and the next section, let s range over $*$ and $\square$ in the definitions of the rules.

$$\frac{\Gamma, x : A \vdash b\ : B \qquad \Gamma \vdash A \to B\ : s}{\Gamma \vdash \lambda x{:}A.\ b\ : A \to B}\ \text{Abstraction}$$

The rule for application, $a\,b$, is just as before:

$$\frac{\Gamma \vdash a\ : A \to B \qquad \Gamma \vdash b\ : B}{\Gamma \vdash a\,b\ : B}\ \text{Application}$$

The final interesting rule is the conversion rule. It says that if it can be proved that an expression A has sort B, and that sort B' is equivalent to B under beta reduction[3], then it can be concluded that A has sort B'. Formally:

$$\frac{\Gamma \vdash A\ : B \qquad \Gamma \vdash B\ : s \qquad B =_\beta B'}{\Gamma \vdash A\ : B'}\ \text{Conversion}$$

This rule causes the type system to regard types as equal if they are *structurally* equal, not only if they have the same name. E.g. $x : (\lambda\alpha{:}*.\ \alpha)\texttt{int} \vdash x\ : \texttt{int}$, and vice versa, $x : \texttt{int} \vdash x\ : (\lambda\alpha{:}*.\ \alpha)\texttt{int}$.

The final rules are assumption and weakening:

$$\frac{x \notin \Gamma \qquad \Gamma \vdash A\ : s}{\Gamma, x : A \vdash x\ : A}\ \text{Ass} \qquad \frac{x \notin \Gamma \qquad \Gamma \vdash A\ : B \qquad \Gamma \vdash C\ : s}{\Gamma, x : C \vdash A\ : B}\ \text{Weak}$$

A user of $\lambda\underline{\omega}$ may notice that it has a practical shortcoming: it is not possible to define a type operator and use it throughout an expression. For instance, it would be useful to bind the definition of the `list`-type operator to a variable so that it can be used throughout the code, i.e.:

```
(λlist : * → * .
     let nil : list α := ...
     let cons : α → list α → list α := ...
         ...)
```

The problem now is that the function $\lambda\texttt{list}{:}* \to *.\ \cdots$ is typed $(* \to *) \to \tau$ for some type τ, which is not a well typed function in $\lambda\underline{\omega}$.

Recall that the Λ abstraction from $\lambda2$ is essentially what is needed: a functions from types to values. The combination of $\lambda2$ and $\lambda\underline{\omega}$ into one system, called $\lambda\omega$ (Lambda Calculus Omega, equivalent to System Fω), is a quite practical type system, where for instance the above is well-typed.

[3]A single beta reduction step $\to_\beta$ is the rewrite $(\lambda x.M)N \Rightarrow M[N/x]$ applied anywhere in a term. The relation $\twoheadrightarrow_\beta$ is the reflexive, transitive closure of $\to_\beta$. And $=_\beta$ is the symmetric, transitive closure of $\twoheadrightarrow_\beta$.

Example of User Defined Ordered Pair in $\lambda\omega$

In this example, a definition for ordered pairs will be given which is well-typed in $\lambda\omega$. To get an intuition of the implementation, the definitions in untyped lambda calculus is first given:

$$\begin{aligned}\text{make-pair} &\equiv \lambda x, y\,.\,\lambda f.\; f\,x\,y\\ \text{first} &\equiv \lambda p\,.\,p\,(\lambda x,\, y.\; x)\\ \text{second} &\equiv \lambda p\,.\,p\,(\lambda x,\, y.\; y)\end{aligned}$$

This can be expressed in $\lambda\omega$, where each function would have the following type (and kind for pair):

$$\begin{aligned}\text{pair} &:: * \to * \to *\\ \text{make-pair} &:: \forall\alpha : *.\forall\beta : *.(\alpha \to \beta \to (\text{pair}\ \alpha\,\beta))\\ \text{first} &:: \forall\alpha : *.\forall\beta : *.((\text{pair}\ \alpha\,\beta) \to \alpha)\\ \text{second} &:: \forall\alpha : *.\forall\beta : *.((\text{pair}\ \alpha\,\beta) \to \beta)\end{aligned}$$

And the following definitions:

$$\begin{aligned}\text{pair} &\equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \forall\sigma{:}*.\,[(\alpha \to \beta \to \sigma) \to \sigma]\\ \text{make-pair} &\equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \lambda x{:}\alpha.\ \lambda y{:}\beta.\ \lambda\sigma{:}*.\ \lambda f{:}\alpha \to \beta \to \sigma.\ fxy\\ \text{first} &\equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \lambda p{:}\text{pair}\ \alpha\,\beta.\ p\,\alpha\,(\lambda x{:}\alpha.\ \lambda y{:}\beta.\ x)\\ \text{second} &\equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \lambda p{:}\text{pair}\ \alpha\,\beta.\ p\,\beta\,(\lambda x{:}\alpha.\ \lambda y{:}\beta.\ y)\end{aligned}$$

It is remarkable that given the types of a polymorphic function, then it is possible to construct a theorem which any function of that type will have [Wad89]; for instance, a function with the same type as "first" *must* be the first projection from pairs.

2.6 Dependent Types

The final extension will be dependent types, which is types depending on values. While programmers are somewhat familiar with polymorphism and type operators, dependent types is probably unfamiliar.

An example of how dependent types could be used, would be lists where the length of a list is a part of its type. Then nil would have the type list(0), and cons would have the type $A \to \text{list}(n) \to \text{list}(n+1)$.

The syntax of pseudo-terms in λP is as follows:

$$\mathcal{T} ::= V \mid C \mid \mathcal{T}\,\mathcal{T} \mid \lambda V{:}\mathcal{T}.\ \mathcal{T} \mid \Pi V{:}\mathcal{T}.\ \mathcal{T}$$

The only difference from $\lambda\underline{\omega}$ is that the arrow-type has been replaced with the more general constructor for cartesian product type, $\Pi x{:}\mathcal{T}.\ \mathcal{T}'$, which is the type of a lambda abstraction expecting an argument a of type or kind $\mathcal{T}$, and returning a result of type $\mathcal{T}'[a/x]$.

So a normal function from `int` $\to$ `bool` would be encoded as $\Pi x{:}$`int`. `bool`. If this function is called with 5, the resulting type would be `bool`$[5/x]$ which is `bool`. In general can any function of type $A \to B$ be encoded as $\Pi x{:}A.\ B$, where x is not free in B.

Functions in λP can be from values to values (as before), but also from values to types. This is enforced by the following rule for constructing cartesian products.

$$\frac{\Gamma \vdash A\ :\ast \qquad \Gamma, x{:}A \vdash B\ :s}{\Gamma \vdash (\Pi x{:}A.\, B)\ :s}\ \textsc{Type/kind formation}$$

The set $\mathbb{K}$ can thus be defined as $\mathbb{K} ::= \star \mid \mathbb{T} \to \mathbb{K}$.

The rule for applications is similar to the old ones, but with the important difference that the argument is substituted into the return type of the function.

$$\frac{\Gamma \vdash F\ :(\Pi x{:}A.\, B) \qquad \Gamma \vdash a\ :A}{\Gamma \vdash Fa\ :B[a/x]}\ \textsc{Application}$$

The final rule is for abstractions, which also is similar to the earlier rules for abstraction.

$$\frac{\Gamma, x{:}A \vdash b\ :B \qquad \Gamma \vdash (\Pi x{:}A.\, B)\ :s}{\Gamma \vdash (\lambda x{:}A.\, b)\ :(\Pi x{:}A.\, B)}\ \textsc{Abstraction}$$

Example of Intuitionistic Implicational Logic in λP

As an example, intuitionistic implicational logic will be encoded inside λP. We will do this by defining the initial assumptions Γ_0 and describe the intuition behind them, and by describing how to interpret that a formula is valid.

$$\Gamma_0 := \text{prop}\ :\ \ast, \tag{2.1}$$
$$\cdot \supset \cdot\ :\ \text{prop} \to \text{prop} \to \text{prop}, \tag{2.2}$$
$$T\ :\ \text{prop} \to \ast, \tag{2.3}$$
$$\supset_i\ :\ \Pi\varphi{:}\text{prop}.\, \Pi\psi{:}\text{prop}.\, (T\varphi \to T\psi) \to T(\varphi \supset \psi), \tag{2.4}$$
$$\supset_e\ :\ \Pi\varphi{:}\text{prop}.\, \Pi\psi{:}\text{prop}.\, T(\varphi \supset \psi) \to T\varphi \to T\psi \tag{2.5}$$

A proposition in this encoding has type prop, thus 2.1 says that prop is a type. The variable $\supset$, as defined in 2.2, says that given two propositions φ and ψ, $\varphi \supset \psi$ is also a proposition, and the intuition is that it denotes "φ implies ψ".

The dependent type T is used to denote that a proposition φ is valid, and is written $T\varphi$. Thus the assumption of a variable of type $T\varphi$ is equivalent to the assumption that φ is valid.

A function f from $T(\varphi)$ to $T(\psi)$ is considered a proof for "φ implies ψ", thus the variable assumed in 2.4 makes it possible to construct an expression of type $T(\varphi \to \psi)$ from f, by writing $\supset_i\ \varphi\,\psi\,f$.

The reverse is given by the variable assumed in 2.5. If we have an expression (proof) e of type $T(\varphi \supset \psi)$ (for $\varphi \to \psi$), then a function converting proofs of $T(\varphi)$ to proofs of $T(\psi)$ can be constructed, and it is written as $\supset_e\ \varphi\,\psi\,e$.

Finally, a proposition φ is valid if the type $T(\varphi)$ is inhabited. I.e. to prove φ, an expression of type $T(\varphi)$ must given.

As an example, given an proposition A : prop, then $A \to A$ should be provable, which is the case since the type $T(A \supset A)$ is inhabited. The expression $\lambda x{:}T(A).\, x$ is of type $T(A) \to T(A)$, and by using $\supset_i$ we have that $\supset_i\ A\,A\,(\lambda x{:} T(A).\, x)$ is of type $T(A \supset A)$.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\mathrm{P}\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda\mathrm{P}2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\mathrm{P}\underline{\omega} \arrow[uu] \\
\lambda^{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda\mathrm{P} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 2.2: Barendregt's Lambda Cube

2.7 The Lambda Cube

We have seen three extensions of the type system $\lambda{\to}$: polymorphic types ($\lambda 2$), type operators ($\lambda\underline{\omega}$), and dependent types ($\lambda\mathrm{P}$). By combining these extensions, a total of eight different type systems can be made from $\lambda{\to}$. In Figure 2.2 the eight different type systems are put into relation to each other. In the bottom-left corner is the simply-typed lambda calculus, and each axis represents an extension.

We have already looked at $\lambda{\to}$, $\lambda 2$, $\lambda\underline{\omega}$, $\lambda\mathrm{P}$ and $\lambda\omega$. The most powerful type system in the cube is in the top-right corner and is called $\lambda\mathrm{P}\omega$, and it is equivalent to the Calculus of Construction, Coc, which is what Coq is based upon.

The Lambda Cube is actually defined using a common framework to formalize all the type systems. All the systems have the same syntax for expressions, which is the same as for $\lambda\mathrm{P}$. And all the systems have a common set of inference rules, which are similar to those of $\lambda\mathrm{P}$ and $\lambda\underline{\omega}$. The different extensions are realized by varying s and t over $*$ and $\square$ in the rule

$$\frac{\Gamma \vdash A \ : s \qquad \Gamma, x{:}A \vdash B \ : t}{\Gamma \vdash (\Pi x{:}A.\, B) \ : t}$$

which defines what kind of functions which may be constructed. All the systems have the version of the rule where $t = s = *$, which means that one can construct normal functions. With $s = \square$ and $t = *$, functions from types to values is accepted, which gives polymorphic types; and with $t = s = \square$, functions from types to types is accepted, which gives type operators; and with $t = *$ and $t = \square$, functions from values to types is accepted, which gives dependent types.

Chapter 3

About Ott

The use of languages defined by context-free grammars and relations over these languages is ubique in the field of programming language theory. They were used throughout the previous chapter, and are also used in almost all papers about programming language theory. Typical languages are the expressions and statements of programs, types of programs, and contexts for type checking, while typical relations over the languages are operational semantics and type systems. These relations are often given in the style of inference rules, which we also saw used throughout the previous chapter.

Ott is a tool created by Francesco Zappa Nardelli, Peter Sewell, and Scott Owens, which is intended to help researchers in fields related to programming language theory to formalize their work [SNO⁺07]. The main contributions of Ott is a domain-specific language for grammars and relations, which Ott can check for inconsistencies, can typeset in LaTeX, and compile to definitions and relations in the languages of Coq, HOL, Isabelle, and OCaml.

In addition to languages and relations, Ott can automatically generate functions for free variables in a term and for substitution of variables: both single and multi-variable substitution. Experimental support for defining functions and a locally nameless Coq-backend also exists.

The advantage of using Ott to typeset grammars and judgments, as opposed to writing them directly in LaTeX, is that Ott performs some saneness checks when defining relations. By considering the grammar as simple type declaration, Ott can check that the definition of relations is unambiguous and that different kinds of expressions are used consistently.

If one uses Ott to produce code for a combination of LaTeX, Coq, and the other supported back ends, the additional benefit is that there is a single definition (the Ott files), which can be translated to LaTeX, Coq, etc. This means that grammars and relations are consistent between the typeset version in LaTeX, the theorem prover version in for instance Coq, and the implementation version in OCaml.

A general disadvantage with Ott is that one loses the fine-grained control one have when type-setting manually in LaTeX and writing Coq-code manually.

A short introduction to Ott is now provided. For further information see the paper [SNO⁺07] and the user guide[1].

[1]`http://www.cl.cam.ac.uk/~pes20/ott/ott_manual_0.22.pdf`

$$\begin{aligned} e &::= x \mid e\,e \mid \lambda x.e && \text{expression}\\ t &::= \text{int} \mid t \to t && \text{type}\\ \Gamma &::= \epsilon \mid \Gamma, x : t && \text{context}\end{aligned}$$

Figure 3.1: BNF grammar for the simply-typed lambda calculus.

```
metavar variable, x ::= {{com Variable}}

grammar
expr, e :: 'e_' ::= {{com Expression}}
     | x               ::  :: Var {{com Variable}}
     | e e'            ::  :: App {{com Application}}
     | lambda x . e    ::  :: Abs {{com Abstraction}}
                                {{tex \lambda [[x]] . [[e]]}}
     | ( e )           ::  :: Paren {{com Parenthesis}}

type, t {{tex \tau}} :: 't_' ::= {{com Type}}
     | int             ::  :: Int {{com Integer}}
     | t -> t'         ::  :: Arrow {{com Function}}
                                {{tex [[t]] \to [[t']]}}

gamma, G {{tex \Gamma}}:: 'G_' ::= {{com Context}}
     | Empty           ::  :: Empty {{com Empty context}}
                                {{tex \epsilon}}
     | G , x : t       ::  :: Extend {{com Extension}}
```

Figure 3.2: OTT code for the BNF from Figure 3.1.

3.1 Grammars in Ott

Backus-Naur Form (BNF) [NBB+63] is the standard way of describing the syntax of a programming language. BNF is a notation for *context-free grammars* (CFG), a finite description of a language, which describes *context-free languages.* Almost all programming languages can be defined using context-free grammars. These languages correspond to the languages which parsers of acceptable (polynomial) time and space complexity can recognize.

Formally, the symbols of a CFG is divided into *terminals* and *non-terminals*, where the terminals represent symbols which appear in the language, while non-terminals represent sets of words.

