School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Refinement Types for Cogent

by

Blaise Paradeza

Final Thesis Report

Submitted:	2 December 2020
Supervisor:	Christine Rizkallah
Student ID:	z5160298

Acknowledgements

I would first like to thank Kai and Liam for their remarkable work as educators who piqued my interest to pursue a thesis in this field. Thanks to my supervisor, Christine, for giving me this opportunity. I greatly appreciate her guidance throughout the year.

I also acknowledge the contributions of many Data61 staff, who provided valuable insights during my thesis work. In particular, Zilin played a vital role in guiding me through my thesis. His willingness to answer any of my questions, as well as the patience he has shown when explaining new concepts to me is commendable. I am incredibly thankful for the wisdom and assistance that Zilin has provided.

Thanks to my fellow thesis group peers: Harrison, James, Lucy, Luka, Oscar, and Simon. Watching their works progress throughout the year inspired me to continue in my own.

I acknowledge my many friends including Elizabeth, Eric, Jennifer, Jessica, Lewes, Lucy, Mitchell and Niriksha. Special thanks to Lucy for her continuous encouragement and support. I am blessed to have such wonderful friends who care for my wellbeing and helped me to overcome the challenges I faced this year.

Finally, I thank my parents for the endless love they have given. Words are not enough to express my gratitude for their support throughout my life.

Abstract

COGENT is a high-level functional language for writing and verifying systems software. Extending the COGENT type system to support refinement types enables the specification of propositions in the type system. Refinement types can be used to guarantee memory-safety, reduce the amount of dynamic checks, and eliminate dead code, while ensuring the language remains total.

As an extension to the language, a limited form of built-in arrays is being added to COGENT. A use case of refinement types is to ensure array indices are always within the array bounds, preventing unsafe memory access.

The aim of this project is to extend the core type system COGENT compiler, by modifying the core typechecker to support refinement types.

Contents

1 Introduction **1**

2 Background **3**

2.1 Cogent 3

2.2 Refinement Types 4

2.2.1 Liquid Haskell 6

2.2.2 Formalisation of Simple Refinement Types in Coq 7

2.2.3 F* 7

2.2.4 TypeScript 7

2.3 Dependent Types 8

3 Formalisation **9**

3.1 Adding Refinement Types to Cogent 10

3.2 Subtyping Relation 12

3.3 Type Inference Rules 13

3.4 Example 15

4 Implementation **17**

4.1 Implementation of Type Inference Rules 18

4.2 Subtyping 19

4.3 SMT Solver Usage 19

5 Conclusion **22**

Bibliography **24**

Appendix 1: Formalisation **26**

A.1 Syntax 26

A.2 Wellformedness 27

A.3 Context Relations 27

A.3.1 Context Splitting 27

A.3.2 Weakening 27

A.4 Type Inference Rules 27

A.5 Subtyping 28

Chapter 1

Introduction

COGENT is a functional language for developing and verifying systems software (O'Connor et al., 2016; O'Connor, 2019). Systems programmers benefit from writing code in a high-level language, which is compiled to produce low-level C code and an automatically generated correctness proof (Amani et al., 2016; Rizkallah et al., 2016).

COGENT is a heavily restricted language due to its focus on formal verification. It lacks many of the features mainstream programming languages possess, such as recursion and iteration.

Enhancing the expressiveness of the COGENT type system with refinement types will facilitate future developments in the COGENT language. COGENT is currently being extended to support a limited form of arrays. Arrays introduce problems with unsafe memory access, such as accessing memory using an index that is outside the array bounds. The introduction of refinement types to COGENT can ensure array access remains memory-safe.

Refinement types are types that are refined by a logical predicate. For example, we can define a type for natural numbers:

```
type Nat = {v:Int | 0 ⩽ v}
```

The type `Nat` refines the type `Int` with a predicate $0 \leqslant \nu$. Any value of type `Nat` is assured to be greater than or equal to zero.

The use of refinement types reduces the need for dynamic checking, as types are checked statically by the compiler. This results in simpler, more precise code.

Another benefit of refinement types is the elimination of dead code. COGENT functions must be total, that is, defined for all inputs. Consider the `head` function, which returns the first element of a list. It is a partial function because it has no definition for empty lists.

In Haskell, to deal with the case in which an empty list is passed to `head`, it is possible to call an error function which throws an exception.

```
1   head          :: [a] -> a
2   head (x:_)    = x
3   head []       = error "head: empty list"
```

Rather than allow exceptions to occur, we prefer to remove the error case entirely. By creating a refinement type for non-empty lists, `head` can be defined to only accept a non-empty list as input.

We can define `head` as:

```
head :: {v:[a] | length(v) > 0} → a
```

There is no longer a need to handle empty lists, and so the dead error handling code is eliminated.

Additionally, using refinement types can assist the COGENT compiler in verification. The predicates of refinement types provide more information for automatically generated proof tactics to use. As a result, less manual verification is required.

In this thesis, we present a formalisation of refinement types in the COGENT core language. We also implement a subset of refinement types in the COGENT type checker, allowing integer and boolean values to be refined by arithmetic and logical predicates.

Chapter 2

Background

2.1 Cogent

Formal verification of software systems code is regarded as expensive and impractical for software development. The Trustworthy Systems project aims to lower the cost of verification to match that of conventional software engineering methods. This results in a platform for developing operating systems components that are guaranteed to be free of faults (Klein et al., 2017).

Existing programming languages do not possess the following desired attributes for this purpose: a complete formalisation, a purely functional semantics, and no garbage collection. O'Connor (2019) presents Cogent, a high level, purely functional, polymorphic language that satisfies these requirements.

Given Cogent code, the certifying compiler generates C code, a shallow embedding of the Cogent code in Isabelle/HOL (Nipkow and Klein, 2014), and a proof that the C code refines the Cogent embedding (O'Connor et al., 2016). Isabelle is a higher order logic (HOL) theorem prover. Code written in Cogent can be reasoned about in Isabelle/HOL, as the refinement proof ensures that any property proven about the Isabelle/HOL embeddings also applies to the C code.

