Foundations and Trends® in Programming Languages

Refinement Types: A Tutorial

Suggested Citation: Ranjit Jhala and Niki Vazou (2021), "Refinement Types: A Tutorial", Foundations and Trends® in Programming Languages: Vol. 6, No. 3-4, pp 159–317. DOI: 10.1561/2500000032.

Ranjit Jhala
University of California, San Diego
USA
jhala@cs.ucsd.edu

Niki Vazou
IMDEA Software Institute, Madrid
Spain
niki.vazou@imdea.org

now
the essence of knowledge
Boston — Delft

Dedicated to Tom Henzinger, on the occasion of his 60th birthday.

Contents

Refinement Types: A Tutorial

Ranjit Jhala[1] and Niki Vazou[2]

[1] *University of California, San Diego, USA; jhala@cs.ucsd.edu*
[2] *IMDEA Software Institute, Madrid, Spain; niki.vazou@imdea.org*

ABSTRACT

Refinement types enrich a language's type system with logical predicates that circumscribe the set of values described by the type. These refinement predicates provide software developers a tunable knob with which to inform the type system about what invariants and correctness properties should be checked on their code, and give the type checker a way to enforce those properties at compile time. In this article, we distill the ideas developed in the substantial literature on refinement types into a unified tutorial that explains the key ingredients of modern refinement type systems. In particular, we show how to implement a refinement type checker via a progression of languages that incrementally add features to the language or type system.

Ranjit Jhala and Niki Vazou (2021), "Refinement Types: A Tutorial", Foundations and Trends® in Programming Languages: Vol. 6, No. 3-4, pp 159–317. DOI: 10.1561/2500000032.

1

Introduction

The type systems of modern languages like C#, Haskell, Java, Ocaml, Rust, and Scala are *the* most widely used method for establishing guarantees about the correct behavior of software. In essence, types allow the programmer to describe *legal* sets of values for various operations, thereby eliminating, at compile-time, the possibility of a large swathe of unexpected and undesirable run-time errors. Unfortunately, well-typed programs *do* go wrong.

1. ***Divisions by zero*** The fact that a divisor is an `int` does not preclude the possibility of a run-time divide-by-zero, or that a given arithmetic operation will over- or under-flow;

2. ***Buffer overflows*** The fact that an `array` or `string` index is an `int` does not eliminate the possibility of a segmentation fault, or worse, leaking data from an out-of-bounds access;

3. ***Mismatched dimensions*** Moving up a level, the fact that a product operator is given two `matrix` values does not prevent errors arising from the matrices having incompatible dimensions;

4. ***Logic bugs*** Classical type systems can ensure that each `date` structure contains suitable (*e.g.* `int` valued) fields holding the `day`,

`month` and `year`, but cannot guarantee that the `day` is valid for the given `month` and `year`;

5. ***Correctness errors*** Finally, at the extreme end, a type system can ensure that a sorting routine produces a list, and that a compilation routine produces a sequence of machine instructions, but cannot guarantee that the list was, in fact, an ordered permutation of the input, or that the machine instructions faithfully implemented the program source.

Refining Types with Predicates Refinement types allow us to enrich a language's type system with *predicates* that circumscribe the set of values described by the type. For example, while an `int` can be any integer value, we can write the refined type

```
type nat = int[v|0 <= v]
```

that describes only non-negative integers. By combining types and predicates the programmer can write precise *contracts* that describe legal inputs and outputs of functions. For example, the author of an array library could specify that

```
val size : x:array(α) ⇒ nat[v|v = length(x)]
val get  : x:array(α) ⇒ nat[v|v < length(x)] ⇒ α
```

which say that (1) a call `size(arr)` *ensures* the returned integer equals to the number of elements in `arr`, and (2) the call `get(arr, i)` *requires* the index `i` be within the bounds of `arr`. Given these specifications, the refinement type checker can guarantee, at *compile time*, that all operations respect their contracts, to ensure, *e.g.* that all array accesses are safe at run-time.

Language-Integrated Verification Refinements provide a tunable knob whereby developers can inform the type system about what invariants and correctness properties they care about, *i.e.* are important for the particular domain of their code. They could begin with basic safety requirements, *e.g.* to eliminate divisions by zero and buffer overflows, or ensure they don't attempt to access values from an empty stack or collection, and then, incrementally, dial the specifications up to include, *e.g.* invariants about custom data types like dates, or ordered heaps, and, if they desire, ultimately go all the way to specifying and verifying the correctness of various routines at compile-time. Crucially, (refinement) types eliminate the barrier between implementation and proof, by enabling verification within the same language, library and tool ecosystem. This tight integration is essential to create a virtuous cycle of feedback across the phases. The *implementation* dictates what properties are important, and provides hints on how to do the verification. Dually, the *verification* provides guidance on how the code can be restructured, *e.g.* to make the abstractions and invariants explicit enough to enable formal proof.

1.1 A Brief History

Refinement types can be thought of as a type-based formulation of assertions from classical program logic (Turing, 1949; Floyd, 1967; Hoare, 1969). The idea of refining types with logical constraints goes back at least to Cartwright, 1976 who described a means of refining Lisp datatypes with constraints to aid in program verification. The ADA programming language has a notion of *range* types which allow the to define contiguous subsets of integers (Dewar *et al.*, 1980). Nordstrom and Petersson, 1983 and Constable, 1983 introduced the notion of logical-refinements-as-subsets of values, and Constable, 1986 turned this notion into a pillar of the Nuprl proof assistant.

Freeman and Pfenning, 1991a introduced the name "refinement types" in a paper that describes a syntactic mechanism to define subsets of algebraic data[1]. Inspired by the early work on Nuprl, the PVS proof assistant embraced the idea of types as subsets, and Rushby *et al.*, 1998 introduced the notion of *predicate subtyping* which forms the basis of the subtyping relation that remains the workhorse of modern refinement type systems. Zenger, 1997 and Xi and Pfenning, 1998 describe a means of *indexing* types with (symbolic) integers after which constraints can be used to specify function contracts that can be verified by linear programming, to, *e.g.* perform array bounds or list or matrix dimension checking at compile time, and Dunfield, 2007 shows how to combine

[1] See Michael Greenberg's post "A refinement type by any other name" for a more detailed discussion on the history of refinement types

indices with datasort refinements to facilitate the verification of data structure invariants.

The Sage system (Gronski *et al.*, 2006) described how refinement like specifications could be verified in a *hybrid* manner: partly at compile time using SMT solvers, and partly at run-time via dynamic contract checks (Flanagan, 2006). Several groups picked up the gauntlet of moving all the checks to compile time, leading to the F7 (Bengtson *et al.*, 2011) and then F* (Swamy *et al.*, 2011) dialects of ML which has been used to formally verify the implementation of cryptographic routines used in widely used web-browsers (Zinzindohoué *et al.*, 2017). Rondon *et al.*, 2008 introduced the notion of *liquid* types which make refinements easier to use by delegating the task of synthesizing refinements to abstract interpretation.

The last decade has seen refinements spread over to languages outside the ML family. Rondon *et al.*, 2010 and Chugh *et al.*, 2012 show how to verify C and JavaScript programs by refining a low-level language of locations (Smith *et al.*, 2000). Kent *et al.*, 2016 show how refinements can be integrated within Racket's occurrence based type system (Tobin-Hochstadt and Felleisen, 2008). Kazerounian *et al.*, 2017 integrate refinements in Ruby's type system using just-in-time type checking. Finally, Hamza *et al.*, 2019 present a refinement-type based verifier for higher-order Scala programs.

1.2 Goals & Outline

Refinement types can be the vector that brings formal verification into mainstream software development. This happy outcome hinges upon the design and implementation of refinement type systems that can be retrofitted to existing languages, or co-designed with new ones. Our primary goal is to catalyze the development of such systems by distilling the ideas developed in the sprawling literature on the topic into a coherent and unified tutorial that explains the key ingredients of modern refinement type systems, by showing how to implement a refinement type checker.

Background We have tried to make this article as self-contained as possible. However, some familiarity with propositional logic and the simply typed lambda calculus will be helpful.

A Nanopass Approach Inspired by the *nanopass framework* for teaching compilation pioneered by Sarkar *et al.*, 2004, we will show how to implement refinement types via a progression of languages that incrementally add features to the language or type system.

- λ_ϕ (§ 3): We start with the simply typed λ-calculus, which will illustrate the foundations, namely, refinements, functions, and function *application*;

- λ_β (§ 4): Next, we will add branch conditions, and show how refinement type checkers do *path-sensitive* reasoning;

- λ_κ (§ 5): Types are palatable when we have to write down only the interesting ones: hence, next, we will see how to automatically *infer* the refinements to make using refinements pleasant;

- λ_α (§ 6): After adding inference to our arsenal, we will be able to add type polymorphism which, will unlock various forms of *context-sensitive* reasoning;

- λ_δ (§ 7): Once we have polymorphic types, we can add polymorphic *data types* like lists and trees, and see how to specify and verify properties of those structures;

- λ_ρ (§ 8): Type polymorphism allows us to reuse functions and data with different kinds of values. We will see why we often need to reuse functions and data across different kinds of invariants, and to support this, we will develop a form of *refinement polymorphism*;

- λ_τ (§ 9): All of the above methods allow us to verify safety properties, *i.e.* assertions about values of code. Next, we will see how refinements let us verify *termination*;

- λ_π (§ 10): Finally, we will see how to write propositions over arbitrary user defined functions and write proofs of those propositions as well-typed programs, effectively converting the host language into a theorem prover.

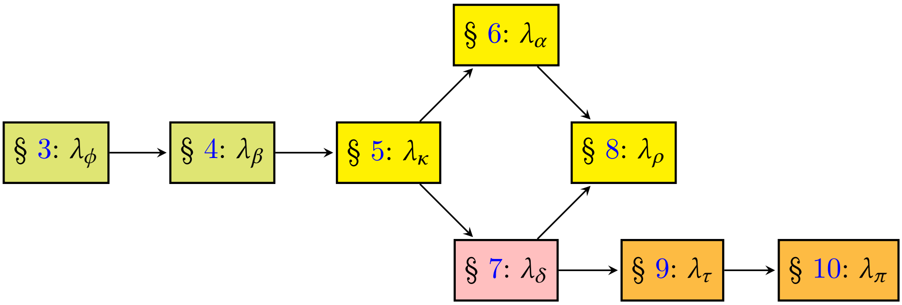

Figure 1.1: Chapter dependencies

Dependencies The ideal reader would, of course, devote several hours of thoughtful contemplation to each of the eight sub-languages. However, life is short, and you may be interested in particular aspects of refinement typing. If so, we suggest reading the chapters in the following order, summarized in Fig. 1.1

- § 3 and § 4 are essential, as they focus on the basics of refinement types and *path-sensitive reasoning*;
- § 5, § 6 and § 8 explain how to support polymorphism via *refinement inference*;
- § 7 explains how refinements allow reasoning about invariants of *algebraic data types*;
- § 9 and § 10 will be of interest to readers who wish to learn how to scale refinements up to *proofs*.