A typical definition of the language for the simply-typed lambda calculus is given in Figure 3.1, where "`e`", "`t`", and "`G`" are non-terminals, and "λ", "`.`", "`(`", "`)`", "`int`", "$\to$", "ϵ", "`,`", and "`:`" are terminals. The status of `x` is not given by the grammar, but it is implicitly understood that it is a non-terminal for some set of symbols which is not defined here.

The translation of this grammar to OTT is shown in Figure 3.2. In the first line, we see that `x` is defined as a *meta-variable*, which means `x` is a non-terminal

variable, x	Variable		
expr, e	::=		Expression
	|	x	Variable
	|	$e\,e'$	Application
	|	$\lambda x.e$	Abstraction
	|	(e)	Parenthesis
type, τ	::=		Type
	|	**int**	Integer
	|	$\tau \rightarrow \tau'$	Function
gamma, Γ	::=		Context
	|	ϵ	Empty context
	|	$\Gamma, x : \tau$	Extension

Figure 3.3: Result of Ott compiling code from Figure 3.2 to LaTeX.

which is not defined by the grammar. Notice the double braces which indicate what Ott calls a *homomorphism*. Homomorphisms usually give some way to translate a symbol or rule into LaTeX or Coq. One can access the "default" homomorphism for an Ott term with the double square brackets, written `[[...]]`, as we will see later. In this case, the homomorphism is for a comment. The `grammar`-keyword indicates that next follows grammars — three in our case.

Each grammar starts with a list of non-terminal names. It is possible to give several names for the same non-terminals. One could for instances have given an extra name, `s`, for the types, which could then be used instead of `t'`. We also see a new homomorphism, `tex`, which is how a rule or symbol should be translated to LaTeX. In the case of `t` and `G`, they are simply translated to another LaTeX symbol. The rule for abstractions and arrow types are more advanced, here the double brackets `[[t]]` means that the LaTeX-representation of `t` should be inserted at that place.

The result of compiling this Ott-file (together with the previous one), results in the output we see in Figure 3.3.

3.2 Relations in Ott

Given a grammar, Ott allows us to define judgments over them, and the notation mimics how judgments are usually written as pure text:

$$\begin{array}{l} \texttt{Premise}_1 \\ \quad\vdots \\ \texttt{Premise}_n \\ \texttt{---------------------------\ ::\ RuleName} \\ \texttt{Conclusion} \end{array}$$

The premises must be on the form described by the non-terminal `formula`, which by default only contains the non-terminal `judgment`, where each judgment in the file is automatically added. By redefining the non-terminal `formula`, one can add those propositions which are needed.

```
grammar
formula :: formula_ ::=
    | judgement        :: :: judgement
    | x notIn FV( e ) :: :: NotFree
                        {{tex [[x]] \notin FV([[e]])}}

defns
    type_judgement :: '' ::=
    defn
    G |- e : t :: ::type_judgement:: ''
        {{tex [[G]] \vdash [[e]] : [[t]] }} by

    ---------------- :: T_Var
    G, x:t |- x:t

    G |- e : t
    x notIn FV(e)
    ---------------- :: T_Weak
    G, x:t' |- e : t

    G |- e  : t' -> t
    G |- e' : t'
    ----------------- :: T_App
    G |- e e' : t

    G, x:t' |- e : t
    ----------------- :: T_Abs
    G |- lambda x . e : t' -> t
```

Figure 3.4: OTT source for typing judgments with user-defined `formula`.

The OTT code for the rules of our running example of the simply-typed lambda calculus is given in Figure 3.4. For the weakening rule, a new formula stating that the variable `x` does not occur freely in the expression `e` must be added to `formula`. The LaTeX output of these rules is shown in Figure 3.5.

3.3 Generating Coq Code

With the OTT-file that we now have, would it be nice to generate source code to a theorem prover assistant. However, there are some things which are missing. We will only concern ourselves with translating to COQ.

The idea of the translation is that each non-terminal is translated to an inductively defined type, living in `Set`[2], while each judgment is translated to

[2]`Set` is the universe in COQ intended for "data-types". COQ also have the universe `Prop`, intended for proposition, and an cumulative hierarchy of universes $\texttt{Type}_i$. When specifying a universe $\texttt{Type}_i$, the index can be omitted, and COQ will infer it. The universes are related as $\texttt{Set} \subseteq \texttt{Type}_1$, $\texttt{Prop} \subseteq \texttt{Type}_1$, and $\texttt{Type}_i \subseteq \texttt{Type}_{i+1}$.

$$\begin{array}{lcl} \textit{judgement} & ::= & \\ & | & \textit{type\_judgement} \\ & & \\ \textit{formula} & ::= & \\ & | & \textit{judgement} \\ & | & x \notin FV(e) \end{array}$$

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{T\_V{\scriptsize AR}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \\ x \notin FV(e)\end{array}}{\Gamma, x : \tau' \vdash e : \tau}\ \text{T\_W{\scriptsize EAK}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau' \to \tau \\ \Gamma \vdash e' : \tau'\end{array}}{\Gamma \vdash e\ e' : \tau}\ \text{T\_A{\scriptsize PP}}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\ \text{T\_A{\scriptsize BS}}$$

Figure 3.5: LaTeX output from OTT code in Figure 3.4.

an inductively defined family of types, indexed by its arguments, and living in `Prop`.

We can notice that there is some information missing in our OTT source code (Figure 3.2 and 3.4) to do this translation, namely, what `x` is and what the formula $x \notin FV(e)$ means.

Again homomorphisms come to our rescue. Adding the homomorphisms `{{coq nat}}` and `{{ coq-equality }}` to `x` tells OTT how `x` should be translated to COQ. The line introducing `x` should be changed to:

```
metavar variable , x ::= {{coq nat}} {{ coq-equality }}
```

The second problem is the $x \notin FV(e)$ formula, which is solved by adding a homomorphism describing how to translate it into COQ. This can be done with the homomorphism

```
          {{coq (~In [[x]] (FV [[e]]))}}
```

where `In` is COQ's standard list membership predicate, and `FV` should be a function giving all the free variables in the term `e`. We can make OTT generate this function automatically for us, by adding the lines

```
freevars
  expr x :: FV
```

which creates a function `FV`, from `expr` to a list of variables, collecting all free variables into a list.

One last thing to change, is the parentheses production rule, as we want the data type `expr` to be the abstract syntax tree. We should therefore change the line

```
  | ( e )          ::  :: Paren {{com Parenthesis}}
```

to

```
  | ( e )          :: S:: Paren {{com Parenthesis}}
                                {{icho [[e]]}}
```

meaning that the parenthesizes are only used for parsing, and the `icho`-homomorphism says that it should be translated to that of `e`.

The source code generated from all of this is shown in Figure 3.6. Notice how each of the non-terminals `expr`, `type`, and `gamma` correspond to an inductive definition in the output. OTT is smart enough to know that it must reorder the inductive definitions so that `type` comes before `expr`. If there are mutual recursive grammars, then OTT will also handle this.

```
(* generated by Ott 0.21.2 from: simple.ott *)

Require Import Arith.
Require Import Bool.
Require Import List.


Definition variable := nat. (*r Variable *)
Lemma eq_variable: forall (x y : variable), {x = y} +↩
   ↪ {x <> y}.
Proof.
  decide equality; auto with ott_coq_equality arith.
Defined.
Hint Resolve eq_variable : ott_coq_equality.

Inductive type : Set :=  (*r Type *)
 | t_Int : type (*r Integer *)
 | t_Arrow (t:type) (t':type) (*r Function *).

Inductive gamma : Set :=  (*r Context *)
 | G_Empty : gamma (*r Empty context *)
 | G_Extend (G:gamma) (x:variable) (t:type) (*r ↩
    ↪Extension *).

Inductive expr : Set :=  (*r Expression *)
 | e_Var (x:variable) (*r Variable *)
 | e_App (e:expr) (e':expr) (*r Application *)
 | e_Abs (x:variable) (e:expr) (*r Abstraction *)
 | e_Paren (e:expr) (*r Parenthesis *).
(** definitions *)

(* defns type_judgement *)
Inductive type_judgement : gamma -> expr -> type -> ↩
   ↪Prop :=    (* defn type_judgement *)
 | T_Var : forall (G:gamma) (x:variable) (t:type),
     type_judgement (G_Extend G x t) (e_Var x) t
 | T_Weak : forall (G:gamma) (x:variable) (t':type) (↩
    ↪e:expr) (t:type),
     type_judgement G e t ->
     (formula_NotFree x e) ->
     type_judgement (G_Extend G x t') e t
 | T_App : forall (G:gamma) (e e':expr) (t t':type),
     type_judgement G e (t_Arrow t' t) ->
     type_judgement G e' t' ->
     type_judgement G (e_App e e') t
 | T_Abs : forall (G:gamma) (x:variable) (e:expr) (t'↩
    ↪ t:type),
     type_judgement (G_Extend G x t') e t ->
     type_judgement G (e_Abs x e) (t_Arrow t' t).
```

Figure 3.6: Generated Coq-output from Ott.

Chapter 4

Embedding of Languages

During formalization, we work with *two* languages, the *object language* and the *meta language*. The object language is the language we are talking about, while the meta language is the language we are talking in. For us, the object language will be the functional programming language with threads and locks, which is described in [PSS13], and the meta language with be those of Coq and Ott.

The main problem of formalization is to implement the meaning of the object language into the meta language. The approaches of how to do this can be categorized along an axis of *shallow* and *deep* embedding [BGG+92]. In practice, a combination of both is usually used.

As an illustration, consider the problem of formalizing propositional intuitionistic logic, where the formulas can be defined as

$$\varphi ::= P \mid \varphi \to \varphi,$$

where P is a set of propositions. The following are two ways to formalize the claim "φ is provable":

Deep embedding Use Coq's inductive type construction to define the syntax of formulas φ and the syntax of contexts Γ, and also define the judgments for intuitionistic propositional logic, $\Gamma \vdash \varphi$, as an inductive relation.

Shallow embedding In some programming language, define a function which translates formulas φ to the Coq-term $\forall P_1 : \text{Prop}, \cdots, P_n : \text{Prop}; \varphi$, where $P_1, \cdots P_n$ are the free variables of φ.

The proposition φ is provable if the corresponding Coq-type is inhabited.

A more in-depth description of each of these approaches follows.

4.1 Deep Embedding

Deep embedding is to model the logic in the meta language. This includes defining both the syntax and the semantics of the object language in the meta language.

For a deep embedding of the intuitionistic logic mentioned above, we need to define both the syntax and what it means for this to be provable in Coq. The formalization is shown in Figure 4.1. The syntax of formulas is defined as an

```
Inductive prop : Set :=
    | Var (n:nat) : prop
    | Imp (p q:prop) : prop.

Definition context := list prop.
Notation "G , p" := (cons p G) (at level 40).

Reserved Notation "G |- p" (at level 40).
Inductive judge : context -> prop -> Prop :=
    | jAxiom (G:context) (p:prop) :
        G, p |- p
    | jWeak (G:context) (p q:prop) :
        G |- p ->
        G, q |- p
    | jIntro (G:context) (p q:prop) :
        G, p |- q ->
        G |- (Imp p q)
    | jElim (G:context) (p q:prop) :
        G |- (Imp p q) ->
        G |- p ->
        G |- q
    where "G |- p" := (judge G p).
```

Figure 4.1: Deep embedding of provability of intuitionistic propositional logic.

inductive type and the rules of the judgment $\Gamma \vdash \varphi$ is defined as an inductive relation on propositions and contexts.

With this approach, everything is formalized in Coq, and statements about the object language can be formulated. Especially if the derivations are placed in the universe `Set` or `Type`, then statements about and transformations on derivations can be made. This is essential for expressing properties such as cut-elimination, and other properties about the structure of derivations.

The main disadvantage with deep embedding is that everything is built from the ground, even things which might not be interesting. For instance, if one does not really care about the structure of derivations, only about derivability, then shallow embedding might be more fitting.

4.2 Shallow Embedding

In a shallow embedding, the formulas of the logic are translated directly to Coq formulas. For a "proper" shallow embedding, this translation happens outside of Coq, for instance by Ott. If this is the case, then the elements of the object language will not be entities in the meta language, so one does not have the possibility to reason about for instance transformations of the syntax.

Derivations will happen inside the logic of Coq, which means that some properties of the formalized systems will be inherited from Coq. For instance, if the object language is some typed lambda calculus, and the lambda terms are translated directly to Coq's lambda terms, then it is guaranteed that accepted translations are terminating.

It can be argued that to implement a shallow embedding, a more in-depth knowledge about the meta language is needed. One must be aware of whether intuitionistic or classical reasoning is needed, and whether one needs functional extensionality[1]. When shallow embedding is applicable, it can lead to short formalizations, as one reuses the meta language's functionality.

A downside with shallow embedding, is that the object language is "lost" through the translation. In a deep embedding, one can reason about the object language, as it is a distinct data type defined in Coq, as opposed to some "informally" restricted class of Coq terms.

A translation happening inside Coq it can still be considered a shallow embedding; as mentioned in the introduction, often a mixture of deep and shallow embedding is used.

A shallow embedding of our example with intuitionistic propositional logic is given in Figure 4.2. The idea for the translation is to create the Coq-proposition $\forall \vec{p}, \varphi$, where `Imp p q` is translated to Coq-implication, and each variable `P` is translated to the corresponding bound variable.

The main part of the translation happens in the `den_prop` functions which takes a formula `p` and a mapping from variables to Coq-propositions `G`, and creates a Coq-proposition for `p`.

To create the mapping of variables, the function `den_quantify` will add an universal quantifier for each free variable in `p`, and update the mapping.

There are a few things to note:

1. When classical axioms are assumed in Coq, then suddenly the denotation is for classical propositional logic.

2. Since the translation gives us simple propositions of `Prop`, all the tactics of Coq are available, especially, decision procedures for propositional logic.

3. Changing the quantifier in `den_quantify` from an universal to an existential changes the meaning from validity to satisfiability.

[1]Functional extensionality means that if two functions f and g are pointwise equal, then f and g are propositionally equal.

```
Require Import Arith ListSet.

Fixpoint free (p:prop) : set nat :=
    match p with
    | Var n => set_add eq_nat_dec n (empty_set _)
    | Imp p q => set_union eq_nat_dec (free p) (free q)
    end.

Definition context := nat -> Prop.
Definition extend (G:context) x value x' :=
    if beq_nat x x' then value else (G x').

Fixpoint den_prop (G:context) (p:prop) : Prop :=
    match p with
    | Var n => (G n)
    | Imp q r => (den_prop G q) -> (den_prop G r)
    end.

Fixpoint den_quantify (vars:set nat) (G:context)
        (p:prop) : Prop :=
    match vars with
    | nil => den_prop G p
    | (cons n vars') =>
        forall (n':Prop),
            den_quantify vars' (extend G n n') p
    end.

Definition denote (p:prop) : Prop :=
    den_quantify (free p) (fun _ => False) p.
```

Figure 4.2: Shallow embedding of provability of intuitionistic propositional logic.

Chapter 5

Deadlock Detection by Reduction to Data Race Condition Checking

The contribution of this thesis is the formalization of some of the results presented in the article [PSS13]; this chapter is an introduction to the work which is formalized. First an overview of the problem is given, then the syntax, rules, and proofs are presented. For an in-depth explanation of what is described in this chapter, see the above reference. Problems related with the formalization are not discussed in this chapter, but in the next chapter.

5.1 Overview

The paper introduces the novel idea to reduce deadlock checking to race condition checking. They claim that state of the art static detection of race conditions are further ahead than static deadlock detectors. Therefore, the reduction takes advantage of the research done for race condition detection, and applies it to deadlock detection.

The first part of the reduction is to extract information about the use of locks in the program, specifically, to approximate how many times a reentrant lock may be taken. There are several ways this could be done, but in the technical report, this is done using a type and effect system, together with constraints.