During type checking, Isabelle is instructed to store the typing judgements established by the compiler, to aid in proving refinement (O'Connor et al., 2016). This is one area of the COGENT compiler that benefits from adding refinement types to COGENT. A more expressive type system will lower the burden on the Isabelle theorem prover by providing more information about types.

COGENT features a linear and uniqueness type system (O'Connor, 2019) to ensure memory safety. Linear types must be used once. Using a variable after it has been freed is not permitted, because the act of freeing the variable counts as one use of the variable. It also prevents memory leaks by not allowing variables to be unused.

Variables of unique type have exactly one reference at a time. If any code referencing an object is also accountable for that object's disposal, then the uniqueness type system statically enforces memory management. This eliminates the need for a garbage collector.

COGENT is also restricted by the requirement for functions to be total: they must be well defined for all inputs and cannot diverge. This greatly simplifies the generated Isabelle embeddings.

COGENT has been successfully used to implement two Linux file systems, ext2 and BilbyFs (Amani et al., 2016), demonstrating its suitability for systems programming and verification.

2.2 Refinement Types

A base type is any type over which refinement is permitted. Refinement types extend base types with refinement predicates. For example:

```
1   type Nat = {v:Int | 0 ⩽ v}
2   type Pos = {v:Int | 0 < v}
```

`Nat` refines the type `Int` with a predicate that restricts the value `v` to be greater than or equal to zero. Similarly, `Pos` describes integers strictly greater than zero.

Using the above types, we can specify a function contract:

```
div :: n:Nat → d:Pos → {v:Nat | v ⩽ n}
```

The function `div` prohibits division by zero by restricting the type of the divisor to strictly positive values. If the program type checks, it is guaranteed that `div` will not throw a divide by zero exception.

Another use of refinement types is global refinement of data types. Data types can be refined such that every instance must satisfy the constraints. Consider a data type for CSV files:

```
data CSV a = CSV { cols :: [String], rows :: [[a]] }
```

This definition is undesirable because it allows rows to have a different length as the header row. To reject malformed instances, we can refine the type (Vazou, 2016):

```
1   type ListL a X = {v:[a] | len v = len X}
2   data CSV a     = CSV {cols :: [String],
3                         rows :: [ListL a cols]}
```

The `CSV` type now enforces that every row has the same length as the header row. Any `CSV`-typed expression that does not satisfy this type definition is rejected by the compiler.

Subtyping and Verification Conditions

Consider the following functions (Vazou, 2016):

```
1   good     :: Nat → Nat → Int
2   good x y = let z = y + 1 in x ‘div‘ z
3
```

```
4   bad      :: Nat → Nat → Int
5   bad x y = x `div` y
```

For the function `bad`, the refinement type system checks that `y` is a subtype of the corresponding type in `div`'s signature, `Nat`, via a subtyping query:

$$\texttt{x:\{x}\geqslant\texttt{0\}, y:\{y}\geqslant\texttt{0\}} \vdash \texttt{\{v}\geqslant\texttt{0\}} \preceq \texttt{\{v>0\}}$$

The refinement type system generates a verification condition, which states that under the assumptions from the environment, the refinement predicate of the sub-type implies the refinement predicate of the super-type.

$$(x \geqslant 0) \wedge (y \geqslant 0) \Rightarrow (v \geqslant 0) \Rightarrow (v > 0)$$

These verification conditions can be efficiently checked by satisfiability modulo theories (SMT) solvers (De Moura and Bjørner, 2008). A solver will reject the above verification condition as invalid, thus, the system rejects the function `bad`. The function `good` is accepted because its corresponding verification condition is valid:

$$(x \geqslant 0) \wedge (y \geqslant 0) \wedge (z = y + 1) \Rightarrow (v = y + 1) \Rightarrow (v > 0)$$

2.2.1 Liquid Haskell

Liquid Haskell is a program verifier that aims to integrate formal verification as part of the software development process (Vazou, 2016). Liquid Haskell provides a layer on top of GHC that uses the framework of Liquid Typing (Rondon et al., 2008) and an SMT solver for generating and solving constraints. As input, it takes Haskell source code annotated with correctness specifications, and checks whether the code satisfies the specifications. These correctness specifications take the form of refinement types.

Liquid Haskell can be seen as an optional type checker for Haskell. Refinements do not affect the dynamic semantics of the language, so Liquid Haskell can be easily applied to existing libraries.

2.2.2 Formalisation of Simple Refinement Types in Coq

Lehmann and Tanter (2016) formalised a simple form of refinement types in Coq (Bertot and Castéran, 2013), an interactive theorem prover. Types are refined using formulas p, which are in terms of logical values w. A logical value may be a variable, constant, lambda abstraction, or function over other logical values.

Subtyping reduces to an entailment judgement. The type system is parameterised with a judgement $\Delta \vDash p$. In a type environment Γ, a refinement type $\{x : B \mid p\}$ is a subtype of another $\{x : B \mid q\}$ if p entails q in the context extracted from Γ. We use <: to denote a subtype relationship between two types.

$$\frac{\begin{array}{c} extract(\Gamma) \cup \{p\} \vDash q \\ \Gamma \vdash \{x : B \mid p\} \qquad \Gamma \vdash \{x : B \mid q\} \end{array}}{\Gamma \vdash \{x : B \mid p\} <: \{x : B \mid q\}}$$

The formalisation of refinement types in Cogent in this thesis draws inspiration from the work of Lehmann and Tanter, in particular, the structure of predicates and the subtyping relation.

2.2.3 F*

Swamy et al. (2016) present a redesigned version of F*, which is both a proof assistant and an effectful verification-oriented language designed around a lattice of monadic effects. F* refinement types are more powerful than systems such as Liquid Typing (Rondon et al., 2008) that only support type-based reasoning. The monadic lattice structure of F* enables reasoning over pure functions directly in extrinsic proofs.