Implementation This article is accompanied by an implementation

`https://github.com/ranjitjhala/sprite-lang`

The `README` that accompanies the code has directions on how to build, modify and execute the sequence of type checkers that we will develop over the rest of this article. We welcome readers who like to get their hands dirty to clone the repository and follow along with the code.

And now, let's begin!

2

Refinement Logic

Refinements are *predicates* drawn from logics whose validity can be decidably checked by Satisfiability Modulo Theory (SMT) solvers. Refinement type checking yields *constraints* whose validity implies that the program is well-typed. Let's start with a quick overview of the logic of predicates and constraints that we will use in this article, and our rationale for choosing it. Readers familiar with SMT solvers can skip ahead, and those interested in learning more should consult Nelson, 1980 or Kroening and Strichman, 2008.

2.1 Syntax

The syntax of predicates and constraints is summarized in Fig. 2.1.

2.1.1 Quantifier-free Predicates

We will work with predicates p drawn from the *quantifier-free* fragment of linear arithmetic and uninterpreted functions (QF-UFLIA) (Barrett *et al.*, 2010). These include boolean literals (*true*, *false*), integer literals $(0, 1, 2, \ldots)$, variables ranging over booleans and integers $(v, x, y, z, \ldots)$ with the variable v usually represents the refined value, linear arithmetic

$$
\begin{array}{llcll}
\textbf{\textit{Predicates}} & p,q & ::= & v, x, y, z, \ldots & \textit{variables} \\
 & & \mid & \textit{true}, \textit{false} & \textit{booleans} \\
 & & \mid & 0, -1, 1, \ldots & \textit{numbers} \\
 & & \mid & p_1 \bowtie p_2 & \textit{interp. ops.} \\
 & & \mid & p_1 \wedge p_2 & \textit{conjunction} \\
 & & \mid & p_1 \vee p_2 & \textit{disjunction} \\
 & & \mid & \neg p & \textit{negation} \\
 & & \mid & \textit{if } p \textit{ then } p_1 \textit{ else } p_2 & \textit{conditional} \\
 & & \mid & f(\overline{p}) & \textit{uninterp. Function} \\
\textbf{\textit{Constraints}} & c & ::= & p & \textit{predicates} \\
 & & \mid & c_1 \wedge c_2 & \textit{conjunction} \\
 & & \mid & \forall x\!:\!b.\ p \Rightarrow c & \textit{implication}
\end{array}
$$

Figure 2.1: Syntax of Predicates and Constraints

operators ($p_1 + p_2, \ldots$), boolean combinations ($p_1 \wedge p_2, \ldots$), and the ternary choice operator ($\textit{if } p \textit{ then } p_1 \textit{ else } p_2$). We use $\bowtie$ to represent all interpreted operators; this set can be extended to include those from other SMT decidable logics *e.g.* operations over sets, strings or bitvectors.

Uninterpreted Functions All other operations will be modeled as applications of uninterpreted functions $f(\overline{x})$. These functions earn their name from the fact that the *only* information that the SMT solver has about their behavior is encoded in the axiom of *congruence*, *i.e.* applying a function to equal arguments yields equal results:

$$\forall \overline{x}, \overline{y}.\ \overline{x} = \overline{y} \Rightarrow f(\overline{x}) = f(\overline{y})$$

We will see how this provides an extremely general mechanism to encode all manner of specifications, from the sizes and heights of trees (chapter 7), to polymorphic refinement variables (chapter 8), to arbitrary user-defined functions (chapter 10).

Example: Predicates Examples of predicates include $0 \leq v$ that we used to denote non-negative integers, and $0 \leq v \wedge v < \textit{length}(x)$ that we used to specify valid indices for the array x, where *length* is an uninterpreted function in the logic. □

2.1.2 Constraints

Refinement type checking will produce *verification condition* (VC) constraints (Floyd, 1967; Hoare, 1971) whose syntax is summarized in Fig. 2.1. A constraint is either a single (quantifier-free) predicate p, or a *conjunction* of two sub-constraints $c_1 \wedge c_2$, or an *implication* of the form $\forall x\!:\!b.\ p \Rightarrow c$ which says that for each x of type b, if the condition p holds then so must c. The type b is defined in the rest of the chapters and as you will notice, includes polymorphic variables and functions. In the implementation, we syntactically map these types to SMT sorts using the standard techniques of monomorphization and defunctionalization. For brevity, we will often omit the sort b of the quantifier bound variable when it is clear from the context.

Example: Constraints The constraints c and c' could be generated from source programs that use an index `i` to access the last element of an array `x`

$$c \doteq \forall x\!:\!\texttt{array}.\ 0 \leq \mathit{length}(x) \Rightarrow \forall n\!:\!\texttt{int}.\ n = \mathit{length}(x) \Rightarrow \forall i\!:\!\texttt{int}.\ i = n - 1 \Rightarrow 0 \leq i \wedge i < \mathit{length}(x)$$

$$c' \doteq \forall x\!:\!\texttt{array}.\ 0 < \mathit{length}(x) \Rightarrow \forall n\!:\!\texttt{int}.\ n = \mathit{length}(x) \Rightarrow \forall i\!:\!\texttt{int}.\ i = n - 1 \Rightarrow 0 \leq i \wedge i < \mathit{length}(x)$$

The variable x is assigned an *uninterpreted* sort `array` that is distinct from all others in the refinement logic. These two constraints seem almost identical, but as we will see in a moment, they have a crucial difference. □

2.2 Semantics

Programs are well-typed when their VCs are valid, as defined next.

Substitution We will write $p[x := q]$ (resp. $c[x := q]$) to denote the (capture avoiding) substitution of all (free) occurrences of x in p (resp. c) with q.

Validity Let Σ denote an interpretation mapping each uninterpreted function f to a corresponding function from the domain to the co-domain of f. We define the notion that Σ models a constraint c, written $\Sigma \models c$ as follows. $\Sigma \models p$ if p has no free variables and p is a tautology under Σ (*i.e.*

$\Sigma(p) \equiv \mathit{true}$). $\Sigma \models c_1 \wedge c_2$ if if $\Sigma \models c_1$ and $\Sigma \models c_2$. Finally, $\Sigma \models \forall x{:}b.\ p \Rightarrow c$ if for every constant v of sort b such that $\Sigma \models p[x := v]$ we have $\Sigma \models c[x := v]$. A constraint c is *valid* if $\Sigma \models c$ for all interpretations Σ.

Checking Validity via SMT SMT solvers can algorithmically determine whether a constraint c is valid by *flattening* it into a collection of sub-formulas of the form $c_i \doteq \forall \overline{x}.p_i \Rightarrow q_i$, such that c is valid iff c_i is valid. The validity of each c_i is determined by checking the *satisfiability* of the (quantifier free) predicate $p_i \wedge \neg q_i$: the formula is valid *iff* no satisfying assignment exists.

Example: Validity The constraint c shown above is *invalid*, as demonstrated by the interpretation where $\mathit{length}(x) \doteq 0$, and then $n \doteq 0$ and $i \doteq -1$. However, the modified version c' is *valid* as every interpretation for length yields a model for the constraint. □

2.3 Decidability

Modern SMT solvers do support quantified formulas and SMT-based verifiers like Dafny (Leino, 2010) and F* (Swamy *et al.*, 2011) use this support to automate verification. However, we make a deliberate choice to restrict the predicates to *quantifier-free* formulas to ensure that the generated VCs remain *decidable*. Decidability is important as a matter of *principle*, as we do not want typability to rely upon the heuristics implemented in different solvers. Instead we want to provide a precise, solver-agnostic, language-based characterization of when a program is well-typed. Decidability is also crucial in *practice*, to ensure that type checking remains predictable. In particular, we want to circumscribe the possible causes that a user need investigate when type checking fails (chapter 12) and decidability ensures that when debugging failures, we can safely avoid worrying about the brittleness of solver heuristics.

3

The Simply Typed λ-calculus

Our first language is the simply typed λ-calculus equipped with primitive arithmetic values and operations. This language has just enough constructs to orient our understanding of refinements, and hence, equips us with the tools needed to explore more advanced features.

3.1 Examples

First, let's build up a mental model of refinements *should* work by working through a few simple examples. Here are two refinement types that describe the subsets of *non-negative* and *positive* integers

$$\textbf{type}\ \texttt{nat} = \texttt{int}\{\nu{:}0 \leq \nu\} \tag{3.1}$$

$$\textbf{type}\ \texttt{pos} = \texttt{int}\{\nu{:}0 < \nu\} \tag{3.2}$$

In the above, ν is the *value variable* which *names* the value being refined (*e.g.* you can use `this` or `self` if you prefer). The condition $0 \leq \nu$ is the *refinement* (from Fig. 2.1) that must be satisfied by any member of the type. Hence, `0,1,2,3,...` are all elements of the refined type `nat`, but not `-1,-2,-3,...`.

Aliases To save ourselves some typing, and as it is often convenient to name concepts, we will define refinement type *aliases*, such as `nat` which is a name for the type of non-negative integers described above.

Ex1: Primitive Constants First, let's consider the simplest possible example of typing code: we should be able to ascribe the `nat` type to the numeric literal `6`

```
val six: nat
let six = 6;
```

Ex2: Primitive Operations The next example illustrates *sequences* of variable definitions (bindings) and how values can be *combined* with various operators. In the below, the types of `six` and `nine` should be composed with that of `add` to let us assign `fifteen` the type `nat`:

```
val fifteen: nat
let fifteen = {
  let six = 6;
  let nine = 9;
  add(six, nine)
};
```

Ex3: Functions Next, consider a function `inc` that takes an `nat` and returns its successor. We should be able to ascribe `inc` a *dependent function* type $x{:}\mathtt{nat} \rightarrow \mathtt{int}\{v{:}x < v\}$ that specifies that the function *ensures* that the output value exceeds the input x:

```
val inc: x:nat => int[v|x < v]
let inc = (x) => {
  let one = 1;
  add(x, one)
};
```

Alternatively, we can give `inc` the type $x{:}\mathtt{nat} \rightarrow \mathtt{pos}$. In this alternative, the result type does not depend on the argument, and as such, it is less precise, *e.g.* we can verify that $0 < \mathtt{inc}\ 6$, but not that $6 < \mathtt{inc}\ 6$.