The second part, is to use the information from the type and effect system, to insert variables into the program, which in the case of a deadlock in the original program will cause a race condition.

It is the first part which is the focus of my work; formalizing the type and effect system, and the proofs that are given.

5.2 The Language

The language of study is a statically typed, functional programming language with threads and reentrant locks.

The class P represents programs, which are essentially sets of processes. P can either be the empty program $\varnothing$, a single thread $p\langle t\rangle$, where p is a process identifier and t is a thread, or the composition of two programs $P \| P$. The process identifier for threads are assumed to be unique for each thread.

The expressions of the language is divided into three categories, threads t, expressions e, and values v, similar to the notion of A-normal form (ANF)[1] [FSDF93], which is related to continuation-passing style (CPS). This restriction on the form of expressions does not cause a loss in expressiveness, as there exists an effective conversions from general lambda terms into ANF-form. The advantage of the A-normal form, is that the semantics, type systems, and proofs, have stronger assumptions to work with, which makes them more elegant and simple.

The grammar for the abstract syntax is given in Figure 5.1, and consists of:

- variables x, truth values, and lock references (the labeling r of lock references is only used by the proofs about type systems);
- a let binding constructions, which are needed because of a poly-let type system [2],
- both recursive and non-recursive abstraction;
- function application;
- conditional expressions;
- spawning threads;
- locking and unlocking.

The underlying type system is not really of interest, and is not specified, but it can be inferred to be a minimally, simply-typed lambda calculus with explicit, general recursion. Terms are given in the style of Church, i.e. variable binders are annotated with types.

5.3 Semantics of the Language

The semantics of the language is given as a small step operational semantics. The semantic is divided into “local” and “global” rewrite rules. The “local” rules describe the steps done by a single thread which has no effect on other threads and which cannot be affected by other threads, while the “global” rules describe the steps which requires synchronization between threads, or which may affect other threads.

Note that the semantics has an emphasis on storing information about locks, and does not store any information about the values of variables, as they are substituted with their value.

[1]ANF usually just have two categories, expressions and values.

[2]In some type systems, the expression let $x = e$ in e' is different from $(\lambda x.e')\ e$, because of let-polymorphism, as described in Chapter 2.

$P ::= \varnothing \mid p\langle t\rangle \mid P \| P$	Program
$t ::= v$	Value
$\mid \texttt{let } x : T = e \texttt{ in } t$	Sequential Composition
$e ::= t$	thread
$\mid v\, v$	application
$\mid \texttt{if } v \texttt{ then } e \texttt{ else } e$	conditional
$\mid \texttt{spawn } t$	spawn thread
$\mid \texttt{new L}$	new lock
$\mid v\texttt{.lock}$	acquire lock
$\mid v\texttt{.unlock}$	release lock
$v ::= x$	variable
$\mid l^r$	lock reference
$\mid \texttt{true} \mid \texttt{false}$	truth values
$\mid \texttt{fn } x : T.t$	function
$\mid \texttt{fun } f : T, x : T.\ t$	recursive function

Figure 5.1: Abstract syntax of programs.

5.4 Type and Effect Systems

The paper defines three type systems: a specification, a syntax directed, and an algorithmic one. The goal of all of the systems, is to approximate how many times a lock is taken, and at which places in the program.

5.4.1 Specification Type System

The first type system given is used as a specification. It is written in a declarative style, with an emphasis on being general and easy to understand. The judgments are on the form

$$C; \Gamma \vdash t : \hat{S} :: \Delta \to \Delta'$$

where C is the set of constraints, Γ is the context mapping variables to type schemes, t is a thread, S is the resulting type scheme, and $\Delta \to \Delta'$ is the effect. Each Δ correspond to an approximation of how many times a lock has been taken, and the effect should be read as "if program t starts evaluating in state Δ, then if it terminates, it will end in state Δ'".

5.4.2 Algorithmic Type System

The second type system given is used as an algorithm. Even though it is written with judgment-rules-notation, it should be suitable as a guide for an implementation.

The main difference between the algorithm and specification, is that the algorithm is deterministic, while the specification is non-deterministic. This is the case because the algorithm is syntax directed; there is only one rule which may be used to fulfill a judgment. This is opposed to the specification, where the rules for instantiating a type schemes, generalization a type, and subsumption can be applied no matter what the term to be type checked is.

Another difference from the specification, is that the algorithm version will compute the minimal set of constraints, as opposed to the specification, where the set of constraints must be given a priori.

5.4.3 Syntax Directed Type System

The syntax directed type system is used as a mean to prove that the specification and algorithm are related. It is syntax directed, just as the algorithm, but it does not generate constraints, just like the specification.

5.5 Theorems and their Proofs

There are five major theorems and lemmas associated with the type systems and the semantics. They are:

5.5.1 Soundness

The first major theorem and proof which appear is the soundness, relating the algorithmic type system and the specification. It is stated as:

Theorem 1 (Soundness). *Given* $\Gamma \vdash_a t : \hat{T} :: \Delta_1 \rightarrow \Delta_2; C$*, then* $C; G \vdash_s t : \hat{T} :: \Delta_1 \rightarrow \Delta_2$*. Where* $\vdash_a$ *is the algorithmic type system, and* $\vdash_s$ *is the specification.*

In other words, if the algorithm is given a term and generates a type, an effect and a constraint, then the specification will hold for the same parameters.

The proof is given by induction on the structure of algorithmic proofs.

5.5.2 Completeness

The second major theorem and proof, is the completeness proof, relating the syntax directed type system with the algorithm. It is stated as:

Theorem 2 (Completeness). *Assume* $\Gamma \lesssim_\theta \Gamma'$, $\Delta_1 \lesssim_\theta \Delta_1'$*, and* $C; \Gamma \vdash_n t : \hat{T} :: \Delta_1 \rightarrow \Delta_2$*, then* $\Gamma \vdash_a t : \hat{T}' :: \Delta_1' \rightarrow \Delta_2'; C'$ *such that*

1. $C \vDash_{\theta'} C'$,

2. $C \vdash \theta' \hat{T}' \leq \hat{T}$*, and*

3. $C \vdash \theta' \Delta_2' \leq \Delta_2$,

for some $\theta' = \theta, \theta''$*.* $\vdash_n$ *is the syntax directed type system.*

This claim is more involved than soundness, since the syntax directed type system will accept many constraints and types, while the algorithm will compute minimal constraints and types.

5.5.3 Subject reduction

The final major theorem is the subject reduction, which means that evaluating a term preserves its type. This property is proved together with a simulation property: the effect simulates the usage of locks by the semantics.

Chapter 6

Formalization

This chapter discusses the choices which had to be made during the formalization of the type systems. When there are more than one viable way to do the formalization, the different choices are presented, and what was chosen will be made clear.

6.1 Syntactical Constructions

The syntactical classes are formalized in OTT. This includes expressions, simple types, annotated types, type schemes, lock sets, lock environments, effects, and constraints.

Note that in this chapter, the phrase "syntactical representation" will be used to mean the inductive data type which closely resembles the grammar. This is opposed to an "abstract representation", where the syntactical object is represented by a more abstract type, for instances sets, which should have inherent properties that are wanted.

6.1.1 Programs

There are two main challenges for formalizing programs. Each problem will be considered separately.

The Parallel Composition Operator

The first challenge is that the $\|$ operator on programs should be associative and commutative, and should have $\varnothing$ as left and right identity. These properties cannot be stated directly in OTT and COQ. The following choices were considered to address the algebraic properties of programs:

Data Type for Sets Representing programs with a data type for finite sets, for instance COQs `MSet` is a feasible solution. Given that one does not break any abstractions of the set, then all relations on programs will be compatible up to the appropriate equivalence relation. In addition, all existing lemmas about the data type are readily available.

Still, for most representations of finite sets, one cannot expect equivalence and identity to coincide.

Augment Rules The second solution is to represent programs as syntactical objects, and then augment the rules where the property is needed. Especially with regards to the semantics, one could then have a rule

$$\frac{P_1 \equiv P_1' \qquad \sigma \vdash P_1' \longrightarrow \sigma' \vdash P_2' \qquad P_2' \equiv P_2}{\sigma \vdash P_1 \longrightarrow \sigma' \vdash P_2,}$$

where $\equiv$ is the desired equivalence relation.

This approach is quite elegant and simple to implement. The downside is that the judgment rules will deviate from the "original" rules, and proofs over these judgments become more complex. Additionally, the relation $\equiv$ must be formalized.

Work on Equivalence Classes The final way is to again represent programs as syntactical objects, and then work on equivalence classes of programs, for a suitable relation $\equiv$. For instance, the R_PAR rule could be defined as

$$\frac{\sigma \vdash [P_1]_\equiv \longrightarrow \sigma' \vdash [P_1']_\equiv}{\sigma \vdash [P_1 \| P_2]_\equiv \longrightarrow \sigma' \vdash [P_1' \| P_2]_\equiv}$$

where $[P]_\equiv$ is the equivalence class of P under $\equiv$.

On the positive side, this rule follows the "meaning" of the informal rules. The downside is that one must prove that each judgment is compatible. For the soundness, this means that if $\sigma \vdash [P_1]_\equiv \longrightarrow \sigma' \vdash [P']_\equiv$ and $P_1 \equiv P_2$, then $\sigma \vdash [P_2]_\equiv \longrightarrow \sigma' \vdash [P']_\equiv$.

The idea of the first solution is used, but is extended, as explained below.

Uniqueness of Process Identifiers

The second challenge is that in a program P, each process identifier should only occur once, i.e. each process identifier uniquely identifies a thread. This cannot be formalized using OTTs grammar, but there are again several ways to solve it.

Predicate The first solution, is to define a relation over programs which check if the property hold. For instance collecting each identifier in a list, and control that no element is repeated.

This approach would need some auxiliary lemmas which state the consequences of a derivation having this property.

Dependent Types The second solution uses dependent types. This means that it has to be defined directly in COQ, not in OTT. Let the type of programs depend on a set of process identifiers, and define parallel composition so that it can only be constructed if the two sets are disjoint. An implementation of this is shown in Figure 6.1.

Mapping The final solution, which solves both this and the above problem, is to implement programs as a finite mapping from process identifiers to threads. This has the advantage that it captures the idea of the process identifiers.

```
Require Import ListSet Arith.
Inductive thread : Set := Thread.
Definition process_id : Set := nat.

Notation "{}" := (empty_set process_id).
Notation "{ p }" := (set_add eq_nat_dec p (empty_set _)).
Definition union S U := set_union eq_nat_dec S U.
Definition inter S U := set_inter eq_nat_dec S U.

Inductive program : set process_id -> Type :=
    | EmptyProgram : program {}
    | Single (p:process_id) (t:thread) : program {p}
    | Parallell {S U} (P1 : program S) (P2 : program U) :
        (inter S U) = {} -> program (union S U).
```

Figure 6.1: Implementation of the program, with the requirement that process identifiers are unique.

Selected Formalization

It could be argued that the first property (associativity, commutativity, and identity) should not be inherent in the type of programs, but should rather be formalized elsewhere. The argument is that the type systems should in principle not depend on this.

A program has been formalized as a finite mapping from process identifiers to threads. The `Tree` data type from library of CompCert is used. Parallel composition is then union, and the empty program is the empty set. This means that parallel composition is associative, commutative, and has the empty program as identity. Also, this means that each process identifier uniquely identifies a thread. Both are properties that we want.

6.1.2 Type-Level Variables and Type Schemes

In the paper, the class of variables Y is defined as the combination of lock set variables and lock environment variables, which is then used to define type schemes as $\forall \vec{Y} : C.\hat{T}$. For reasons which are discussed in section 6.2, this has been changed to $\forall \vec{\rho}\vec{X} : C.\hat{T}$, where the vector $\vec{Y}$ has been split into two vectors, $\vec{\rho}$ and $\vec{X}$.

6.1.3 Threads, Expressions, Types, and Values

Threads, expressions, simple types, annotated types, program types, and values are represented directly and straightforwardly in OTT. The underlying types were formalized as those of simply-typed lambda calculus. The representation of variables is naïve; variables are identified by natural numbers, and there is no distinction between free and bound variables.

The expression `new L` should be annotated with a program point π. The point should be unique, but this is not forced by the current implementation. Not enforcing this will only mean that the program can contains less “information”, so the analysis may not give the most precise answer. Solutions to this are the same as those which enforces unique process identifiers in the `program`-type.

6.1.4 Constraints

If one chose a syntactical representation of constraints, there is still a choice to be made for how to formalize the semantics of $C \vDash C'$. It is also possible to have a non-syntactical representation of the constraints. Regarding this, the following possibilities were considered:

1. Constraints can be represented with "syntactical" objects, and an axiomatization of the judgment $C \vdash C'$ can be given. Since we are not doing research about the fundamentals of constraint satisfaction, this approach only needs a sound axiomatization, completeness is not needed. Typical properties axiomized would be that the order of inequality does not matter, and similar.

2. Constraints can be represented with "syntactical" objects, and an implementation of an algorithm, for instance the *Worklist Algorithm* [NNH99], can be implemented to decide judgments.

   One of the major drawbacks with this approach would be that this would be a lot of work. Even though one correctly completes the implementation, one would still have to prove that it has the desired properties.

3. Constraints can be "syntactical" objects, and a function $[\![C]\!]$ can be given, which translates a constraint C to `Prop`. The translation would use an underlying representation for sets, for instance `MSet` from the standard library.

   If C and C' does not contain any free variables, the claim $C \vDash C'$ would be $[\![C]\!] \to [\![C']\!]$. If there are free variables in C or C', the translation could be

   $$\forall \theta,\ [\![\theta C]\!] \to [\![\theta C']\!],$$

   where θ is a mapping from variables to the underlying representation of sets.

   The main issue with this embedding, is that one must make sure that one is only able to prove the translated statements of those constraint-checks which actually hold. Given that we are working with finite sets, represented with a plain data type, this should hold.

4. A final solution is if constraints are translated directly to `Prop`, using the Coq-hom in Ott. This means there will not be a syntactic representation in Coq of the constraints.

   The problem with this is that the syntactic version seems to be needed, mainly because of the variables, as these are needed in type schemes, lock sets, and lock environments. So implementing this deep embedding would presumably cause larger changes in the representation of everything depending on lock sets and lock environments.

The first and the third solutions seem most useful, and it is the third which was chosen. To represent the constraints, the syntax of inequalities is formalized by Ott, and a constraint is then a list of inequalities.

6.1.5 Lock Environments

It could be tempting to represent the lock environments as a mapping from lock sets to a number. This is problematic; first of all, there is a state ordering over the syntax of lock environments; and secondly, they are really lock environment *expressions*, so if they contain a free variable, the translation to a single mapping cannot be done.

Therefore, a syntactical representation is used, faithful to the "original" definition from [PSS13]. It could be the case, that the form $\Delta, r : n$ could be restricted to $\Delta, \rho : n$, which is indicated by [PSS13], but this has not been done.

6.1.6 Lock Sets and Effects

The grammar for lock sets, and effects are implemented directly as syntactical objects in OTT.

Note that lock sets are *expressions* which represent sets; one cannot represent a lock set as a finite set of lock references, unless one manage to represent lock set variables with COQ's variables, in other works, with a shallow embedding.

6.1.7 Context

Contexts are defined as lists, where new bindings are added to the right, and a relation is defined in COQ for variable look up. It would also be possible to define contexts as a finite mapping from variables to type schemes. For simplicity, the context is defined as a syntactical object in OTT, and a judgment $\Gamma(x) = \hat{S}$ is defined, by induction over the context.