2.2.4 TypeScript

The work on refinement types described so far have focused on functional languages such as Haskell and F*, however, there is also demand for refinement types in more mainstream, imperative languages. Refined TypeScript is a lightweight refinement type

system for TypeScript (Vekris et al., 2016). Similar to LIQUID HASKELL, subtyping obligations are reduced to verification conditions, which are proven by an SMT solver.

2.3 Dependent Types

Dependent types are types that depend on a value. They are similar to refinement types, in the sense that the base type is restricted to a limited set of values.

Agda (Bove and Dybjer, 2008) is a dependently typed functional programming language. Consider the dependent type A^n of vectors of length n with components of type A. We can define vectors inductively, encoding the length of the vector within the constructor:

```
1  data Vec (A: Set) : Nat -> Set where
2      []    : Vec A zero
3      _::_  : {n: Nat} -> A -> Vec A n -> Vec A (succ n)
```

The benefit of this type definition is that more accurate specifications can be made. For instance, a vector addition function can now demand that both input vectors have the same length, ensuring type safety.

An important difference between refinement types and dependent types is that refinement types include predicates, allowing proof obligations to be outsourced to an SMT solver.

Dependent types carry proof objects that are constructed by the user. Manually discharging proofs can be burdensome especially for systems programmers, who may not have the expertise to complete this task.

Refinement types are more intuitive and usable for systems programmers, and proofs can be delegated to an SMT solver. For these reasons, refinement types are preferred over dependent types for COGENT.

Chapter 3

Formalisation

Our formalisation of refinement types in COGENT relies on an understanding of COGENT's type system. COGENT uses a substructural type system, so care must be taken when manipulating contexts. The structural laws of exchange, weakening, and contraction listed below are not generally available in a substructural system.

$$\frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_2, \Gamma_1 \vdash e : \tau}\ \textsc{Exchange} \qquad \frac{\Gamma_1 \vdash e : \tau}{\Gamma_1, \Gamma_2 \vdash e : \tau}\ \textsc{Weaken} \qquad \frac{\Gamma_1, \Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_1, \Gamma_2 \vdash e : \tau}\ \textsc{Contract}$$

Permitting the laws of weakening and contraction without constraint can break the uniqueness properties of COGENT's type system. The weakening law allows for the discard of linear resources that have not yet been used. Admitting contraction enables the creation of many references to the same mutable object.

Having every type restricted by linearity may be inconvenient for the programmer. We include the **Share** and **Drop** judgements to determine whether contraction or weakening on a type is allowed. Instead of using the structural laws, we manipulate contexts with explicit relations for context-splitting and weakening (O'Connor, 2019):

Under assumptions A, the context Γ can be split into Γ_1 and Γ_2. Each assumption from Γ is assigned to either Γ_1 or Γ_2. A type can only be placed in both Γ_1 and Γ_2 if it is

shareable.

$$\frac{A \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \leadsto x : \tau, \Gamma_1 \boxplus \Gamma_2} \qquad \frac{A \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \leadsto \Gamma_1 \boxplus x : \tau, \Gamma_2}$$

$$\frac{}{A \vdash \varepsilon \leadsto \varepsilon \boxplus \varepsilon} \qquad \frac{A \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad A \vdash \tau \ \textbf{Share}}{A \vdash x : \tau, \Gamma \leadsto x : \tau, \Gamma_1 \boxplus x : \tau, \Gamma_2}$$

The context Γ can be weakened to a smaller context Γ' by removing an assumption that has a discardable type.

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\leadsto} \Gamma'}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\leadsto} x : \tau, \Gamma'} \qquad \frac{A \vdash \Gamma \overset{\text{weak}}{\leadsto} \Gamma' \qquad A \vdash \tau \ \textbf{Drop}}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\leadsto} \Gamma'}$$

$$\frac{}{A \vdash \varepsilon \overset{\text{weak}}{\leadsto} \varepsilon}$$

The above context relations for splitting and weakening are used in many of the type inference rules shown in this chapter.

3.1 Adding Refinement Types to Cogent

Figure 3.1 describes the syntax of Cogent updated with refinement types. Highlighted sections indicate changes we have made to the original Cogent syntax by O'Connor (2019).

We add a new type to Cogent to represent refinement types, $\{x : \mathtt{B} \mid p(x)\}$. Any base type $\mathtt{B}$ can be used to construct a refinement type. For simplicity, this formalisation has primitive types `T` as the only possible base type, however `B` can be extended to include more types in the future.

Refinement predicates p are expressed in terms of logical values w. A logical value may be a literal, variable, or logical function over other logical values.

Values	$v ::=$	x	(variables)
	$\mid$	ℓ	(literals)
Literals	$\ell ::=$	$\mathbb{N} \mid \texttt{True} \mid \texttt{False}$	
Types	$\tau ::=$	a	(type variables)
	$\mid$	$\texttt{T}$	(primitive types)
	$\mid$	$\{x : \texttt{B} \mid p(x)\}$	(refinement types)
	$\mid$	$x : \tau_1 \rightarrow \tau_2$	(functions)
	$\mid$	$\texttt{Array}\ \tau\ l$	(arrays)
Primitive Types	$\texttt{T} ::=$	$\texttt{U8} \mid \texttt{U16} \mid \texttt{U32} \mid \texttt{U64}$	(integers)
	$\mid$	$\texttt{Bool}$	(booleans)
Base Types	$\texttt{B} ::=$	$\texttt{T}$	
Expressions	$e ::=$	v	(values)
	$\mid$	$e_1 \wr e_2$	(primitive operators)
	$\mid$	$e_1\ e_2$	(function application)
	$\mid$	$f[\overrightarrow{\tau_i}]$	(type application)
	$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	(let)
	$\mid$	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	(if)
	$\mid$	$e :: \tau$	(type annotation)
	$\mid$	$\textit{index}\ e_a\ e_i$	(array indexing)
Operators	$\wr ::=$	$+ \mid \leqslant \mid \neq \mid \wedge \mid \ldots$	
Formulas	$p, q ::=$	$P(\overline{w}) \mid p \wedge q \mid p \vee q \mid \neg p$ (P is the set of predicates)	
Logical values	$w ::=$	$v \mid F(\overline{w})$ (F is the set of logical functions)	
Context	$\Gamma ::=$	$\varnothing$	(empty)
	$\mid$	$\Gamma, x : \tau$	(variable binding)
	$\mid$	Γ, p	(predicate)
Axiom Sets	$A ::=$	$\overline{a_i\ \textbf{Drop}}, \overline{b_j\ \textbf{Share}}$	

Harpoons ($\overrightarrow{x}$) indicate lists.
Overlines ($\overline{x}$) indicate sets.