Ex4: Function Calls All the functions that call `inc` should satisfy the requirement that they pass in a `nat`ural parameter. As an example, we present the function `inc2` below that calls `inc` with the predecessor of its input `y` and returns that result. We should be able to give `inc2` the type which states that the function *requires* that the input `y` is `pos`itive and *ensures* its output is also `pos`itive:

Base Types	b	$::=$	`int`	*integers*
Refinements	r	$::=$	$\{v\!:\!p\}$	*known*
Types	t,s	$::=$	$b\{r\}$	*refined base*
		$\mid$	$x\!:\!t \to t$	*dependent function*
Kinds	k	$::=$	B	*base kind*
		$\mid$	$\star$	*star kind*
Environments	Γ	$::=$	$\emptyset$	*empty*
		$\mid$	$\Gamma; x\!:\!t$	*variable-binding*

Figure 3.1: Syntax of Types and Environments

```
val inc2: y:pos => pos
let inc2 = (y) => {
  let one = 1;
  let y1  = sub(y, one);
  inc(y1)
};
```

3.2 Types and Terms

We summarize the language of types and terms for λ_ϕ in Fig. 3.1.

Types A *base* type b is an atomic primitive type, *e.g.* the set of all integers `int`. A *refinement* $\{v\!:\!p\}$ is a pair of a *value variable* v and a logical predicate p drawn from the SMT logic Fig. 2.1, *e.g.* $0 \leq v$. A *refined base type* $b\{v\!:\!p\}$ is a base type combined with a refinement *e.g.* `nat` shown in (3.1). A *function type* $x\!:\!s \to t$ comprises an input binder x of type s and an output type t in which x may appear (within a refinement), *e.g.* $x\!:\!\texttt{nat} \to \texttt{int}\{v\!:\!x < v\}$, the type assigned to `inc`.

Kinds We use a simple kind system to check whether a type is base, and hence, can be refined. The base kind B is given only to refined base types, while all types can have the kind $\star$.

Terms The different kinds of terms of λ_ϕ are summarized in Fig. 3.2. The simplest terms are *constants* `c` which include primitive values like $0, 1, 2, \ldots$ as well as arithmetic operators like *add* (addition), *sub* (subtraction), and so on. Next, we have *variables* x that are introduced

Terms	e	$::=$	c	*constants*
		$\mid$	x	*variables*
		$\mid$	$\mathbf{let}\ x = e\ \mathbf{in}\ e$	*let-binding*
		$\mid$	$\lambda x.\ e$	*functions*
		$\mid$	$e\ x$	*application*
		$\mid$	$e : t$	*type-annotation*

Figure 3.2: Syntax of Terms

via *let-binders* $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ that bind the value of e_1 to x before evaluating e_2 and *function* definitions $\lambda x.\ e$ that create functions with a parameter x that produce the value of e as the result. We can *apply* functions $e\ x$ where e is the function and x its argument.[1] Finally, the *annotation* form $e : t$ lets us ascribe the type t to the term e.

Abbreviations We write b to abbreviate $b\{v : true\}$. We abbreviate $b\{v : p\}$ to $\{v : p\}$ when b is clear from the context and to $b\{p\}$ when p does not refer to the binder v. For environment bindings $x : b\{v : p\}$ we assume that the environment and refinement binder names are the same, *i.e.* $x : b\{x : p\}$ and omit the refinement binder name to write $x : b\{p\}$.

3.3 Declarative Typing

We are now ready to look at the different judgments and rules that establish when a term e *has* type t.

3.3.1 Substitution

We use the notation $t[y := z]$ to denote the type where all free occurrences of y are substituted with the variable z. That is, substitution is defined in a capture avoiding manner:

$$
\begin{aligned}
b\{v : p\}[v := z] &\doteq b\{v : p\} \\
b\{v : p\}[y := z] &\doteq b\{v : p[y := z]\} \\
(x : s \to t)[x := z] &\doteq x : s[x := z] \to t \\
(x : s \to t)[y := z] &\doteq x : s[y := z] \to t[y := z]
\end{aligned}
$$

[1] We will explain why the syntax restricts arguments to variables in § 3.3.3.

Well-formedness $\boxed{\Gamma \vdash t : k}$

$$\frac{\Gamma; x{:}b \vdash p}{\Gamma \vdash b\{x{:}p\} : B}\textsc{Wf-Base} \qquad \frac{\Gamma \vdash s : k_s \qquad \Gamma; x{:}s \vdash t : k_t}{\Gamma \vdash x{:}s \rightarrow t : \star}\textsc{Wf-Fun}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \star}\textsc{Wf-Kind}$$

Figure 3.3: Well-formedness of types

3.3.2 Judgments

A *context* Γ is a sequence of variable-type bindings $x_1{:}t_1, \ldots, x_n{:}t_n$. The type system uses contexts to define five kinds of judgments.

Well-formedness judgments $\Gamma \vdash t : k$ state that in the context Γ the type t is *well-formed* with kind k, *i.e.*, that each refinement is a `bool`-valued proposition over variables bound in the context or type. Fig. 3.3 summarizes the rules that establish the well-formedness of types. The rule Wf-Base says that a base type $b\{x{:}p\}$ is well-formed with base kind in a context Γ if the refinement p is a well-sorted predicate in the context extended with x. Well-sortedness of predicates $\Gamma \vdash p$ is using the standard, unrefined type checking to check that the predicate p is boolean under the environment Γ with all refinements erased. The rule Wf-Fun says that a function type $x{:}s \rightarrow t$ is well-formed with star kind in a context Γ if the *input* type s is well-formed for some kind under Γ and the *output* type is well-formed for some kind under the context extended with the parameter x, thereby allowing refinements in the output to *depend upon* the inputs. The rule Wf-Kind says that any well formed type with base kind also has star kind.

The next two judgments formalize when the set of values of one type are *subsumed by* (*i.e.* contained within) another.

Entailment judgments $\Gamma \vdash c$ state that in the context Γ the logical constraint c is *valid i.e.*, “is true”. For example the entailment judgment

Entailment $\boxed{\Gamma \vdash c}$

$$\frac{\mathsf{SmtValid}(c)}{\emptyset \vdash c}\textsc{Ent-Emp} \qquad \frac{\Gamma \vdash \forall x{:}b.\ p \Rightarrow c}{\Gamma; x{:}b\{x{:}p\} \vdash c}\textsc{Ent-Ext}$$

Subtyping $\boxed{\Gamma \vdash t_1 <: t_2}$

$$\frac{\Gamma \vdash \forall v_1{:}b.\ p_1 \Rightarrow p_2[v_2 := v_1]}{\Gamma \vdash b\{v_1{:}p_1\} <: b\{v_2{:}p_2\}}\textsc{Sub-Base}$$

$$\frac{\Gamma \vdash s_2 <: s_1 \qquad \Gamma; x_2{:}s_2 \vdash t_1[x_1 := x_2] <: t_2}{\Gamma \vdash x_1{:}s_1 \rightarrow t_1 <: x_2{:}s_2 \rightarrow t_2}\textsc{Sub-Fun}$$

Figure 3.4: Entailment and Subtyping

$$x{:}\mathtt{int}\{0 \le x\} \vdash \forall y{:}\mathtt{int}.\ y = x + 1 \Rightarrow 0 \le y \tag{3.3}$$

reduces, via the rule Ent-Ext, to the *verification condition*

$$\forall x{:}\mathtt{int}.\ 0 \le x \Rightarrow \forall y{:}\mathtt{int}.\ y = x + 1 \Rightarrow 0 \le y \tag{3.4}$$

which, via the rule Ent-Emp, is deemed to be *valid* by an SMT solver.

Subtyping judgments $\Gamma \vdash t_1 <: t_2$ state that t_1 is a subtype of t_2 in a typing context Γ comprising a sequence of type bindings. The rule Sub-Base reduces subtyping on (refined) base types into an entailment. For example, the subtyping judgment

$$x{:}\mathtt{int}\{0 \le x\} \vdash \mathtt{int}\{y{:}y = x + 1\} <: \mathtt{int}\{v{:}0 \le v\} \tag{3.5}$$

reduces, via the rules Sub-Base to the entailment (3.3), which, as stated before, is deemed valid by SMT. The rule Sub-Fun decomposes subtyping on functions into *contra-variant* subtyping on the input types, and *co-variant* subtyping on the output types. For example, the subtyping judgment

$$\emptyset \vdash x{:}\mathtt{int} \rightarrow \mathtt{int}\{y{:}y = x + 1\} <: x{:}\mathtt{int}\{0 \le x\} \rightarrow \mathtt{int}\{v{:}0 \le v\} \tag{3.6}$$

Type Synthesis $\boxed{\Gamma \vdash e \rhd t}$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \rhd t}\textsc{Syn-Var} \qquad \frac{\mathsf{prim}(\mathsf{c}) = t}{\Gamma \vdash \mathsf{c} \rhd t}\textsc{Syn-Con}$$

$$\frac{\Gamma \vdash e \rhd x{:}s \to t \quad \Gamma \vdash y \lhd s}{\Gamma \vdash e\ y \rhd t[x := y]}\textsc{Syn-App} \qquad \frac{\Gamma \vdash t : k \quad \Gamma \vdash e \lhd t}{\Gamma \vdash e{:}t \rhd t}\textsc{Syn-Ann}$$

Figure 3.5: Bidirectional Typing: Synthesis

holds because it reduces to following judgments on the respective input and output types:

$$\emptyset \vdash \mathtt{int}\{x{:}0 \leq x\} <: \mathtt{int} \tag{3.7}$$

$$x{:}\mathtt{int}\{0 \leq x\} \vdash \mathtt{int}\{y{:}y = x + 1\} <: \mathtt{int}\{v{:}0 \leq v\} \tag{3.8}$$

The former holds trivially, as $\Gamma \vdash$ *true*. The latter subtyping judgment is the same as (3.5), and so, holds via the entailment (3.3).

Bidirectional Typing The next two kinds of judgments present typing in a *bidirectional* style (Pierce and Turner, 1998; Dunfield and Krishnaswami, 2020), where we separate the terms where types are *checked* from those for which the types are *synthesized*.

