Refinement Types in Real-World Programming

Xiu Hong Kooi
Wolfson College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: xhk20@cam.ac.uk

30th May 2021

Declaration

I, Xiu Hong Kooi of Wolfson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,706

Signed: ***KooiXiuHong***

Date: ***30-05-2021***

This dissertation is copyright ©2021 Xiu Hong Kooi.
All trademarks used in this dissertation are hereby acknowledged.

Acknowledgement

I am extremely grateful to my supervisor, Prof. Alan Mycroft, for providing guidance and feedback throughout this project and the academic year.

I would like to thank everyone at Wolfson College for making my year at Cambridge a memorable one.

Deepest gratitude to my family for their support and making my dream a reality.

Finally, I would like to thank my friends for keeping me sane during these uncertain times.

Abstract

Refinement Type in Real-World Programming

Refinement Types provide a more expressive type checking and allow more errors to be caught automatically, however it is not a feature the general programmer is aware of. Existing research in the area is focused predominantly on the theoretical aspects of refinement types rather than their implementation in real-world programming. While there are certain programming languages available that support refinement types, these languages are often limited to a small number of research projects that are no longer maintained or simply open-source libraries that simulate a restricted form of refinement types. The introduction of refinement types in mainstream programming proves to be difficult because of features like mutation. Refinement types are able to make use of free variables in the program, this means that a change in the program state might affect the meaning of types, making type checking ambiguous. Furthermore, in general showing that a predicate holds is undecidable, making static type checking of refinement types hard. Our project aims to present refinement types with an emphasis on their behaviour in real-world programming languages. We achieve this by describing an imperative language named Simple-R which strongly resembles C. We address the problems stated above by adapting multiple well-known concepts in programming languages such as immutable variables, closures and hybrid type checking. Using a combination of these techniques we formalise the rules for variables and pointers in Simple-R in order to specify the behaviour of refinement types precisely.

Total word count: 14,706

Contents

List of Figures

Chapter 1

Introduction

Type systems [23] have been one of the most researched field in programming languages theory. They improve the reliability of a language by enforcing rules, preventing operations being applied on incompatible data. Type systems can be broken down into various categories. In addition to the well known *Static* [4] and *Dynamic* [38] typing, programming languages have included more powerful and flexible type systems over the years. Languages like *C#* [29] and *Go* [18] for example allow *Type Inference* [37].

Types are fundamental in order to show a program's correctness, the use of types restricts and therefore eliminates illegal programs at compile time. However, even a well-typed program still leaves room for various errors such as

1. **Out of bounds access** In the case of arrays or buffers, the type checker can guarantee that the programmer is using an integer to index elements but it makes no guarantee that the index is indeed within the valid range.

2. **Arithmetic errors** An arithmetic operation can type check that it is indeed operating on numerical values however it is unable to verify the legality. Forbidden arithmetic operations such as as zero divisors or square root of a negative number cannot be caught by the type checker.

3. **Logical error** Finally, a type checker cannot verify any logical mistakes made by the programmer. Mistakes are inevitable, while the programmer can ensures that the arguments passed to a function are two integers, there is no way for the type checker to ensure that the result is the sum of the two.

Dependent Types [2] appears to be a potential solution to the problem, dependent types allow the programmer to create types whose definition depends on values, e.g.

a type of vector that is of length n. A type system that provides such refined control over the values it can take unlocks possibility that are previously unavailable such as domain-specific type checking.

A slight restriction on dependent types can be found in *Refinement Types* [22] where one is allowed to define specify subtypes of existing types, e.g. a type of non-zero integers. Refinement Types can be seen as a generalisation of *Floyd-Hoare Logic* assertions [12], they are constrained by a decidable predicate which brings a better ease of programming compared to dependent types. While dependent types are significantly more powerful than refinement types, the latter provides a good balance between expressiveness and ease of use.

1.1 Overview of Refinement Types

In this section we discuss at a very high level the behaviour one would expect from refinement types and the benefits of using refinement types.

```
type EvenInt = {v : int | v % 2 == 0}

function f(int x) {
  if (x % 2 != 0)
    throw error ``x has to be even''
  ...
}

function refined_f(EvenInt x) {
  ...
}
```

Figure 1.1: Example of Refinement Types

Consider the pseudo-code in figure 1.1 and assume that the function f is an arbitrary function that requires the argument x to be even. The implementation first checks if the value provided is indeed even, if it is not then it throws a runtime error. Alternatively, the function *refined_f* achieves the same through the use of refinement types. *EvenInt* is a integer type whose value is guaranteed to be even. If one passes an odd integer to the function it will be caught automatically either at compile time or runtime by the type checker.

Refinement types allows programmer to define much richer and expressive types, they allow for safer code as it is able to guarantee more properties are checked by

the type checker. Furthermore, using refinement types eliminates the need for many trivial error handling code. Around 4% of code in every program is dedicated to error handling [5], refinement types can reduce the numbers of checks needed and allow the programmer more time to focus on writing the logical part of the code, indirectly leading to more robust codebases [58].

1.2 Motivation

Desipte their usefulness, dependent and refinement types remain uncommon in real-world programming. While their theory has been established several decades ago, only a small number of programming languages support these advanced type systems. Furthermore, the languages that do support them are limited to functional languages and theorem provers that are rarely used in real-world programming.

Mainstream languages often follow the imperative programming paradigm and one of the challenges of introducing these type systems in imperative languages is the presence of mutation. Imperative languages rely on statements that change the state of the program, these changes have the potential making the semantics of refinement types unclear. For instance, if one defined a refinement type as $\{v : int \mid v < N\}$ where its values have to be less than N, what is the behaviour if the value of N changes?

Past research has shown the feasibility of using dependent and refinement types in imperative languages, however these studies are only focused on the theoretical aspects which are highly technical and complex, requiring prerequisite knowledge in the literature. While we find these works novel and made important advancements in the field, they fail to capture how these types systems could relate to mainstream programming languages.

We argue that the research in the area of dependent types and refinement types can be made more accessible. Research in the area often assumes full knowledge of *The Lambda Calculus* [44] and uses the concept of *Monads* [54] for state changes. The semantics of the λ-calculus is simple and naturally relates to functional programming languages, however, when one tries to model imperative programming by extending the calculus, we begin to lose the simple nature.

Our project is motivated by this lack of accessible literature on the theory of refinement types especially involving mainstream languages with features such as mutation. We aim to present a simple, restricted framework that captures the essence of refinement types. The said framework will be presented as an extension to *WHILE*

language [51]. The WHILE language was created as a tool to aid the study of programming language theory, thus we believe it strongly satisfies our requirement of accessibility with its simple syntax and semantics. Using the framework, we study the interactions between mutation and refinement types, furthermore we relate our findings to concepts in mainstream languages.

1.3 Key Contributions

At the end of the project we achieved the following,

1. Survey the state of dependent and refinement types in real-world programming languages and analyse their differences.
2. Construct Simple-R, an imperative language with refinement types based on the WHILE language that resembles C.
3. Explore the ambiguities of refinement types in a mutable environment and propose solutions to resolve these ambiguities.
4. Discuss how the strategies used to handle refinement types in Simple-R are closely tied to concepts in real-world languages.
5. Discuss how our work complements the existing literature and possible future extensions.

This dissertation serves as a document for our findings and is organised as follows. This first chapter outlines our motivation for this project and our contributions.

In chapter 2 we provide a background of the area, we first give the formal definition of dependent types and how it differs from refinement types. We then survey the current state of dependent types and refinement types in existing programming languages. We explore different languages with varying paradigms and summarise their differences in order to have a better understanding of how they support these advanced type systems. The languages that are studied include Agda, Haskell, Idris, Scala, TypeScript and Xanadu. Finally, we studied C++ and how one can construct a limited form of refinement type using the template mechanism.

In chapter 3 we identify three key concepts in programming language that are relevant to our project. These concepts include immutability, closures and hybrid type checking. Their mechanics in programming languages are non-trivial, we explain and clarify some of their behaviours. Finally, we explained how these concepts can be used to implement refinement types in an imperative language.

In chapter 4 we introduce an imperative language with refinement type support. Our language, named Simple-R, is a First-Order Programming Language which resembles the C programming language. Our aim with the language is to capture the core behaviour of refinement types, for the pursuit of simplicity, the language is kept at a minimal with only the fundamental features. In this chapter, we raise multiple scenarios where refinement types causes ambiguity when mutation is possible. We address this problem by starting with a simple approach where mutation is restricted on variables associated with refinement types. We then propose a solution based on closures that allows variables in types to be independent from program variables. Finally we tackle the problem where mutation happens indirectly via pointers.

In chapter 5 we discuss our work with regards to the existing research in the area and how our project complements the literature. Finally, we conclude by summarising our project and discuss future work in chapter 6.

Chapter 2

Background Research

In this chapter we provide a formal introduction to the theory of dependent types and refinement types. Building on the definition, we identify how these two advanced type system differ. As functional languages prove to be more commonly associated with these type systems, we review the Agda, Idris and Haskell programming languages to understand the natural behaviour of dependent types and refinement types. During our research, we noted several research projects that bring these type systems to imperative languages, we review the *Xanadu* [56], *Ynot* [32] and *RSC* [53] projects. Open-source programmers have also made strides in the area, we review a Scala library, *refined* [17], which brings refinement type to the Scala language. Finally, we study the C++ templating mechanism and identify its similarity to refinement types.

2.1 Dependent Types

At a very high level dependent types are types that depend on the value of another type. For example, we can define a type that captures the integers less than x using the definition `type BoundedInt :=` $\Pi x : int.\{i : int \mid i < x\}$. In this case we can say that the type *BoundedInt* depends on the type *int* x.

2.1.1 Dependent Π Types

We can capture the definition mathematically using the notion of *dependent product types*, written as Π type. This is also sometimes referred to as *dependent function type* as in this definition we construct a function $F : A \to B$. The function F takes an element of type A and gives us an element of type B which may depend on A.

We express it mathematically using the Π notation as

$$\prod x : A.F(x)$$

In this definition, $F(x)$ is the type family for the type B that depends on A. However F could be a constant function, so we can also express the definition as $\Pi x : A.B$ where in this case B does not depend on the value x. One can define an integer type that is bounded by a certain value x as $\Pi x : int.\ \{i : int \mid i < x\}$.

Interestingly, the dependent product type correspond to the *forall quantifier* as per the *Curry–Howard correspondence* [20]. The idea is that the dependent function $F(x)$ corresponds to predicate $P(x)$ and thus the dependent product type has a one-to-one correspondence to $\forall x : A.P(x)$.

2.1.2 Dependent Σ Types

In addition to the dependent product type, we have the notion of *dependent sum types*, written as Σ type. This is often referred to as the *dependent pair type* as the resulting type here is an ordered pair. Specifically the resulting pair $\langle a, b\rangle$ is ordered such that the second element depends on the first element. The mathematical definition is similar to that of the product type

$$\langle a, b\rangle : \sum x : A.F(x)$$

In the case $a : A, b : F(x)$, similarly, F could be a constant function and thus the expression is $\Sigma x : A.B$. Consider the following example,

$\Sigma x : Int.\{y : Int \mid y = x * 2\}$, then the type would contain values like $\langle 1, 2\rangle$ and $\langle 4, 8\rangle$ where the second element in the pair is doubled the first.

Like the dependent product type, the dependent sum type correspond to a universal quantifier, in this case, the *existential quantifier*. As per the Curry–Howard correspondence, $F(x)$ corresponds to predicate $P(x)$ thus $\Sigma x : A.F(x)$ corresponds to $\exists x : A.P(x)$.

2.2 Refinement Types

Refinement Types, sometimes referred to as *Subset Types* [33] or *Liquid Types* [45] is a type system where a certain base type is refined by a *decidable* predicate, a refinement type A has the form $\{\upsilon : B \mid P(\upsilon)\}$. The *value variable* υ represents the values that A can take. The type B is the base type and $P(\upsilon)$ is the *refinement predicate*, a boolean expression involving υ and any free variables in the program.

Formally, a value x is well typed with regards to a refinement type if $P(x)$ evaluates to *true*.

2.3 Refinement Types vs Dependent Types

The key differentiating factor between refinement types and dependent types is that the latter is able to depend on values in abritary manner whereas the former only restrict using a decideable predicate.

In many cases refinement types and dependent types can achieve the same thing, $\{v : int \mid v < x\}$ and $\Pi x : int.\ \{i : int \mid i < x\}$ effectively define the same type, a bounded integer whose value cannot be greater than x. They differ only in the way x is provided, in a refinement type x is a free variable whereas in a dependent type x is a parameter.

However, there a certain scenarios where a type is possible in dependent types but not in refinement types. Consider the type `Πn : int.{v : Vec<int> | len(v) = n}` where given an integer n one gets a type that is a vector of length n. This is not possible in a refinement type system as the base type *int* is not the same as the refined type `Vec<int>`.

Dependent types prove to be more powerful than refinement types but the latter for all intents and purposes does the job in providing the improved expressivity for general programming. For this reason, we base our project on refinement types.

The decidability of refinement types allows one to have a high level of automation and provides a simpler understanding. In various refinement type systems, the authors restrict the refinement predicate to be SMT decidable, this allows many properties to be checkable at compile time through the use of theorem provers. However, this is only possible in cases where values are known at compile time which limits it to functional languages where mutation is not possible.

2.4 Functional Languages

In this section we analyse functional languages that suports dependent types and refinement types. Functional languages use the *functional programming* [27] paradigm which is a programming paradigm whose evaluation consists of using a series of function applications. Functional languages has the notion of purity which states that i) a function will always return the same value when given the same arguments, ii) the

evaluation of a function has no side effects (changes to the state of the program). In this paradigm, functions return values as opposed to altering the state of the program. This programming paradigm is *declarative* in that it focuses on describing *what* the program will accomplish rather than *how* it accomplish it.

2.4.1 Agda

Agda [34] is a purely functional language originally developed by Ulf Norell in 1999 however the first appearance the current version known as Agda 2 is in 2007. Agda has all the necessary constructs one would expect in a functional language such as first-class functions, inductive definitions, pattern matching, etc. In addition to being a functional language, Agda also serves as an automated theorem prover. Agda is one of the few programming language with native dependent type support.