6.2 Substitutions

The paper uses substitutions, but there is no elaboration about their meaning. The substitutions are from several lock set variables and lock environment variables, to respectively lock sets and lock environments. The substitutions are applied to to constraints, annotated types, type schemes, and therefore implicitly also to lock sets and lock environment as well.

Via OTT, the substitutions are given by four finite lists, one for lock set variables, and a corresponding one for lock sets, and similar for the lock environments. Trying to model the approach of the paper, where the different kind of variables are mixed together, would require either a custom list type, or heavy use of dependent types. The advantages over simply using four lists are probably small.

The importance of enforcing absence of repeated variables is low, and similar for enforcing that the lists matching a variable to its value have the same length; simply dropping the last elements should not cause any problems.

6.2.1 Ott Generated Substitution Functions

Using the grammar of the programming language, OTT can generate COQ functions for both single and multi-variable substitution. Simply by stating

```
substitutions
    multi lockset ls_var :: subst_ls
```

OTT will generate a function `subst_ls_in_constraints`, which will substitute free occurrences of lock set variables (`ls_var`) in constraints.

However, these functions are intended for substituting one kind of variable, while the substitutions needed by our type system requires simultaneous substitution of two kinds of variables, lock set variables and lock environment variables.

It is in general not the case that composing two substitutions, one for each variable kind, would be equivalent to one substitution which simultaneously substitutes both variable kinds. This is because the terms substituted by the first substitution could contain variables which the second substitution then would change. With simultaneous substitution, the substitution will not be applied to the terms which are inserted.

Fortunately, this is not the case for our type system, as lock environment variables do not occur in lock sets. So if lock set variables are substituted first, no new lock environment variables will be introduced, and then one can then apply the substitution for lock environment variables.

Unfortunately, trying to generate a substitution function for lock set variables causes OTT to output no function for substitution variables in constraints. The reason is that OTT is not responsible for the generation of COQ code for constraints, as this is translated manually. Therefore, OTT does not know how to do substitution in constraints.

In addition, the generated substitution-function is not capture-avoiding.

6.2.2 Deep Embedding of Substitutions

The final proposal is to represent the substitution as finite lists, mapping variables to values, and when applied to a term, the a function will traverse the term and look up in the list every time a variable occur. This could be considered a deeper embedding than what will be described below.

6.2.3 Shallow Embedding of Substitutions

Another proposal, is to immediately translate a given substitution to a function from variables to corresponding terms, and then when the substitution is applied to a term, lift this substitution of variables up to substitution on terms.

One challenge with this is the typing of the substitution functions, as there is a relationship between the actual parameter and its return type — if it is called with a lock set variable, a lock set should be returned, and similar for lock environments.

This can be solved quite elegantly with dependent types, as shown in Figure 6.2. The type of a substitution is defined as `subst`, and it is the type of a dependent function, taking an argument which is either a lock environment variable or lock set variable, and depending on which it is, it will return either a lock environment or a lock set.

Notice that defining the type of substitutions as

```
Definition subst : Type :=
    (le_var+ls_var) -> (lockset + lockenv).
```

is not the same, as this would allow calling the substitution with a lock set variable, and get a lock environment returned.

```
Definition subst_type (y : le_var + ls_var) : Set :=
    match y with
    | inl _ => lockenv
    | inr _ => lockset
    end.

Definition subst : Set :=
    forall (y:le_var + ls_var), subst_type y.

Definition empty_subst : subst :=
    fun (y : le_var + ls_var) =>
        match y with
        | inl X => le_Var X
        | inr rho => ls_Var rho
        end.

Definition extend_ls (x:ls_var) (v:lockset)
        (th:subst) : subst :=
    fun (y : le_var + ls_var) =>
        match y as y' return (subst_type y') with
        | inl X => th (inl X)
        | inr x' =>
            if beq_le_var x x'
            then v
            else th (inr x')
        end.
```

Figure 6.2: Implementation of substitution functions with dependent types.

The empty substitution takes a variable, and applies the correct constructor to make it a term — it is essentially the identity function.

To extend a substitution (the function `extend_ls`), an anonymous function is returned, which compares the variable to be substituted, and either returns the value given value, or delegates the problem to the substitution that is extended.

After building such a function, one function which lifts it up to constraints, types, lock environments, and lock sets is needed. This function will be total, without the use of arbitrary "default" values or the `option` type operator.

To make sure that the substitution is capture-avoiding, all bound variables in a type scheme is increased with the maximum identifier which occur in the substitution and in the type scheme. Since type schemes are the only binders of these variables, and type schemes cannot be nested, this is sufficient.

It is this last proposal which was implemented.

6.3 Type and Effect System: Specification

The formalization of the type and effect system which is the specification includes the relations for abstract state ordering relation, subtype relation, and the type system itself.

The function $\lfloor \hat{S} \rfloor$, which removes the annotations from a type, was defined in Ott as a function. All occurrences of the "function" $\lceil T \rceil$ in the type system are on the form $\lceil T \rceil = \hat{T}$, and was replaced with the equivalent claim $T = \lfloor \hat{T} \rfloor$. The main reason for this is that $\lceil T \rceil$ is not a function, as there are several types $\hat{T}$ such that $\lceil T \rceil = \hat{T}$.

Some formulas are used in the definitions of the relations, but are not formalized as judgments. These are added as rules to the `formula`-language of Ott, and a Coq-homomorphism is given, which translates them to Coq code. The formulas include the point-wise greater relation on abstract states ($\Delta \leq \Delta'$), the constraint check $C \vdash C'$, and a checks of free variables ($\vec{Y} \notin FV(G, C)$).

6.4 Type and Effect System: Algorithm

The function $\lceil \hat{T} \rceil_A$ used in the algorithm version, is implemented in Coq. The formalized version takes as arguments a simple type, and two numbers which represents the next fresh variable identifier for lock set and lockenv variables. It returns a triple, consisting of the generated annotated type, and the new numbers to be used for generating fresh variables.

The relations of constraint generation ($\hat{T} \leq \hat{T}' \vdash C$); greatest lower bound ($\hat{T}_1 \wedge \hat{T}_2 = \hat{T}; C$) and least upper bound ($\hat{T}_1 \vee \hat{T}_2 = \hat{T}; C$) for annotated types; the greatest lower bound and least upper bound for lock environment, and the algorithm itself, is defined directly in Ott.

6.4.1 Freshness of Variables

Several of the inference rules in the algorithmic type systems and operational semantics introduces variables (or other entities) with the claim that they are *fresh*. A formal definition of what it means to be fresh is not obvious, but one thing which certainly is clear is that derivations containing freshness-claims are in general not composable.

There are also claims of "being free in" in the rules, which have a straightforward meaning. In one case, the claim of freshness can be changed with a "being free" claim: the rule R-NewL have a claim of freshness, but claiming that it is free in σ suffices. Since σ contains all previously introduced lock references, is it sufficient to check if the lock reference does not occur in σ.

Using global freshness destroys the compositionality of the inference rules. For instance, if one has two derivations D_1 and D_2, both containing a claim that x is fresh, then applying an inference rule using these two derivations yields an incorrect derivation

Here are some ideas which were considered for how to implement a freshness claim in Coq.

Predicates on Derivations

Instead of enforcing the correctness of a derivation locally, one can create a predicate, which given a derivation, will check if all freshness-claims holds. How this can be done is shown in Figure 6.3.

The main problem with this is that this will not work if the `derivation` was defined to be of type `Prop`, since one cannot inspect object of type `Prop`. It is possible to change which universe Ott places a predicate, so Ott's default behavior is not a problem.

```
Inductive derivation : Type :=
  | IsFresh (x : nat) : derivation
  | UseVar (x : nat) : derivation
  | And (a b : derivation) : derivation.

Fixpoint free_variables (d : derivation) : list nat :=
  match d with
  | UseVar x => x::nil
  | IsFresh _ => nil
  | And a b => (free_variables a) ++ (free_variables b)
  end.

Fixpoint fresh_variables (d : derivation) : list nat :=
  match d with
  | UseVar _ => nil
  | IsFresh x => x::nil
  | And a b => (fresh_variables a) ++ (fresh_variables b)
  end.

Definition FreshClaimsHolds (d : derivation) : Prop :=
  List.NoDup (fresh_variables d) /\
  List.Forall (fun x y => x <> y)
    (list_prod (fresh_variables d) (free_variables d)).
```

Figure 6.3: Checking freshness with a predicate.

The major drawback with this method, is that the proof that freshness claims holds has to be carried around, and lemmas describing the consequences of freshness holding would probably be needed.

Make Type System Composable

The second approach is to change the rules, so that information about freshness flows through the system. In our case, this can be achieved by adding two counters, one for each kind of variables, as "input" to the algorithm, which represents the next free identifiers for lock set and lock environment variables, and make the algorithm return an updated version of these two numbers.

Whenever a fresh variable is needed, any number larger than the input-number and which is also not free in its local surroundings can be chosen. The output numbers will then be the maximum number for a free variable at that point.

This is the approach followed used in the formalization. The new signature for the algorithm is

$$\rho_{max}; X_{max}; \Gamma \vdash t : \hat{T} :: \phi; C; \rho'_{max}; X'_{max}$$

where ρ_{max} and X_{max} are the input for counters of fresh variables, and ρ'_{max} and X'_{max} is the output.

6.5 Syntax Directed Type System

The final type system which was formalized is the syntax directed type system. No surprises or new challenges arose during the formalization of this system, since it is very similar to the specification.

6.6 Formalization of the Soundness Proof

The soundness proof establishes that the algorithm is sound with regards to the specification, and it is stated as:

Theorem 3 (Soundness). *Given* $\Gamma \vdash_a t : \hat{T} :: \Delta_1 \to \Delta_2, C$*, then* $C;\ \Gamma \vdash_s t : \hat{T} :: \Delta_1 \to \Delta_2$.

To prove this, there are several properties related to the structural properties of constraints which needs to be proven. These lemmas are discussed in Section 6.6.2.

The proof of soundness is by induction over the given algorithmic derivation. The way it has been formalized is to, at least some degree, follow the outline of the proof given in [PSS13]. All cases have been completed, except the case for TA_Abs2, which has a problem that will be described below.

6.6.1 Proving the Case of TA_Abs2

In the case for TA_Abs2 there is a step which is not justified, and which is not obviously true. The given proof is:

Given is the derivation

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_A \qquad \hat{T}_2 = \lceil T_2 \rceil_A \qquad X_1,\, X_2 \text{ fresh} \qquad \begin{array}{c} C_2 \vdash \hat{T}_2 \geq \hat{T}_2' \end{array} \\ \Gamma, f : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x : \hat{T}_1 \vdash_a e : \hat{T}_2' :: X_1 \to \Delta_2 \qquad C_3 = X_2 \leq \Delta_2}{\Gamma \vdash_a \texttt{fun}\ \ f : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x : \hat{T}_1 .\ e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1 ; C}$$

where $C = C_1, C_2, C_3$. From this they give a derivation concluding with

$$\frac{C; \Gamma, f : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x : \hat{T}_1 \vdash_s e : \hat{T}_2 :: X_1 \to X_2}{C \vdash_s \texttt{fun}\ \ f : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x : \hat{T}_1 .\ e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1 .}\ \text{T\_ABS2}$$

Everything before this point in the proof is correct, and also accepted by COQ, but the application of the T_ABS2 is wrong, as X_1 in the final type and the two Δ_1's in the final effect must be exactly the same.

Possible Fix: Evaluation of Values Should have no Effect

The idea of the first possible fix is that a function is a value, and as the evaluation of a value causes no effect, it should be possible to conclude from $C; \Gamma \vdash_s v : \hat{T} :: \Delta \to \Delta$ that $\Gamma \vdash_s v : \hat{T} :: \Delta' \to \Delta'$ holds for any Δ'. This is a property one ideally want the type system to have, and it is certainly the case that algorithmic version has this property, as all the derivation rules for values places not constraint on their effect. It is unfortunately not a trivial proof for that the type specification has this property, as the T_ABS2-rule places restrictions on which lock environments may be used.

To prove this property, one would probably have to construct a new derivation.

For the purpose of proving soundness, it would be sufficient to prove the weaker statement where from an effect $X \to X$ one can get the effect $\Delta \to \Delta$.

Possible Fix: Change Type System

Another possibility would be to change the specification so that the above property of effect of values is easier to prove. There is in a way a miss-match between the specification and the algorithm, since the property is so apparent in one of them, but not in the other.

The problem remains open in the formalization. In the proof of soundness, the inductive step of T_ABS2 has been assumed without proof.

6.6.2 Strengthening of Constraints

There are several judgments which depend on constraints. Those of the judgments which only use the constraints as the antecedent of constraint checks should obey structural rules about constraints. More specific, for a judgment $J : \mathit{Constraints} \to \mathit{Prop}$, the following properties should hold:

Weakening If $J(C)$, then $J(C \sqcup C')$.

Exchange If $J(C \sqcup C')$, then $J(C' \sqcup C)$.

Weakening is used in many of the proofs, and must therefore be formalized. In addition, associativity and exchange of constraint must also be proved to preserve derivability. The property can also be called strengthening, dependent on one's perspective. In the formalization, there is a large class of lemmas related to this. Note that associativity holds immediately, as list concatenation is associative.

Some or all of these properties have been stated, and either proved, partially proved, or been admitted[1], for the following judgments.

Constraint denotation: Weakening and exchange, and some more, have been stated and completely proved to hold for the denotation of constraints.

Type specification: Commutativity has been proved. Strengthening has been partially proved. There is one case in the inductive proof has not been completed, as α-conversion is needed. More on this in Section 7.1.

Subtyping: Strengthening stated and proved.

The most interesting of these proofs is the strengthening of constraints for the type system. Proving this by induction over the assumption, is easy for all cases except T_Gen, and for a good reason: at first sight it might not be the case that it holds.

To see this, consider the T_Gen-case, where the derivation

$$\dfrac{C_1,\ C_2;\ \Gamma \vdash t : \hat{T} :: \phi \qquad \vec{Y} \text{ not free in } \Gamma, C_1}{C_1;\ \Gamma \vdash t : \forall \vec{Y}.C_2 : \hat{T} :: \phi}\ \textsc{T\_Gen}$$

is given, and

$$C_1, C;\ \Gamma \vdash t : \forall \vec{Y}.C_2 : \hat{T} :: \phi$$

is the goal, with the induction hypothesis

$$C_1,\ C,\ C_2;\ \Gamma \vdash t : \hat{T} :: \phi.$$

Then an application of the T_Gen-rule will not work, as we then need to prove that $\vec{Y}$ is not free in Γ and C_1, C. But $\vec{Y}$ is fixed by the goal, and the C is arbitrary, so this cannot possibly work.

Given that it is an actual[2] type scheme in the conclusion, the only other applicable rule is T_Var.

$$\dfrac{\dfrac{\rho \sqsupseteq \{\pi\} \vdash \rho \sqsupseteq \{\pi\}}{\rho \sqsupseteq \{\pi\}; \epsilon \vdash \texttt{New L}_\pi : \texttt{L}^\rho :: \phi}\ \textsc{T\_NewL} \qquad \rho \notin FV(\varnothing, \epsilon)}{\varnothing, \rho \sqsupseteq \{\pi\};\ \epsilon \vdash \texttt{New L}_\pi : \forall \rho.\rho \sqsupseteq \{\pi\} : \texttt{L}^\rho :: \phi}\ \textsc{T\_Gen}$$

Notice that if we, for instance, tries to strengthen it with $C := \rho \sqsupseteq \pi'$, then the T_Gen-rule is not applicable, as shown here

$$\dfrac{\cdots \qquad \rho \notin FV(\rho \sqsubseteq \pi', \epsilon)}{C, \varnothing, \rho \sqsupseteq \{\pi\};\ \epsilon \vdash \texttt{New L}_\pi : \forall \rho.\rho \sqsupseteq \{\pi\} : \texttt{L}^\rho :: \phi.}\ \textsc{T\_Gen}$$

[1]Admitted is a term used in Coq which means that a proof is not completed, but is assumed. One can admit a complete proof, or parts of a proof.