- ***Synthesis*** judgments $\Gamma \vdash e \rhd t$ state that the type t can be *generated* for the term e in the context Γ.

- ***Checking*** judgments $\Gamma \vdash e \lhd t$ state that a given type t is indeed *valid* for a term e in the context Γ, by *pushing* typing goals for terms into typing goals for sub-terms.

Let's take a closer look at the checking and synthesis rules.

3.3.3 Synthesis

Fig. 3.5 shows the rules for deriving synthesis judgments $\Gamma \vdash e \rhd t$ for terms e whose type can be generated from the context Γ.

Variables x synthesize the type ascribed to x in the context Γ (Syn-Var). For example, we can deduce that $\Gamma_0 \vdash x \triangleright \mathtt{nat}$ when

$$\Gamma_0 \doteq x{:}\mathtt{nat};\ one{:}\{one = 1\} \tag{3.9}$$

Constants c synthesize their "built-in" primitive type denoted by $\mathsf{prim}(\mathsf{c})$, which is usually the most precise reflection of the semantics of the constant that can be represented in the refinement logic (Syn-Con). For example, primitive `int` values like 0 and 1 are assigned the *singleton* types inhabited only by those values

$$\begin{aligned}\mathsf{prim}(0) &\doteq \mathtt{int}\{v{:}v = 0\}\\ \mathsf{prim}(1) &\doteq \mathtt{int}\{v{:}v = 1\}\end{aligned}$$

and arithmetic operators have primitive types that reflect their semantics

$$\begin{aligned}\mathsf{prim}(add) &\doteq x{:}\mathtt{int} \to y{:}\mathtt{int} \to \mathtt{int}\{v{:}v = x + y\}\\ \mathsf{prim}(sub) &\doteq x{:}\mathtt{int} \to y{:}\mathtt{int} \to \mathtt{int}\{v{:}v = x - y\}\end{aligned}$$

For exposition, we deliberately use ***add*** and ***sub*** for the primitive arithmetic operators of the language, to syntactically distinguish the names from + and − in the refinement logic.

Applications $e\ x$ synthesize the *output* type of e after substituting the input binder with the actual x (Syn-App). In the environment Γ_0 from (3.9), the term $add\ x\ one$, would synthesize the type

$$\Gamma_0 \vdash add\ x\ one \triangleright \mathtt{int}\{v{:}v = x + one\} \tag{3.10}$$

Applications & ANF The rule Syn-App returns the function's output type after substituting the input binder with the actual argument. We require that the application terms be in *Administrative Normal Form* (ANF) so that this substitution only replaces binders with other binders, and not arbitrary expressions (Flanagan *et al.*, 1993). This restriction ensures that all the intermediate refinements produced during type checking belong to the same (decidable) fragment of the refinement logic that the user-defined specifications are from. In particular, it prevents arbitrary terms from seeping into the refinements, which would complicate SMT-based subtyping. Knowles and Flanagan, 2009 propose an alternative dependent application rule that uses an *existential type* to ensure that terms do not seep into refinements:

$$\frac{\Gamma \vdash e_1 \triangleright x{:}s \rightarrow t \qquad \Gamma \vdash e_2 \triangleright s}{\Gamma \vdash e\ y \triangleright \exists x{:}s.t}\textsc{Syn-App-Ex}$$

The above rule ensures that the existentials only appear on the left-side of subtyping obligations, at which point they can simply be pulled into the environment, *i.e.* via a subtyping rule:

$$\frac{\Gamma; x{:}s \vdash t_1 \prec: t_2}{\Gamma \vdash \exists x{:}s.t_1 \prec: t_2}\textsc{Sub-Ex}$$

This existential-based rule requires an extra subtyping step, but has the benefit of not requiring terms to be in ANF which is problematic for the meta-theory, as the small-step evaluation does not preserve ANF structure. However, apart from the meta-theory, the two rules are equivalent and we pick ANF for ease of exposition and implementation.

For brevity and readability, we will present programs that do not follow the ANF structure and assume they are converted to ANF form before type checking.

Annotated terms $e{:}t$ synthesize the (annotated) type t, after ensuring that the annotation t is well-formed in the given context and checking that e indeed can be ascribed the type t (Syn-Ann).

3.3.4 Checking

Fig. 3.6 shows the rules for deducing checking judgments $\Gamma \vdash e \triangleleft t$. Typically, we use these judgments to verify that a λ-term has its given (annotated) type, and to push those top-level obligations inside let-binders to get localized obligations for the inner expressions.

Functions $\lambda x.\ e$ can be checked against the type $x{:}t_1 \rightarrow t_2$ in a context Γ if their bodies e can be checked against the output type t_2 in the environment extended by binding the parameter to the input type t_1 (Chk-Lam). For example, we check `inc` (from section 3.1) against its ascribed type

$$\emptyset \vdash \lambda x.\ \texttt{let}\ one = 1\ \texttt{in}\ add\ x\ one \triangleleft x{:}\texttt{nat} \rightarrow \texttt{int}\{v{:}x < v\}$$

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\frac{\Gamma; x{:}t_1 \vdash e \triangleleft t_2}{\Gamma \vdash \lambda x.\ e \triangleleft x{:}t_1 \rightarrow t_2}\textsc{Chk-Lam}$$

$$\frac{\Gamma \vdash e_1 \triangleright t_1 \qquad \Gamma; x{:}t_1 \vdash e_2 \triangleleft t_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \triangleleft t_2}\textsc{Chk-Let}$$

$$\frac{\Gamma \vdash e \triangleright s \qquad \Gamma \vdash s <: t}{\Gamma \vdash e \triangleleft t}\textsc{Chk-Syn}$$

Figure 3.6: Bidirectional Typing: Checking

by checking the body against the output type in the extended context

$$x{:}\mathtt{nat} \vdash \mathbf{let}\ one = 1\ \mathbf{in}\ add\ x\ one \triangleleft \mathtt{int}\{v{:}x < v\} \tag{3.11}$$

Let-bindings $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ can be checked against the type t_2 if we can check that e_2 has type t_2 in the environment extended by binding x to the type t_1 synthesized for e_1 (Chk-Let). In effect, this rule *pushes* the obligation t_2 for the whole expression, into an obligation for the let-body e_2. For example, the judgment (3.11) reduces to the below check that pushes the obligation inside:

$$x{:}\mathtt{nat};\ one{:}\{one = 1\} \vdash add\ x\ one \triangleleft \mathtt{int}\{v{:}x < v\} \tag{3.12}$$

That is, we must check the term *add x one* in the environment extended by binding the local *one* to the type synthesized from its (constant) expression 1.

Subsumption rule Chk-Syn weakens the synthesized type of a term to match against its required type: if the term e synthesizes a type s which is subsumed by t, then we can check e against t. For example, the judgment (3.12) that checks the body of `inc` (from section 3.1) is established by using Chk-Syn to yield the obligation

$$x{:}\mathtt{nat};\ one{:}\{one = 1\} \vdash \mathtt{int}\{v{:}v = x + one\} <: \mathtt{int}\{v{:}x < v\} \tag{3.13}$$

which checks that the type synthesized for ***add x one*** (3.10) is a subtype of the goal $\texttt{int}\{v : x < v\}$.

Readers familiar with Floyd-Hoare program logics may be reminded of the *consequence rule*

$$\frac{P \Rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

which allows us to strengthen the preconditions and weaken the post-conditions (via implication) for a command C similar to how Chk-Sym lets us weaken the type for a term e (via subtyping).

3.4 Verification Conditions

The typing rules show how refinement verification works in a *declarative* fashion: let's finish our discussion with a concrete *implementation* of a verifier. For readers familiar with program logics, the declarative rules are akin to Floyd-Hoare style rules. Instead, we will describe an implementation of a *verification condition* (VC) generator that takes as input a program annotated with refinement types and returns a VC, *i.e.* a constraint (§ 2.1.2) whose *validity* can (a) be determined by an SMT solver and (b) implies the typability of the program. In particular, we will describe the implementation of three functions, that are algorithmic counterparts of the respective typing judgments.

Implication Constraints We write $(x :: t) \Rightarrow c$ for the *implication* constraint which is defined as:

$$(x :: t) \Rightarrow c \doteq \begin{cases} \forall x : b.\ p[v := x] \Rightarrow c & \text{if } t \equiv b\{v : p\} \\ c & \text{otherwise} \end{cases}$$

For example, the implication contraint $(x :: \texttt{int}\{v : 0 < v\}) \Rightarrow 0 \leq x$ generates the valid verification condition $\forall x : \texttt{int}.\ 0 < x \Rightarrow 0 \leq x$. Note that the otherwise case omits all the non base types from the verification condition derivation. For instance, the implication constraint $(x :: y : t_1 \rightarrow t_2) \Rightarrow \mathit{false}$ generates the invalid verification condition *false*.

Subtyping $\texttt{sub}(s,\ t)$ summarized in Fig. 3.7, mirrors the subtyping rules Fig. 3.4. The function takes as input two types s and t and returns as output a constraint c whose validity implies that s is a subtype of t:

sub	:	$(T \times T) \rightarrow C$
$\mathsf{sub}(b\{v_1 : p_1\},\ b\{v_2 : p_2\})$	$\doteq$	$\forall v_1 : b.\ p_1 \Rightarrow p_2[v_2 := v_1]$
$\mathsf{sub}(x_1 : s_1 \rightarrow t_1,\ x_2 : s_2 \rightarrow t_2)$	$\doteq$	$c_i \wedge (x_2 :: s_2) \Rightarrow c_o$
where		
c_i	$=$	$\mathsf{sub}(s_2,\ s_1)$
c_o	$=$	$\mathsf{sub}(t_1[x_1 := x_2],\ t_2)$

Figure 3.7: Algorithmic Subtyping for λ_ϕ

Proposition 3.1. If $\mathsf{sub}(s,\ t) = c$ and $\Gamma \vdash c$, then $\Gamma \vdash s <: t$.

The two cases shown in Fig. 3.7 correspond directly to the rules Sub-Base and Sub-Fun from Fig. 3.4. For refined base types, the generated constraint states that sub-refinement p_1 implies the super-refinement p_2. For function types, we recursively invoke sub to conjoin the constraint on the input type and an implication constraint that checks the output subtyping holds assuming the stronger input type.