The code listing below is an example of defining a fixed length vector in dependent type.

```
data Nat : Set where
zero : Nat
suc  : Nat -> Nat

data EvenNat : Nat -> Set where
even-zero  : EvenNat zero
even-plus2 : {n : Nat} -> EvenNat n -> EvenNat (suc (suc n))

data Vec (A : Set) : Nat -> Set where
[] : Vec A zero
_::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

Figure 2.1: Dependent Types in Agda

While Agda provides dependent type support, it remains a niche language. One of the reason being its paradigm, Agda is *purely functional* [46], meaning that all functions are pure (i.e. not relying on the program state or other mutable data). Functional languages are generally considered harder to learn and grasp compared to other paradigms [25]. Furthermore, the general lack of awareness and experienced users who can program with dependent types contribute to Agda's niche in the programming world and is predominantly used for theorem proving.

Idris

Idris [3] is a dependently typed functional language which first appeared in 2007 and bears similarity with Agda, both in terms of paradigm and type system. However they differ in one crucial way, Idris was designed to emphasise general purpose programming rather than theorem proving. It provides interoperability with systems libraries and C programs, as well as language constructs for domain specific language implementation [10]. However, Idris remain predominantly a research tool. Idris is not production ready [10] as it is missing certain libraries and more importantly nobody is working on Idris full time. Furthermore Idris is still a functional language, hence suffering from the limitations stated earlier.

2.4.2 Haskell and LiquidHaskell

Haskell [19] is a purely functional programming language first appeared in 1990. In contrast to Agda, Haskell is often considered a more general purpose programming language. Haskell is among the most popular programming languages and argubly the most popular "pure" language in the world [8], even being adopted by software companies such as Facebook [15].

Haskell is not a language that supports dependent types natively however many extensions has been developed to simulate the experience. Generalised Algebraic Data Types (GADTs) are a generalization of the algebraic data types, GADTs allow the programmer to explicitly write down the types of the constructors [41].

Using the power of GADT one could define some dependent types as so

```
data EvenNat (n :: Nat) where
  EvenZero   :: EvenNat 0
  EvenPlusTwo :: EvenNat n -> EvenNat (n + 2)

data Vec a (n :: Nat) where
  Nil  :: Vec a 0
  (:>) :: a -> Vec a n -> Vec a (n + 1)
```

Figure 2.2: GADTs in Haskell

Although GADTs provide a way of simulating dependent types in Haskell and argubly enough for simple dependent typing enough for many cases. Haskell does not qualify as a fully dependent language due to the lack of certain features such as

dependently typed functions. There have been proposals to add full dependent type support to Haskell however a lot of work remains to be done [43, 55].

LiquidHaskell

LiquidHaskell [36] is a static type-checker that allows programmers to specify and check program correctness by applying refinement types theory. The programmer annotates refinement types in the form of Haskell comments `{-@ annotation @-}`, the properties must be SMT decideable and is checked using a variety of theorem provers.

Type checking in LiquidHaskell happens in three phases, phase one is the normal compilation using the Haskell compiler. In phase two LiquidHaskell generates the set of types constraints provided by the annotations, these are checked using a SMT solver in phase three. After these verifications, LiquidHaskell outputs "SAFE" if the program is well-typed. Otherwise, type errors are reported with along with the line it occurs.

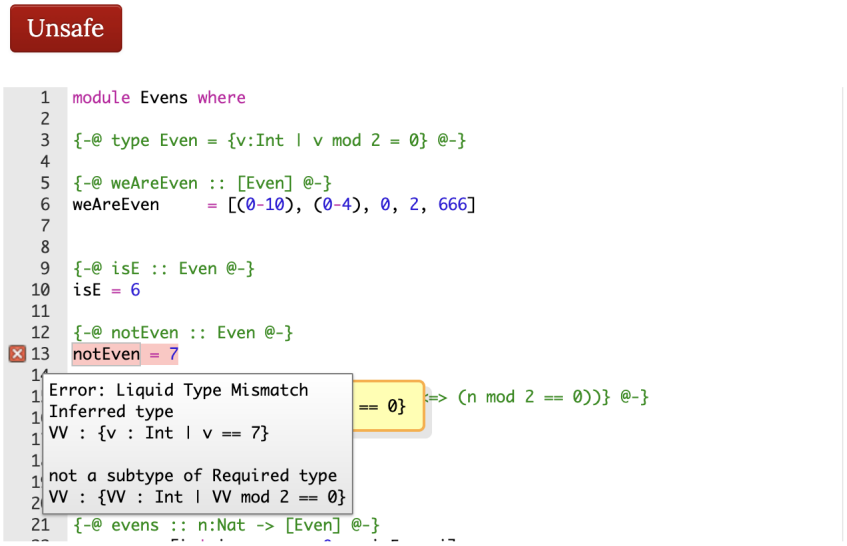

Figure 2.3: LiquidHaskell Demo

Static refinement type checking is made possible in Haskell because of the lack of assignment statements. Once a variable x is declared in a scope, its value cannot be changed. This allows SMT solvers to know precisely the value of x and prove the refinement predicate statically. In the case of I/O their properties cannot be verified using LiquidHaskell.

2.5 Imperative Languages

In this section we discuss dependent and refinement typing with regards to imperative languages. Imperative languages use the *imperative programming* [49] paradigm, in contrast to the functional paradigm, this paradigm emphasises *how* a program will operate by using a series of statements to change the program state.

2.5.1 Research Languages

Currently there are no production imperative programming language with native support for dependent types or refinement types. However, previous research has looked at creating new languages with these capabilities or extending existing languages. In this section we survey previous works in the area.

Ynot

Ynot is an axiomatic extension to the *Coq Proof Assistant* [50] created by Aleksandar Nanevski, et al [32]. It adds support for writing and reasoning about dependently-typed programs with side-effects, i.e. written in an imperative manner.

The formal aspects of Ynot is based on *Hoare Type Theory* [30], a previous work by the authors and the concept of Monads, specifically the monadic system of Haskell. This allows Ynot to write ML or Haskell-style programs and still be able to formally reason about their values and effects.

Ynot is publicly available for download [31] however it suffers from the same issues as Agda, Coq is a niche language and conventionally used as a theorem prover. Furthermore, in order to take advantage of dependent types in Ynot, one has to write monadic program as one would in Haskell which most programmers are unfamiliar with. Finally, the paper written on Ynot is theoretically complex, requiring prerequisite knowledge of advanced concepts such as Hoare Logic, Hoare Type Theory and Monads, making the paper challenging to new readers seeking to understand the concept of dependent types or refinement types in imperative languages.

Xanadu

Xanadu [56] is a dependently typed imperative language created by a team at the University of Cincinnati. The language was implemented in OCaml and is based on a previous project by the authors named *Dependent ML* (DML) where the authors used a *restricted form of dependent types* which essentially takes the form of refinement types.

One aspect we found appealing in the paper is the use of real-world examples to show the concept of dependent types. The authors used examples such as binary search trees, array bound checking, etc to introduce the concepts, allowing new readers to easily understand the basic behaviour of these type systems. However, the Xanadu language described in the paper still uses many complex theoretical concepts that most readers are unlikely to be familiar with.

There are no examples of the actual language available online except some code snippets shown by the author in the original paper. As such we are unable to provide any analysis on the language itself. Xanadu was stated to be available online in the original paper, however the original cited webpage appears to have been taken offline. It is safe to conclude that the language itself is also no longer in active development. However, the language is worth a mention here as it shows the feasibility of imperative languages having dependent typing. The research by the authors are impressive and it serves as inspiration for our project.

Refined TypeScript

TypeScript is a superset of JavaScript that brings static typing to the language. A team from the University of California, San Diego wrote a paper [53] in 2016 showcasing Refined TypeScript (RSC), a lightweight refinement type system for TypeScript.

The key observation is that RSC behaves like LiquidHaskell, it serves as a program verifier that verifies the refinement predicate using an SMT solver. However, unlike Haskell, Typescript is an imperative language and mutation is possible. The authors handled the mutable code in two ways, i) refinements are restricted to immutable variables and ii) reassignment is handled using Static single assignment (SSA) translation.

The paper provides a unique way of handling assignment via SSA translation however this method introduces complexity into the design which are unnecessary in our project. We find this paper intriguing because of their idea of introducing refinement types in a mainstream programming language.

2.5.2 Refined Scala

The *Scala* [40] Programming Language is a statically typed multi-paradigm language, it is compiled into Java ByteCode and executed on the *Java Virtual Machine*. As such, Scala shares similarity with Java and provides language interoperability.

The Scala Type System is richer than the Java Type System, Scala supports certain types not available in Java such as Monads, Singleton Types, etc. Despite the rich type system, refinement types and dependent types are not native features of the language. However, an open source project named *refined* [17] has brought refinement types into Scala.

```scala
def main(args: Array[String]): Unit = {

  val l: Int Refined Less[10] = 3
  println("Compile Time Refinement Type Check: " + l)

  val i: Int = -1
  val i1: Either[String, Int Refined Greater[1]] = refineV[Greater[1]](i)
  println("Runtime Refinement Type Check: " + i1.getOrElse("invalid ref predicate"))

}
```

Figure 2.4: Refinement Types in Scala

Figure 2.4 is an example of refinement types using refined. While the library does not have a formal write up of its implementation and design decisions, the informal documentation is sufficient for us to identify and discuss multiple interesting design.

The first interesting observation we made is that the library does not allow refinement types with any arbitrary refinement predicate. Instead, the library has a set of predefined predicates that one is able to associate with refinement types. In the example provided in figure 2.4 we use the predicate `Less[N]` to refine the integer type such that its value is always less than `N`. As Scala is an imperative language where mutation is possible, the library insist that `N` must be a literal and not a variable.

Furthermore, in order to have compile time refinement type checking, one must initialise variables that is of refinement type with a literal. Using the example in figure 2.4, the statement `val l: Int Refined Less[10] = x` would be forbidden as one tries to initialise a refinement typed variable with a non-literal.

If one wishes to initialise a refinement type using a variable or function, one must use the `refineV[P](v)` function, where `P` is a predicate and `v` is a value. This function tells refined to perform type check at runtime and will either give an error if `v` does not satisfy `P` or it gives the value `v` which is refined using `P`. An example of this use case in shown in figure 2.4.

The refined Scala library provides a simple of using refinement types in a modern

language through a restrictive manner. There are many design decisions one can take away from this project such as the combination of compile-time and runtime type checking.

2.5.3 C++ Templates

C++ Templates [13] are a way of passing the type of a data as a parameter so certain code can be reused. For instance, the same sorting algorithm can be used on multiple data types such as *int* and *double*, using templates the programmer will not need to write the same sorting function multiple times for different data types.

Templates are often compared to Java's *Generics* [11] as the C++ equivalent. While this statement is mostly true, C++ templates has an additional feature. Generics only allow the template parameter to be a class, templates on the other hand allow the parameter to be a class, values or pointers. This feature allows one to create types that resembles refinement types.

C++ 11 introduced the notion of *constexpr* and *static_assertion*. These allow for certain compile time checking, however the values checked must fulfill the *constant expression* requirements [14].

```
template <int N, int M>
struct LessThanN {
  const int value;
  LessThanN(): value(M) {
    static_assert(M < N, ``type is invalid'');
  }
};
```

Figure 2.5: Compile Time Checking in C++

Figure 2.5 shows a struct that simulate a refined integer type whose values can only be less than `N`. It is defined as `LessThanN<10, 5> x = LessThanN<10, 5>;`. Type checking is done at compile time using the `static_assert` in the constructor. If one provides an invalid definition then a compile time error is thrown.

`LessThanN<10, 12> x = LessThanN<10, 12>;` yields the following message:

static_assert failed due to requirement '12 < 10' "type is invalid"

Notice in our example we made the `LessThanN` type immutable, the reason here is that there is no trivial way to ensure that if the programmer changes the value using `x.value = 100` the new value coheres with the refinement type.

Mutation of Template Variable

In our previous example we have been passing an "integer literal" as the template parameter. Often programmers are required to work with varied values through variables, how will this effect the behaviour?

Consider the following hypothetical situation using the struct defined in figure 2.5:

```
int n = 5;
LessThanN<n, 3> ltf = LessThanN<n, 3>();
n = 2;
```

Figure 2.6: Modifying Template Variable in C++

Compiling the above code leads to a compile time error, C++ requires template arguments like n above to be constant expressions. In order to use variables as template arguments, they have to be defined **const**. In which case the above situation is impossible as the variable n now cannot be changed at line 3.

We see glimpses of refinement typing in C++, while it is still far from full refinement types. We are able to take note of certain mechanics such as ensuring variables associated with types are constant and the use of immutability.

2.6 Summary

In this chapter we formally introduced dependent types and refinement types, these two type systems are similar on the surface but are formally very different. Certain functional languages such as Agda and Idris have native support for dependent types however we observed that there are no imperative language with the same support. Research has attempted to bring dependent types and refinement types to imperative programming; these projects have been successful but the languages that were created are not widespread.

We found several open-source projects that attempted to bring refinement types to mainstream languages. We analysed *refined*, a Scala based library that introduces refinement types to the language in a restrictive manner. We found the project

interesting however it remains informal and lacks formal documentation. Finally, we studied C++ templates and notice the potential of creating types that simulate refinement types.

We identify that work in the area are missing in certain aspects, dependent types and refinement types are conventionally associated with functional languages because of the lack of mutation. Previous research that attempts to bring the more advanced type system to imperative languages remains mainly theoretical and does not relate to real-world languages. Practical projects that address these form of typing in real-world languages remains informal and are mainly documented as online articles. Our project addresses these gaps by providing formal semantics for refinement types and relating them to real-world imperative languages.

Chapter 3

Key Concepts in Programming Languages

Throughout our research, we observed certain ideas that commonly appear across different language constructs. In this chapter, we introduce three concepts, immutability, closures and hybrid type checking. We discuss these concepts in depth and identify how they relate to refinement types which is critical in the development of our language.

3.1 Immutable Objects in Programming

In this section we analyse the crucial differences between an immutable object and a constant in different programming lanaguages. We follow up our analysis by stating the importance of immutability and constancy with regards to type systems.

3.1.1 Immutable vs Constant

Immutability is an important concept in many programming languages, an object is immutable if its state cannot be modified after creation. *Constancy* is often mistaken as an alternate name for immutability however they are two different concepts in programming languages. A constant is a variable that cannot be reassigned after initialisation, one can say that a constant is an immutable variable. The difference between an immutable object and a constant is subtle but crucial, these two notions does not provide the same guarantees that a value remains the same.

Consider the following Java code in figure 3.1, assume a class `Number` with a single field `value` that is not `final`.