Figure 3.1: Syntax for Cogent, extended with refinement types

In addition to typing information, the context Γ may also include predicates. This allows information from branching constructs to be utilised in typing the branches.

The context Γ is wellformed if all free variables in types and predicates have been previously defined.

$$\frac{}{\vdash \varnothing} \qquad \frac{\vdash \Gamma \qquad fv(p) \subseteq dom(\Gamma)}{\vdash \Gamma, p}$$

$$\frac{\vdash \Gamma \qquad x \notin \mathsf{dom}(\Gamma) \qquad \mathsf{fv}(\tau) \subseteq \mathsf{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

3.2 Subtyping Relation

One refinement type $\{x : \mathtt{B} \mid p\}$ is a subtype of another $\{x : \mathtt{B} \mid q\}$ if under the typing context Γ, p entails q. Our Sub-Base rule is inspired by previous work of Lehmann and Tanter (2016).

$$\frac{\mathsf{extract}(\Gamma), p \vDash q}{\Gamma \vdash \{x : \mathtt{B} \mid p\} <: \{x : \mathtt{B} \mid q\}} \textsc{Sub-Base}$$

The $\mathsf{extract}$ function recursively traverses the context, gathering both predicates from refinement types and raw predicates. Any other cases are ignored. It returns a collection of logical predicates, which, together with p, must entail q for the subtyping relation to be true.

$$\begin{aligned}
\mathsf{extract}(\varnothing) &= \varnothing \\
\mathsf{extract}(\Gamma, p) &= \mathsf{extract}(\Gamma), p \\
\mathsf{extract}(\Gamma, x : \{y : \tau \mid p\}) &= \mathsf{extract}(\Gamma), p[x/y] \\
\mathsf{extract}(\Gamma, x : a) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \mathtt{T}) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \tau_1 \rightarrow \tau_2) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \mathtt{Array}\ \tau\ l) &= \mathsf{extract}(\Gamma)
\end{aligned}$$

To better understand why we have extended the context to contain predicates, consider the program:

```
1   f : U8 -> {v: U8 | v > 1}
2   f x = if x > 2 then x else 5
```

For the above program to be well typed, both branches of the **if** statement must be a subtype of the return type of `f`. In particular, the type of the expression `x` (from the **then** branch) needs to be a subtype of $\{\nu : \texttt{U8} \mid \nu > 1\}$. Without the information from the **if** guard that $x > 2$, there is no way of establishing this.

Sub-Base is utilised in conjunction with a rule for subsumption to upcast types.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Subsumption}$$

We also introduce an axiom for casting refinement types to base types:

$$\frac{}{\Gamma \vdash \{\nu : \texttt{B} \mid p\} <: \texttt{B}}\ \textsc{Ref-Base}$$

3.3 Type Inference Rules

Many of the type inference rules presented are standard for refinement type systems (Bierman et al., 2010; Lehmann and Tanter, 2016; Vazou, 2016).

For clarity of the rules, we omit the translation from the Cogent language to the predicate language. When a Cogent expression e_i appears inside a predicate, we really mean the Cogent expression e_i converted to the predicate language.

The rules for literals ILit and BLit have predicates that reflect their exact values, so that these values can be utilised in subtyping judgements. Inference for primitive operators (IOp, Div, COp, BOp) behaves in a similar fashion.

There are three inference rules for variables. If x is a base type Var-Base or a refinement type Var-Ref, the synthesised type is a refinement type where values are equal to x. Otherwise, Var is used, and the inferred type of x is the same as from the context.

Some primitive operators such as division can have undefined behaviours. In Div, we restrict the type of the divisor to values greater than zero, which eliminates undefined

behaviours that may arise from dividing by zero. Similarly, random access of arrays (Arr-Idx) is only permitted for any index value that is within the array bounds.

For If, we add the branching condition to the context so that it may be used for typing within the branches. In the **then** branch, the context is augmented with a predicate containing the branching condition. In the **else** branch, the negation of this predicate is added to the context.

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

$$\frac{\ell \in \mathbb{N} \qquad \ell < |\mathtt{T}|}{\Gamma \vdash \ell : \{\nu : \mathtt{T} \mid \nu = \ell\}}\ \text{ILit} \qquad \frac{\ell \in \{\mathtt{True}, \mathtt{False}\}}{\Gamma \vdash \ell : \{\nu : \mathtt{Bool} \mid \nu = \ell\}}\ \text{BLit}$$

$$\frac{\Gamma \overset{\text{weak}}{\rightsquigarrow} x : \mathtt{B}}{\Gamma \vdash x : \{\nu : \mathtt{B} \mid \nu = x\}}\ \text{Var-Base} \qquad \frac{\Gamma \overset{\text{weak}}{\rightsquigarrow} x : \{\nu : \mathtt{B} \mid p\}}{\Gamma \vdash x : \{\nu : \mathtt{B} \mid \nu = x\}}\ \text{Var-Ref}$$