[2]By actual I mean that the list of variables and the constraints are not empty.

6.6.3 Status of the Proof

To summarize the status of the soundness proof, the following cases have not been proved:

- The case of TA_Abs is not completed because of the error in the original proof.
- Two cases in the proof of strengthening of the type specification is not completed, as α-equivalence is needed.
- The proof have, as planned, been cut off at "obviously true" lemmas.

Chapter 7

Technical and Practical Experiences

This chapter will first discuss the major technical shortcomings of the formalization, namely α-equivalence of type schemes (Section 7.1), and technical and practical problems related to the use of Ott (Section 7.2), finally, the overall experience with formalizing a typical article is discussed (Section 7.3).

7.1 α-Equivalence between Type Schemes

α-equivalence is a general phenomenon in programming languages and logics with bound variables. The main reason why some technical lemmas needed for the soundness proof were not completed, is that α-equivalence of type schemes was not formalized. Two type schemes $\forall \vec{Y}.\, C : \hat{T}$ and $\forall \vec{Y}'.\, C[\vec{Y}'/\vec{Y}] : \hat{T}'[\vec{Y}'/\vec{Y}]$, where the substitution does not cause any variable to be bound, should be considered equal. Since the problem was discovered too late, it has not been corrected in the formalization. A survey of the problem in general is first described, then how these can be applied to our case is discussed.

7.1.1 Representation of Syntax and Bound Variables

A major problem when formalizing a language with variables and bindings is to get α-equivalence and *capture-avoiding* substitution.

α-equivalence means that the names of bound variables in an expression, formula, etc, do not matter. For instance, the two lambda terms $\lambda x.\ x\ x$ and $\lambda y.\ y\ y$ denote the same function, and one would like to identify them as "the same".

Capture-avoiding substitution is also related to bound variables. For instance, if one naïvely replaces x for y in $\lambda x.\ x\ y$, the result is $\lambda x.\ x\ x$, which is wrong: the replacement caused the variable y to be "captured" by the binder. A capture-avoiding substitution would have resulted in $\lambda x'.\ x'\ x$, or an equivalent term.

Naïve Representation

A naïve representation of expressions would use identifiers (for instance numbers or strings) to differentiate between variables. This corresponds to the syntactic representation used by humans. With this representation, both checking α-equivalence and doing capture-avoiding substitution is cumbersome.

To check for α-equivalence, one would have to systematically rename bound variables, and then check for syntactic equivalence.

Substitution is even more work: one must have a way to generate a fresh variable from a list of variables, and also potentially rename all bound variables, so that the free variables of an inserted expression do not accidentally get bound.

Locally-Nameless Representation

One can represent bound variables in an expression as a number which indicates which of the enclosing binding operator it refers to. For instance, the expression $\lambda x.\ \lambda y.\ xy$ can be represented as $\lambda\lambda\, 1\, 0$. These numbers are called *De Bruijn indexes* [DB72]. There are several options for how to represent free variables; if the free variables uses identifiers as in the naïve approach, this is called a *locally-nameless* representation [Gor94].

The expression $\lambda y.\ \lambda x.\ yx$, which is α-equivalent to the expression above, is also represented as $\lambda\lambda\, 1\, 0$. In general, the De Bruijn indexed version of two terms t_1 and t_2 are identical iff t_1 and t_2 are α-equivalent. The advantage with this representation, is that checking for α-equivalence is trivial: it is syntactical equality.

Substitution on the other hand, is more complex, but not as complex as in the naïve representation. Given that a term has exactly one free variable, indicated with 0 at the top-level, then substituting this variable with the term e is done by replacing each variable n enclosed by n binding operators, with the term $e\uparrow^n$. The last operation, $e\uparrow^n$, is called *lifting* or *shifting*, and means that every free variable in e is increased with n. This causes free variables in e not to be bound by the substitution.

Higher Order Abstract Syntax

The idea of Higher Order Abstract Syntax (HOAS) [PE88] is that the language of formalization already has support for substitution, namely function application. One could then *imagine* a formalization in Coq of lambda calculus as:

```
Inductive expr : Type :=
    | App : expr -> expr -> expr
    | Abs : (expr -> expr) -> expr.
```

If this definition was accepted, it would lead to an inconsistent theory in Coq [Chl08], so, the definition is rejected by Coq. The reason why Coq rejects the definition is because it violates the *positivity restriction.* A type defined inductively cannot occur as an argument in a function type in its own definition. The `Abs`-constructor takes an argument of type `(expr -> expr)`, which is a violation of the positivity-restriction. Still, we will look at what this definition "could" mean in the next paragraphs[1].

[1]This definition is still used in programming languages like Haskell, where non-termination is not a grave problem, and in logics like that of Twelf, where it does not lead to an inconsistency.

The expression $\lambda x.\, \lambda y.\, x$ is then be represented as `Abs (fun x => fun y => x)`, and substitution is then simply to call the function with the value to be inserted.

Checking for α-equivalence is trivial, as Coq already does this.

Unfortunately, the definition also allows elements which do not correspond to any lambda term, called *exotic terms* [Chl08]. Consider the expression

```
Abs (fun x => match x with
       | App e _ => e
       | _ => x
     end)
```

which is of type `expr`. This does not correspond to any lambda term, as a term cannot change itself based on what is substituted into it. It is possible to remove the existence of exotic terms [WW03], but it will still not pass the positivity requirement.

Parametric Higher Order Abstract Syntax

Parametric Higher Order Abstract Syntax (PHOAS) [Chl08] is a weakening of HOAS which both rules out unwanted terms, and is accepted by Coq.

The idea is that one make variables explicit, drawn from some type `V`, as shown here:

```
Inductive expr' (V : Type) : Type :=
    | Var : V -> expr' V
    | App : (expr' V) -> (expr' V) -> (expr' V)
    | Abs : (V -> expr' V) -> expr' V.
```

With this definition, one could still make exotic terms. For instance, if one chooses `V` to be natural numbers, one could make a function which behaves differently for odd and even numbers. To solve this, the following definition of an expression is given:

```
Definition expr : Type := forall (V : Type), expr' V.
```

The universal quantification rules out any possibility for members of `expr` to inspect what is substituted into it.

Checking for α-equivalence is equality, just as with HOAS.

7.1.2 Application to Type Schemes

The problem of α-equivalence of type schemes could be solved using all the techniques above (except HOAS). How each technique applies to our specific problem is now described.

Naïve Representation

One could continue to use the current, naïve representation, but add a new inference rule to each of the type systems, which allows α-conversion. This would be, at least in the short term, the quickest fix. For the longer run however, the solution is not desirable. It is conceptually ugly, as one needs to formalize properties which could be inherent in the representation, and the definitions of substitution and α-equivalence make the proofs unnecessarily complex.

Locally-Nameless Representation

The implementation of De Bruijn indexes would introduce a strict difference between bound and free variables. Bound variables would use De Bruijn indexes, while free variables would use identifiers. The binding structure also must be changed, and where variables can occur must now be duplicated to two versions,

$$scheme, \hat{S} ::= \forall n, m, C : \hat{T}$$

where n is the number of lock set variables introduces, and m is the number of lock-environment variables introduced.

There is experimental support for locally-nameless representation in OTT, but it is not directly applicable to the given representation of type schemes, only supports binders which introduce a fixed numbers of variables. It could be possible to change the syntax and rules related to type schemes, so that type schemes only introduce one variable at the time.

Additionally, a predicate is needed which checks if a structure with variables is locally closed, i.e. that it does not contain bound variables which do not point to any binder.

Parametric Higher Order Abstract Syntax

The initial challenge with PHOAS is that the introduction of several variables at once is needed, so a similar transformation which is needed for OTTs locally-nameless representation is needed. And also here, free variables must be indicated as named variables.

If the system were to be formalized again, this would be the way I would formalize bound variables for type schemes, as long as the type schemes can be coerced into introducing a single variable at the time. The "free" and correct variable substitution and semantics for bound variables is very appealing.

7.1.3 Representation of Expressions

The problem of binding and substitution also occurs in the context of expressions, but it has not affected the current work, as proofs relating the type systems and the semantics have not been formalized. Though it is important to note that this would still affect the current proofs, though the implication of this has not been researched.

As the functions work with closed terms, both De Bruijn indexes and PHOAS would should work fine. It would be natural to try to reformulate the syntax using PHOAS.

7.2 Experience with Ott

During the process of learning and using OTT, as the size of the project grew, several lessons were learned. OTT feels to some degree unreliable, in that changes in the OTT-file can have unintended effects on the generated COQ-code.

One problem encountered early, was that using COQ-homomorphisms to give a definition of a grammar can be unreliable, as it could produce COQ-code where the order of definitions were incorrect. The code was on the form as shown

```
grammar
foo :: foo_ ::=
 | Foo1          ::   :: Foo1_Intro
 | Foo2          ::   :: Foo2_Intro

bar :: ''    ::= {{coq (list foo)}}
 | empty         :: S:: EmptyList {{coq (@nil foo)}}
 | foo :: bar    :: S:: ConsList {{coq (cons [[foo]] [[bar]])}}
```

Figure 7.1: The form of the code which caused the generation of an invalid Coq-file.

in Figure 7.1, with two definitions. Ott will generate an inductive definition for `foo`, while the definition of `bar` is given by Coq-homomorphisms. In the concrete example I had, the definition of `bar` was given before that of `foo`, which caused Coq to reject the program. Changing the order in the Ott-file had no effect on the produced output. Splitting the Ott-files across several files, as described next, seems to remedy this.

7.2.1 Separation of Ott-files

Early on in the formalization, only a single Ott-file was used. From a software engineering point of view, we know this is not a good idea. Ott has fairly good support of separating a project into several files. Having two Ott-files `syntax.ott` and `semantics.ott`, the command

```
$ ott -i syntax.ott -o syntax.v \
    -i semantics.ott -o semantics.v
```

causes two Coq-files to be created, `syntax.v`, which will contain the definitions from `syntax.ott`, and `semantics.v`, which will contain the definitions from `semantics.ott`. The files cannot be mutually dependent, but the definitions within each file can.

While this lets us break up a project into several files, there can still be problems where a definition ideally should be split across several files, but is forced to stay in a single file. This is mainly if there are mutually dependent definitions.

An example of this is the definition of `formula`. In the beginning of the project, `formula` was defined in a single file. This is not very practical, ideally, the best would be to introduce a kind of formula where it is defined and used. This was achieved by changing the definition of `formula` to what is shown in Figure 7.2. With this definition, each file can define their own formulas, together with inline Coq-code.

One must then define all of the extra formula-rules as meta-rules (with the `M`). The only downside, is that one will get rules such as

```
Inductive syntax_formula : Set := .
```

which litter the code, but they cause no harm. An interesting project would be to make it possible to make Ott drop definitions like this, if wanted.

```
formula, F :: formula_ ::=
 | judgement             ::  :: judgement
 | syntax_formula        :: M::
   syntax_formula {{coq [[syntax_formula]]}}
 | type_formula          :: M::
   type_formula {{coq [[type_formula]]}}
 | substitution_formula :: M::
   substitution_formula {{coq [[substitution_formula]]}}
 | constraint_formula    :: M::
   constraint_formula {{coq [[constraint_formula]]}}
 | free_formula          :: M::
   free_formula {{coq [[free_formula]]}}
```

Figure 7.2: Main definition of formulas, which delegates the definition of formulas to where they are used.

7.2.2 Substitution and Free Variables

In the beginning the substitution functions generated by OTT were used. But when the syntax of constraints was changed to use COQ's `list`, then OTT could no longer generate a correct substitution function, which means that a function has to be defined manually.

The auto-generated functions become unusable as soon as some part of the syntax is defined using COQ-homomorphisms.

7.3 Experience with Formalization

To take a highly technical article and formalize its content was a challenge. The most typical problems are the "informal" changes of definitions, and inconsistent use of notations. Here are some examples:

1. The type systems uses the type `Thread`, which is not mentioned in the syntax of types.

2. In the semantics then expression `spawn t` will return the process identifier of the created thread, but that is not a value according to the syntax.

3. The operational semantics which is given in the paper is unlabeled; later in the paper, it is said its transitions should be labeled with information about lock usage.

4. Notation which is not property defined is used, such as $C \vDash_\theta C'$, $\Gamma \lesssim_\theta \Gamma'$, and $\Delta \lesssim_\theta \Delta'$.

5. The syntax of the base types T is not defined. In the formalization, these have been implemented as the base types `Lock`, `Thread`, and `Bool`, closed under function-types.

6. There were examples of typographical errors in the type systems; for instance the rule TA_COND, there are primes which are placed wrongly.

7. The rule LE_Arrow is wrong, also here is primes missing.

8. Only the least upper bound of types, abstract states, and effects is given explicit, the definition of greatest lower bound must be inferred.

Having the authors of the paper available to answer questions during the formalization was of great value.

Chapter 8

Conclusion

8.1 Contribution

The formalization presented in this thesis, shows that the theory introduced in the article [PSS13] is precise enough to be formalized in a rigorous, unambiguous, and machine-checkable way. Furthermore, the formalized proof of soundness done in Coq shows that the arguments used in the article are logically sound, and increase the faith in the correctness of the claim of soundness. The additional formalized technical proofs about the type systems further increase the faith in the correctness of the type systems.

The main limitation of the thesis is that there are admitted proofs. This was expected from the start, but it could be that some properties which have been assumed, but not proven, do not actually hold. Ideally, all proofs should be completed. Furthermore, only one major proof from the article is formalized; all the proofs should be formalized.

8.2 General Challenges

To quote the POPLMark challenge [ABF+05]:

> How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?

The nature of the problems which had to be solved in this thesis indicates that the gap between machine checkable and "English" proofs and definitions are still wide apart. The skills needed to do a proper formalization is a research topic in itself.

The first major problem is that of bindings and capture-avoiding substitutions. In the theorem prover Twelf, this is partially solved with higher-order abstract syntax. In other theorem provers, a first-order representation, such as De Bruijn indexes, is usually used, which is not ideal. An approach to solve the problem, which do not include changing the underlying logic of theorem provers, is to create high-level libraries, such as "Lambda Tamer" by Adam Chlipala [Chl10], implementing a framework around PHOAS.

Another major problem with "practical" formalization is the different kinds of equivalences which theorem provers supply and the equivalences which users want. Type systems usually have two kinds of equivalences: judgmental equality and propositional equality[1]. Furthermore, user-defined equivalences is also needed; in this thesis both constraints, programs, and types have natural equivalence relations. There are mainly two approaches for working with user-defined equivalence relations in type theory: setoids (a set equipped with an equivalence relation) and quotient types (essentially the type of equivalence classes). Unfortunately, quotient types are not directly available in intentional type theories such as COQ, so setoids must be used instead. Using setoids means that relations might not be compatible with the equipped equivalence relation, which means that additional lemmas about compatibility needs to be proven. There has been work on finding restricted quotient types to intensional type theories [AAL].

Further work on creating and improving specialized tools, such as OTT, is needed to minimize the overhead of formalizing.

[1]In extensional type theory, these are collapsed to one.

Bibliography

[AAL] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li, *Definable quotients in type theory.*

[ABF$^+$05] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic, *Mechanized metatheory for the masses: The poplmark challenge*, Theorem Proving in Higher Order Logics, Springer, 2005, pp. 50–65.

[Bar92] Henk P. Barendregt, *Lambda calculi with types*, Handbook of Logic in Computer Science (Samson Abramsky, Dov Gabbay, and Thomas Maibaum, eds.), vol. 1: Mathematical Structures, Oxford University Press, 1992, pp. 117–309.