Example: Subtyping Constraint For example, let s and t respectively be the sub- and super-type in subtyping judgment (3.6). Then $\mathsf{sub}(s,\ t)$ returns the VC constraint:

$$\begin{aligned} &\forall x : \mathtt{int}.\ 0 \leq x \Rightarrow \mathit{true} \\ \wedge\ &\forall x : \mathtt{int}.\ 0 \leq x \Rightarrow \forall y : \mathtt{int}.\ y = x + 1 \Rightarrow 0 \leq y \end{aligned}$$

whose first and second conjuncts correspond to the input (3.7) and output (3.8) subtyping obligations respectively. □

Synthesis $\mathsf{synth}(\Gamma,\ e)$ summarized in Fig. 3.8, is the analogue of the synthesis rules from Fig. 3.5. The function takes as input a context Γ and a term e whose type we want to synthesize under Γ and returns a pair $(c,\ t)$ such that the validity of c implies that e synthesizes type t:

Proposition 3.2. If $\mathsf{synth}(\Gamma,\ e) = (c,\ t)$ and $\Gamma \vdash c$, then $\Gamma \vdash e \triangleright t$.

The cases for variables x and constants c simply return the types for the respective element by looking up the context Γ or the primitive type

synth	:	$(\Gamma \times E) \rightarrow (C \times T)$
$\mathsf{synth}(\Gamma,\ x)$	$\doteq$	$(\mathit{true},\ \Gamma(x))$
$\mathsf{synth}(\Gamma,\ \mathsf{c})$	$\doteq$	$(\mathit{true},\ \mathsf{prim}(\mathsf{c}))$
$\mathsf{synth}(\Gamma,\ e\ y)$	$\doteq$	$(c \wedge c',\ t[x := y])$
where		
$(c,\ x{:}s \rightarrow t)$	$=$	$\mathsf{synth}(\Gamma,\ e)$
c'	$=$	$\mathsf{check}(\Gamma,\ y,\ s)$
$\mathsf{synth}(\Gamma,\ e{:}t)$	$\doteq$	$(c,\ t)$
where		
c	$=$	$\mathsf{check}(\Gamma,\ e,\ t)$

Figure 3.8: Algorithmic Synthesis for λ_ϕ

respectively. In both these cases, the constraint (VC) is simply *true* as the synthesized types hold unconditionally as shown in SYN-VAR and SYN-CON from Fig. 3.5. For application terms $e\ y$, we recursively invoke synth to synthesize the type and VC for the function e, invoke check to generate a VC that holds if the argument y is of the expected input type, and then return the conjunction of the VCs for the function and argument, together with the function's output type. For annotated terms $e{:}t$ we generate the VC by invoking check, mimicking SYN-ANN. Consider the following environment that arises in the body of `inc2` (from section 3.1):

$$\Gamma_2 \ \doteq \mathit{inc}{:}x{:}\mathtt{nat} \rightarrow \mathtt{int}\{v{:}x < v\};\ y{:}\mathtt{nat};\ y_1{:}\{y_1 = y - 1\}$$

Here, $\mathsf{synth}(\Gamma_2,\ \mathit{inc}\ y_1)$ returns the VC and type

$$c \ \doteq \forall v{:}\mathtt{int}.\ v = y_1 \Rightarrow 0 \leq v$$
$$t \ \doteq \mathtt{int}\{v{:}y_1 < v\}$$

where the VC c checks that the input y_1 meets the required precondition of *inc* (*i.e.* is a `nat`), and the synthesized t is the output of *inc* with the arguments y_1 substituted for the formal x.

Checking $\mathsf{check}(\Gamma,\ e,\ t)$ summarized in Fig. 3.9, implements the checking judgment. The function takes as input a context Γ, a term e,

check	:	$(\Gamma \times E \times T) \rightarrow C$
$\mathsf{check}(\Gamma,\ \lambda x.\ e,\ x{:}s \rightarrow t)$	$\doteq$	$(x :: s) \Rightarrow c$
where		
c	$=$	$\mathsf{check}(\Gamma; x{:}s,\ e,\ t)$
$\mathsf{check}(\Gamma,\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2,\ t_2)$	$\doteq$	$c_1 \wedge (x :: t_1) \Rightarrow c_2$
where		
(c_1, t_1)	$=$	$\mathsf{synth}(\Gamma,\ e_1)$
c_2	$=$	$\mathsf{check}(\Gamma; x{:}t_1,\ e_2,\ t_2)$
$\mathsf{check}(\Gamma,\ e,\ t)$	$\doteq$	$c \wedge c'$
where		
(c, s)	$=$	$\mathsf{synth}(\Gamma,\ e)$
c'	$=$	$\mathsf{sub}(s,\ t)$

Figure 3.9: Algorithmic Checking for λ_ϕ

and a type t we want to check for e and returns as output a constraint c whose validity implies that e checks against t:

Proposition 3.3. If $\mathsf{check}(\Gamma,\ e,\ t) = c$ and $\Gamma \vdash c$, then $\Gamma \vdash e \triangleleft t$.

The first case for terms $\lambda x.\ e$ implements CHK-LAM from Fig. 3.6, by using check to generate the VC for the body e under the environment extended with the type for the formal x, which is added as an antecedent to the returned VC. The second case for terms $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$, follows CHK-LET to synthesize a VC c_1 and type t_1 for e_1, which is used to generate a VC c_2 for the body e_2. The VCs for the two terms are conjoined after weakening the body's with antecedent constraining the local binder x. The last case implements the subsumption rule CHK-SYN by generating a VC c and type s for e and then conjoining c with the VC stating s is subsumed by the (goal) super-type t.

***Example: VC for* `inc`** Let's see how check and synth yield a VC for `inc` from § 3.1. Let

$$\Gamma_0 \doteq x{:}\mathtt{nat};\ one{:}\{one = 1\}$$
$$t_0 \doteq \mathtt{int}\{v{:}x < v\}$$

The invocation $\mathsf{check}(\Gamma_0,\ \mathit{add}\ x\ \mathit{one},\ t_0)$ matches the last case (Chk-Syn) to return the VC

$$c_0 \doteq \forall v{:}\texttt{int}.\ v = x + \mathit{one} \Rightarrow x < v$$

The antecedent in c_0 comes from the (sub-) type synthesized by the call $\mathsf{synth}(\Gamma_0,\ \mathit{add}\ x\ \mathit{one})$ (3.10), and the consequent comes from the (super-) type corresponding to the goal t_0 (3.13). Next, let

$$\begin{aligned} \Gamma_1 &\doteq x{:}\texttt{nat} \\ e_1 &\doteq \textbf{let}\ \mathit{one} = 1\ \textbf{in}\ \mathit{add}\ x\ \mathit{one} \end{aligned}$$

The invocation $\mathsf{check}(\Gamma_1,\ e_1,\ t_0)$ matches the second case (*i.e.* Chk-Let) to return the VC

$$c_1 \doteq \forall \mathit{one}{:}\texttt{int}.\ \mathit{one} = 1 \Rightarrow c_0$$

obtained by recursively invoking check on the body to get c_0 and then weakening it with the type synthesized by synth for the binder *one*. Finally, $\mathsf{check}(\emptyset,\ \lambda x.\ e_1,\ x{:}\texttt{nat} \rightarrow t_0)$ matches the first case (*i.e.* Chk-Lam) to return the VC

$$c_2 \doteq \forall x{:}\texttt{int}.\ 0 \leq x \Rightarrow c_1$$

which is the VC for the body e_1 weakened with the type for the binder x. The constraint c_2 is the formula

$$\forall x{:}\texttt{int}.\ 0 \leq x \Rightarrow \forall \mathit{one}{:}\texttt{int}.\ \mathit{one} = 1 \Rightarrow \forall v{:}\texttt{int}.\ v = x + \mathit{one} \Rightarrow x < v$$

which is proved valid by the SMT solver, thereby verifying `inc`. □

3.5 Discussion

Before we move on to our next language feature let's ponder some key lessons learned from developing refinement types for λ_ϕ.

Primitive Types connect the semantics of terms and types. As in every typed language, programs with refinement types have *two* levels of semantics. First, the *dynamic* ("operational") semantics corresponding to how terms reduce to values. Second, the abstract or *static* ("logical") semantics that describe the sets of values that a term can reduce to. In refinement types, the types of the primitive constants *e.g.* prim(c) are the glue that connects the two semantics. We start by giving primitive constants like `3` and `4` very precise *singleton* types like $\texttt{int}\{v : v = 3\}$ that reflect their dynamic semantics in the refinement logic. Next, we give arithmetic operators like ***add*** similarly precise types like $x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \texttt{int}\{v : v = x + y\}$, and after this, the function application and subtyping suffice to logically (overapproximate) the sets of values that a term can reduce.

Arithmetic Overflow Consequently, we can use different specifications to more precisely track machine operations. For example, for simplicity of exposition, we give ***add*** the type that says that the returned integer is in fact the logical (mathematical) addition of the two arguments. Note that, this may not hold if `int` is implemented as 32- or 64- bit integers, which may *overflow*. However, it is quite straightforward to write more restrictive specifications for primitives like ***add*** such that verification statically guarantees the absence of arithmetic overflows [2].

Assumptions for Soundness Our system relies on the assumption that the primitives of the language satisfy their specified types prim(c). This assumption does not always hold. For example, returning to overflows, we can `increase` the "maximum" (fixed-width) integer by one (`inc(maxInt, 1)`) to get back a result than is *smaller* than the input, violating the specification of `inc`. Similarly, one can break the system when primitive operations, like addition, equality, comparisons, *etc.* do not satisfy the laws of the respective primitive logic operators. It is important to note that the goal of our refinement type system is not to validate these assumptions but instead, to verify more sophisticated properties on a programming model where these assumptions hold.

Types as Program Logics The development for λ_ϕ already shows that refinement types can be viewed as a generalization of Floyd-Hoare style program logics. Such logics typically have *monolithic* assertions that describe the entire state of the machine at a given program point. Types allow us to *decompose* those assertions into more fine-grained refinements on the values of individual terms. Similarly, pre- and post-

[2]See Ranjit Jhala's post "Arithmetic Overflows" for how to reason about arithmetic overflows.

conditions correspond directly to input- and output-types for functions. The function application rule *checks* pre-conditions (input types) and *assumes* the returned value satisfies the post-condition (output type). Dually, the function definition rule *assumes* the pre-conditions (input types) hold for the input parameters and *checks* that the returned value satisfies the post-condition (output type).