$$\frac{\tau \notin \mathtt{B} \qquad \tau \text{ is not a ref.type} \qquad \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau}{\Gamma \vdash x : \tau}\ \text{Var} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e :: \tau : \tau}\ \text{Sig}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \mathtt{T} \neq \mathtt{Bool} \qquad \wr \in \{+, -, \times\} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \mathtt{T}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{\nu : \mathtt{T} \mid \nu = e_1 \wr e_2\}}\ \text{IOp} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathtt{T} \neq \mathtt{Bool} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \{\nu : \mathtt{T} \mid \nu > 0\}\end{array}}{\Gamma \vdash e_1 \div e_2 : \{\nu : \mathtt{T} \mid \nu = e_1 \div e_2\}}\ \text{Div}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \mathtt{T} \neq \mathtt{Bool} \qquad \wr \in \{=, \neq, <, \leqslant, >, \geqslant\} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \mathtt{T}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{x : \mathtt{Bool} \mid x = e_1 \wr e_2\}}\ \text{COp} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \wr \in \{\wedge, \vee\} \\ \Gamma_1 \vdash e_1 : \mathtt{Bool} \qquad \Gamma_2 \vdash e_2 : \mathtt{Bool}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{x : \mathtt{Bool} \mid x = e_1 \wr e_2\}}\ \text{BOp}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash e_1 : (x : \tau_1 \to \tau_2) \qquad \Gamma_2 \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1 e_2 : \tau_2[e_2/x]}\ \text{App} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}\ \text{Let}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathtt{Bool} \\ \Gamma_2, e_1 \vdash e_2 : \tau \qquad \Gamma_2, \neg e_1 \vdash e_3 : \tau\end{array}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ \text{If} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash a : \mathtt{Array}\ \tau\ l \\ \Gamma_2 \vdash i : \{\nu : \mathtt{U32} \mid \nu < l\}\end{array}}{\Gamma \vdash \textit{index}\ a\ i : \tau}\ \text{Arr-Idx}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \text{Subsumption}$$

Figure 3.2: Type inference rules

$$\boxed{\Gamma \vdash \tau_1 <: \tau_2}$$

$$\frac{\mathsf{extract}(\Gamma), p \vDash q}{\Gamma \vdash \{x : \mathtt{B} \mid p\} <: \{x : \mathtt{B} \mid q\}} \text{ SUB-BASE}$$

$$\frac{}{\Gamma \vdash \{\nu : \mathtt{B} \mid p\} <: \mathtt{B}} \text{ REF-BASE}$$

$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \qquad \Gamma, x : \tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash x : \tau_{11} \to \tau_{12} <: x : \tau_{21} \to \tau_{22}} \text{ SUB-FUN}$$

Figure 3.3: Subtyping rules

3.4 Example

To illustrate the usage of these typing rules, we will examine the process of type inference for a small function.

```
1   fun : {v: U8 | v < 3} -> {v: U8 | v < 8}
2   fun x = 5 + x
```

Typing the body of our function, `5 + x`, requires our IOP rule with the operator +. According to IOP, both operands `5` and `x` must be of the same base type. At this point, the base type is still to be determined. Highlighted in blue are premises yet to be constructed.

$$\frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash 5 : \mathtt{B} \qquad \Gamma_2 \vdash x : \mathtt{B}}{\Gamma \vdash 5 + x : \{\nu : \mathtt{B} \mid \nu = 5 + x\}} \text{ IOP}$$

As `5` is an integer literal, we can deduce its type `U8`.

$$\frac{\dfrac{\dfrac{}{5 \in \mathbb{N}} \qquad \dfrac{}{5 < |\mathtt{U8}|}}{\Gamma_1 \vdash 5 : \{\nu : \mathtt{U8} \mid \nu = 5\}} \text{ ILIT} \qquad \dfrac{}{\Gamma_1 \vdash \{\nu : \mathtt{U8} \mid \nu = 5\} <: \mathtt{U8}} \text{ REF-BASE}}{\Gamma_1 \vdash 5 : \mathtt{U8}} \text{ SUBSUMPTION}$$

We can derive x is also of type `U8` by using information from the function's type signature that $x : \{\nu : \texttt{U8} \mid \nu < 3\}$, along with our rule for subsumption.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \overset{\text{weak}}{\rightsquigarrow} x : \{\nu : \texttt{U8} \mid \nu < 3\}}}{\Gamma_2 \vdash x : \{\nu : \texttt{U8} \mid \nu = x\}}\ \textsc{Var-Ref} \qquad \dfrac{}{\Gamma_2 \vdash \{\nu : \texttt{U8} \mid \nu = x\} <: \texttt{U8}}\ \textsc{Ref-Base}}{\Gamma_2 \vdash x : \texttt{U8}}\ \textsc{Subsumption}$$

We have now inferred that `5 + x` is of type $\{\nu : \texttt{U8} \mid \nu = 5 + x\}$:

$$\dfrac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash 5 : \texttt{U8} \qquad \Gamma_2 \vdash x : \texttt{U8}\end{array}}{\Gamma \vdash 5 + x : \{\nu : \texttt{U8} \mid \nu = 5 + x\}}\ \textsc{IOp}$$

However, the return type of the function `fun` is not $\{\nu : \texttt{U8} \mid \nu = 5{+}x\}$, but $\{\nu : \texttt{U8} \mid \nu < 8\}$. To show that our inferred type is a subtype of the expected return type we use Sub-Base.

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash 5 : \texttt{U8} \qquad \Gamma_2 \vdash x : \texttt{U8}\end{array}}{\Gamma \vdash 5 + x : \{\nu : \texttt{U8} \mid \nu = 5 + x\}}\ \textsc{IOp} \qquad \dfrac{\mathtt{extract}(\Gamma), \nu = 5 + x \vDash \nu < 8}{\Gamma \vdash \{\nu : \texttt{U8} \mid \nu = 5 + x\} <: \{\nu : \texttt{U8} \mid \nu < 8\}}\ \textsc{Sub-Base}}{\Gamma \vdash 5 + x : \{\nu : \texttt{U8} \mid \nu < 8\}}\ \textsc{Subsumption}$$

The context Γ contains $x : \{\nu : \texttt{U8} \mid \nu < 3\}$. Therefore, the entailment judgement in the premise of Sub-Base is as follows:

$$x < 3,\ \nu = 5 + x \ \vDash\ \nu < 8$$

This is true, so the program successfully passes type checking.

Chapter 4

Implementation

The first four stages of the Cogent compiler (figure 4.1) are relevant to the implementation of refinement types in Cogent. Cogent programs are written in the surface language. This is desugared to the core language, which is a simpler subset of the surface language. Type checking is performed at both the surface and the core level. Some typing information gained from surface type checking is utilised by the core type checker.