[BC04] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development: Coq'art: the calculus of inductive constructions*, springer, 2004.

[BGG$^+$92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel, *Experience with embedding hardware description languages in HOL*, TPCD, vol. 10, 1992, pp. 129–156.

[BM79] Robert S. Boyer and J. Strother Moore, *A computational logic*, vol. 5, Academic press New York, 1979.

[CC99] Catarina Coquand and Thierry Coquand, *Structured type theory*, Workshop on Logical Frameworks and Metalanguages, 1999.

[Chl08] Adam Chlipala, *Parametric higher-order abstract syntax for mechanized semantics*, ACM Sigplan Notices **43** (2008), no. 9, 143–156.

[Chl10] ———, *A verified compiler for an impure functional language*, ACM Sigplan Notices, vol. 45, ACM, 2010, pp. 93–106.

[Chl11] ———, *Certified programming with dependent types*, 2011.

[Cur34] Haskell B. Curry, *Functionality in combinatory logic*, Proceedings of the National Academy of Sciences of the United States of America **20** (1934), no. 11, 584.

[DB72] Nicolaas Govert De Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem*, Indagationes Mathematicae (Proceedings), vol. 75, Elsevier, 1972, pp. 381–392.

[FSDF93] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, *The essence of compiling with continuations*, ACM Conference on Programming Language Design and Implementation (PLDI), ACM, June 1993, In *SIGPLAN Notices* 28(6).

[Gir72] Jean-Yves Girard, *Interprétation fonctionelle et élimination des coupure dans l'arithmetique d'ordre supérieur*, Ph.D. thesis, Université Paris VII, 1972.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor, *Proofs and types*, Cambridge University Press, 1989.

[Gon08] Georges Gonthier, *Formal proof–the four-color theorem*, Notices of the AMS **55** (2008), no. 11, 1382–1393.

[Gor94] Andrew D. Gordon, *A mechanisation of name-carrying syntax up to alpha-conversion*, Springer, 1994.

[Ler09] Xavier Leroy, *Formal verification of a realistic compiler*, Communications of the ACM **52** (2009), no. 7, 107–115.

[NBB$^+$63] Peter Naur, John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al., *Revised report on the algorithmic language algol 60*, Communications of the ACM **6** (1963), no. 1, 1–17.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, *Principles of program analysis*, Springer, 1999.

[Nor07] Ulf Norell, *Towards a practical programming language based on dependent type theory*, Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Pau88] Lawrence C Paulson, *A preliminary user's manual for Isabelle*, University of Cambridge, Computer Laboratory, 1988.

[PE88] Frank Pfenning and Conal Elliot, *Higher-order abstract syntax*, ACM SIGPLAN Notices, vol. 23, ACM, 1988, pp. 199–208.

[PSS13] Ka I Pun, Martin Steffen, and Volker Stolz, *Deadlock checking by data race detection*, Submitted for journal publication, under review.

[Rey74] John Reynolds, *Towards a theory of type structure*, Colloque sur la programmation (Paris, France) (B. Robinet, ed.), Lecture Notes in Computer Science, vol. 19, Springer-Verlag, 1974, pp. 408–425.

[SNO$^+$07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša, *Ott: effective tool support for the working semanticist*, Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ICFP '07, ACM, 2007, pp. 1–12.

[Wad89] Philip Wadler, *Theorems for free!*, Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM, 1989, pp. 347–359.

[Wel94] J. B. Wells, *Typability and type checking in the second order λ-calculus are equivalent and undecidable*, Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France), IEEE, Computer Society Press, July 1994, pp. 176–185.

[WW03] Geoffrey Washburn and Stephanie Weirich, *Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism (extended version)*, Tech. report, Department of Computer & Information Science, University of Pennsylvania, 2003, MS-CIS-03-26.

Appendix A

Ott output

processid, p	Process id
variable, x, f	Variable for value
programpoint, π	Program point
lockref, l	Lock reference
lockcount, lc	Times lock is taken
index, k, j, m, z	
n	

formula, F	::=			
	|	*judgement*		
	|	*syntax_formula*	M	
	|	*type_formula*	M	
	|	*substitution_formula*	M	
	|	*constraint_formula*	M	
	|	*free_formula*	M	
	|	$\rho_1, \,..\, , \rho_k > \rho_{max}$		
terminals	::=			
	|	$\forall$		
	|	$\neq$		
syntax_formula	::=			
	|	$\rho = \rho'$	M	
	|	$\rho \neq \rho'$	M	
	|	$\rho < \rho'$	M	
	|	$x \neq x'$	M	
	|	$p \neq p'$	M	
program, P	::=			Program
	|	$\varnothing$		Empty Program
	|	$p\langle t\rangle$		Single Program
	|	$P_1 \| P_2$		Parallel Composition
thread, t	::=			Thread
	|	v		Value
	|	$\textbf{let}\, x : T = e \,\textbf{in}\, t$	bind x in t	Let
	|	$t[v/x]$	M	Substitute v for x in t
	|	(t)	S	

expr, *e*	::=			Expression
	|	t		Thread
	|	$v_1\ v_2$		Application
	|	**if** v **then** e_1 **else** e_2		Conditional
	|	**spawn** t		Spawn a thread
	|	$\mathbf{newL}_\pi$		Create a new lock
	|	$v.$ **lock**		Aqcuire lock
	|	$v.$ **unlock**		Release lock
	|	(e)	S	
value, *v*	::=			Value
	|	x		Variable reference
	|	l^r		Lock reference
	|	**true**		Truth value
	|	**false**		Truth value
	|	**fn** $x : T.t$	bind x in t	Function abstraction
	|	**fun** $f : T_1.x : T_2.t$	bind f in t bind x in t	Recursive function abstraction
	|	p		Proccess id
	|	(v)	S	
lockset, *r*	::=			Lock set expression
	|	ρ		Variable
	|	$\{\pi\}$		Singleton
	|	$r \cup r'$		Union
	|	(r)	M	
ls_var, ρ_{max}, ρ	::=			
	|	$\rho + 1$	M	
	|	$max(\rho_1 \,..\, \rho_k)$	M	
	|	$min(\rho_1 \,..\, \rho_k)$	M	
	|	(ρ)	S	
simpletype, *T*	::=			Simple Type
	|	**Bool**		Boolean
	|	**Thread**		Thread
	|	**L**		Lock
	|	$T_1 \rightarrow T_2$		Function
	|	(T)	S	
	|	$\lfloor \hat{T} \rfloor$	M	
	|	$\lfloor \hat{S} \rfloor$	M	
lockstate	::=			
	|	**free**		

	|	$p(lc)$	
	|	$p(1)$	M
	|	$p(lc+1)$	M
	|	$\mathbf{dec}\,(p(lc))$	M
$\mathit{state},\ \sigma$	::=		
	|	ϵ	M
	|	$\sigma[l \mapsto \mathit{lockstate}]$	M
$\mathit{free\_ls\_var}$	::=		
	|	$FV(\Delta)$	M
	|	$FV(\Gamma)$	M
	|	$FV(C)$	M
$\mathit{free\_le\_var}$	::=		
	|	$FV(\Delta)$	M
	|	$FV(\Gamma)$	M
	|	$FV(C)$	M
$\mathit{free\_formula}$	::=		
	|	$\rho_1\,..\,\rho_k\ \mathbf{notIn}\,\mathit{free\_ls\_var}$	M
	|	$X_1\,..\,X_k\ \mathbf{notIn}\,\mathit{free\_le\_var}$	M
$\mathit{constraint\_formula}$	::=		
	|	$C \vDash C'$	M
	|	$C \vdash r \sqsubseteq r'$	M
$\mathit{type\_formula}$	::=		
	|	$\varphi = \varphi'$	M
	|	$T = T'$	M
	|	$\hat{S} = \hat{S}'$	M
	|	$\sigma = \sigma'$	M
	|	$X < X'$	M
	|	$X \neq X'$	M
	|	$\mathbf{unique}\,X_1\,..\,X_k$	M
	|	$\mathbf{unique}\,\rho_1\,..\,\rho_k$	M
	|	$\Delta <= \Delta'$	M
$\mathit{le\_var},\ X,\ X_{max}$	::=		
	|	$X+1$	
	|	$max(X_1\,..\,X_k)$	
	|	$min(X_1\,..\,X_k)$	
	|	$\mathbf{const}\,X$	M

lockenv, Δ	::=			Lock Environment expression
	|	$\bullet$		
	|	X		
	|	$\Delta, r : n$		
	|	$\Delta \oplus \Delta'$		
	|	$\Delta \ominus \Delta'$		
	|	$(r : 1)$	M	
	|	(Δ)	M	
ineq	::=			Single Constraint
	|	$r \sqsubseteq r'$		
	|	$\Delta \leq \Delta'$		
constraints, C	::=			Constraint
	|	$\emptyset$	M	
	|	*ineq*	M	
	|	C, C'	M	
	|	$\theta\, C$	M	
effect, φ	::=			Effect
	|	$\Delta_1 \rightarrow \Delta_2$		
annotatedtype, $\hat{T}$	::=			Annotated Type
	|	**Bool**		
	|	**Thread**		
	|	$\mathbf{L}^r$		
	|	$\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$		
	|	$\theta\, \hat{T}$	M	
	|	$(\hat{T})$	S	
scheme, $\hat{S}$	::=			Typescheme
	|	$\forall\, \rho_1 \,..\, \rho_k\, X_1 \,..\, X_j : C.\hat{T}$	bind $\rho_1..\rho_k$ in C bind $\rho_1..\rho_k$ in $\hat{T}$ bind $X_1..X_j$ in C bind $X_1..X_j$ in $\hat{T}$	
	|	$\hat{T}$	S	
	|	$\theta\, \hat{S}$	M	
	|	$\mathit{close}(\Gamma, C, \hat{T})$	M	
program_type, Φ	::=			Program type
	|	$p\langle \varphi; C\rangle$		
	|	$\Phi \| \Phi'$		
context, Γ	::=			

	$\vert$	ϵ		Empty context
	$\vert$	$\Gamma, x : \hat{S}$		Extend Γ with $x : \hat{S}$
	$\vert$	(Γ)	S	
substitution_formula	::=			
	$\vert$	$\theta = \theta'$	M	
subst, θ	::=			
	$\vert$	$[\Delta_1 \mathinner{..} \Delta_m / X_1 \mathinner{..} X_k][r_1 \mathinner{..} r_j / \rho_1 \mathinner{..} \rho_z]$	M	
LocalStep	::=			
	$\vert$	$t_1 \rightarrow t_2$		As a local step, t_1 reduc
GlobalStep	::=			
	$\vert$	$\sigma \vdash P \rightarrow \sigma' \vdash P'$		
ContextLookup	::=			
	$\vert$	$\Gamma(x) = \hat{S}$		x is bound to $\hat{S}$ in Γ
DownAT	::=			
	$\vert$	$\lfloor \hat{T} \rfloor$		
DownSC	::=			
	$\vert$	$\lfloor \hat{S} \rfloor$		
AbstrSateOrder	::=			
	$\vert$	$C \vdash \Delta \leq \Delta'$		
SubType	::=			
	$\vert$	$C \vdash \hat{T} \leq \hat{T}'$		
TypeAndEffect	::=			
	$\vert$	$C; \Gamma \vdash e : \hat{S} :: \varphi$		
	$\vert$	$\vdash P :: \Phi$		
ConstraintGeneration	::=			
	$\vert$	$\hat{T} \leq \hat{T}' \vdash C$		
	$\vert$	$\Delta \leq \Delta' \vdash C$		
LeastUpperBound	::=			
	$\vert$	$\hat{T} \vee \hat{T}' = \hat{T}''; C$		
	$\vert$	$\hat{T} \wedge \hat{T}' = \hat{T}''; C$		
	$\vert$	$\Delta_1 \vee \Delta_2 = \Delta; C$		

	$\mid$	$\Delta_1 \wedge \Delta_2 = \Delta; C$
	$\mid$	$\varphi_1 \vee \varphi_2 = \varphi; C$
	$\mid$	$\varphi_1 \wedge \varphi_2 = \varphi; C$
TypeAndEffectAlgo	::=	
	$\mid$	$(\rho_{max}, X_{max})\Gamma \vdash e : \hat{T} :: \varphi; C(\rho'_{max}, X'_{max})$
SyntaxDirected	::=	
	$\mid$	$C; \Gamma \vdash e : \hat{T} :: \varphi$
judgement	::=	
	$\mid$	*LocalStep*
	$\mid$	*GlobalStep*
	$\mid$	*ContextLookup*
	$\mid$	*AbstrSateOrder*
	$\mid$	*SubType*
	$\mid$	*TypeAndEffect*
	$\mid$	*ConstraintGeneration*
	$\mid$	*LeastUpperBound*
	$\mid$	*TypeAndEffectAlgo*
	$\mid$	*SyntaxDirected*
user_syntax	::=	
	$\mid$	*processid*
	$\mid$	*variable*
	$\mid$	*programpoint*
	$\mid$	*lockref*
	$\mid$	*lockcount*
	$\mid$	*index*
	$\mid$	*n*
	$\mid$	*formula*
	$\mid$	*terminals*
	$\mid$	*syntax_formula*
	$\mid$	*program*
	$\mid$	*thread*
	$\mid$	*expr*
	$\mid$	*value*
	$\mid$	*lockset*
	$\mid$	*ls_var*
	$\mid$	*simpletype*
	$\mid$	*lockstate*
	$\mid$	*state*
	$\mid$	*free_ls_var*
	$\mid$	*free_le_var*

| *free_formula*
| *constraint_formula*
| *type_formula*
| *le_var*
| *lockenv*
| *ineq*
| *constraints*
| *effect*
| *annotatedtype*
| *scheme*
| *program_type*
| *context*
| *substitution_formula*
| *subst*

$\boxed{t_1 \rightarrow t_2}$ As a local step, t_1 reduces to t_2

$$\frac{}{\mathbf{let}\, x : T = v\, \mathbf{in}\, t \rightarrow t[v/x]} \quad \text{R\_RED}$$

$$\frac{}{\mathbf{let}\, x_2 : T_2 = (\mathbf{let}\, x_1 : T_1 = e_1\, \mathbf{in}\, t_1)\, \mathbf{in}\, t_2 \rightarrow \mathbf{let}\, x_1 : T_1 = e_1\, \mathbf{in}\, (\mathbf{let}\, x_2 : T_2 = t_1\, \mathbf{in}\, t_2)} \quad \text{R\_LET}$$

$$\frac{}{\mathbf{let}\, x : T = (\mathbf{if}\, \mathbf{true}\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)\, \mathbf{in}\, t \rightarrow \mathbf{let}\, x : T = e_1\, \mathbf{in}\, t} \quad \text{R\_IF1}$$

$$\frac{}{\mathbf{let}\, x : T = (\mathbf{if}\, \mathbf{false}\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)\, \mathbf{in}\, t \rightarrow \mathbf{let}\, x : T = e_2\, \mathbf{in}\, t} \quad \text{R\_IF2}$$

$$\frac{}{\mathbf{let}\, x : T = ((\mathbf{fn}\, x' : T'.t')\ v)\, \mathbf{in}\, t \rightarrow \mathbf{let}\, x : T = t'[v/x']\, \mathbf{in}\, t} \quad \text{R\_APP1}$$

$$\frac{}{\mathbf{let}\, x : T = ((\mathbf{fun} f : T_1.x' : T_2.t')\ v)\, \mathbf{in}\, t \rightarrow \mathbf{let}\, x : T = (t'[v/x'])[\mathbf{fun} f : T_1.x' : T_2.t'/f]\, \mathbf{in}\, t} \quad \text{R\_APP2}$$