```
final Number n = new Number(10);
n.value = 12;
```

Figure 3.1: Modifying Constant Reference in Java

In our simple example, we defined a constant `n`, however the object held by `n` is not immutable which allowed us to modify the value despite the variable `n` being a constant.

C++ provides a more expressive form of constant where the previous example is not possible, however C++ allow a different scenario where the value of a constant can be changed. Consider the code snippet in figure 3.2 where one declares a pointer `p` that points to a constant integer. C++ ensures that one is not able to modify the value of `&x` using the pointer `p` however it does not guarantee `x` cannot be modified in a different manner. The value pointed to by `p` can change even though it is defined as a constant.

```
int x = 10;
const int *p = &x;
x = 5;
```

Figure 3.2: Modifying Pointer to Constant in C++

3.1.2 Immutability in Types

The concept of immutability is fundamental and is taught to new programmers early in the curriculum. However, the use of constants are often introduced as a good software engineering practice that reduces careless errors in code. While it is a key use case of constants, they are often used to achieve a lot more.

Recall from section 2.5.3, constants are used to enforce the behaviours of certain language features such as templates in C++. We see the same being done in Java's Lambda expression whereby any local variables referenced in the expression must be a constant.

Immutability is a key idea in our project as it is one of the biggest hurdle in bringing dependent types and refinement types into imperative programming. A good understanding of the difference between constant and immutable proves to be a critical in the development of our language. Furthermore, we see how different imperative languages enforce constancy and immutability in areas where changes to variables are

problematic. We identify that the same idea can be incoporated into our language, enforcing immutable variables in refinement types.

3.2 Closures

In this section we introduce the concept of closures using practical examples.

3.2.1 Introducing Closures

In programming languages, *Closures* are a way of enclosing functions with its surrounding environment, it allows a function to have access to local variables in the outer scope even if these variables have gone out of scope. Closures are present in many languages with first-class functions such as JavaScript, Python and Ruby.