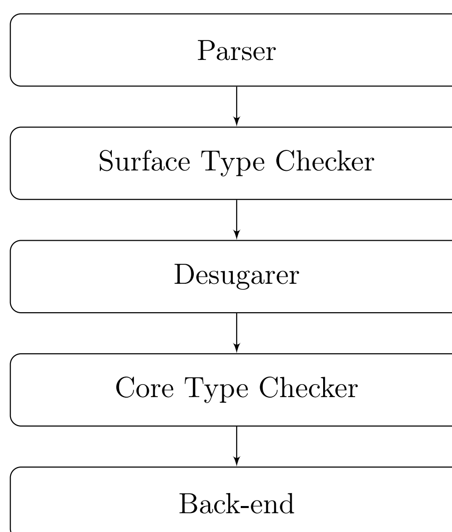

Figure 4.1: Stages of the Cogent Compiler

4.1 Implementation of Type Inference Rules

To represent predicates in the COGENT core language, a separate language for logical expressions was introduced. This logical expression language is essentially a one to one mapping from COGENT expressions to the logic language. Conceptually, it is a different language, and it is convenient for the implementation to have COGENT expressions and logical expressions be distinct.

Recall that our formalisation omitted the translation of COGENT expressions to the logic language. Consider the rule for **if**:

$$\frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \texttt{Bool} \qquad \Gamma_2, e_1 \vdash e_2 : \tau \qquad \Gamma_2, \neg e_1 \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ \textsc{If}$$

The COGENT expression e_1 is transformed into the logic language, then added to the context in the **then** branch. The negation of the logic language equivalent of e_1 is added to the context in the **else** branch.

$$\frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \texttt{T} \neq \texttt{Bool} \qquad \wr \in \{+, -, \times\} \qquad \Gamma_1 \vdash e_1 : \texttt{T} \qquad \Gamma_2 \vdash e_2 : \texttt{T}}{\Gamma \vdash e_1 \wr e_2 : \{\nu : \texttt{T} \mid \nu = e_1 \wr e_2\}}\ \textsc{IOp}$$

$$\frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \texttt{T} \neq \texttt{Bool} \qquad \Gamma_1 \vdash e_1 : \texttt{T} \qquad \Gamma_2 \vdash e_2 : \{\nu : \texttt{T} \mid \nu > 0\}}{\Gamma \vdash e_1 \div e_2 : \{\nu : \texttt{T} \mid \nu = e_1 \div e_2\}}\ \textsc{Div}$$

In performing type inference for primitive operators, we first infer the types of the parameters. Each primitive operator has a set of allowed input types, which can be determined from the first parameter. For example, if division is used with the first parameter having base type `U16`, then the second parameter must be a subtype of $\{\nu : \texttt{U16} \mid \nu > 0\}$.

To simplify pattern matching in the compiler code, we allow x : `B` to be treated as a refinement type $\{x : \texttt{B} \mid \text{True}\}$. When checking allowed input types for primitive operators, we normalise base types to refinement types so that we do not need to deal with base types separately.

4.2 Subtyping

The typing rule for SUBSUMPTION is nondeterministic.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Subsumption}$$

For any given type τ_1, there are many possible choices of τ_2 such that $\Gamma \vdash \tau_1 <: \tau_2$. The undecidable nature of this rule makes it impossible to implement.

Types can be annotated in the surface language to aid in type inference. On its own, the SUBSUMPTION rule is undecidable, however, under the assumption that types are given annotations from the surface language, the rule becomes deterministic. Rather than inferring an arbitrary τ_2, we utilise the annotated type τ_2 from the surface as an input, and check whether $\Gamma \vdash \tau_1 <: \tau_2$.

4.3 SMT Solver Usage

Propositional satisfiability (SAT) is the problem of deciding whether there exists an assignment to variables in a Boolean formula such that the formula is true. Satisfiability Modulo Theories (SMT) formulas are a generalisation of SAT formulas, but use a richer language (De Moura and Bjørner, 2011). For example, an SMT instance can include arithmetic and quantified variables.

$$\frac{extract(\Gamma), p \vDash q}{\Gamma \vdash \{x : B \mid p\} <: \{x : B \mid q\}}\ \textsc{Sub-Base}$$

The premise of our SUB-BASE rule is an entailment judgement. Refinement predicates range over variables, literals, arithmetic operations and boolean conjunctives; therefore, we rely on an SMT solver to discharge this proof obligation.

Recall the definition of $\mathsf{extract}$:

$$
\begin{aligned}
\mathsf{extract}(\varnothing) &= \varnothing \\
\mathsf{extract}(\Gamma, p) &= \mathsf{extract}(\Gamma), p \\
\mathsf{extract}(\Gamma, x : \{y : \tau \mid p\}) &= \mathsf{extract}(\Gamma), p[x/y] \\
\mathsf{extract}(\Gamma, x : \tau) &= \mathsf{extract}(\Gamma) \qquad \text{for other } \tau
\end{aligned}
$$

The $\mathsf{extract}$ function traverses the context Γ and collects each predicate, including predicates inside refinement types in the typing context. For the premise of Sub-Base to be true, the conjunction of the logical expressions from $\mathsf{extract}(\Gamma)$ and the refinement predicate p must imply q. We require an SMT solver to determine the truth of this expression:

$$
\left(\bigwedge_i p_i \wedge p \right) \Rightarrow q
$$

$$
\text{where } p_i \in \mathsf{extract}(\Gamma)
$$

We use the SBV library to invoke the SMT solver [1]. SBV supports many third-party SMT solvers such as CVC4 (Barrett et al.), MathSAT (Bruttomesso et al., 2008) and z3 (De Moura and Bjørner, 2008). They share a common SMT-Lib interface, so the choice of solver used can be easily changed. In the Cogent core type checker, we have selected Microsoft's z3 solver.

SBV.Dynamic is used instead of the standard SBV interface because expressions in the Cogent core language are deeply embedded. Care is taken so that terms created are are type correct.