$\boxed{\sigma \vdash P \rightarrow \sigma' \vdash P'}$

$$\frac{t_1 \rightarrow t_2}{\sigma \vdash p\langle t_1 \rangle \rightarrow \sigma \vdash p\langle t_2 \rangle} \quad \text{R\_LIFT}$$

$$\frac{\begin{array}{l}\sigma \vdash P_1 \rightarrow \sigma' \vdash P_1' \\ \texttt{NoDup (dom\_P (}P_1 || P_2\texttt{))}\end{array}}{\sigma \vdash P_1 || P_2 \rightarrow \sigma' \vdash P_1' || P_2} \quad \text{R\_PAR}$$

$$\frac{p_1 \neq p_2}{\sigma \vdash p_1\langle \mathbf{let}\, x : T = \mathbf{spawn}\, t_2\, \mathbf{in}\, t_1 \rangle \rightarrow \sigma \vdash p_1\langle \mathbf{let}\, x : T = p_2\, \mathbf{in}\, t_1 \rangle || p_2\langle t_2 \rangle} \quad \text{R\_SPAWN}$$

$$\frac{\begin{array}{l}\sigma' = \sigma[l \mapsto \mathbf{free}] \\ \texttt{lockref\_fresh}\ \sigma\ \ l\end{array}}{\sigma \vdash p\langle \mathbf{let}\, x : T = \mathbf{newL}_\pi\, \mathbf{in}\, t \rangle \rightarrow \sigma' \vdash p\langle \mathbf{let}\, x : T = l^r\, \mathbf{in}\, t \rangle} \quad \text{R\_NEWL}$$

$$\frac{\begin{array}{l}\texttt{lockref\_lookup}\ \sigma\ \ l\ \texttt{= Some}\ \mathbf{free} \\ \sigma' = \sigma[l \mapsto p(1)]\end{array}}{\sigma \vdash p\langle \mathbf{let}\, x : T = l^r.\, \mathbf{lock}\, \mathbf{in}\, t \rangle \rightarrow \sigma' \vdash p\langle \mathbf{let}\, x : T = l^r\, \mathbf{in}\, t \rangle} \quad \text{R\_LOCK}$$

$$\frac{\begin{array}{l}\texttt{lockref\_lookup}\ \sigma\ l\ \texttt{=}\ \texttt{Some}\ p(lc)\\ \sigma' = \sigma[l \mapsto p(lc+1)]\end{array}}{\sigma \vdash p\langle \mathbf{let}\, x : T = l^r.\, \mathbf{lock\, in}\, t\rangle \to \sigma' \vdash p\langle \mathbf{let}\, x : T = l^r\, \mathbf{in}\, t\rangle}\ \textsc{R\_Relock}$$

$$\frac{\begin{array}{l}\texttt{lockref\_lookup}\ \sigma\ l\ \texttt{=}\ \texttt{Some}\ p(lc)\\ \sigma' = \sigma[l \mapsto \mathbf{dec}\,(p(lc))]\end{array}}{\sigma \vdash p\langle \mathbf{let}\, x : T = l^r.\, \mathbf{unlock\, in}\, t\rangle \to \sigma' \vdash p\langle \mathbf{let}\, x : T = l^r\, \mathbf{in}\, t\rangle}\ \textsc{R\_Unlock}$$

$\boxed{\Gamma(x) = \hat{S}}$ x is bound to $\hat{S}$ in Γ

$$\frac{}{(\Gamma, x : \hat{S})(x) = \hat{S}}\ \textsc{LookupAx}$$

$$\frac{\begin{array}{c}x \neq x'\\ \Gamma(x) = \hat{S}\end{array}}{(\Gamma, x' : \hat{S}')(x) = \hat{S}}\ \textsc{LookupNext}$$

$\boxed{\lfloor \hat{T} \rfloor}$

$$\begin{array}{l}\lfloor \mathbf{Bool} \rfloor \equiv \mathbf{Bool}\\ \lfloor \mathbf{Thread} \rfloor \equiv \mathbf{Thread}\\ \lfloor \mathbf{L}^r \rfloor \equiv \mathbf{L}\\ \lfloor \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 \rfloor \equiv (\lfloor \hat{T}_1 \rfloor) \to (\lfloor \hat{T}_2 \rfloor)\end{array}$$

$\boxed{\lfloor \hat{S} \rfloor}$

$$\lfloor \forall\, \rho_1 \mathinner{..} \rho_k\, X_1 \mathinner{..} X_j : C.\hat{T} \rfloor \equiv \lfloor \hat{T} \rfloor$$

$\boxed{C \vdash \Delta \leq \Delta'}$

$$\frac{}{C \vdash \Delta \leq \Delta}\ \textsc{SO\_Refl}$$

$$\frac{\begin{array}{c}C \vdash \Delta_1 \leq \Delta_2\\ C \vdash \Delta_2 \leq \Delta_3\end{array}}{C \vdash \Delta_1 \leq \Delta_3}\ \textsc{SO\_Trans}$$

$$\frac{}{\Delta \leq \Delta' \vdash \Delta \leq \Delta'}\ \textsc{SO\_Ax}$$

$$\frac{\Delta_1 <= \Delta_2}{C \vdash \Delta_1 \leq \Delta_2}\ \textsc{SO\_Base}$$

$$\frac{C \vdash \bullet \leq \Delta_1}{C \vdash \Delta_2 \leq \Delta_2 \oplus \Delta_1}\ \textsc{SO\_Plus1}$$

$$\frac{C \vdash \Delta_1 \leq \bullet}{C \vdash \Delta_2 \oplus \Delta_1 \leq \Delta_2}\ \textsc{SO\_Plus2}$$

$$\frac{C \vdash \bullet \leq \Delta_1}{C \vdash \Delta_2 \ominus \Delta_1 \leq \Delta_2}\ \textsc{SO\_Minus1}$$

$$\frac{C \vdash \Delta_1 \leq \bullet}{C \vdash \Delta_2 \leq \Delta_2 \ominus \Delta_1}\ \textsc{SO\_Minus2}$$

$\boxed{C \vdash \hat{T} \leq \hat{T}'}$

$$\frac{}{C \vdash \hat{T} \leq \hat{T}} \quad \text{S\_Refl}$$

$$\frac{C \vdash \hat{T}_1 \leq \hat{T}_2 \qquad C \vdash \hat{T}_2 \leq \hat{T}_3}{C \vdash \hat{T}_1 \leq \hat{T}_3} \quad \text{S\_Trans}$$

$$\frac{C \vDash r_1 \sqsubseteq r_2}{C \vdash \mathbf{L}^{r_1} \leq \mathbf{L}^{r_2}} \quad \text{S\_Lock}$$

$$\frac{C \vdash \hat{T}_1' \leq \hat{T}_1 \qquad C \vdash \hat{T}_2 \leq \hat{T}_2' \qquad C \vdash \Delta_1' \leq \Delta_1 \qquad C \vdash \Delta_2 \leq \Delta_2'}{C \vdash \hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{\Delta_1' \to \Delta_2'} \hat{T}_2'} \quad \text{S\_Arrow}$$

$\boxed{C; \Gamma \vdash e : \hat{S} :: \varphi}$

$$\frac{\Gamma(x) = \hat{S}}{C; \Gamma \vdash x : \hat{S} :: \Delta \to \Delta} \quad \text{T\_Var}$$

$$\frac{C \vdash \{\pi\} \sqsubseteq \rho}{C; \Gamma \vdash \mathbf{newL}_\pi : \mathbf{L}^\rho :: \Delta \to \Delta} \quad \text{T\_NewL}$$

$$\frac{C \vdash \rho \sqsubseteq \rho'}{C; \Gamma \vdash l^\rho : \mathbf{L}^{\rho'} :: \Delta \to \Delta} \quad \text{T\_LRef}$$

$$\frac{\lfloor \hat{T}_1 \rfloor = T_1 \qquad C; \Gamma, x : \hat{T}_1 \vdash t : \hat{T}_2 :: \varphi}{C; \Gamma \vdash \mathbf{fn}\, x : T_1.t : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta \to \Delta} \quad \text{T\_Abs1}$$

$$\frac{\lfloor \hat{T}_1 \rfloor = T_1 \qquad \lfloor \hat{T}_2 \rfloor = T_2 \qquad C; \Gamma, f : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x : \hat{T}_1 \vdash t : \hat{T}_2 :: \Delta_1 \to \Delta_2}{C; \Gamma \vdash \mathbf{fun}\, f : T_1 \to T_2.x : T_1.t : \hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2 :: \Delta_1 \to \Delta_1} \quad \text{T\_Abs2}$$

$$\frac{C; \Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta_1 \to \Delta_1 \qquad C; \Gamma \vdash v_2 : \hat{T}_2 :: \Delta_1 \to \Delta_1}{C; \Gamma \vdash v_1\, v_2 : \hat{T}_1 :: \Delta_1 \to \Delta_2} \quad \text{T\_App}$$

$$\frac{C; \Gamma \vdash v : \mathbf{Bool} :: \Delta_1 \to \Delta_1 \qquad C; \Gamma \vdash e_1 : \hat{T} :: \Delta_1 \to \Delta_2 \qquad C; \Gamma \vdash e_2 : \hat{T} :: \Delta_1 \to \Delta_2}{C; \Gamma \vdash \mathbf{if}\, v\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : \hat{T} :: \Delta_1 \to \Delta_2} \quad \text{T\_Cond}$$

$$\frac{C; \Gamma \vdash e_1 : \hat{S}_1 :: \Delta_1 \to \Delta_2 \qquad \lfloor \hat{S}_1 \rfloor = T_1 \qquad C; \Gamma, x : \hat{S}_1 \vdash t_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3}{C; \Gamma \vdash \mathbf{let}\, x : T_1 = e_1\, \mathbf{in}\, t_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3} \quad \text{T\_Let}$$

$$\frac{C;\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2}{C;\Gamma \vdash \textbf{spawn}\, t : \textbf{Thread} :: \Delta_1 \to \Delta_1}\ \text{T\_SPAWN}$$

$$\frac{\begin{array}{c} C;\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_1 \\ C \vdash \Delta_1 \oplus (\rho : 1) \leq \Delta_2 \end{array}}{C;\Gamma \vdash v.\,\textbf{lock} : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_2}\ \text{T\_LOCK}$$

$$\frac{\begin{array}{c} C;\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_1 \\ C \vdash \Delta_1 \ominus (\rho : 1) \leq \Delta_2 \end{array}}{C;\Gamma \vdash v.\,\textbf{unlock} : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_2}\ \text{T\_UNLOCK}$$

$$\frac{\begin{array}{l} C_1, C_2;\Gamma \vdash e : \hat{T} :: \Delta_1 \to \Delta_2 \\ \rho_1 \mathinner{..} \rho_j\ \textbf{notIn}\, FV(\Gamma) \\ X_1 \mathinner{..} X_k\ \textbf{notIn}\, FV(\Gamma) \\ \rho_1 \mathinner{..} \rho_j\ \textbf{notIn}\, FV(C_1) \\ X_1 \mathinner{..} X_k\ \textbf{notIn}\, FV(C_1) \end{array}}{C_1;\Gamma \vdash e : \forall\, \rho_1 \mathinner{..} \rho_j\ X_1 \mathinner{..} X_k : C_2.\hat{T} :: \Delta_1 \to \Delta_2}\ \text{T\_GEN}$$

$$\frac{\begin{array}{l} C_1;\Gamma \vdash e : \forall\, \rho_1 \mathinner{..} \rho_j\ X_1 \mathinner{..} X_k : C_2.\hat{T} :: \Delta_1 \to \Delta_2 \\ \theta = [\Delta_1 \mathinner{..} \Delta_m / X_1 \mathinner{..} X_k][r_1 \mathinner{..} r_z / \rho_1 \mathinner{..} \rho_j] \\ C_1 \vDash \theta\, C_2 \end{array}}{C_1;\Gamma \vdash e : \theta \hat{T} :: \Delta_1 \to \Delta_2}\ \text{T\_INST}$$

$$\frac{\begin{array}{l} C;\Gamma \vdash e : \hat{T}_2 :: \Delta_1 \to \Delta_2 \\ C \vdash \hat{T}_2 \leq \hat{T}_1 \\ C \vdash \Delta_1' \leq \Delta_1 \\ C \vdash \Delta_2 \leq \Delta_2' \end{array}}{C;\Gamma \vdash e : \hat{T}_1 :: \Delta_1' \to \Delta_2'}\ \text{T\_SUB}$$

$\boxed{\vdash P :: \Phi}$

$$\frac{C;\epsilon \vdash t : \hat{T} :: \varphi}{\vdash p\langle t \rangle :: p\langle \varphi; C \rangle}\ \text{T\_THREAD}$$

$$\frac{\begin{array}{c} \vdash P_1 :: \Phi_1 \\ \vdash P_2 :: \Phi_2 \end{array}}{\vdash P_1 \| P_2 :: \Phi_1 \| \Phi_2}\ \text{T\_PAR}$$

$\boxed{\hat{T} \leq \hat{T}' \vdash C}$

$$\frac{}{\textbf{Bool} \leq \textbf{Bool} \vdash \varnothing}\ \text{C\_BASIC1}$$

$$\frac{}{\textbf{Thread} \leq \textbf{Thread} \vdash \varnothing}\ \text{C\_BASIC2}$$

$$\frac{}{\mathbf{L}^{\rho_1} \leq \mathbf{L}^{\rho_2} \vdash \rho_1 \sqsubseteq \rho_2}\ \text{C\_LOCK}$$

$$\frac{\begin{array}{c} \hat{T}_1' \leq \hat{T}_1 \vdash C_1 \\ \hat{T}_2 \leq \hat{T}_2' \vdash C_2 \end{array}}{\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{X_1' \to X_2'} \hat{T}_2' \vdash C_1, C_2, X_1' \leq X_1, X_2 \leq X_2'}\ \text{C\_ARROW}$$

$\boxed{\Delta \leq \Delta' \vdash C}$

$$\frac{}{\Delta \leq \Delta' \vdash \Delta \leq \Delta'}\ \textsc{C\_Id}$$

$\boxed{\hat{T} \vee \hat{T}' = \hat{T}''; C}$

$$\frac{}{\mathbf{Bool} \vee \mathbf{Bool} = \mathbf{Bool}; \varnothing}\ \textsc{LT\_Bool}$$

$$\frac{}{\mathbf{Thread} \vee \mathbf{Thread} = \mathbf{Thread}; \varnothing}\ \textsc{LT\_Thread}$$

$$\frac{\begin{array}{l}\rho \neq \rho_1 \\ \rho \neq \rho_2 \\ \mathbf{L}^{\rho_1} \leq \mathbf{L}^{\rho} \vdash C_1 \\ \mathbf{L}^{\rho_2} \leq \mathbf{L}^{\rho} \vdash C_2\end{array}}{\mathbf{L}^{\rho_1} \vee \mathbf{L}^{\rho_2} = \mathbf{L}^{\rho}; C_1, C_2}\ \textsc{LT\_Lock}$$

$$\frac{\begin{array}{c}\hat{T}_1' \wedge \hat{T}_1'' = \hat{T}; C_1 \\ \hat{T}_2' \vee \hat{T}_2'' = \hat{T}'; C_2 \\ \varphi_1 \vee \varphi_2 = \varphi; C_3\end{array}}{\hat{T}_1' \xrightarrow{\varphi_1} \hat{T}_2' \vee \hat{T}_1'' \xrightarrow{\varphi_2} \hat{T}_2'' = \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2; C_1, C_2, C_3}\ \textsc{LT\_Arrow}$$