```
function outer_function(x) {
    function inner_function(y) {
      return x + y;
    }
    return inner_function;
}
let closure = outer_function(10);
closure(5); // 15
```

Figure 3.3: Closures in JavaScript

Figure 3.3 is an example of a function closure written in JavaScript. The function `inner_function` is a closure as the variable `x` from the outer scope is accessible within. The line `let closure = outer_function(10)` binds the value 10 to `x`, even after the `outer_function` exits and `x` is out of scope, the closure retains the value of `x`.

3.2.2 Closures in Types

Closures are predominantly associated with functions, however the idea itself is useful more generally. The ability to bind values to variables within a scope can also be applied to types in order to preserve state and achieve a more consistent behaviour when performing type checking.

Recall that one of the difficulty of implementing refinement types in an imperative language is the presence of mutation. We propose that using the idea of closure one can prevent mutation to variables in a refinement type. For example, given

the refinement type $\{v : int \mid v > x\}$, one can close x within the type and make it independent of any changes to a different variable x in the program. This idea is explored in depth further in the project.

3.3 Hybrid Type Checking

In this section we introduce *Hybrid Type Checking* [26], a combination of static and dynamic type checking. We show practical example of hybrid type checking in mainstream languages. Finally we propose a way of type checking refinement types using a hybrid type checking appraoch.

3.3.1 Static and Dynamic Type Checking

Programming languages handle type checking in two main ways, *statically* and *dynamically*. Static type checking verifies the type safety of a program by analysing the program's text. Dynamic type checking verifies the same at runtime, after consideration the execution of statements.

One often associates strong and weak typing with static and dynamic type checking. While it is true that most weakly typed languages employ dynamic type checking, strongly typed languages do not exclusively use static typing. Modern programming languages offer complex features, some of which are hard or impossible to decide statically. In order to accomodate the evolving complexity of programming languages, the two distinct approachs are incorporated together. Gordon Plotkin, et al discussed the idea of dynamically checking a static language in this paper [1]. In practice, modern languages like Java now supports the combination of static and dynamic type checking, e.g. Java's runtime checks for downcasting which is discussed in the next subsection.

Example of Hybrid Type Checking in Java

Downcasting is the act of converting a reference type of a base class to one of its derived class. However, unlike upcasting, downcasting is only possible if the variable of the base class is a value of the derived class being downcasted to. Discovering if a variable can be downcasted relies on dynamic runtime environment and thus it cannot be checked statically however there are certain properties about downcasting that can be checked statically. As such, Java employs a combination of static and dynamic checking to verify the type safety of this advanced feature.

Consider a program with three classes `A`, `B extends A` and `C extends B`. The statements `A x = new B();(C) x` type checks at compile time as there is a possibility it will succeed at runtime, however in this case it fails at runtime as `x` is not a reference of type `C`. Note that in other cases such as `Integer i = 10; (String) i`, type checking fails at compile time because there is no way for the casting to succeed at all. The Java type checker verifies as much as possible statically, requiring only the most complicated type checks to be done at runtime.

3.3.2 Hybrid Type Checking in Refinement Types

Functional languages such as LiquidHaskell have shown the ability of type checking certain refinement types statically at compile time, however with the presence of mutation in imperative languages it turns out to be more difficult. The refinement predicate may contains variables whose value is only available at runtime or could have changed since declaration. This is an area that hybrid type checking proves to be useful.

Interestingly, Cormac Flanagan proposed a similar idea with the same name to verify the correctness of Dependent Types and Refinement Types in a λ-calculus [26]. The paper inspired our idea by proposing a type checker that is "a synthesis of these two approaches that enforces precise interface specifications, via static analysis where possible, but also via dynamic checks where necessary".

Statically, we would not be able to verify that a value satisfy a refinement predicate, however we are able to verify some important properties. Consider the following example, $4 : \{v : int \mid v < x\}$, we cannot check the validity statically as the value of x is not available. However, we can verify that there is a possibility for success. Similarly, we can identify cases that will not succeed at all such as $false : \{v : int \mid v < x\}$ where the value is different from the base type. Interestingly, in certain cases refinement types can be completely checked statically. For example, $x : \{v : int \mid v < x\}$ obviously does not type check, however in order to verify this statically one requires the help of automated theorem provers. In our project we have not consider these cases and left them as future work.

We propose that we can a use hybrid type checking technique to type check refinement types in an imperative language. We first statically verify that a value type checks against the base type, at runtime we verify that the refinement predicate holds. This idea is incorporated into our language and discussed further in the next chapter.

3.4 Summary

In this chapter we introduced several key ideas in programming languages, explained the meaning behind them and how they relates to refinement types. We identified the difference between a constant and immutable object which is key in the study of refinement types in imperative languages. We introduced the concept of closures and showed examples of it in action using JavaScript, we propsed a way of using closures in types that enables refinement types in imperative programming. We explained the difference in static and dynamic type checking, we followed up by observing how these two type checking mechanisms are combined in feature-rich langauges like Java in order to verify the correctness of complicated operations such as downcasting. Finally, we note that refinement types are difficult to verify statically and proposed using hybrid type checking to verify its safety.

Chapter 4

The Simple-R Language

In this chapter we describe Simple-R, a procedural language with support for refinement types. We introduce certain unclear behaviour relating to refinment types and mutation. We extend the language with different variants that address these unclear behaviour using two approaches. Finally, we introduce heap allocated memory and discuss how this feature interacts with refinement types.

4.1 Language Syntax

Types	T	: $int \mid bool \mid \{x : T \mid e\}$
Function Type		: $T_1, T_2, T_3, \ \ldots\ , T_n \longrightarrow T_0$
Function Definition	f	: $(x_0, x_1, \ \ldots\ , x_n) \longrightarrow C; e \mid$ $(x_0, x_1, \ \ldots\ , x_n) \longrightarrow$ **begin** $D; C; e$ **end**
Operators	ops	: $+ \mid - \mid \wedge \mid \vee \mid < \mid >$
Expressions	e	: $vals \mid x \mid e_1\ ops\ e_2 \mid f(e_1 ... e_n)$
Commands	C	: $C_1; C_2 \mid$ **skip** $\mid x\ := e_1 \mid$ **while** e_1 **do** $C_1 \mid$ **if** e_1 **then** C_1 **else** $C_2 \mid$ **begin** $D; C_1$ **end**
Declarations	D	: **var** $x : T = e \mid D_0; D_1$
Variables	$vars$	: $x \in a, b, c ... z$
Values	$vals$	: $\forall v \in \mathbb{Z} \mid \forall v \in \mathbb{B}$
Variable Types Context	Γ	: $vars \mapsto T$
Variable Values Context	σ	: $vars \mapsto vals$

Figure 4.1: Language Syntax for Simple-R

Figure 4.1 is the syntax of Simple-R, the paragraphs below explain the language in details. In contrast to other literature in this area, Simple-R is a *First Order Programming Language* [9] modeled after the WHILE [51] programming language but with functions and refinement types. Rather than building it on the λ-calculus

as one would in a Higher Order Language, we opted for a "C-like" syntax, with concepts such as expressions, commands and top-level functions. In order to achieve simplicity and observe the core behaviour of refinement types, the language is kept at a minimum with only the necessary construct.

Types The Simple-R language has support the following data types.

- The basic integer type *int*
- The basic boolean type *bool*
- The refinement type, written as $\{x : A \mid e\}$. This defines a type that is a refinement of type A that satisfies a boolean expression e.

Functions Function Types are of the form $(T_1, T_2, T_3, \ ... \ , T_n \longrightarrow T_0)$, it takes up to n types and returning a type. Note that functions are not first-class in this language, unlike a higher order language, functions types are its own entity and are distinct from the regular types. This distinction means that functions cannot be passed to other functions, nor can it return a function as a result. One can compare functions in Simple-R to methods in Java. The function body is either a command or declarations followed by command and it must end in an expression that is the return value. Note that the return keyword is omitted. Function returns can be difficult to represent, in many languages functions can be terminated in certain branches using the return keyword. In order to achieve a simple model for functions, we came to the decision to disallow return in the middle of the function, values can only be returned at the very end of the function.

Operators The language supports the following built in infix operators.

- Integer addition using the $+$ operator.
- Integer subtraction using the $-$ operator.
- Integer inequality $<$ as the less than operator.
- Integer inequality $>$ as the greater than operator.
- Logical conjunction using the $\wedge$ operator.
- Logical disjunction using the $\vee$ operator.

Expressions The following expressions make up the language.

- Variables and values represent the most basic form of an expression.

- The six operators which operates on two other expressions e_1 and e_2 and return an expression.
- The function call $f(e_1, ..., e_n)$ invokes function f with arguments $e_1, ..., e_n$ and return the result. Function calls are classified as expression as one can utilise the computed result in other expressions.

Commands Commands are statements that can be executed to perform an action. Simple-R supports the following commands,

- The assignment statement $(:=)$ assigns an expression e_1 to variable x, note x must have already been declared.
- The **if then** and **while do** statements are the standard control flow operations one would expect.
- The **skip** statement is use to indicate an empty expression and does not perform any meaningful action.
- The **begin** $D; C$ **end** command allows one to open a new scope. A begin statement is followed by a declaration D, the declared variables would then be accessible in the scope of command C. Upon reaching **end**, the variables are out of scope and deallocated.

Declarations Declarations are the method one use to declare new construct in the program. We start the language off having the most basic variable declaration **var** $x : T = e$ which declare a new variable x of type T and initialise it the resulting value of expression e. Multiple declarations can be made by concatenating multiple declaration as $D_1; D_2$.

Variables and Values In Simple-R, all variables are allocated locally, we use x to denote a variable. Variables can take values as suggested by the types, all the integer literals $\mathbb{Z}$, booleans $\mathbb{B}$.

Variable Values Context Variables are mapped to values in the program context σ. Each variable is allowed to only appear once in the context. We use $\mathbf{dom}(\sigma)$ to retrieve the domain which represent the set of declared variables. We use the notation $\sigma(x)$ to retrieve the value bound to variable x.

Variable Types Context Variables are also mapped to their types in the type context Γ. Each variable is allowed to only appear once in the context. This context

will be used to type check programs in the next few sections.

4.2 Metavariables

Throughout the text we will be extending the syntax and semantics in order to accommodate more features. We will use the following metavariables and notations throughout our definitions.

- The metavariable *op* to range over all operators.
- The metavariable v to range over all values.
- The uppercase letter T to range over all types.
- The following notations to operate on states and contexts.
  - $\sigma + \{x \mapsto v\}$ adds $\{x \mapsto v\}$ to σ provided initially $x \notin \mathbf{dom}(\sigma)$.
  - $\sigma[x \mapsto v]$ updates σ such that now $\sigma(x) = v$.
  - $\sigma \setminus \{x\}$ to remove x from σ, provided that $x \in \mathbf{dom}(\sigma)$.
- $\langle e, \sigma \rangle \Longrightarrow \langle e', \sigma \rangle$ as the transition relation for expression.
- $\langle C, \sigma \rangle \longrightarrow \langle C', \sigma' \rangle$ and $\langle C, \sigma \rangle \longrightarrow \sigma'$ as the transition relation for commands.
- $\langle f_{body}, \sigma \rangle \dashrightarrow \langle e, \sigma \rangle$ as the transition relation for function body.

4.3 Typing Rules

This section defines the typing rules for the Simple-R language.

We begin by defining the typing judgements. The typing judgements used in Simple-R takes the form of the standard notation.

$$\Gamma \vdash e : T$$
$$\Gamma \vdash_C C : \mathit{void}$$
$$\Gamma \vdash_D D \dashv \Gamma'$$

This first judgement represents that expression e under the context Γ has the type T. For a more consistent syntax, we use a pseudotype *void* to represent the type of a command, the second judgement states that under environment Γ the command C is well-formed. The third states that a well-formed declaration produces the output context Γ'.

Using these judgements we can easily define the first few typing rules for *int*, *bool*, **skip** and variables.

$$\forall n \in \mathbb{Z}.\ \Gamma \vdash n : int$$
$$\forall b \in \mathbb{B}.\ \Gamma \vdash b : bool$$
$$\Gamma \vdash_C \textbf{skip} : void$$
$$x : T, \Gamma \vdash x : T$$

Figure 4.2: Typing Rules for Base Values

The remaining typing rules for the operators, statements and function calls are presented below. First we define the basic rules in figure 4.3, most of the typing rule are straightforward and what one would expect from the type checker. The typing rules for refinement types and functions are covered in the later subsections.

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}\ \text{(Op-Add)} \qquad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 - e_2 : int}\ \text{(Op-Minus)}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 < e_2 : bool}\ \text{(Op-Less)} \qquad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 > e_2 : bool}\ \text{(Op-Greater)}$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \wedge e_2 : bool}\ \text{(Op-And)} \qquad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \vee e_2 : bool}\ \text{(Op-Or)}$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash e : T}{\Gamma \vdash_C x := e : void}\ \text{(Assign)} \qquad \frac{\Gamma \vdash e : bool \quad \Gamma \vdash_C C_1 : void \quad \Gamma \vdash_C C_2 : void}{\Gamma \vdash_C \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 : void}\ \text{(If-Else)}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash_C C : void}{\Gamma \vdash_C \textbf{while } e \textbf{ do } C : void}\ \text{(While-Do)} \qquad \frac{\Gamma \vdash_C C_1 : void \quad \Gamma \vdash_C C_2 : void}{\Gamma \vdash_C C_1; C_2 : void}\ (C\text{-Concat})$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash_D \textbf{var } x : T = e \dashv \Gamma + \{x \mapsto T\}}\ (\text{D-}Vars)$$

$$\frac{\Gamma \vdash_D D_1 \dashv \Gamma' \quad \Gamma' \vdash_D D_2 \dashv \Gamma''}{\Gamma \vdash_D D_1; D_2 \dashv \Gamma''}\ (D\text{-Concat})$$

$$\frac{\Gamma \vdash D \dashv \Gamma' \quad \Gamma' \vdash_C C : void}{\Gamma \vdash_C \textbf{begin } D; C \textbf{ end} : void}\ \text{(Begin)}$$

Figure 4.3: Typing Rules for Simple-R

Functions

In order to type check a function we require access to the function signature, we define a mapping $fts : f \mapsto T_1, T_2 ... T_n \longrightarrow T_0$ that allows us to retrieve the signature of a function. We first type check the function definition using the following new judgements.

$$\Gamma \vdash_f (x_1, x_2, \ ... \ , x_n) \longrightarrow C; e$$
$$\Gamma \vdash_f (x_1, x_2, \ ... \ , x_n) \longrightarrow \textbf{begin } D; C; e \textbf{ end}$$

Using the judgements, we check that the function definition matches the function signature using the rules in figure 4.4.

$$\frac{\begin{array}{c} fts((x_1, x_2, \ ... \ , x_n) \longrightarrow C; e) = T_1, \ ... \ , T_n \longrightarrow T_0 \\ \Gamma \vdash x_1 : T_1 \qquad \Gamma \vdash x_2 : T_2 \qquad ... \qquad \Gamma \vdash x_n : T_n \\ \Gamma \vdash_C C : void \qquad\qquad \Gamma \vdash e : T_0 \end{array}}{\Gamma \vdash_f (x_1, x_2, \ ... \ , x_n) \longrightarrow C; e} \text{ (Func-Def-}C; e\text{)}$$

$$\frac{\begin{array}{c} fts((x_1, x_2, \ ... \ , x_n) \longrightarrow \textbf{begin } D; C; e \textbf{ end}) = T_1, \ ... \ , T_n \longrightarrow T_0 \\ \Gamma \vdash x_1 : T_1 \qquad \Gamma \vdash x_2 : T_2 \qquad ... \qquad \Gamma \vdash x_n : T_n \\ \Gamma \vdash_D D \dashv \Gamma' \qquad \Gamma' \vdash_C C : void \qquad \Gamma' \vdash e : T_0 \end{array}}{\Gamma \vdash_f (x_1, x_2, \ ... \ , x_n) \longrightarrow \textbf{begin } D; C; e \textbf{ end}} \text{ (Func-Def-}D; C; e\text{)}$$

Figure 4.4: Typing Rules for Function Definitions

We use first order logic to check that a n-nary function that takes the parameters x_1 to x_n has the same types that is specified in the signature. Furthermore, we check that the function body is well formed and the returned expression matches the return type in the signature.

For a function call, we use the typing rule in figure 4.5. If $f(e_1 ... e_n)$ is a n-nary function of type T_1 to T_n, then the arguments e_1 to e_n must be of same type. Evaluation of the function will then satisfy the return type T_0 in the signature.

$$\frac{\begin{array}{c} fts(f) = T_1, T_2, \ ... \ , T_n \longrightarrow T_0 \\ \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad ... \quad \Gamma \vdash e_n : T_n \end{array}}{\Gamma \vdash f(e_1, \ ... \ , e_n) : T_0} \text{ (Func-Call)}$$

Figure 4.5: Typing Rules for Function Call

Refinement Types

Finally we define typing rules for a refinement types, a refinement type is represented in our system as $\{v : T \mid e\}$ and is constructed by a base type T and a refinement

predicate e which may contain the variable v and any free variables bound in an outer scope in the program. A refinement type represent all values v of T where the expression $e[u/v]$ holds.