Using SBV to call the SMT solver required expressions in the logic language to be converted to the SBV symbolic language. Logical expressions are traversed recursively and transformed into symbolic representations. Integers, booleans, and primitive operators are translated to equivalent constructs in the SBV language.

[1] https://hackage.haskell.org/package/sbv

The case for variables is not as straightforward. Creating a symbolic variable requires knowledge of the variable's type, which is not included in the original logical expression. To resolve this issue, we also pass in the typing context, which is used to lookup the types of variables encountered in expressions.

Variables are represented using de Bruijn indices. To refer to a variable within a refinement predicate, we must upshift the index to account for the new binder associated with the refinement type.

As an example, suppose that within the context Γ contains just one entry $x : \texttt{U8}$, with de Bruijn index 0. In constructing a refinement type $\{v : \mathsf{U}8 \mid v > x\}$, within the predicate expression $v > x$, we upshift x to be index 1, since index 0 is taken by v. All variables in logical expressions from $\mathsf{extract}(\Gamma)$ are also upshifted by 1 to align with this change.

Chapter 5

Conclusion

In this thesis, we presented a formalisation of a subset of refinement types in the COGENT core language, allowing integers and boolean values to be refined by arithmetic and logical predicates. This has been implemented in the core type checker, with the help of an SMT solver.

One use case of refinement types is eliminating undefined behaviours, such as division by zero. Usage of the division operator is now statically checked to prevent errors from occurring at runtime.

This work has paved the way for arrays to be added into the COGENT language. Arrays introduce the potential for memory to be accessed unsafely through array indexing. Refinement types have enhanced the type system to specify that arrays can only be indexed by values within the array bounds.

Regarding future work, the implementation of dependent functions is important for having a complete refinement type system. Dependent functions are functions for which the result type depends on the input value:

```
f : (x: a) -> {v: b | P(x,v)}
```

This complicates refinement predicates because they are expressed not only in terms of the base type `v`, but also the input type `x`. Dependent functions will enable more expressive specification of function return types.

Some constructs in COGENT are not yet supported by refinement types. Variant types, for example, can benefit from refinement. Consider a large variant type *ErrorType* consisting of all possible error types that a system can return, e.g. error types A-Z. Suppose we had a function that can only return error types X, Y or Z. Currently, we must pattern match over every possible option in *ErrorType*, from A to Z, to find the returned error type. This is an expensive operation. We can refine the variant type *ErrorType* with a predicate that the only possible error types that arise from the function call are X, Y and Z. This improves performance as less cases need to be checked.

Support for unboxed records is not difficult to implement. This can allow us to refine records based on the values of their fields. For refining other COGENT types, the main complication is representing them in the SMT symbolic language, which is less expressive than COGENT. However, the SMT solver is only concerned with values, and does not need to know type information such as linearity to determine the truth of entailment judgements.

Bibliography

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *ACM SIGARCH Computer Architecture News*, 44(2):175–188, 2016.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan JovanoviĂĞ, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 3642221092.

Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: CoqâĂŹArt: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

Gavin M Bierman, Andrew D Gordon, CĂČtĂČlin HriÅčcu, and David Langworthy. Semantic subtyping with an SMT solver. *SIGPLAN notices*, 45(9):105–116, 2010. ISSN 0362-1340.

Ana Bove and Peter Dybjer. Dependent types at work. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 57–99. Springer, 2008.

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *International Conference on Computer Aided Verification*, pages 299–303. Springer, 2008.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69âĂŞ77, September 2011. ISSN 0001-0782. doi: 10.1145/1995376.1995394. URL `https://doi.org/10.1145/1995376.1995394`.

Gerwin Klein, June Andronick, Gabriele Keller, Daniel Matichuk, Toby Murray, and Liam O'Connor. Provably trustworthy systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104): 20150404, 2017.

Nico Lehmann and É Tanter. Formalizing simple refinement types in Coq. In *2nd International Workshop on Coq for Programming Languages (CoqPLâĂŹ16), St. Petersburg, FL, USA*, 2016.

Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014.

Liam O'Connor. *Type Systems for Systems Types*. PhD thesis, Computer Science & Engineering, Faculty of Engineering, UNSW, 2019.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. *ACM SIGPLAN Notices*, 51(9): 89–102, 2016.

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam OâĂŹConnor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*, pages 323–340. Springer, 2016.

Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, UC San Diego, 2016.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 310–325, 2016.

Appendix 1: Formalisation

A.1 Syntax

Values	$v ::=$	x	(variables)
	$\mid$	ℓ	(literals)
Literals	$\ell ::=$	$\mathbb{N} \mid \texttt{True} \mid \texttt{False}$	
Types	$\tau ::=$	a	(type variables)
	$\mid$	$\texttt{T}$	(primitive types)
	$\mid$	$\{x : \texttt{B} \mid p(x)\}$	(refinement types)
	$\mid$	$x : \tau_1 \to \tau_2$	(functions)
	$\mid$	$\texttt{Array}\ \tau\ l$	(arrays)
Primitive Types	$\texttt{T} ::=$	$\texttt{U8} \mid \texttt{U16} \mid \texttt{U32} \mid \texttt{U64}$	(integers)
	$\mid$	$\texttt{Bool}$	(booleans)
Base Types	$\texttt{B} ::=$	$\texttt{T}$	
Expressions	$e ::=$	v	(values)
	$\mid$	$e_1 \wr e_2$	(primitive operators)
	$\mid$	$e_1\ e_2$	(function application)
	$\mid$	$f[\overrightarrow{\tau_i}]$	(type application)
	$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	(let)
	$\mid$	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	(if)
	$\mid$	$e :: \tau$	(type annotation)
	$\mid$	$\textit{index}\ e_a\ e_i$	(array indexing)
Operators	$\wr ::=$	$+ \mid \leqslant \mid \neq \mid \wedge \mid \ldots$	
Formulas	$p, q ::=$	$P(\overline{w}) \mid p \wedge q \mid p \vee q \mid \neg p$ (P is the set of predicates)	
Logical values	$w ::=$	$v \mid F(\overline{w})$ (F is the set of logical functions)	
Context	$\Gamma ::=$	$\emptyset$	(empty)
	$\mid$	$\Gamma, x : \tau$	(variable binding)
	$\mid$	Γ, p	(predicate)
Axiom Sets	$A ::=$	$\overline{a_i\ \textbf{Drop}}, \overline{b_j\ \textbf{Share}}$	

Harpoons ($\overrightarrow{x}$) indicate lists.
Overlines ($\overline{x}$) indicate sets.