Higher-Order Contracts One immediate benefit of using types instead of monolithic assertions, is that they naturally scale up to handling higher-order functions, such as `incf` shown below.

```
val incf: x:nat => pos
let incf = (x) => {
  val tmp : f:(nat => nat) => pos
  let tmp = (f) => {
    add(f(x), 1)
  };
  tmp(inc)
};
```

The specification for `incf` says that if it is given a `nat` value `x` then it also returns a `nat`. To implement this contract, the function creates a higher-order `tmp` which increments the value returned by invoking its argument `f` on `x`. Types help in two ways. First, they let us *specify* a suitable contract for `f`, namely it takes a `nat` and returns one. Second, they let us *verify* the application `tmp(inc)` by using the function subtyping rule Sub-Fun to check that `inc` (section 3.1) whose type is $z{:}\texttt{int} \to \texttt{int}\{v{:}z < v\}$ is a valid input for `tmp`, hence verifying the call, and that `incf` returns a `pos`. While this example is contrived, we will see how this form of type-directed decomposition of the verification goals greatly simplifies the verification of programs with polymorphic data and higher-order functions[3].

[3] See Ranjit Jhala's post "Types vs. Floyd-Hoare Logic" on a comparison between types and Floyd-Hoare logic.

4

Branches and Recursion

The programs one can write in λ_ϕ are dreadfully predictable: they compute simple *arithmetic* expressions over their inputs. Next, let's study λ_β which enriches λ_ϕ with two constructs – conditional branching and recursion – that are essential to facilitate *computation*. Consequently, we will see how to extend typing and VC generation to account for these constructs to enable *path-sensitive* verification that precisely accounts for the conditions accumulated along the course of evaluation.

4.1 Examples

Let us get appetized with some examples that showcase the new features. Let's assume that λ_β has a new primitive type `bool` with two values `true` and `false`:

```
type bool = true | false
```

We will see how to support user-defined algebraic data types and pattern-matching in λ_δ (§ 7).

Example: Branches First, suppose that the two `bool` constants of λ_β are given the following refinement types that connect the values to the truth or falsehood of refinement predicates:

$$\mathsf{prim}(\mathsf{true}) \doteq \mathsf{bool}\{b{:}b\} \tag{4.1}$$

$$\mathsf{prim}(\mathsf{false}) \doteq \mathsf{bool}\{b{:}\neg b\} \tag{4.2}$$

That is the constants `true` and `false` map directly to the corresponding proposition in the refinement logic. Consider the function `not` that implements negation for `bool` values. We would like to verify the specification for `not`, `and` and `or`, that reflects their semantics in their types:

```
val not: x:bool => bool[b|b ⇔ ¬x]
let not = (x) => { if (x) { false } else { true } };

val and: x:bool => y:bool => bool[b|b ⇔ x ∧ y]
let and = (x, y) => { if (x) { y } else { false } };

val or: x:bool => y:bool => bool[b|b ⇔ x ∨ y]
let or = (x, y) => { if (x) { true } else { y } };
```

□

Example: Recursion Next, suppose that λ_β has primitive arithmetic comparison operators, analogous to `add` and `sub` from § 3.3.3

$$\mathsf{prim}(leq) \doteq x{:}\mathsf{int} \to y{:}\mathsf{int} \to \mathsf{bool}\{v{:}v \Leftrightarrow x \leq y\}$$

$$\mathsf{prim}(geq) \doteq x{:}\mathsf{int} \to y{:}\mathsf{int} \to \mathsf{bool}\{v{:}v \Leftrightarrow x \geq y\}$$

As we can now test `int` values, we ought to be able to write recursive functions, such as `sum` that takes as input a number n and returns the summation $0+1+\ldots+n$. Again, we would like to verify that the returned value is a `nat` that exceeds `n`:

```
val sum : n:int => nat[v|n <= v]
let rec sum = (n) => {
  let c = leq(n, 0);
  if (c) {
    0
  } else {
    let n1 = sub(n,1);
    let t1 = sum(n1);
    n + t1
  }
}
```

□

$$
\begin{array}{rrcll}
\textbf{\textit{Base Types}} & b & ::= & \ldots & \textit{from } \lambda_\phi \textit{ (§ 3.1)} \\
 & & | & \texttt{bool} & \textit{booleans} \\
\textbf{\textit{Terms}} & e & ::= & \ldots & \textit{from } \lambda_\phi \\
 & & | & \textbf{if } x \textbf{ then } e \textbf{ else } e & \textit{branches} \\
 & & | & \textbf{let rec } x = e : t \textbf{ in } e & \textit{recursion}
\end{array}
$$

Figure 4.1: Syntax of Types and Terms

4.2 Types and Terms

λ_β extends the syntax of λ_ϕ with a few extensions summarized in Fig. 4.1.

Types The syntax of types is extended with the base type `bool`.

Terms The syntax of terms is extended in two ways. First, λ_β introduces a *conditional* expression **if** x **then** e_1 **else** e_2, that evaluates to e_1 when x evaluates to *true* and to e_2 otherwise. We require that the condition be a variable x instead of an expression for reasons similar to that of the ANF conversion required for Syn-App from § 3.3.3. Second, we introduce a *recursive binder* expression **let rec** $x = e_1 : t_1$ **in** e_2, where the binder x may appear free in e_1. Unlike plain let-binders, we require that the types of such recursive binders be annotated with their type t_1.

4.3 Declarative Typing

Next, let us consider the rules used to determine whether a term e has a given type t. As before, we have four kinds of judgments. The rules for entailment and subtyping are exactly the same as for λ_ϕ. However, to support branching, path-sensitivity and recursion, we must extend the rules that establish the checking and synthesis judgments.

4.3.1 Checking

Conditionals and recursive binders are handled by the checking rules summarized in Fig. 4.2.

Conditionals **if** x **then** e_1 **else** e_2 can be checked to have the type t in a context Γ when x is a `bool` and *both* branches e_1 and e_2 can be

Type Checking $\boxed{\Gamma \vdash e \lhd t}$

$$\frac{y \text{ is fresh} \qquad \Gamma \vdash x \lhd \texttt{bool} \qquad \Gamma; y{:}\{\texttt{int}{:}x\} \vdash e_1 \lhd t \qquad \Gamma; y{:}\{\texttt{int}{:}\neg x\} \vdash e_2 \lhd t}{\Gamma \vdash \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \lhd t}\ \textsc{Chk-If}$$

$$\frac{\Gamma \vdash t_1 : k \qquad \Gamma; x{:}t_1 \vdash e_1 \lhd t_1 \qquad \Gamma; x{:}t_1 \vdash e_2 \lhd t_2}{\Gamma \vdash \textbf{let rec } x = e_1 : t_1 \textbf{ in } e_2 \lhd t_2}\ \textsc{Chk-Rec}$$

Figure 4.2: Bidirectional Checking: Other rules from λ_ϕ (Fig. 3.6)

checked to have type t (Chk-If). Unlike with classical type checking, we want to check e_1 (resp. e_2) in a context that is extended with the fact that x evaluated to `true` (resp. `false`). Without this extra information, we cannot, *e.g.*, establish that the body of `not` (section 4.1) returns a boolean b that is the logical negation of the input x. The rule Chk-If incorporates branch condition by binding a *fresh* variable y to a refinement that captures the value of the condition x. That is we check the "then" branch e_1 by extending the context with a binding $y{:}\{\texttt{int}{:}x\}$ that says that x is *true*. On the other hand, we check the "else" branch e_2 by extending the context with $y{:}\{\texttt{int}{:}\neg x\}$ which records the fact that e_2 is evaluated when x is `false`. The binder y is used only to capture the value of the condition and could, in theory, be of *any* type. In practice, since the binder y has no runtime information, we can give it a unit type, but here we give it an `int` type, since for simplicity our core language does not have unit type.

Example: Checking not Let's see how this method of *branch strengthening* allows us to check the implementation of `not` from section 4.1 against its specification. First, rule Chk-Lam gives us the following obligation where the context has only $x{:}\texttt{bool}$ from the specification for the input parameter x:

$$x{:}\texttt{bool} \vdash \textbf{if } x \textbf{ then } \texttt{false} \textbf{ else } \texttt{true} \Leftarrow \texttt{bool}\{b{:}b \Leftrightarrow \neg x\}$$

Chk-If splits the above into two obligations, for the then- and else-branch respectively:

$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{x\} \vdash \texttt{false} \Leftarrow \texttt{bool}\{b\!:\!b \Leftrightarrow \neg x\}$$
$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{\neg x\} \vdash \texttt{true} \Leftarrow \texttt{bool}\{b\!:\!b \Leftrightarrow \neg x\}$$

Chk-Syn and Syn-Con, using the constant types for `true` and `false`, reduce the above to the respective subtyping obligations:

$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{x\} \vdash \texttt{bool}\{b\!:\!\neg b\} <: \texttt{bool}\{b\!:\!b \Leftrightarrow \neg x\}$$
$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{\neg x\} \vdash \texttt{bool}\{b\!:\!b\} <: \texttt{bool}\{b\!:\!b \Leftrightarrow \neg x\}$$

Sub-Base boils the above down to the subsumption checks:

$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{x\} \vdash \forall b\!:\!\texttt{bool}.\ \neg b \Rightarrow b \Leftrightarrow \neg x$$
$$x\!:\!\texttt{bool};\ y\!:\!\texttt{int}\{\neg x\} \vdash \forall b\!:\!\texttt{bool}.\ b \Rightarrow b \Leftrightarrow \neg x$$

Ent-Ext turns these into the VCs that are proved valid by SMT:

$$\forall x\!:\!\texttt{bool}.\ \mathit{true} \Rightarrow \forall y\!:\!\texttt{int}.\ x \Rightarrow \forall b\!:\!\texttt{bool}.\ \neg b \Rightarrow b \Leftrightarrow \neg x \tag{4.3}$$
$$\forall x\!:\!\texttt{bool}.\ \mathit{true} \Rightarrow \forall y\!:\!\texttt{int}.\ \neg x \Rightarrow \forall b\!:\!\texttt{bool}.\ \neg b \Rightarrow b \Leftrightarrow \neg x \tag{4.4}$$

Note that validity depends crucially on the hypotheses x and $\neg x$ introduced by branch strengthening. Without those, the VCs would be invalid and hence `not` would fail to typecheck. □

Recursive binders $\textbf{let rec}\ x = e_1 : t_1\ \textbf{in}\ e_2$ have type t_2 in a context Γ if in the context where x is also *assumed* to have type t_1, (1) the recursive term e_1 can be *guaranteed* to have type t_1 and (2) the body e_2 can be checked to have type t_2. Note that λ_β requires explicit type annotations for recursive binders to facilitate bidirectional checking, so the rule Chk-Rec additionally checks that the annotation t_1 is well-formed in Γ. (While top-level signatures are invaluable for design and documentation, we will see how they may be elided via refinement inference in chapter 5.)