Recall that in section 3.3.2 that we stated that the refinement predicate in general is undecideable at compile time and one requires dynamic checks for refinement types. However, we proposed that there are certain properties that can be checked statically such as a value must at least type checks against the base type for a chance of success.

We adopted a type checking approach that validates as much as possible statically and defers complicated type checking until runtime. In this section we focus on static refinement type checking, we will discuss runtime type checking later in section 4.4.3.

We first define the typing rule for refinement types using the idea of *universes* [28] in type theory, using the judgement form $\Gamma \vdash T : Type$ to represent that T is a valid type. We begin by defining the formation rule for a refinement type.

$$\frac{\Gamma \vdash v : T \qquad \Gamma, v : T \vdash e : bool}{\Gamma \vdash \{v : T \mid e\} : Type} \text{ (Refine-Type-Form)}$$

Figure 4.6: Refinement Types Formation Rule

We then define an *unsound* typing judgement to capture this behaviour statically, soundness will be regained further on using dynamic checks.

$$\frac{\Gamma \vdash e_1 : T}{\Gamma \vdash e_1 : \{v : T \mid e\}} \text{ (Refine-Base-US)}$$

Figure 4.7: Static Refinement Type-Checking Rule

Additionally, we define the typing rule for a subtypes, we express T_1 is a subtype of T_2 as $T_1 \preceq T_2$. The subtype rules defined in figure 4.8 allow us to perform perform basic polymorphism. (Subtype-Base 1 and 2) defines the basic definition of subtype. The first states that a refinement type of T is trivially a subtype of T and the second states that a refinement type T_1 is a subtype of another refinement type T_2 if the set of values in T_1 is a subset of or equal the values in T_2. (Subtype Trans) defines the transitivity of subtype and (Subtype Reflex) defines the reflexivity.

One should be able to use a value of type T_1 in a context expecting type T_2 provided T_1 is a subtype of T_2. For example, an expression e of type $EvenNat$ when passed to a context expecting a type Nat should be valid. We enable this behaviour using the (Subtype) rule.

$$\Gamma \vdash \{v : T \mid e\} \preceq T \text{ (Subtype-Base 1)}$$

$$\frac{\Gamma \vdash \forall v_1 : T.\ e_1 \Rightarrow e_2[v_1/v_2]}{\Gamma \vdash \{v_1 : T_1 \mid e_1\} \preceq \{v_2 : T_1 \mid e_2\}} \text{ (Subtype-Base 2)}$$

$$\frac{\Gamma \vdash T_1 \preceq T_2 \quad \Gamma \vdash T_1 \preceq T_3}{\Gamma \vdash T_1 \preceq T_2} \text{ (Subtype-Trans)}$$

$$\Gamma \vdash T_1 \preceq T_1 \text{ (Subtype-Reflex)}$$

$$\frac{\Gamma \vdash T_1 \preceq T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash e : T_2} \text{ (Subtype)}$$

Figure 4.8: Refinement Types Subtyping Rules

In this section we covered the typing rules for Simple-R and how refinement types are verified statically. In the next section we discuss the dynamic behaviours.

4.4 Operational Semantics

In this section we define the *Operational Semantics* [42] for Simple-R using the *Small-Step Structural Operational Semantics* [39] (SOS) rules. Figure 4.9 defines the reduction rules for expressions. The rules for function calls will be covered in section 4.4.2.

$$\frac{v = v_1 + v_2}{\langle v_1 + v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-Add)} \qquad \frac{v = v_1 - v_2}{\langle v_1 - v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-Minus)}$$

$$\frac{v = v_1 < v_2}{\langle v_1 < v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-Less)} \qquad \frac{v = v_1 > v_2}{\langle v_1 > v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-Greater)}$$

$$\frac{v = v_1 \wedge v_2}{\langle v_1 \wedge v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-AND)} \qquad \frac{v = v_1 \vee v_2}{\langle v_1 \vee v_2, \sigma\rangle \Longrightarrow \langle v, \sigma\rangle} \text{ (Op-OR)}$$

$$\frac{\langle e_1, \sigma\rangle \Longrightarrow \langle e_1', \sigma\rangle}{\langle e_1\ op\ e_2, \sigma\rangle \Longrightarrow \langle e_1'\ op\ e_2, \sigma\rangle} \text{ (Op-}E\text{ 1)} \qquad \frac{\langle e_2, \sigma\rangle \Longrightarrow \langle e_2', \sigma\rangle}{\langle v\ op\ e_2, \sigma\rangle \Longrightarrow \langle v\ op\ e_2', \sigma\rangle} \text{ (Op-}E\text{ 2)}$$

$$\frac{x \in \mathbf{dom}(\sigma)}{\langle x, \sigma\rangle \Longrightarrow \langle \sigma(x), \sigma\rangle} \text{ (Vars)}$$

Figure 4.9: Operational Semantics for Expression

Figure 4.10 defines the operational semantics for the basic commands in the language. The semantics for **begin** $D; C$ **end** is complicated compared to the other

commands and thus will be covered with more details in section 4.4.1.

$$\langle \textbf{skip}, \sigma \rangle \longrightarrow \sigma \text{ (Skip)} \quad \frac{\langle C_1, \sigma \rangle \longrightarrow \sigma'}{\langle C_1; C_2, \sigma \rangle \longrightarrow \langle C_2, \sigma' \rangle} \text{ (Cmd-Concat)}$$

$$\langle \textbf{if } \mathit{true} \textbf{ then } C_1 \textbf{ else } C_2, \sigma \rangle \longrightarrow \langle C_1, \sigma \rangle \text{ (If-True)}$$

$$\langle \textbf{if } \mathit{false} \textbf{ then } C_1 \textbf{ else } C_2, \sigma \rangle \longrightarrow \langle C_2, \sigma \rangle \text{ (If-False)}$$

$$\frac{\langle e_1, \sigma \rangle \Longrightarrow \langle e_1', \sigma \rangle}{\langle \textbf{if } e_1 \textbf{ then } C_1 \textbf{ else } C_2, \sigma \rangle \longrightarrow \langle \textbf{if } e_1' \textbf{ then } C_1 \textbf{ else } C_2, \sigma \rangle} \text{ (IF-}E\text{)}$$

$$\langle \textbf{while } e_1 \textbf{ do } C_1, \sigma \rangle \longrightarrow \langle \textbf{if } e_1 \textbf{ then } C_1; (\textbf{while } e_1 \textbf{ do } C_1) \textbf{ else skip}, \sigma \rangle \text{ (While)}$$

$$\frac{x \in \textbf{dom}(\sigma)}{\langle x := v, \sigma \rangle \longrightarrow \sigma[x \mapsto v]} \text{ (Assign-}V\text{)} \quad \frac{\langle e, \sigma \rangle \Longrightarrow \langle e', \sigma \rangle}{\langle x := e, \sigma \rangle \longrightarrow \langle x := e', \sigma \rangle} \text{ (Assign-}E\text{)}$$

Figure 4.10: Operational Semantics for Commands

4.4.1 Scoping

Scoping has an important role in programming languages, variables can be defined in scopes and are unavailable once outside. In order to help us formalise the notion of scopes we define a new notation, $\Longrightarrow^*$ and $\longrightarrow^*$ as a multi-step evaluation. The evaluation rule $\langle e, \sigma \rangle \Longrightarrow^* \langle v, \sigma' \rangle$ simply evaluate the expression e one or more times until it evaluates to a value v. The same applies for a multistep evaluation for commands, $\langle C, \sigma \rangle \longrightarrow^* \sigma'$ execute the command C until there is no evaluation rules to apply.

The multi-step evaluation provides us to easily express scoping rules, the alternative would be to model an explicit call stack, which complicates the formulas and syntax.

Using a multi step evaluation, We can easily define the scoping rules as shown in figure 4.11. The (Begin-V) rule opens a new scope σ' which assigns x the value v, this scope allows the command C_1 to be evaluated using a multi step evaluation. Notice that the resulting state uses the $\setminus$ operator which removes all the variables in the set $\{x\}$ from the domain of σ.

This is necessary as it is possible to modify the range of σ through assignments. Upon exiting a scope the alteration to the variables should still be visible, the only difference is that the variable x is no longer in the state, i.e. has ran out of scope.

$$\frac{\sigma' = \sigma + \{x \mapsto v\} \quad x \notin \mathbf{dom}(\sigma) \quad \langle C_1, \sigma' \rangle \longrightarrow^* \sigma''}{\langle \mathbf{begin\ var}\ x : T = v; C_1\ \mathbf{end}, \sigma \rangle \longrightarrow \sigma'' \setminus \{x\}}\ \text{(Begin-}V\text{)}$$

$$\frac{\langle e, \sigma \rangle \Longrightarrow \langle e', \sigma \rangle}{\langle \mathbf{begin\ var}\ x : T = e; C_1\ \mathbf{end}, \sigma \rangle \longrightarrow \langle \mathbf{begin\ var}\ x : T = e'; C_1\ \mathbf{end}, \sigma \rangle}\ \text{(Begin-}E\text{)}$$

$$\frac{\sigma' = \sigma + \{x \mapsto v\} \quad x \notin \mathbf{dom}(\sigma) \quad \langle \mathbf{begin}\ D_1; C_1\ \mathbf{end}, \sigma' \rangle \longrightarrow^* \sigma''}{\langle \mathbf{begin\ var}\ x : T = v; D_1; C_1\ \mathbf{end}, \sigma \rangle \longrightarrow \sigma'' \setminus \{x\}}\ \text{(Begin-}D\text{-Concat-}V\text{)}$$

$$\frac{\langle e, \sigma \rangle \Longrightarrow \langle e', \sigma \rangle}{\langle \mathbf{begin\ var}\ x : T = e; D; C_1\ \mathbf{end}, \sigma \rangle \longrightarrow \langle \mathbf{begin\ var}\ x : T = e'; D; C_1\ \mathbf{end}, \sigma \rangle}\ \text{(Begin-}D\text{-Concat-}E\text{)}$$

Figure 4.11: Scoping Rules in Simple-R

Concatenated declarations are handled by evaluating the declarations in the order they are written. The rules (Begin-D-Concat-V) and (Begin-D-Concat-E) formalise the behaviour. To minimise space, in future text when we introduce new form of *Declarations*, we will only formalise the rules where only a single declaration is present. The corresponding concetenation rules, which can be derived easily using similar semantics to the ones used in in figure 4.11, are omitted.

4.4.2 Function Calls

Functions in imperative language often operate in their own scope, they do not have the access to the local variables out of their scope however one can pass data using arguments. Functions in Simple-R are call by values, changing the value of an argument in the function does not change the value in the caller's scope. Our multi-step evaluation rule defined previously helps us evaluate functions in their own scope.

In order to define the operational semantics of a function we will need to specify the function body. We define a mapping $fns : f \mapsto ((x_1, x_2, \ \ldots\ , x_n), f_{body})$, f_{body} can either be $C; e$ or **begin** $D; C; e$ **end**, using this mapping we can retrieve the body of a function. Since function definitions cannot be changed at runtime, this does not need to be part of the configuration and can be consider as a static context.

We first define the rules for the simple case where the function body is simply $C; e$, the rules for **begin** $D; C; e$ **end** will be defined later in the section.

$$\frac{\begin{array}{c} \mathit{fns}(f) = ((x_1,\ \ldots\ , x_n), C; e) \qquad \langle C, \sigma' \rangle \longrightarrow^* \sigma'' \\ \sigma' = \{x_1 \mapsto v_1,\ \ldots\ , x_n \mapsto v_n\} \quad \langle e, \sigma'' \rangle \Longrightarrow^* \langle v, \sigma'' \rangle \end{array}}{\langle f(v_1,\ \ldots\ , v_n), \sigma \rangle \Longrightarrow \langle v, \sigma \rangle} \text{ (Func-}V\text{)}$$

$$\frac{\langle e_k, \sigma \rangle \Longrightarrow \langle e_k', \sigma' \rangle}{\langle f(v_1,\ \ldots\ , v_{k-1}, e_k,\ \ldots\ , e_n), \sigma \rangle \Longrightarrow \langle f(v_1,\ \ldots\ , v_{k-1}, e_k',\ \ldots\ , e_n), \sigma \rangle} \text{ (Func-}E\text{)}$$

Figure 4.12: Operational Semantics for Functions

Figure 4.12 shows the basic evaluation rules for functions, the first rule (Func-V) creates a new state where only the argument values are available. We then evaluates the function body until it evaluates to a value which is then returned to the caller's context. Unlike scopes, functions only update its local state which is dellalocated upon return.

The second rule (Func-E) evaluates the arguments passed into the function in order to get their values. Our language defines a strict behaviour that all arguments are evaluated in the order they are written in, i.e. from left to right and all arguments are call-by-value. This design parallels that of Java and ML but not like C where arguments can be evaluated out of order or Haskell where lazy evaluation is applied.

In certain cases one might declare new local variables in functions and subsequently use these local variables in the return expression. Figure 4.13 defines the rules, recall the notation $\dashrightarrow$ is the transition relation for function body **begin** $D; C; e$ **end**.

$$\frac{\langle C, \sigma \rangle \longrightarrow^* \sigma' \qquad \langle e, \sigma' \rangle \Longrightarrow^* v}{\langle \textbf{begin } C; e \textbf{ end}, \sigma \rangle \dashrightarrow \langle v, \sigma' \rangle} \text{ (Func-Body)}$$

$$\frac{\begin{array}{c} \sigma' = \sigma + \{x \mapsto v_0\} \qquad x \notin \textbf{dom}(\sigma) \\ \langle \textbf{begin } C; e \textbf{ end}, \sigma' \rangle \dashrightarrow^* \langle v, \sigma'' \rangle \end{array}}{\langle \textbf{begin } x : T = v_0; C; e \textbf{ end}, \sigma \rangle \dashrightarrow \langle v, \sigma'' \rangle} \text{ (Func-Body-}D\text{)}$$

$$\frac{\begin{array}{c} \sigma' = \sigma + \{x \mapsto v_0\} \qquad x \notin \textbf{dom}(\sigma) \\ \langle \textbf{begin } D; C; e \textbf{ end}, \sigma' \rangle \dashrightarrow^* \langle v, \sigma'' \rangle \end{array}}{\langle \textbf{begin } x : T = v_0; D; C; e \textbf{ end}, \sigma \rangle \dashrightarrow \langle v, \sigma' \rangle} \text{ (Func-Body-}D\text{-Concat)}$$

$$\frac{\begin{array}{c} \mathit{fns}(f) = ((x_1,\ \ldots\ , x_n), \textbf{begin } D; C; e \textbf{ end}) \\ \sigma' = \{x_1 \mapsto v_1,\ \ldots\ , x_n \mapsto v_n\} \quad \langle \textbf{begin } D; C; e \textbf{ end}, \sigma' \rangle \dashrightarrow^* \langle v, \sigma'' \rangle \end{array}}{\langle f(v_1,\ \ldots\ , v_n), \sigma \rangle \Longrightarrow \langle v, \sigma \rangle} \text{ (Func-}D\text{)}$$

Figure 4.13: Semantics for Declarations in Functions

The first three rules in the figure behave similar to the operational semantics for **begin end**, however, instead of simply returning the state, we return the value

of the return expression. Note that in order to minimise space, we omited the rule where an expression is used in the declaration, however one can easily derived the rule using semantics similar to (Begin-E) and (Begin-D-Concat-E) from figure 4.11. Finally, the rule (Func-D) ties everything together by specifying the rules for evaluating a function where the body contains new variable declarations.

4.4.3 Runtime Refinement Type Checking

A declaration of a variable that is of refinement types requires specific evaluation rules. When one declares a variable of a refinement type, the language has to ensure that the refinement predicate evaluates to true. While we have defined a statically typed language, even in a statically typed language, there are cases when the type of some data cannot be determined statically as discussed in section 3.3.

In section 4.3 we demonstrated how some basic properties of refinement types are checked statically in Simple-R using the unsound typing judgement (Refine-Base-US), now we show how these refinement types are validated dynamically in order to regain soundness. We define a special form of typing judgement to type check refinement types dynamically.

$$\sigma, \Gamma \vdash_R e_1 : \{v : T \mid e\}$$

Figure 4.14: Typing Judgement for Refinement Types

Figure 4.14 shows a typing judgement for refinement types which extends $\Gamma \vdash e : T$ with the ability to access the runtime state σ. We can then perform a type check on refinement type using the rule defined below.

$$\frac{\Gamma \vdash e_1 : T \qquad \langle e[e_1/v], \sigma\rangle \Longrightarrow^* \langle \mathit{true}, \sigma\rangle}{\sigma, \Gamma \vdash_R e_1 : \{v : T \mid e\}} \text{ (Runtime-Refine)}$$

Figure 4.15: Type Checking Refinement Types

During runtime, the typing judgement in figure 4.15 asserts that an expression e_1 is a valid refinement type if it evaluates to the base type T and the refinement predicate $e[e_1/v]$ evaluates to true under state σ.

Our type check rule looks to capture the behaviour of refinement type however it turns out the current definition is underspecified in the presence of mutation. In the next section we put forward certain ambiguities with the design and discuss different choices one can make to address them.

4.5 Refinement Types and Variables

4.5.1 Ambiguity of Refinement Types involving Mutation

Programming languages have a wide range of features, often the team behind the language have to put extensive thought into the design decisions. As there are often multiple ways of handling a certain feature, rarely is there a definite answer to which is the most optimal, they are usually a matter of human judgement or aesthetics.

This problem is known as *feature interactions* [6], it describes the problem where in some situations multiple features would modify the behaviours of each others. The same applies to refinement types, which can be seen as a feature. For instance, the inclusion of refinement types would modify the behaviour of assignments and vice versa. This section covers the different ways of handling refinement types and how they interact with other features. Consider the ambiguity of using variables in refinement types in figure 4.16.

begin var $y \ : \ int = 10;$
    begin typedef $BoundedIntY = \{\upsilon : Int \mid \upsilon < y\};$
        begin var $x \ : \ BoundedIntY = 6; y := 5$ **end**
    end
end

Figure 4.16: Refinement Type Ambiguity involving Mutation

The simple program shown in figure 4.16 declares a bounded integer type $BoundedIntY$ whose value cannot be greater than another integer y, a free variable in the program. A variable x is declared to be of type $BoundedIntY$ with the appropriate value. Later in the program, the value of y changes. resulting to x now being ill-typed. The behaviour is now undefined, should x still be well-typed, or is the behaviour invalid?

4.5.2 Type Declaration

First we need to introduce a new declaration statement for types, we use a **typedef** statement to create a new refinement type with a name x. It is important to note that a **typedef** can only declare a new refinement type, one is not able to use **typedef** to provide an alias for a base type, i.e. the type definition **typedef** $newint = int$ is forbidden, however one is able to achieve the same effect by declaring **typedef** $newint = \{\upsilon : int \mid true\}$.

Figure 4.17 extends the syntax of Simple-R to introduce the new form of declaration.

Declaration	D	$: ... \mid \textbf{typedef}\ x = \{v : T \mid e\}$
Variable Types Context	Γ	$: vars \mapsto T \mid Type$
Type Name Context	τ	$: vars \mapsto \{v : T \mid e\}$

Figure 4.17: Type Name Context Extension for Simple-R

In our syntax, we have a context that maps $vars$ to $vals$, we now need to add a similar mapping for types which we will use τ to represent, we call this the Type Name Context. In addition, we modified our Variable Types Context so that we can identify if a certain variable is a value or a type.

The typing judgement has to be updated to accommodate type checking using a declared type. All typing judgements are updated from $\Gamma \vdash e : T$ to $\Gamma, \tau \vdash e : T$ and the typing rule for typedef is defined below.

$$\frac{\Gamma \vdash \{v : T \mid e\} : Type}{\Gamma, \tau \vdash_D \textbf{typedef}\ x = \{v : T \mid e\} \dashv \Gamma + \{x \mapsto Type\}, \tau + \{x \mapsto \{v : T \mid e\}\}} \text{ (Typedef)}$$

We now have to define the rules for type checking an expression against a declared type. As a declared type is only ever a refinement type, we have established that they cannot be type checked statically. However, we must still validate the basic properties using the *Refine-Base-US* rule.

$$\frac{\Gamma, \tau \vdash y : Type \qquad \Gamma, \tau \vdash e : \tau(y)}{\Gamma, \tau \vdash e : y} \text{ (Refine-Base-Vars-US)}$$