$\boxed{\hat{T} \wedge \hat{T}' = \hat{T}''; C}$

$$\frac{}{\mathbf{Bool} \wedge \mathbf{Bool} = \mathbf{Bool}; \varnothing}\ \textsc{GT\_Bool}$$

$$\frac{}{\mathbf{Thread} \wedge \mathbf{Thread} = \mathbf{Thread}; \varnothing}\ \textsc{GT\_Thread}$$

$$\frac{\begin{array}{l}\rho \neq \rho_1 \\ \rho \neq \rho_2 \\ \mathbf{L}^{\rho} \leq \mathbf{L}^{\rho_1} \vdash C_1 \\ \mathbf{L}^{\rho} \leq \mathbf{L}^{\rho_2} \vdash C_2\end{array}}{\mathbf{L}^{\rho_1} \wedge \mathbf{L}^{\rho_2} = \mathbf{L}^{\rho}; C_1, C_2}\ \textsc{GT\_Lock}$$

$$\frac{\begin{array}{c}\hat{T}_1' \vee \hat{T}_1'' = \hat{T}; C_1 \\ \hat{T}_2' \wedge \hat{T}_2'' = \hat{T}'; C_2 \\ \varphi_1 \wedge \varphi_2 = \varphi; C_3\end{array}}{(\hat{T}_1' \xrightarrow{\varphi_1} \hat{T}_2') \wedge (\hat{T}_1'' \xrightarrow{\varphi_2} \hat{T}_2'') = (\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2); C_1, C_2, C_3}\ \textsc{GT\_Arrow}$$

$\boxed{\Delta_1 \vee \Delta_2 = \Delta; C}$

$$\frac{\begin{array}{c}\Delta_1 \leq X \vdash C_1 \\ \Delta_2 \leq X \vdash C_2\end{array}}{\Delta_1 \vee \Delta_2 = X; C_1, C_2}\ \textsc{LE\_States}$$

$\boxed{\Delta_1 \wedge \Delta_2 = \Delta; C}$

$$\frac{\begin{array}{c}X \leq \Delta_1 \vdash C_1 \\ X \leq \Delta_2 \vdash C_2\end{array}}{\Delta_1 \wedge \Delta_2 = X; C_1, C_2}\ \textsc{GE\_States}$$

$\boxed{\varphi_1 \vee \varphi_2 = \varphi; C}$

$$\frac{\begin{array}{c}\Delta_1' \wedge \Delta_1'' = \Delta_1; C_1 \\ \Delta_2' \vee \Delta_2'' = \Delta_2; C_2\end{array}}{\Delta_1 \to \Delta_2 \vee \Delta_1' \to \Delta_2' = \Delta_1 \to \Delta_2; C_1, C_2} \quad \text{LE\_Arrow}$$

$$\boxed{\varphi_1 \wedge \varphi_2 = \varphi; C}$$

$$\frac{\begin{array}{c}\Delta_1' \vee \Delta_1'' = \Delta_1; C_1 \\ \Delta_2' \wedge \Delta_2'' = \Delta_2; C_2\end{array}}{\Delta_1 \to \Delta_2 \wedge \Delta_1' \to \Delta_2' = \Delta_1 \to \Delta_2; C_1, C_2} \quad \text{GE\_Arrow}$$

$$\boxed{(\rho_{max}, X_{max})\Gamma \vdash e : \hat{T} :: \varphi; C(\rho'_{max}, X'_{max})}$$

$$\frac{\begin{array}{l}\Gamma(x) = \forall\, \rho_1 \mathinner{..} \rho_j\ X_1 \mathinner{..} X_k : C.\hat{T} \\ \theta = [X_1' \mathinner{..} X_m' / X_1 \mathinner{..} X_k][\rho_1' \mathinner{..} \rho_z' / \rho_1 \mathinner{..} \rho_j] \\ \textbf{unique}\, \rho_1' \mathinner{..} \rho_z' \\ \textbf{unique}\, X_1' \mathinner{..} X_m' \\ \rho_1' \mathinner{..} \rho_z'\ \textbf{notIn}\, FV(\Gamma) \\ \rho_1' \mathinner{..} \rho_z'\ \textbf{notIn}\, FV(C) \\ \rho_1' \mathinner{..} \rho_z'\ \textbf{notIn}\, FV(\Delta) \\ X_1' \mathinner{..} X_m'\ \textbf{notIn}\, FV(\Gamma) \\ X_1' \mathinner{..} X_m'\ \textbf{notIn}\, FV(C) \\ X_1' \mathinner{..} X_m'\ \textbf{notIn}\, FV(\Delta) \\ \rho_{max} < min(\rho_1' \mathinner{..} \rho_z') \\ X_{max} < min(X_1' \mathinner{..} X_m')\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash x : \theta\,\hat{T} :: \Delta \to \Delta; \theta\, C(max(\rho_1' \mathinner{..} \rho_z') + 1, max(X_1' \mathinner{..} X_m') + 1)} \quad \text{TA\_Var}$$

$$\frac{\begin{array}{l}X_{max} < X'_{max} \\ \rho_{max} < \rho \\ \rho < \rho'_{max}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \textbf{newL}_\pi : \mathbf{L}^\rho :: \Delta \to \Delta; \{\pi\} \sqsubseteq \rho(\rho'_{max}, X'_{max})} \quad \text{TA\_NewL}$$

$$\frac{\begin{array}{l}X_{max} < X'_{max} \\ \rho_{max} < \rho' \\ \rho' < \rho'_{max}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash l^\rho : \mathbf{L}^{\rho'} :: \Delta \to \Delta; \rho \sqsubseteq \rho'(\rho'_{max}, X'_{max})} \quad \text{TA\_Lref}$$

$$\frac{\begin{array}{l}\texttt{lift\_A}\ T_1\ \rho_{max}\ X_{max}\ \texttt{=}\ (\hat{T}_1\texttt{,}\ \rho'_{max}\texttt{,}\ X'_{max}) \\ (\rho'_{max}, X'_{max})\Gamma, x : \hat{T}_1 \vdash t : \hat{T}_2 :: X_1 \to \Delta_2; C(\rho''_{max}, X''_{max}) \\ X''_{max} < X_1 \\ X_1 < X_2 \\ X_2 < X'''_{max}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \textbf{fn}\, x : T_1.t : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C, \Delta_2 \leq X_2(\rho''_{max}, X'''_{max})} \quad \text{TA\_Abs1}$$

$$\frac{\begin{array}{l}\texttt{lift\_A}\ T_1 \to T_2\ \rho_{max}\ X_{max}\ \texttt{=}\ (\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2\texttt{,}\ \rho'_{max}\texttt{,}\ X'_{max}) \\ (\rho'_{max}, X'_{max})\Gamma, f : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x : \hat{T}_1 \vdash t : \hat{T}_2' :: X_1 \to \Delta_2; C_1(\rho''_{max}, X''_{max}) \\ \hat{T}_2' \leq \hat{T}_2 \vdash C_2 \\ \Delta_2 \leq X_2 \vdash C_3 \\ \texttt{(* No fresh claim X1, X2? *) True}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \textbf{fun} f : T_1 \to T_2.x : T_1.t : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C_1, C_2, C_3(\rho''_{max}, X''_{max})} \quad \text{TA\_Abs2}$$

$$\frac{\begin{array}{l}(\rho_{max}, X_{max})\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta;\, C_1(\rho'_{max}, X'_{max}) \\ (\rho'_{max}, X'_{max})\Gamma \vdash v_2 : \hat{T}'_2 :: \Delta \to \Delta;\, C_2(\rho''_{max}, X''_{max}) \\ \hat{T}'_2 \le \hat{T}_2 \vdash C \\ X''_{max} < X \\ X < X'''_{max}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash v_1\, v_2 : \hat{T}_1 :: \Delta \to X;\, C_1, C_2, C, \Delta \le \Delta_1, \Delta_2 \le X\, (\rho''_{max}, X'''_{max})} \quad \textsc{TA\_App}$$

$$\frac{\begin{array}{l}T = \lfloor \hat{T}_1 \rfloor \\ T = \lfloor \hat{T}_2 \rfloor \\ \hat{T}_1 \vee \hat{T}_2 = \hat{T};\, C \\ \Delta_1 \vee \Delta_2 = \Delta';\, C' \\ (\rho_{max}, X_{max})\Gamma \vdash v : \mathbf{Bool} :: \Delta_0 \to \Delta_0;\, C_0(\rho'_{max}, X'_{max}) \\ (\rho'_{max}, X'_{max})\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1;\, C_1(\rho''_{max}, X''_{max}) \\ (\rho''_{max}, X''_{max})\Gamma \vdash e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2;\, C_2(\rho'''_{max}, X'''_{max})\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \mathbf{if}\, v\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : \hat{T} :: \Delta_0 \to \Delta';\, C_0, C_1, C_2, C, C'(\rho'''_{max}, X'''_{max})} \quad \textsc{TA\_Cond}$$

$$\frac{\begin{array}{l}(\rho_{max}, X_{max})\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2;\, C_1(\rho'_{max}, X'_{max}) \\ \lfloor \hat{T}_1 \rfloor = T_1 \\ (\rho'_{max}, X'_{max})\Gamma, x : \hat{S}_1 \vdash t_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3;\, C_2(\rho''_{max}, X''_{max}) \\ \hat{S}_1 = close(\Gamma, C_1, \hat{T}_1)\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \mathbf{let}\, x : T_1 = e_1\, \mathbf{in}\, t_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3;\, C_2(\rho''_{max}, X''_{max})} \quad \textsc{TA\_Let}$$

$$\frac{\begin{array}{l}(\rho_{max}, X_{max})\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2;\, C(\rho'_{max}, X'_{max}) \\ \texttt{(* Should it really be } \Delta_1 \to \Delta_1 \texttt{ in conclusion? *) True}\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash \mathbf{spawn}\, t : \mathbf{Thread} :: \Delta_1 \to \Delta_1;\, C(\rho'_{max}, X'_{max})} \quad \textsc{TA\_Spawn}$$

$$\frac{\begin{array}{l}(\rho_{max}, X_{max})\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta \to \Delta;\, C_1(\rho'_{max}, X'_{max}) \\ X'_{max} < X \\ X < X''_{max} \\ \Delta \oplus (\rho : 1) \le X \vdash C_2\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash v.\,\mathbf{lock} : \mathbf{L}^{\rho} :: \Delta \to X;\, C_1, C_2(\rho'_{max}, X''_{max})} \quad \textsc{TA\_Lock}$$

$$\frac{\begin{array}{l}(\rho_{max}, X_{max})\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta \to \Delta;\, C_1(\rho'_{max}, X'_{max}) \\ X'_{max} < X \\ X < X''_{max} \\ \Delta \ominus (\rho : 1) \le X \vdash C_2\end{array}}{(\rho_{max}, X_{max})\Gamma \vdash v.\,\mathbf{unlock} : \mathbf{L}^{\rho} :: \Delta \to X;\, C_1, C_2(\rho'_{max}, X''_{max})} \quad \textsc{TA\_Unlock}$$

$$\boxed{C; \Gamma \vdash e : \hat{T} :: \varphi}$$

$$\frac{\begin{array}{l}\Gamma(x) = \forall\, \rho_1 \,..\, \rho_k\, X_1 \,..\, X_j : C'.\hat{T} \\ \theta = [\Delta_1 \,..\, \Delta_m / X_1 \,..\, X_j][r_1 \,..\, r_z / \rho_1 \,..\, \rho_k] \\ C \vDash \theta\, C'\end{array}}{C; \Gamma \vdash x : \theta\, \hat{T} :: \Delta \to \Delta} \quad \textsc{TD\_Var}$$

$$\frac{C \vdash \{\pi\} \sqsubseteq \rho}{C; \Gamma \vdash \mathbf{newL}_{\pi} : \mathbf{L}^{\rho} :: \Delta \to \Delta} \quad \textsc{TD\_NewL}$$

$$\frac{C \vdash \rho \sqsubseteq \rho'}{C; \Gamma \vdash l^{\rho} : \mathbf{L}^{\rho'} :: \Delta \to \Delta} \quad \textsc{TD\_LRef}$$

$$\frac{\begin{array}{l}\lfloor\hat{T}_1\rfloor = T_1\\ C;\Gamma, x:\hat{T}_1 \vdash t:\hat{T}_2 :: \varphi\end{array}}{C;\Gamma \vdash \mathbf{fn}\, x:T.t : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta \to \Delta}\ \textsc{TD\_Abs1}$$

$$\frac{\begin{array}{l}\lfloor\hat{T}_1\rfloor = T_1\\ \lfloor\hat{T}_2\rfloor = T_2\\ C;\Gamma, f:\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x:\hat{T}_2 \vdash t:\hat{T}_2 :: \varphi\\ \varphi = \Delta_1 \to \Delta_2\end{array}}{C;\Gamma \vdash \mathbf{fun}\, f:T_1 \to T_2.x:T_1.t : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta_1 \to \Delta_1}\ \textsc{TD\_Abs2}$$

$$\frac{\begin{array}{l}C;\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2 :: \Delta \to \Delta\\ C;\Gamma \vdash v_2 : \hat{T}_2 :: \Delta \to \Delta\end{array}}{C;\Gamma \vdash v_1\, v_2 : \hat{T}_1 :: \Delta \to \Delta'}\ \textsc{TD\_App}$$

$$\frac{\begin{array}{l}C \vdash \hat{T}_1 \le \hat{T}\\ C \vdash \hat{T}_2 \le \hat{T}\\ C \vdash \Delta_1 \le \Delta'\\ C \vdash \Delta_2 \le \Delta'\\ C;\Gamma \vdash v : \mathbf{Bool} :: \Delta \to \Delta\\ C;\Gamma \vdash e_1 : \hat{T}_1 :: \Delta \to \Delta_1\\ C;\Gamma \vdash e_2 : \hat{T}_2 :: \Delta \to \Delta_2\end{array}}{C;\Gamma \vdash \mathbf{if}\, v\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : \hat{T} :: \Delta \to \Delta'}\ \textsc{TD\_Cond}$$

$$\frac{\begin{array}{l}C_1, C_2;\Gamma \vdash e : \hat{T}_1 :: \Delta_1 \to \Delta_2\\ \rho_1 .. \rho_j\ \mathbf{notIn}\, FV(\Gamma)\\ \rho_1 .. \rho_j\ \mathbf{notIn}\, FV(C)\\ X_1 .. X_k\ \mathbf{notIn}\, FV(\Gamma)\\ X_1 .. X_k\ \mathbf{notIn}\, FV(C)\\ C_2;\Gamma, x : \forall\, \rho_1 .. \rho_j\, X_1 .. X_k : C_1.\hat{T}_1 \vdash e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3\end{array}}{C_2;\Gamma \vdash \mathbf{let}\, x:T_1 = e\, \mathbf{in}\, t : \hat{T}_2 :: \Delta_1 \to \Delta_3}\ \textsc{TD\_Let}$$

$$\frac{C;\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2}{C;\Gamma \vdash \mathbf{spawn}\, t : \mathbf{Thread} :: \Delta_1 \to \Delta_1}\ \textsc{TD\_Spawn}$$

$$\frac{\begin{array}{l}C;\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_1\\ C \vdash \Delta_1 \oplus (\rho:1) \le \Delta_2\end{array}}{C;\Gamma \vdash v.\,\mathbf{lock} : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_2}\ \textsc{TD\_Lock}$$

$$\frac{\begin{array}{l}C;\Gamma \vdash v : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_1\\ C \vdash \Delta_1 \ominus (\rho:1) \le \Delta_2\end{array}}{C;\Gamma \vdash v.\,\mathbf{unlock} : \mathbf{L}^{\rho} :: \Delta_1 \to \Delta_2}\ \textsc{TD\_Unlock}$$