Example: Checking **`sum`** Let's see how the assume-guarantee method allows us to verify the implementation of `sum`. Let us introduce a few abbreviations:

$$t_s \doteq \texttt{int}\{v\!:\!0 \le v \wedge n \le v\} \tag{4.5}$$
$$\Gamma_s \doteq \mathit{sum}\!:\!n\!:\!\texttt{int} \rightarrow t_s;\ n\!:\!\texttt{int} \tag{4.6}$$
$$e_s \doteq \text{the body of } \texttt{sum} \tag{4.7}$$

Chk-Rec and then Chk-Lam get the ball rolling by yielding the checking obligation:

$$\Gamma_s \vdash e_s \Leftarrow t_s$$

Chk-Let and then Chk-If split the above into two obligations:

$$\Gamma_s; c:\{c \Leftrightarrow n \leq 0\}; y:\{c\} \vdash 0 \Leftarrow t_s \tag{4.8}$$

$$\Gamma_s; c:\{c \Leftrightarrow n \leq 0\}; y:\{\neg c\} \vdash e_2 \Leftarrow t_s \tag{4.9}$$

where e_2 is the **`else`**-branch of the body of `sum`. Chk-Syn, Syn-Con, Sub-Base and Ent-Ext turn (4.8) to the valid VC:

$$\forall n, c, y, v.\ (c \Leftrightarrow n \leq 0) \Rightarrow c \Rightarrow (v = 0) \Rightarrow (0 \leq v \wedge n \leq \overset{\rightarrow}{)} \tag{4.10}$$

Chk-Let reduces the judgment (4.9) for the **`else`** branch to the following subtyping obligation that checks that the value of `n + t1` is indeed a subtype of the return t_s:

$$\Gamma_s' \ \vdash \mathtt{int}\{v : v = n + t_1\} <: t_s \tag{4.11}$$

where the context Γ_s' is Γ extended with the **`else`**-branch condition and bindings for n_1 and t_1:

$$\begin{aligned}\Gamma_s' \ \doteq\ & \Gamma_s; c:\{c \Leftrightarrow n \leq 0\}; y:\{\neg c\};\\ & n_1:\{n_1 = n - 1\}; t_1:\{0 \leq t_1 \wedge n_1 \leq t_1\}\end{aligned}$$

In Γ_s', the binding for n_1 is the output type of `sub` with formals replaced with n and 1. Similarly, the binding for t_1 is the (assumed) output type of `sum`, with the formal n replaced with the actual parameter t_1. Sub-Base and Ent-Ext reduce the above subtyping (4.11) to the VC:

$$\begin{aligned}&\forall n, c, y, n_1, t_1, v.\\ &\quad (c \Leftrightarrow n \leq 0) \Rightarrow (\neg c) \Rightarrow (n_1 = n - 1) \Rightarrow (0 \leq t_1 \wedge n_1 \leq t_1) \Rightarrow (v = n + t_1)\\ &\quad \Rightarrow (0 \leq v \wedge n \leq \overset{\rightarrow}{)}\end{aligned} \tag{4.12}$$

which the SMT solver proves valid, guaranteeing that `sum` indeed meets its given specification. □

self	:	$(X \times T) \to T$
$\mathsf{self}(x, b\{v : p\})$	$\doteq$	$b\{v : p \wedge v = x\}$
$\mathsf{self}(x, t)$	$\doteq$	t

Figure 4.3: Selfification: Singleton Type Strengthening

4.3.2 Synthesis

Both the new language constructs in λ_β, *i.e.* branches and recursive binders have checking judgments. However, to precisely use the information gleaned from branch conditions, to enable path-sensitive verification, we will need to modify the rule for *variables*.

Example: Path Sensitivity The function `abs` returns the absolute value of its input `x`.

```
val abs : x:int => nat[v|x ≤ v]
let abs = (x) => {
  let c = leq(0, x);
  if (c) {
    x
  } else {
    sub(0, x)
  };
};
```

However, note that the **then** branch is simply `x`. If we directly applied Syn-Var from Fig. 3.5, then the synthesized type would be `int`, the type that `x` is bound to in the context, which is clearly not a subtype of `nat`! Hence, we require another way to type the variable lookup `x` in a manner that is precise enough to let us use the branch condition to prove that the result is a `nat`. □

Variables and Selfification This problem can be solved by the so-called *selfification* rule introduced by Ou *et al.*, 2004 where the variable x is given a singleton type $b\{v : v = x\}$, *i.e.* where the refinement says the value is equal to x. We formalize this idea in the updated Syn-Var in Fig. 4.4. In context Γ, the type synthesized for x is $\mathsf{self}(x, t)$, where t is what x is bound to in Γ. Fig. 4.3 summarizes definition of self.

Type Synthesis $\boxed{\Gamma \vdash e \rhd t}$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \rhd \mathsf{self}(x, t)}\textsc{Syn-Var}$$

Figure 4.4: Bidirectional Synthesis: Other rules from λ_ϕ (Fig. 3.6)

When invoked on a base type b – for which equality is defined in the refinement logic – the function strengthens the refinement p with the singleton conjunct $v = x$. When invoked on other (*e.g.* function) types, the input is returned as is.

***Example: Selfification in* `abs`** Let's use selfification to verify `abs`. Let

$$\Gamma_a \ \doteq\ x{:}\mathtt{int};\ c{:}\{c \Leftrightarrow 0 \leq x\}$$

As we've seen before with `not` and `sum`, Chk-Lam, Chk-Let and Chk-If produce the two obligations for the **`then`** and **`else`** branches:

$$\Gamma_a;\ y{:}\{c\} \vdash x \Leftarrow \mathtt{int}\{v{:}0 \leq v \wedge x \leq v\}$$
$$\Gamma_a;\ y{:}\{\neg c\} \vdash (\mathit{sub}\ \mathtt{0}\ x) \Leftarrow \mathtt{int}\{v{:}0 \leq v \wedge x \leq v\}$$

Chk-Syn, Syn-Var and Syn-App reduce the above to, respectively:

$$\Gamma_a;\ y{:}\{c\} \vdash \mathtt{int}\{v{:}v = x\} <: \mathtt{int}\{v{:}0 \leq v \wedge x \leq v\}$$
$$\Gamma_a;\ y{:}\{\neg c\} \vdash \mathtt{int}\{v{:}v = 0 - x\} <: \mathtt{int}\{v{:}0 \leq v \wedge x \leq v\}$$

Finally, Sub-Base and Ent-Ext establish the above by verifying the validity of the VCs:

$$\forall x, c, y, v.\ (c \Leftrightarrow 0 \leq x) \Rightarrow c \Rightarrow (v = x) \Rightarrow (0 \leq v \wedge x \leq v) \tag{4.13}$$
$$\forall x, c, y, v.\ (c \Leftrightarrow 0 \leq x) \Rightarrow \neg c \Rightarrow (v = 0 - x) \Rightarrow (0 \leq v \wedge x \leq v) \tag{4.14}$$

Notice that the hypothesis $v = x$ obtained from selfification is essential for the validity of the first (**`then`**) VC; without it, *i.e.* if we simply gave the term x its type from the context `int`, we would get the *invalid* VC

$$\forall x, c, y, v.\ (c \Leftrightarrow 0 \leq x) \Rightarrow c \Rightarrow \mathit{true} \Rightarrow (0 \leq v \wedge x \leq v)$$

which would make us foolishly reject `abs`. □

4.4 Verification Conditions

Next, let's implement the checking and synthesis rules as a pair of VC generation functions check and synth.

Checking Recall that $\mathsf{check}(\Gamma,\ e,\ t)$ returns the VC c whose validity implies that $\Gamma \vdash e \triangleleft t$ holds (proposition 3.3). Fig. 4.5 summarizes the new cases of check for λ_β.

For conditional expressions $\mathtt{\textbf{if}}\ x\ \mathtt{\textbf{then}}\ e_1\ \mathtt{\textbf{else}}\ e_2$ the VC is the conjunction of the VCs c_1 and c_2 that are generated by invoking check on the branches e_1 and e_2 respectively, and then conditioning the result to track whether the branch condition was true ($\{x\}$) or false ($\{\neg x\}$). For example, $\mathsf{check}(\emptyset,\ e_n,\ t_n)$ where e_n and t_n are respectively the implementation and specification of not (from section 4.1) returns the VC constraint which is the conjunction of the two VCs (4.3, 4.4):

$$\forall x, y, b.\ (x \Rightarrow \neg b \Rightarrow (b \Leftrightarrow \neg x))\ \wedge\ (\neg x \Rightarrow b \Rightarrow (b \Leftrightarrow \neg x))$$

For recursive binders $\mathtt{\textbf{let rec}}\ x = e_1 : t_1\ \mathtt{\textbf{in}}\ e_2$ the VC is the conjunction of the VCs obtained for e_1 and e_2, both generated using the environment Γ extended by binding x to its specified type t_1. For example, $\mathsf{check}(\Gamma_s,\ e_s,\ t_s)$, where Γ_s, e_s and t_s are the environment, body and output type of sum as shown in (4.6), (4.7), (4.5), yields the following VC, which is the conjunction of the two VCs (4.10, 4.12)

$$\begin{aligned}
&\forall n, c, y, n_1, t_1, v.\ (c \Leftrightarrow n \leq 0) \Rightarrow \\
&\quad (c \Rightarrow v = 0 \Rightarrow (0 \leq v \wedge n \leq v)) \wedge \\
&\quad (\neg c \Rightarrow n_1 = n - 1 \Rightarrow (0 \leq t_1 \wedge n_1 \leq t_1) \Rightarrow v = n + t_1 \Rightarrow (0 \leq v \wedge n \leq v))
\end{aligned}$$