During runtime, the refinement predicate has to be validated to regain soundness using the *Refine-Runtime* rule.

$$\frac{\Gamma \vdash x : Type \qquad \Gamma \vdash_R e_1 : \tau(x)}{\sigma, \Gamma \vdash_R e_1 : x} \text{ (Runtime-Refine-Vars)}$$

In our operational semantics, our configuration is similarly updated from $\langle C, \sigma \rangle$ to $\langle C, \sigma, \tau \rangle$ in order to incorporate this new feature. All previous operational semantics should be updated consistently.

$$\frac{\langle C_1, \sigma, \tau' \rangle \longrightarrow \langle \sigma', \tau' \rangle \qquad x \notin \textbf{dom}(\sigma) \qquad x \notin \textbf{dom}(\tau) \qquad \tau' = \tau + \{x \mapsto \{v : T \mid e\}\}}{\langle \textbf{begin typedef}\ x = \{v : T \mid e\}; C_1\ \textbf{end}, \sigma, \tau \rangle \longrightarrow \langle \sigma', \tau \rangle} \text{ (Type-}D\text{)}$$

Note that unlike the rule for variable declaration, we did not have to remove the type name x from the context after evaluation. This is because unlike variables, types

cannot change after declaration. The new type simply goes out of scope, hence the new type context will simply be our initial one.

The operational semantics above simply add a new type definition with name x into the Type Name context τ. Importantly, the variable x must not have already been defined as a value variable in σ and a type of the same name can not already be defined. Logically it is equivalent to the semantics for variable declaration.

4.5.3 Complete Immutability

The main issue one encounters when associating refinement types with imperative languages is the presence of mutation and assignments. The obvious way to work around this problem is to remove the ability to make changes to the state of the program, all variables declared are final.

Using this approach does definitely resolves the ambiguity but the usability of the language is poor. Imperative programming relies on state changes, forbidding that goes against the paradigm. However, if we strategically enforce immutability we can resolve the ambiguity while preserving a flexible imperative style language. In the next few sections we discuss two approaches that improve the specification of our type checker through restricting changes to variables.

4.5.4 Immutability in Refinement Types

In order to avoid feature interaction between refinement types and assignments, we observe that it is suffice to require all free variables appearing in a refinement predicate to be immutable. This approach improves flexibility of our language as one will still be able to have mutation in context irrelevant to refinement types. In this section we extends Simple-R with this functionality and dubbed the new variant Simple-R_I.

In order to capture the notation of immutable types, the set of variables $vars$ is partitioned into two different distinct categories. Mutable and immutable, we define a predicate $is\_immutable$ that specify if a given variable is immutable. $is\_immutable(x)$ holds if and only if x is immutable.

$$\textit{Declaration} \quad D \quad : ... \mid \textbf{const}\ x : T = e_1$$

Figure 4.18: Immutable Variables Declaration for Simple-R_I

One can declare an immutable variable similar to defining a mutable variable, using

the **const** $x : T = e$ statement. We establish the following rules for immutable variables.

The typing judgement is in analogous to that of variable declaration using **var**.

$$\frac{\Gamma, \tau \vdash e_1 : T_1}{\Gamma, \tau \vdash_D \textbf{const } x : T_1 = e_1 \dashv \Gamma + \{x \mapsto T_1\}, \tau} \text{ (Const-}D\text{)}$$

The operational semantics in figure 4.19 describes the process of declaring an immutable variable, the semantics are similar to that of mutable variables. The only new addition here is extending the assignment statement to only accept the latter.

$$\frac{\langle C_1, \sigma', \tau\rangle \longrightarrow^* \langle \sigma'', \tau'\rangle \quad \sigma' = \sigma + \{x \mapsto v\} \quad x \notin \textbf{dom}(\sigma)}{\langle \textbf{begin const } x : T = v; C_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \sigma'' \setminus \{x\}, \tau'\rangle} \text{ (Begin-Const)}$$

$$\frac{\langle e_1, \sigma, \tau\rangle \Longrightarrow \langle e_1', \sigma, \tau\rangle}{\langle \textbf{begin const } x : T = e_1; C_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \textbf{begin const } x : T = e_1'; C_1 \textbf{ end}, \sigma, \tau\rangle} \text{ (Begin-Const-E)}$$

$$\frac{x \in \textbf{dom}(\sigma) \quad \neg \mathit{is\_immutable}(x)}{\langle x := v, \sigma, \tau\rangle \longrightarrow \langle \sigma[x \mapsto v], \tau\rangle} \text{ (Assign-Const)}$$

Figure 4.19: Immutable Variables Rules for Simple-R_I

Once we define the ability to define immutable variables, we can utilise it in our type definition. First we refine our specification for **typedef** with regards to refinement types using the evaluation rules.

We require a new notation to capture all the free variables that appear in an expression e, we use the notation $\mathit{free\_vars}(e)$ to give the set of all variables that appear in an expression e.

Notice that in our syntax the refinement predicate is simply a boolean expression, this means that it is not possible for the refinement predicate to declare any new variables within it. So it is easy to see that for a refinement predicate e, we require $\mathit{free\_vars}(e) \subseteq \textbf{dom}(\sigma)$.

Using the new notation we can easily verify that all free variables in a refinement predicate *is_immutable* using the rules defined in figure 4.20. The same applies when one declares a variable of type $\{v : T \mid e\}$ without using the **typedef** command.

We have now address the ambiguity brought up using this semantics, the program in figure 4.16 is no longer a valid program as y is a mutable variable and is forbidden from appearing in a refinement type. One would have to rewrite the declaration of y using **const** which forbids it from being assigned to after initialisation. It is easy to see by restricting free variables in the refinement predicate we ensure that

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\sigma) \qquad x \notin \mathbf{dom}(\tau) \\ \forall \alpha \in \mathit{free\_vars}(e).\ \mathit{is\_immutable}(\alpha) \\ \langle C_1, \sigma, \tau + \{x \mapsto \{v : T \mid e\}\}\rangle \longrightarrow \langle \sigma', \tau\rangle \end{array}}{\langle \textbf{begin typedef } x = \{v : T \mid e\}; C_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \sigma', \tau\rangle} \text{ (Refine-Typdef-Immu)}$$

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\sigma) \qquad x \notin \mathbf{dom}(\tau) \\ \forall \alpha \in \mathit{free\_vars}(e).\ \mathit{is\_immutable}(\alpha) \\ \langle C_1, \sigma + \{x \mapsto v_1\}, \tau\rangle \longrightarrow \langle \sigma', \tau\rangle \end{array}}{\langle \textbf{begin var } x : \{v : T \mid e\} = v_1; C_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \sigma' \setminus \{x\}, \tau\rangle} \text{ (Refine-Immu)}$$

Figure 4.20: Operational Semantics for Typedef in Simple-R_I

the expression stays the same throughout its existence, avoiding any complications regarding mutation.

This is a simple approach that deals with the ambiguity of having mutation and refinement types, note that C++ utilises the same strategy for templates arguments. We could potentially adapt a clever approach used by the Java programming language to overcome variable mutation in Anonymous classes. Java requires that any variables accessed by an anonymous class that is part of its enclosing scope to be final or *effectively final* [21]. An effectively final variable is a variable that is not declared immutable but its value is not changed after declaration. While this does improves some flexibity as one do not need to use the **const** statement for declaration, it behaves the same for all intents and purposes. For this reason we consider this approach as unimportant and considered a more general solution.

The limitation of this approach is that one has to strategically declare variables, in some cases one might not be able to declare an immutable variable but still wants to use it in a refinement type.

4.5.5 Type Closure

If we further analyse our design, we notice that our main concern is for the refinement predicate to stay static, any changes in the program context should not change its definition. In this section, we propose a design based on closures which allows us to ensure the values of variables in a refinement predicate remain constant without the need of declaring constant variables. We define a new variant of Simple-R named Simple-R_C that extends the former and implements this technique to handle refinement types.

Recall in section 3.2 we introduced the idea of closures, a closure is simply a function enclosed with an environment. Closures have access to variables from the calling

context, even if the original variable goes out of scope. A value is bound to the variable in the closure such that changes does not reflect within. In order to make free variables in a predicate independent, we designed a similar approach where we *snapshot* variables, i.e. bind free variables to a value. We name this approach of closing variables used in types as *Type Closure*.

Taking the example program in figure 4.16 if the value of variable y in the refinement predicate is snapshotted at the time of creation, then any changes to the variable y will not affect the type definition and the program will stay well typed.

Semantically, closures can be modeled as a data structure mapping functions to environment as demonstrated in the extended lambda calculus proposed by Gerald J. Sussman and Guy L. Steele Jr in this paper [48]. We adapted a similar approach, however instead of a function mapping to an environment, we have a type mapping to an environment. We modify the definition of τ to hold this information.

$$\textit{Type Name Context} \quad \tau \quad : vars \mapsto (\{v : T \mid e\}, \sigma, \tau)$$

The Type Name Context now associate a refinement type with its evaluation context. The said context is simply the σ, τ at the time of the refinement type creation. In figure 4.21, we define σ_1 and τ_1 as a copy of the program state σ and τ and attach this to the type.

$$\frac{\begin{array}{c}\sigma_1 = \sigma \qquad\qquad \tau_1 = \tau \\ \langle C_1, \sigma, \tau + \{y \mapsto (\{v : T \mid e\}\}, \sigma_1, \tau_1)\rangle \longrightarrow^* \langle \sigma', \tau\rangle\end{array}}{\langle \textbf{begin typedef } y = \{v : T \mid e\}; C_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \sigma', \tau\rangle} \text{ (Refine-Typedef-Closure)}$$

Figure 4.21: Operational Semantics for Typedef in Simple-R_C

In order to type check refinement types with closures, we update our typing judgement for refinement types. Notice that we only require type closures for named types; if one declares a refinement type without a name i.e. **var** $x : \{v : int \mid v < x\}$, then the rule (Refine-Runtime) defined in section 4.4.3 that evaluates the predicate using the current environment will hold.

Our new typing rule now retrieves the associated context and evaluates appropriately, the new rule is defined in figure 4.22.

$$\frac{\begin{array}{c}\langle e[e_1/v], \sigma_1, \tau_1\rangle \Longrightarrow^* \langle true, \sigma_1, \tau_1\rangle \\ \Gamma, \sigma, \tau \vdash x : Type \quad (\{v : T \mid e\}, \sigma_1, \tau_1) = \tau(x)\end{array}}{\Gamma, \sigma, \tau \vdash_R e_1 : x} \text{ (Refine-Runtime-Closure)}$$

Figure 4.22: Refinement Type Checking in Simple-R_C

The typing rule behaves similarly to our previous rule, the only difference here is the context being evaluated in. When type checking against a refinement type, the evaluation context σ_1, τ_1 is retrieved from τ, the predicate $e[e_1/\upsilon]$ is then evaluated under these context.

It address the ambiguity in figure 4.16 by specifying a concrete behaviour. $BoundedIntY$ being bound to 10 throughout, even when the free variable y is reassigned to another value.

Unlike C++, this form of semantics allows one to utilise non-constant expression in type definition. This is achieved by using the idea of closures and snapshotting the value of variables at the time of type definition.

4.5.6 Summary

In this section we gave a scenario where our refinement type system is underspecified in the presence of mutation. We proposed two different approaches to complete our specification, the first utilises constants and immutable variables to disallow modification of free variables in a refinement predicate. The second utilises closures and value bindings to ensure consistency.

While the proposed solutions are effective in dealing with mutation on local variables. As Simple-R starts incorporating more features, our design becomes underspecified yet again. In the next section, we analyse the behaviour of our system in the presence of dynamic memory allocation and aliasing.

4.6 Refinement Types and Pointers

We previously proposed a set of rules that specify behaviour of refinement types with regards to local variables and we learn that we can keep behaviour consistent if we restrict changes to the variable. However, our solution does not accommodate reference variables and pointers. If a refinement type relies on pointers in the refinement predicate then the simple act of declaring the pointers immutable is not enough as pointer values can be altered indirectly.

In this section we first extend our syntax with the notion of dynamically allocated memory. Later we study the feature interaction between pointers and refinement types in order to identify a consistent behaviour.

4.6.1 Introducing Heap Allocated Memory

In programming languages, there are three different types of memory allocation, static, stack and heaps [47]. While we have not explicitly discussed our memory management model, it is obvious that Simple-R operates on a stack based model. It is apparent in the behaviour of local variables which are inaccessible once out of scope as discussed in our scoping rules. We introduce a new variant of Simple-R called Simple-R^* that supports dynamic memory allocation.

Syntax Extension

First we extend our language syntax to accommodate these new features, we define the new syntax in figure 4.23. For simplicity and safety, we have omitted the obligation to use manual memory management and we assume that *Automatic Garbage Collection* [24] is present in the language.

Types	T	$: ... \mid T\ ref$
Expressions	e	$: ... \mid !e_1 \mid \mathbf{new}(T, e_1)$
Commands	C	$: ... \mid !e_1 := e_2$
Values	$vals$	$: ... \mid h \in \{h_1, h_2, h_3 ...\}$
Heap	Δ	$: h \mapsto vals$

Figure 4.23: Heap Memory in Simple-R^*

Types We introduce a new type, T *ref* as a reference type that behaves like a pointer in C/C++, like pointers, references are typed and the value of a reference is a heap address.

Expressions The following expressions are introduced into the language, unlike C/C++ where one is allowed to take the address of a local variable, in our system one may only create a pointer through the use of **new** as one would in a language like Java.

- $!e_1$ to dereference the heap address held by expression e_1.
- $\mathbf{new}(T, e_1)$ to allocate and return a T *ref* with the value obtained after evaluating e_1.

Commands We add a new command that dereference a heap address and modifies its underlying value. The command $!e_1 := e_2$ changes the value at heap address after evaluating e_1 to the value after evaluating e_2.

Values We add new values into the language, h is an element in the set $\{h_0, h_1, h_2...\}$ and repesents a heap address.

Heap Finally, we define a heap context Δ as a mapping from heap addresses to values. It represents the set of heap addresses that are currently in use and we use the notation, $\Delta(h)$ to get the value held at a particular heap address.

Typing Rules

We define the rules for pointers in figure 4.24. The typing rules are straightforward, dereferencing a heap address of type $T\,ref$ yields a value of type T and allocating a new heap address with type T gets a reference of the same type.

$$\frac{\Gamma \vdash e_1 : T\ ref}{\Gamma \vdash !e_1 : T}\ \text{(Deref)} \qquad \frac{\Gamma \vdash e_1 : T}{\Gamma \vdash \mathbf{new}(T, e_1) : T\ ref}\ \text{(New-Ref)}$$

$$\frac{\Gamma \vdash e_2 : T \qquad \Gamma \vdash e_1 : T\ ref}{\Gamma \vdash_C\ !e_1 := e_2 : void}\ \text{(Deref-Assign)}$$

Figure 4.24: Typing Rules for Ref Type

Operational Semantics

Lastly, we define the operational semantics for the new expression forms in figure 4.25. Pointers are allocated using the **new** expression which takes a type T and a value v, the evaluation rules allocate a new heap address h that is not currently present, assigns it the value v and returns the address. The value can be retrieved and modified through dereferencing, shown in the first two rules.

4.6.2 Ambiguity of Refinement Types involving Pointers

The introduction of pointers provides the programmer with multiple new tools. Features such as pass-by-reference and aliasing are now at their disposal. However with the introduction of new features, the feature interaction problem arises once again and the relation between pointers and refinement types is unclear. Previously we showed that we can avoid ambiguous behaviour between mutation and refinement types using immutability or closures. This appears not to be the case with pointers, consider the code snippet in figure 4.26. In this scenario, we declare a constant y as a pointer to an int, initialised with the value 10. The expression $!y := 5$ indirectly modifies the definition of the type $BoundedIntY$, making z ill-typed.

$$\frac{\langle e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle e_1', \sigma, \tau, \Delta'\rangle}{\langle !e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle !e_1', \sigma, \tau, \Delta'\rangle}\ \text{(Deref-E)} \qquad \frac{h_1 \in \mathbf{dom}(\Delta) \quad v = \Delta(h_1)}{\langle !h_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle v, \sigma, \tau, \Delta\rangle}\ \text{(Deref-V)}$$

$$\frac{\langle e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle e_1', \sigma, \tau, \Delta'\rangle}{\langle !e_1 := e_2, \sigma, \tau, \Delta\rangle \longrightarrow \langle !e_1' := e_2, \sigma, \tau, \Delta'\rangle}\ \text{(P-Assign-}E\text{ 1)}$$

$$\frac{\langle e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle e_1', \sigma, \tau, \Delta'\rangle}{\langle !h_1 := e_1, \sigma, \tau, \Delta\rangle \longrightarrow \langle !h_1 := e_1', \sigma, \tau, \Delta'\rangle}\ \text{(P-Assign-}E\text{ 2)}$$

$$\frac{h_1 \in \mathbf{dom}(\Delta) \quad \Delta' = \Delta[h_1 \mapsto v_1]}{\langle !h_1 := v_1, \sigma, \tau, \Delta\rangle \longrightarrow \langle \sigma, \tau, \Delta'\rangle}\ \text{(P-Assign)}$$

$$\frac{\langle e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle e_1', \sigma, \tau, \Delta'\rangle}{\langle \mathbf{new}(T, e_1), \sigma, \tau, \Delta\rangle \Longrightarrow \langle \mathbf{new}(T, e_1'), \sigma, \tau, \Delta'\rangle}\ \text{(P-Alloc-}E\text{)}$$

$$\frac{h \notin \mathbf{dom}(\Delta) \quad \Delta' = \Delta + \{h \mapsto v\}}{\langle \mathbf{new}(T, v), \sigma, \tau, \Delta\rangle \Longrightarrow \langle h, \sigma, \tau, \Delta'\rangle}\ \text{(P-Alloc-}V\text{)}$$

Figure 4.25: Operational Semantics for Pointers