A.2 Wellformedness

$$\boxed{\vdash \Gamma}$$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \qquad fv(p) \subseteq dom(\Gamma)}{\vdash \Gamma, p}$$

$$\frac{\vdash \Gamma \qquad x \notin dom(\Gamma) \qquad fv(\tau) \subseteq dom(\Gamma)}{\vdash \Gamma, x : \tau}$$

A.3 Context Relations

A.3.1 Context Splitting

$$\boxed{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus \Gamma_2} \qquad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow \Gamma_1 \boxplus x : \tau, \Gamma_2}$$

$$\frac{}{A \vdash \varepsilon \rightsquigarrow \varepsilon \boxplus \varepsilon} \qquad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A \vdash \tau \textbf{ Share}}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus x : \tau, \Gamma_2}$$

A.3.2 Weakening

$$\boxed{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau, \Gamma'} \qquad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma' \qquad A \vdash \tau \textbf{ Drop}}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}$$

$$\frac{}{A \vdash \varepsilon \overset{\text{weak}}{\rightsquigarrow} \varepsilon}$$

A.4 Type Inference Rules

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

$$\frac{\ell \in \mathbb{N} \qquad \ell < |\texttt{T}|}{\Gamma \vdash \ell : \{\nu : \texttt{T} \mid \nu = \ell\}} \text{ ILit} \qquad \frac{\ell \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash \ell : \{\nu : \texttt{Bool} \mid \nu = \ell\}} \text{ BLit}$$

$$\frac{\Gamma \overset{\text{weak}}{\rightsquigarrow} x : \texttt{B}}{\Gamma \vdash x : \{\nu : \texttt{B} \mid \nu = x\}} \text{ Var-Base} \qquad \frac{\Gamma \overset{\text{weak}}{\rightsquigarrow} x : \{\nu : \texttt{B} \mid p\}}{\Gamma \vdash x : \{\nu : \texttt{B} \mid \nu = x\}} \text{ Var-Ref}$$

$$\frac{\tau \notin \mathtt{B} \qquad \tau \text{ is not a ref.type} \qquad \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e :: \tau : \tau}\ \textsc{Sig}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \mathtt{T} \neq \mathtt{Bool} \qquad \wr \in \{+, -, \times\} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \mathtt{T}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{\nu : \mathtt{T} \mid \nu = e_1 \wr e_2\}}\ \textsc{IOp} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathtt{T} \neq \mathtt{Bool} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \{\nu : \mathtt{T} \mid \nu > 0\}\end{array}}{\Gamma \vdash e_1 \div e_2 : \{\nu : \mathtt{T} \mid \nu = e_1 \div e_2\}}\ \textsc{Div}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \mathtt{T} \neq \mathtt{Bool} \qquad \wr \in \{=, \neq, <, \leqslant, >, \geqslant\} \\ \Gamma_1 \vdash e_1 : \mathtt{T} \qquad \Gamma_2 \vdash e_2 : \mathtt{T}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{x : \mathtt{Bool} \mid x = e_1 \wr e_2\}}\ \textsc{COp} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \wr \in \{\wedge, \vee\} \\ \Gamma_1 \vdash e_1 : \mathtt{Bool} \qquad \Gamma_2 \vdash e_2 : \mathtt{Bool}\end{array}}{\Gamma \vdash e_1 \wr e_2 : \{x : \mathtt{Bool} \mid x = e_1 \wr e_2\}}\ \textsc{BOp}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash e_1 : (x : \tau_1 \to \tau_2) \qquad \Gamma_2 \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1 e_2 : \tau_2[e_2/x]}\ \textsc{App} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathtt{Bool} \\ \Gamma_2, e_1 \vdash e_2 : \tau \qquad \Gamma_2, \neg e_1 \vdash e_3 : \tau\end{array}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ \textsc{If} \qquad \frac{\begin{array}{c}\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash a : \mathtt{Array}\ \tau\ l \\ \Gamma_2 \vdash i : \{\nu : \mathtt{U32} \mid \nu < l\}\end{array}}{\Gamma \vdash \textit{index}\ a\ i : \tau}\ \textsc{Arr-Idx}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Subsumption}$$

A.5 Subtyping

$$\boxed{\Gamma \vdash \tau_1 <: \tau_2}$$

$$\frac{\mathsf{extract}(\Gamma), p \vDash q}{\Gamma \vdash \{x : \mathtt{B} \mid p\} <: \{x : \mathtt{B} \mid q\}}\ \textsc{Sub-Base}$$

$$\begin{aligned}
\mathsf{extract}(\emptyset) &= \emptyset \\
\mathsf{extract}(\Gamma, p) &= \mathsf{extract}(\Gamma), p \\
\mathsf{extract}(\Gamma, x : \{y : \tau \mid p\}) &= \mathsf{extract}(\Gamma), p[x/y] \\
\mathsf{extract}(\Gamma, x : a) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \mathtt{T}) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \tau_1 \to \tau_2) &= \mathsf{extract}(\Gamma) \\
\mathsf{extract}(\Gamma, x : \mathtt{Array}\ \tau\ l) &= \mathsf{extract}(\Gamma)
\end{aligned}$$

$$\frac{}{\Gamma \vdash \{\nu : \mathtt{B} \mid p\} <: \mathtt{B}}\ \textsc{Ref-Base}$$

$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \qquad \Gamma, x : \tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash x : \tau_{11} \to \tau_{12} <: x : \tau_{21} \to \tau_{22}}\ \textsc{Sub-Fun}$$