```
begin const y : int ref = new(int, 10);
    begin typedef BoundedIntY = {i : int | i < !y};
        begin var z : BoundedIntY = 6; !y := 5 end
    end
end
```

Figure 4.26: Refinement Type Ambiguity involving Pointers

Our previously defined semantics proves to be underspecified and does not provide a definite behaviour. The approach where we require all free variables in a refinement predicate to be constant however this is not sufficient for pointers. A constant pointer variable simply restrict the address held by the pointer to be changed, manipulating the underlying value is still very much possible. Moreover, using closures also fails to prevent the ambiguity as one can easily modify the heap address and have the changes reflect within the closure. Our previous semantics resembles Java's *final* mechanics discussed in section 3.1.1, whereby while the reference variable cannot change, the underlying value can.

In order to address this case, our previously proposed semantics has to be extended to address two different features. One is to prevent the pointer from modifying the underlying heap address and the other is that no aliases should be able to modify the address indirectly.

4.6.3 Immutable Heap Locations

In C/C++, one is able to declare a pointer to a constant, we would require the same mechanism in Simple-R^*. We create a new variant of Simple-R^* called Simple-R^*_I to capture this behaviour.

It is important to note the difference between an immutable pointer and a pointer to an immutable. The latter allows the pointer variable to change. In order to concretely differentiate the two in a definite manner, we introduce a new predicate $is\_immutable\_heap(h)$ to specify if the value held at a specific memory address is immutable. The $is\_immutable\_heap(h)$ predicate holds if and only if the heap address h is a constant and cannot be modified.

While this predicate behaves exactly as the $is\_immutable(x)$ predicate defined previously, we made the distinction between the two as the one operates on variables and the other operates on heap addresses.

The $is\_immutable\_heap(h)$ predicate also applies if a heap address has the value of another heap address, e.g. `int **p` in C. If a heap address h_1 has the value of another heap address h_2 then we require $is\_immutable\_heap(h_1) \Leftrightarrow is\_immutable\_heap(h_2)$.

In order for one to define an immutable pointer we added a new expression **newconst**(T, e_1) which allocates a heap allocated constant value and returns a pointer to it.

$$\textit{Expressions} \quad e \quad : ... \mid \textbf{newconst}(T, e_1)$$

Figure 4.27: Heap Memory in Simple-R^*_C

The typing rule in figure 4.28 above formalises the typing rules for this new expression.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{newconst}(T, e) : T \ \textit{ref}} \ \text{(New-Const)}$$

Figure 4.28: Typing Rules for Pointer to Constants

Notice that unlike C/C++, Simple-R^* does not have the notion of *address-taking*, the only way to obtain a pointer is through the use of **new** or **newconst**. Recall from section 3.1.1, this form avoids pitfalls in C/C++ where one gives a pointer to a constant the address of a mutable variable which still allows the value to be modified in a different manner.

The lack of address-taking allows us to define semantics that completely forbid modification on a heap location. We define the operational semantics for the new expression in figure 4.29, the rules simply restrict the ability to modify the value of a constant heap address.

$$\frac{\langle e_1, \sigma, \tau, \Delta\rangle \Longrightarrow \langle e_1', \sigma, \tau, \Delta'\rangle}{\langle \mathbf{newconst}(T, e_1), \sigma, \tau, \Delta\rangle \Longrightarrow \langle \mathbf{newconst}(T, e_1), \sigma, \tau, \Delta'\rangle} \text{ (PConst-Alloc-}E\text{)}$$

$$\frac{h \notin \mathbf{dom}(\Delta) \qquad \Delta' = \Delta + \{h \mapsto v\}}{\langle \mathbf{newconst}(T, v), \sigma, \tau, \Delta\rangle \Longrightarrow \langle h, \sigma, \tau, \Delta'\rangle} \text{ (PConst-Alloc-}V\text{)}$$

$$\frac{h_1 \in \mathbf{dom}(\Delta) \qquad \neg \mathit{is\_immutable\_heap}(h_1) \qquad \Delta' = \Delta[h_1 \mapsto v_1]}{\langle !h_1 := v_1, \sigma, \tau, \Delta\rangle \longrightarrow \langle v_1, \sigma, \tau, \Delta'\rangle} \text{ (P-Assign)}$$

Figure 4.29: Operational Semantics for Pointers to Constants

The introduction of immutable heap address allows us to specify a concrete behaviour for refinement types in the presence of pointers. Adopting a similar strategy to our previous restriction, we require all pointers present in a refinement predicate to not only be a constant itself but it must also be referencing a constant value. In the following text, we incorporate the same idea used in Simple-R_I where we restrict mutation on free variables in a refinement predicate.

We define a new predicate, $\mathit{is\_ref}(x)$, which returns true if and only if x is a pointer variable. Formally, $\mathit{is\_ref}(x) \Leftrightarrow \Gamma(x) = T\ \mathit{ref}$.

The new specification for refinement type definition is proposed in figure 4.30.

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\sigma) \qquad\qquad x \notin \mathbf{dom}(\tau) \\ \forall \alpha \in \mathit{free\_vars}(e).\ \mathit{is\_immutable}(\alpha) \\ \forall \alpha \in \mathit{free\_vars}(e).\ \mathit{is\_ref}(\alpha) \Rightarrow \mathit{is\_immutable\_heap}(\sigma(\alpha)) \\ \langle C_1, \sigma, \tau + \{x \mapsto \{v : T \mid e\}\} \longrightarrow \langle \sigma', \tau\rangle \end{array}}{\langle \mathbf{begin\ typedef}\ x = \{v : T \mid e\}; C_1\ \mathbf{end}, \sigma, \tau\rangle \longrightarrow \langle \sigma', \tau\rangle} \text{ (Refine-Typedef-Immu-P)}$$

Figure 4.30: Declaring Refinement Types with Pointers

The new specification retains all constraints present and make a crucial new addition. In addition to all free variables being immutable, a new rule states that all pointer variables in the expression must to be pointing to a constant. This ensures that the value pointed to by the pointer would not change throughout the type's existence, this also applies to a pointer pointing to another pointer. Furthermore, we can be sure that any aliases to the heap address will not be able to modify the memory address because of the rules specified in figure 4.29.

The example program shown previously in figure 4.26 is now an invalid program. To make it valid we adapt the newly defined rules and rewrite it as in figure 4.31.

begin const y : $int\ ref =$ **newconst**$(int, 10)$;
  begin typedef $BoundedIntY = \{i : int \mid i < !y\}$;
    begin var z : $BoundedIntY = 6$; **skip end**
  end
end

Figure 4.31: Valid Pointers in Refinement Types

4.6.4 Pointers in Type Closures

In this section we discuss how the Simple-R_C variant introduced previously can be adapted to include pointers.

Immutable Heap Locations in Type Closures

The approach of using Type Closures in Simple-R_C can easily be modified to work with pointers using immutable heap locations. If one declares a refinement type T, then in addition to the context being copied, we require any pointer variables in the refinement predicate to be constant. This variant, extended from Simple-R^*, is named Simple-R^*_C

$$\frac{\begin{array}{c}\sigma_1 = \sigma \qquad\qquad \tau_1 = \tau \\ \forall \alpha \in \mathit{free\_vars}(e).\ \mathit{is\_ref}(\alpha) \Rightarrow \mathit{is\_immutable\_heap}(\sigma(\alpha)) \\ \langle C_1, \sigma, \tau + \{y \mapsto (\{v : T \mid e\}, \sigma_1, \tau_1)\}\rangle \longrightarrow^* \langle \sigma', \tau\rangle\end{array}}{\langle \textbf{begin typedef } y = \{v : T \mid e\}; C_1; e_1 \textbf{ end}, \sigma, \tau\rangle \longrightarrow \langle \sigma', \tau\rangle} \text{ (Refine-Typedef-Closure-P)}$$

Figure 4.32: Operational Semantics for Refinement Type in Simple-R^*_C

Deep Copy Pointers

An alternative solution which enhances the flexibility of the language is snapshotting pointers using a *deep copy* method and storing these in closures. We observe that pointers in refinement types simply behaves like local constant variables as their values cannot be modified. One can easily snapshot the values held by pointers and store them in a new local variables, for example, if a refinement type has the form $\{v \mid v <!!!p\}$, then the type system would snapshot the value of at the end of the multi-level pointer p and rewrite the type as $\{v \mid v < p_{val}\}$ where p_{val} is a local variable in the Type Closure that has the value of dereferecing the multi-level pointer p.

Using a deep copy approach improves flexibility as one does not need to declare pointers to immutable in order to use them in refinement types. Due to the limits of time, we propose this as an interesting idea and leave the formal semantics as future work.

4.6.5 Summary

In this section we extended our language to include advanced features such as pointers and aliasing. We observed how introducing new form of variables brings new form of mutation into the language and therefore requires us to adapt new strategies in our specification. We presented a new system that characterised the behaviour of pointers in the context of refinement types. Our system forbids mutation on heap locations used in types to ensure consistency in refinement types while allowing them for local variables. Finally, we incorporate these new requirements into our language by creating two new variants, Simple-R_I^* and Simple-R_C^*, extending our previously defined semantics.

4.7 Overall Summary

In this chapter we constructed Simple-R, an imperative programming language with native support for refinement types. We distinguished our system apart from existing work by basing the underlying calculus on a simple "*WHILE*" language rather than the λ-calculus, to the best of our knowledge this form of specification for refinement type with mutation has not been done previously. The syntax and semantics of our language resembles C/C++, enhancing the accessibility of our work especially compared to those using Monads such as Ynot.

Throughout the chapter we demonstrated how type-checking refinement types statically proves to be undecidable in our system. We presented a form of hybrid type checking that overcomes this problem, our type checker verifies as much as possible statically and defers more complex verification to be done dynamically.

However, we observed that naive runtime type checking has ambiguities because of the presence of mutation. We identified these underspecifications and proposed two ideas that concretely characterise the behaviour of refinement types in a mutable environment. Our first approach is based on the idea of immutable variables and forbidding mutation on variables associated with the refinement predicate. The second approach utilises the notion of closures, a snapshot of the relevant parts of state is made at the time of type declaration and the copied state is always used

for type checking. The base language is extended into two new variants, Simple-R_I and Simple-R_C incorporates the first and second approach respectively.

Finally, we explored the idea of adding dynamically allocated memory into the language and how it leads to further ambiguity. We introduce the concept of immutable heap locations that forbids modifications to pointers used in a refinement predicate, we showed how our variant avoid certain pitfalls in C/C++ by disallowing address taking. We conclude the chapter by extending Simple-R_I and Simple-R_C with two new variants Simple-R_I^* and Simple-R_C^* which provide the semantics for pointers in refinement types.

Chapter 5

Related Work

In this chapter we review existing literature and discuss their contributions to the field. We categorise past research into two groups, those focusing on the theoretical aspects of refinement types and those relating refinement types to mainstream programming, i.e. languages with a significant number of users.

Theory of Refinement Types Research into refinement types in imperative languages often appears under the name of dependent types. Although these projects are investigating dependent types in imperative languages, the dependent types described in the projects are often restricted such that it takes the form of refinement types [7, 35].

A notable example is the type system *DML(C)* described by Hongwei Xi and Frank Pfenning in this 1999 paper [59]. In the paper, the authors described a dependent type system that is restricted over a domain of constraints C, the addition of C allows type checking to be decidable. This paper is significant as it was the first type system that introduced dependent types in a general purpose language. The DML(C) type system is later adapted into different dependently typed imperative language such as *Xanadu* [56] and *ATS* [57].

Hoare Type Theory (HTT) is another notable approach to dependent types in an imperative setting. Proposed by Aleksandar Nanevski, Greg Morrisett and Lars Birkedal in a 2006 paper [30], HTT combines Hoare Logic with dependent types to create a new form of monadic Hoare Triples that encodes effective computation involving dependent types and imperative commands. Type checking in HTT is performed in two phases. The first phase is basic type-checking and verification-condition checking which can be fully automatic. The second phase involves checking

the validity of the generated verification-conditions which is done by an automated theorem provers, manually or even ignored. The two phase type checking used in HTT resembles the two phase type checking we employed in Simple-R. Hoare Type Theory is implemented in *Ynot* which we covered in section 2.5.1.

Refinement Types in Mainstream Programming DML(C) and Hoare Type Theory represent the theoretical aspects of refinement types and dependent types where the authors created a new type system to interact with imperative program. However, a substantial number of research have attempted to bring refinement types to mainstream programming languages.

The concept of refinement types was first introduced by Tim Freeman and Frank Pfenning [16] in 1991 as an extension to the *ML* programming language. The paper introduce a system of subtypes for ML that preserves decideable type inference and compile-time checking. Although the authors only considered refinement types in Mini-ML where imperative code is not possible, the paper introduced a brand new type that would prove to be a popular research area.

LiquidHaskell was introduced in 2014 by Niki Vazou, Eric L. Seidel and Ranjit Jhala as an extension to Haskell to enable refinement types [52]. The programmer specifies refinement types and properties using annotations which will be checked by LiquidHaskell. The authors made contributions by showing how one could extend a mainstream language such as Haskell with refinement types, introducing the concept of refinement types to more programmers.

In 2016 Panagiotis Vekris, Benjamin Cosman and Ranjit Jhala wrote a paper that introduced refinement types into the TypeScript programming language [53]. Unlike LiquidHaskell which operates on a purely functional language, the authors presented *Refined TypeScript* which showed how refinement types would behave in a mainstream imperative language. This was achieved by restricting refinement types to immutable variables as we did in our project, furthermore SSA is used to handle certain reassignments.

Our project complements the existing literature by serving as a link between the two areas described in this chapter. We provide a simple theoretical foundation for refinement types in a C-Style imperative programming language while relating the theoretical concepts to different design choices in existing real-world languages. Another area our work relates closely to is the study of hybrid type checking, we have discuss some of the work in the area in section 3.3.

Chapter 6

Conclusions and Future Work

Refinement Types provide a more expressive type-checking however its applications in real-world programming are not often studied. Existing research mostly focuses on the theoretical aspects and studies how refinement types would behave in an imperative language without relating it to real-world programming. Our research presented a novel approach to the introduction of refinement types in an imperative programming language, we achieved this by basing our research on a system that closely resembles real-world programming languages, allowing our work to be accessible to more readers in constrast to systems like Ynot which requires reexpressing programs in monadic style.

We adapted three key ideas from real-world programming languages, immutable variables, closures and hybrid type checking. Combining these ideas, we proposed Simple-R, an imperative programming language built on the WHILE language that supports refinement types. Type checking refinement types is difficult as in general predicates are undecideable. To overcome this, Simple-R employs a hybird type checking approach that statically checks basic typing properties and defers complicated checks to be done dynamically.

The presence of mutation brings certain complications to refinement types. As the refinement predicate can make use of free variables in the program, changes to these variables will effectively change the meaning of the type. We extended Simple-R with two variants Simple-R_I and Simple-R_C. The former restricts the sort of free variables that can appear in the refinement predicate by allowing only immutable variables, similar to Java's restriction on variables in Anonymous Functions. The latter introduces a concept called Type Closures which snapshots the program state when a type is declared; subsequent type checks use this copied state and ensures a

change in the program state does not impact the type. We conclude our research by extending these languages with two new variants Simple-R^*_I and Simple-R^*_C which introduce pointers into the two previous variants. The addition of pointers means that the simple notion of constant is now inadequate to handle refinement types as the underlying value can still be modified. We address this problem by using the concept of pointers to immutable in C/C++, highlighting the key differences between constant and immutable.

6.1 Future Work

Simple-R serves as a simple foundation for introducing refinement types into imperative languages and it can be extended with different ideas. In this section we outline some possible areas for future work.

Advanced Language Features In a paper [7] published in 2012, Joana Campos and Vasco Vasconcelos presented an object-oriented language with dependent types. An interesting extension would be to extend Simple-R with classes and study how refinement types interacts with them. We previously proposed using deep copy semantics to handle pointers in Simple-R, a potential future work is to formalise the semantics. Lastly, refinement types in an concurrent environment would be a novel extension that has not been studied previously.

Formal Verifications and Modelling In our project we focused on the construction of our language and discuss how it fits into real-world programming. As such we have omited discussion of formal verifications such as the checks for *soundness* and *completeness*. A potential outlet for future work would be to perform these verifications for Simple-R. One can even take it further by modelling the Simple-R language in an automated theorem prover to ensure future modifications keeps the integrity of the language.

Full Dependent Types Support An obvious future work would be to introduce full dependent type support in Simple-R. Dependent types are significantly more powerful than refinement types and their introduction would allow the programmer even more expressive typing. Like refinement types, past research on the topic of dependent types focused on the theoretical aspects, a similar project of presenting dependent types in real-world programming would be interesting and novel.

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce and G. Plotkin. "Dynamic Typing in a Statically-Typed Language". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, 213–227. ISBN: 0897912942. DOI: 10.1145/75277.75296. URL: https://doi.org/10.1145/75277.75296.

[2] Ana Bove and Peter Dybjer. "Dependent Types at Work". In: *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. Ed. by Ana Bove, Luís Soares Barbosa, Alberto Pardo and Jorge Sousa Pinto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–99. ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_2.

[3] Edwin Brady. *Idris: A Language for Type-Driven Development*. URL: https://www.idris-lang.org/ (visited on 11/11/2020).

[4] Kim Bruce. "Foundations of Object-Oriented Programming Languages: Types and Semantics". In: (Jan. 2002).

[5] Bruno Cabral, Paulo Sacramento and Paulo Marques. "Hidden truth behind .NET's exception handling today". In: *Software, IET* 1 (Jan. 2008), pp. 233 –250. DOI: 10.1049/iet-sen:20070017.

[6] Muffy Calder, Mario Kolberg, Evan H. Magill and Stephan Reiff-Marganiec. "Feature Interaction: A Critical Review and Considered Forecast". In: *Comput. Netw.* 41.1 (Jan. 2003), 115–141. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(02)00352-3. URL: https://doi.org/10.1016/S1389-1286(02)00352-3.

[7] Joana Campos and Vasco T. Vasconcelos. "Dependent Types for Class-based Mutable Objects (Artifact)". In: *Dagstuhl Artifacts Ser.* 4.3 (2018), 01:1–01:2. DOI: 10.4230/DARTS.4.3.1. URL: https://doi.org/10.4230/DARTS.4.3.1.

[8] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. URL: https://pypl.github.io/ (visited on 18/12/2020).

[9] Robert Cartwright and John McCarthy. "First Order Programming Logic". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: Association for Computing Machinery, 1979, 68–80. ISBN: 9781450373579. DOI: 10.1145/567752.567759. URL: https://doi.org/10.1145/567752.567759.

[10] The Idris Community. *Idris: Frequently Asked Questions*. URL: http://docs.idris-lang.org/en/latest/faq/faq.html (visited on 04/01/2021).

[11] Oracle Corporation. *The Java Tutorials: Why use Generics*. URL: https://docs.oracle.com/javase/tutorial/java/generics/why.html (visited on 04/01/2021).

[12] Benjamin Cosman and Ranjit Jhala. "Local Refinement Typing". In: *Proceedings of the ACM on Programming Languages* 1 (June 2017). DOI: 10.1145/3110270.

[13] cplusplus.com. *C++: Templates*. URL: https://www.cplusplus.com/doc/oldtutorial/templates/ (visited on 04/01/2021).

[14] cppreference.com. *C++: constexpr specifier*. URL: https://en.cppreference.com/w/cpp/language/constexpr (visited on 04/01/2021).

[15] Facebook Engineering. *Fighting spam with Haskell*. URL: https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/ (visited on 18/12/2020).

[16] Tim Freeman and Frank Pfenning. "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, 268–277. ISBN: 0897914287. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[17] fthomas. *refined: simple refinement types for Scala*. URL: https://github.com/fthomas/refined (visited on 24/03/2021).

[18] Golang. *A Tour of Go: Type inference*. URL: https://tour.golang.org/basics/14 (visited on 16/12/2020).

[19] haskell.org. *Haskell: An advanced, purely functional programming language*. URL: https://www.haskell.org/ (visited on 18/12/2020).

[20] W. A. Howard. "The formulae-as-types notion of construction". In: 1969.

[21] Oracle Inc. *The Java™ Tutorials: Anonymous Classes*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html (visited on 13/02/2020).

[22] Ranjit Jhala and Niki Vazou. *Refinement Types: A Tutorial*. 2020. arXiv: 2010.07763 [cs.PL].

[23] Hui Jiang, Dong Lin, Xingyuan Zhang and Xiren Xie. "Type system in programming languages". In: *Journal of Computer Science and Technology* (2001), pp. 286–292.

[24] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Aug. 1996. ISBN: 978-0-471-94148-4.

[25] Abdullah Khanfor and Ye Yang. "An Overview of Practical Impacts of Functional Programming". In: Dec. 2017, pp. 50–54. DOI: 10.1109/APSECW.2017.27.

[26] Kenneth Knowles and Cormac Flanagan. "Hybrid Type Checking". In: *ACM Trans. Program. Lang. Syst.* 32.2 (Feb. 2010). ISSN: 0164-0925. DOI: 10.1145/1667048.1667051. URL: https://doi.org/10.1145/1667048.1667051.

[27] Jagatheesan Kunasaikaran and Azlan Iqbal. "A Brief Overview of Functional Programming Languages". In: *electronic Journal of Computer Science and Information Technology* 6 (Dec. 2016), p. 32.

[28] Per Martin-Löf. *Intuitionistic type theory*. Vol. 1. Studies in proof theory. Bibliopolis, 1984. ISBN: 978-88-7088-228-5.

[29] Microsoft. *C documentation*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/ (visited on 16/12/2020).

[30] Aleksandar Nanevski, Greg Morrisett and Lars Birkedal. "Polymorphism and separation in Hoare type theory". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. Ed. by John H. Reppy and Julia L. Lawall. ACM, 2006, pp. 62–73. DOI: 10.1145/1159803.1159812. URL: https://doi.org/10.1145/1159803.1159812.

[31] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau and L. Birkedal. *The Ynot Project*. URL: http://ynot.cs.harvard.edu/ (visited on 15/04/2021).

[32] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau and L. Birkedal. "Ynot: dependent types for imperative programs". In: *ICFP 2008*. 2008.

[33] Bengt Nordström and Kent Petersson. *Types and specifications*. Chalmers Tekniska Högskola/Göteborgs Universitet. Programming Methodology Group, 1983.

[34] Ulf Norell, Andreas Abel, Nils A. Danielsson and Catarina Coquand M. Takeyama. *Agda*. URL: https://agda.readthedocs.io/ (visited on 02/11/2020).

[35] Xinming Ou, Gang Tan, Yitzhak Mandelbaum and David Walker. "Dynamic Typing with Dependent Types". In: *Exploring New Frontiers of Theoretical Informatics*. Ed. by Jean-Jacques Levy, Ernst W. Mayr and John C. Mitchell. Boston, MA: Springer US, 2004, pp. 437–450. ISBN: 978-1-4020-8141-5.

[36] Ricardo Peña. "An Introduction to Liquid Haskell". In: *Proceedings XVI Jornadas sobre Programación y Lenguajes, PROLE 2016, Salamanca, Spain, 14-16th September 2016*. Ed. by Alicia Villanueva. Vol. 237. EPTCS. 2016, pp. 68–80. DOI: 10.4204/EPTCS.237.5. URL: https://doi.org/10.4204/EPTCS.237.5.

[37] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

[38] Marco Pil. "Dynamic Types and Type Dependent Functions". In: *Selected Papers from the 10th International Workshop on 10th International Workshop*. IFL '98. Berlin, Heidelberg: Springer-Verlag, 1998, 169–185. ISBN: 3540662294.

[39] Gordon D Plotkin. "The origins of structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60-61 (2004). Structural Operational Semantics, pp. 3 –15. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2004.03.009. URL: http://www.sciencedirect.com/science/article/pii/S1567832604000268.

[40] École Polytechnique Fédérale Lausanne. *The Scala Programming Language*. URL: https://www.scala-lang.org/ (visited on 24/03/2021).

[41] François Pottier and Yann Régis-Gianas. "Stratified Type Inference for Generalized Algebraic Data Types". In: *SIGPLAN Not.* 41.1 (Jan. 2006), 232–244. ISSN: 0362-1340. DOI: 10.1145/1111320.1111058. URL: https://doi.org/10.1145/1111320.1111058.

[42] Sanjiva Prasad and S. Arun-Kumar. "Introduction to Operational Semantics". In: *The Compiler Design Handbook: Optimizations and Machine Code Generation* (Sept. 2002). DOI: 10.1201/9781420040579.ch22.

[43] ghc proposals. *GitHub: ghc-proposals*. URL: https://github.com/ghc-proposals/ghc-proposals (visited on 18/12/2020).

[44] Raúl Rojas. "A Tutorial Introduction to the Lambda Calculus". In: *CoRR* abs/1503.09060 (2015). arXiv: 1503.09060. URL: http://arxiv.org/abs/1503.09060.

[45] Patrick M. Rondon, Ming Kawaguci and Ranjit Jhala. "Liquid Types". In: *SIGPLAN Not.* 43.6 (June 2008), 159–169. ISSN: 0362-1340. DOI: 10.1145/1379022.1375602. URL: https://doi.org/10.1145/1379022.1375602.

[46] A. Sabry. "What is a Purely Functional Language?" In: *J. Funct. Program.* 8 (1998), pp. 1–22.

[47] S.Poornima, Chakunta Rao, Nazia Thabassum and Dr.S.P. Anandaraj. "Memory Management by Using the Heap and the Stack in Java". In: *International Journal of Research In Computer Science Engineering (IJRCSE)* 4 (Dec. 2014), pp. 1977–1982.

[48] Gerald Sussman and Guy Steele. "Scheme: A Interpreter for Extended Lambda Calculus". In: *Higher-Order and Symbolic Computation* 11 (Dec. 1998), pp. 405–439. DOI: 10.1023/A:1010035624696.

[49] Don Syme, Adam Granicz and Antonio Cisternino. "Introducing Imperative Programming". In: *Expert F# 2.0*. Ed. by Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh, Anita Castro and Tiffany Taylor. Berkeley, CA: Apress, 2009, pp. 67–96. ISBN: 978-1-4302-2432-7. DOI: 10.1007/978-1-4302-2432-7_4. URL: https://doi.org/10.1007/978-1-4302-2432-7_4.

[50] The Coq development team. *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (visited on 02/11/2020).

[51] Cláudio Vasconcelos and António Ravara. "The While language". In: *CoRR* abs/1603.08949 (2016). arXiv: 1603.08949. URL: http://arxiv.org/abs/1603.08949.

[52] Niki Vazou, Eric L. Seidel and Ranjit Jhala. "LiquidHaskell: Experience with Refinement Types in the Real World". In: *SIGPLAN Not.* 49.12 (Sept. 2014), 39–51. ISSN: 0362-1340. DOI: 10.1145/2775050.2633366. URL: https://doi.org/10.1145/2775050.2633366.

[53] Panagiotis Vekris, Benjamin Cosman and Ranjit Jhala. "Refinement Types for TypeScript". In: *CoRR* abs/1604.02480 (2016). arXiv: 1604.02480. URL: http://arxiv.org/abs/1604.02480.

[54] Philip Wadler. "Monads for Functional Programming". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, 24–52. ISBN: 3540594515.

[55] Stephanie Weirich, Pritam Choudhury, Antoine Voizard and Richard A. Eisenberg. *A Role for Dependent Types in Haskell (Extended version)*. 2019. arXiv: 1905.13706 [cs.PL].

[56] Hongwei Xi. "Imperative Programming with Dependent Types". In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000, pp. 375–387. DOI: 10.1109/LICS.2000.855785. URL: https://doi.org/10.1109/LICS.2000.855785.

[57] Hongwei Xi. "Applied Type System: An Approach to Practical Programming with Theorem-Proving". In: *CoRR* abs/1703.08683 (2017). arXiv: 1703.08683. URL: http://arxiv.org/abs/1703.08683.

[58] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking through Dependent Types". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, 249–257. ISBN: 0897919874. DOI: 10.1145/277650.277732. URL: https://doi.org/10.1145/277650.277732.

[59] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, 214–227. ISBN: 1581130953. DOI: 10.1145/292540.292560. URL: https://doi.org/10.1145/292540.292560.