Synthesis The synthesis function $\mathsf{synth}(\Gamma,\ e)$ returns a pair of a VC c and type t such that the validity of c implies that $\Gamma \vdash e \triangleright t$ holds (proposition 3.2). Fig. 4.5 summarizes the updated case for variable lookup using selfification. Here, the generated VC is trivial *i.e. true*, but the synthesized type is $\mathsf{self}(x, t_x)$ where t_x is the type that x is bound to in the context Γ. For example, using the updated selfified version of synth, the invocation of $\mathsf{check}(\emptyset,\ e_a,\ t_a)$ — where e_a and t_a are the implementation and specification of abs — yields the following

check	:	$(\Gamma \times E \times T) \rightarrow C$
$\mathsf{check}(\Gamma,\ \mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2,\ t)$	$\doteq$	$c_1 \wedge c_2$
where		
c_0	$=$	$\mathsf{check}(\Gamma,\ x,\ \mathsf{bool})$
c_1	$=$	$(y :: \mathsf{int}\{x\}) \Rightarrow \mathsf{check}(\Gamma,\ e_1,\ t)$
c_2	$=$	$(y :: \mathsf{int}\{\neg x\}) \Rightarrow \mathsf{check}(\Gamma,\ e_2,\ t)$
y	$=$	fresh binder
$\mathsf{check}(\Gamma,\ \mathbf{let\ rec}\ x = e_1 : t_1\ \mathbf{in}\ e_2,\ t)$	$\doteq$	$c_1 \wedge c_2$
where		
c_1	$=$	$(x :: t_1) \Rightarrow \mathsf{check}(\Gamma_1,\ e_1,\ t_1)$
c_2	$=$	$(x :: t_1) \Rightarrow \mathsf{check}(\Gamma_1,\ e_2,\ t)$
Γ_1	$=$	$\Gamma; x : t_1$
. . . plus cases from λ_ϕ (*Fig.* 3.9)		
synth	:	$(\Gamma \times E) \rightarrow (C \times T)$
$\mathsf{synth}(\Gamma,\ x)$	$\doteq$	$(\mathit{true},\ \mathsf{self}(x, t_x))$
where		
t_x	$=$	$\Gamma(x)$
. . . plus cases from λ_ϕ (*Fig.* 3.8)		

Figure 4.5: Algorithmic Checking for λ_β

VC which is the conjunction of the **then** and **else** VCs (4.13, 4.14)

$$\forall x, c, y, v.\ (c \Leftrightarrow 0 \leq x) \Rightarrow (c \Rightarrow v = x \Rightarrow (0 \leq v \wedge x \leq v)) \wedge$$
$$(\neg c \Rightarrow v = 0 - x \Rightarrow (0 \leq v \wedge x \leq v))$$

4.5 Discussion

At this point, we have seen enough to write refinement type checkers for interesting languages, with functions, branching and recursion. Let's glance back at the mechanisms that make verification tick in λ_β.

Recursion via Assume-Guarantee Reasoning First, we account for recursion using the classic *assume-guarantee* method where, to check **let rec** $x = e_1 : t_1$ **in** e_2 we *assume* that the recursive binder x has the type t_1, and then, *guarantee* that fact by checking its implementation e_1 against t_1. As in classical Floyd-Hoare logic, this only gives us a so-called *partial* correctness guarantee; we will look at verifying *total* correctness later in chapter 9.

Path-Sensitivity via Branch Strengthening We incorporate path-sensitive reasoning in conditional expressions **if** x **then** e_1 **else** e_2 by introducing a fresh variable (*i.e.* y in CHK-IF) and binding it to a refinement that states that the condition x is true (resp. false) when check the **then** branch e_1 (resp. **else** branch e_2). We will generalize this strategy to account for user-defined data-types and pattern matching in chapter 7.

Occurrence Typing via Selfification Finally, the presence of branches allows binders to have strong or more precise types under branches (as in `abs`). We account for this form of path-sensitive strengthening by updating the variable lookup rule SYN-VAR with *selfification* which says the type of x is a singleton whose value equals x (Ou *et al.*, 2004). This method, dubbed "occurrence typing" by Komondoor *et al.*, 2005 and Tobin-Hochstadt and Felleisen, 2008, allows us to then use the rest of the refinement typing machinery (*e.g.* branch strengthening) to precisely type each *occurrence* of a variable x under different branches.

5

Refinement Inference

Bidirectional typing's separate checking and synthesis modes ensure that the programmer need only write type signatures for functions, after which the refinement checker can synthesize the types of intermediate sub-expressions to produce verification conditions for the SMT solver to validate. However, as Pierce and Turner, 1998 observe, to make higher-order programming pleasant, we will want to spare the programmer the tedium of having to type local function definitions like those passed as arguments to `map` or `fold`. Similarly, to make type polymorphism (§ 6) usable, we want to avoid cluttering the code with explicit (refinement) type annotations at polymorphic instantiation sites. Thus, let's study λ_κ which extends λ_β with a mechanism for *inferring* refinements via the following strategy.

- ***Step 1: Types to Templates*** First, we generalize refinement type signatures to allow them to contain *holes* denoting unknown refinements. Type checking begins by replacing these holes with *Horn Variables* that represent the unknown refinements.

- ***Step 2: VCs to Horn Constraints*** Second, we run the VC generation procedures as described in the preceding chapters. Now, however, these procedures yield *Horn Constraints* which are VCs containing Horn (Variable) applications in addition to predicates.

- ***Step 3: VC Validity to Horn Solving*** Third, instead of asking an SMT solver to determine the validity of a VC, we will invoke a Horn Solver that repeatedly queries an SMT solver in a *fixpoint* computation that determines whether there are refinements that can be substituted for the Horn variables that make the resulting VCs valid.

5.1 Examples

Before plunging into the formal details of λ_κ, let's build up some intuition by studying how the three-step strategy plays out on an example.

Encoding Assertions Many languages have an *assertion* statement which allows the programmer to test, typically at run-time, that some condition holds and to halt execution otherwise. The following `assert` function allows the programmer to write such *assertions* but the refined (input) type ensures that any client that calls `assert(cond)` only typechecks if the refinement type checker can verify that `cond` *always* evaluates to `true` at run-time.

```
val assert : bool[b|b] => int
let assert = (b) => { 0 };
```

Recall the `abs` function from § 4.3.2 whose type signature has been deliberately elided

```
let abs = (x) => {
  let c = leq(0, x);
  if (leq(0, x)) { x } else { sub(0, x) };
};
```

Finally, consider `main` that calls `abs` and `assert` s that the returned value is non-negative:

```
val main : int => int
let main = (y) => {
  let z  = abs(y);
  let c  = leq(0, z);
  assert(c)
}
```

Recap: Verification Conditions Suppose that we are given a type signature for `abs`, for example

```
val abs : x:int => int[v|0 ≤ v]
```

Then the type checker from chapter 4 would produce the VC:

$$
\begin{array}{llllll}
 & \forall x, c, v. \quad (c \Leftrightarrow 0 \leq x) & \Rightarrow c & \Rightarrow v = x & \Rightarrow 0 \leq v & (a) \\
 & & \wedge \neg c & \Rightarrow v = 0 - x & \Rightarrow 0 \leq v & (b) \\
\wedge & \forall y, z, c, b. \quad 0 \leq z \Rightarrow (c \Leftrightarrow 0 \leq z) \Rightarrow (b \Leftrightarrow c) & & & \Rightarrow b & (c)
\end{array}
$$

The first two conjuncts of the VC arise from verifying that the implementation of `abs` satisfies the output the above signature, *i.e.* its specified post-condition. The conjuncts respectively state that in the **then** (conjunct (a)) and **else** (conjunct (b)) branches, the output value v must be non-negative. The last conjunct comes from checking the call to `assert` *i.e.* verifying that the boolean value `c` that `assert` is invoked on, is indeed always `true`. Note that the third conjunct *uses* (1) the output type of `abs` to assume that `z` is non-negative, (2) the output type of the primitive `leq` which we typed as (§ 4.1)

$$\mathsf{prim}(leq) \doteq x{:}\mathtt{int} \rightarrow y{:}\mathtt{int} \rightarrow \mathtt{bool}\{v{:}v \Leftrightarrow x \leq y\}$$

to assume that `c` is `true` if and only if `z` is non-negative. The above assumptions suffice to prove that the `assert`'s input `b` (which at this call-site is `c`) is indeed always `true`.

5.1.1 Step 1: Holes and Templates

Suppose that we wrote the following specification where ⋆ denotes a *refinement-hole*: an unknown refinement that we want to *infer*

```
val abs : x:int => int[★]
```

Readers may be reminded of Haskell's notion of a *type-hole* which allows programmers to partially specify type signatures that can then be automatically filled by type inference (Winant *et al.*, 2014). The key trick to filling refinement holes is to generalize VCs to constraints containing Horn applications that represent unknown refinements. To do so, every type signature with a hole is transformed into a *template* containing (distinct) Horn variables that represent the unknown refinements.

***Example: Template for* `abs`** For example, the signature for `abs` yields the template:

$$\texttt{abs} \ : \ x{:}\texttt{int} \rightarrow \texttt{int}\{v{:}\kappa(x, v)\} \tag{5.1}$$

where κ is a *Horn variable* such that $\kappa(z_1, z_2)$ denotes an (unknown) refinement (relation) over the Horn variable's *parameters* z_1 and z_2. □

5.1.2 Step 2: Horn Constraints

Next, we use the templates to run exactly the same VC generation procedure as before. However, instead of producing a VC we get a *Horn* constraint which is a VC with Horn variable applications appearing at various positions.

***Example: Constraints for* `abs`** For example, if we run `check` on the above code with `abs` and `main`, but using the template (5.1) as the specification for `abs`, we get the Horn constraint:

$$\begin{array}{llllll}
 & \forall x, c, v. & (c \Leftrightarrow 0 \leq x) \ \Rightarrow c & \Rightarrow v = x & \Rightarrow \kappa(x, v) & (a') \\
 & & \qquad\qquad\qquad\quad \wedge \neg c & \Rightarrow v = 0 - x & \Rightarrow \kappa(x, v) & (b') \\
\wedge & \forall y, z, c, b. & \kappa(y, z) \Rightarrow (c \Leftrightarrow 0 \leq z) \Rightarrow (b \Leftrightarrow c) & & \Rightarrow b & (c')
\end{array} \tag{5.2}$$

Notice that this constraint is mostly identical to the VC shown above, with three conjuncts (a), (b) and (c), except that instead of: (1) the consequent $0 \leq v$ that appears in the conjuncts (a) and (b) stipulating that the output value v is non-negative, we have the Horn application $\kappa(x, v)$ representing the output v is related to the input x via an (as yet not known) refinement κ, and (2) the assumption $0 \leq z$ that appears as a hypothesis in (c) stating that z is non-negative, we have the Horn application $\kappa(y, z)$ that says that the value of z is related to that of (the argument) y by the as yet unknown refinement κ. □

5.1.3 Step 3: Horn Solving

At this point, we cannot ask an SMT solver to simply check the validity of a VC, as the constraints contain unknown Horn relations. Instead we invoke a *Horn solver* to determine whether *there exist* a satisfying assignment for the Horn variables. A Horn *assignment* is a mapping of Horn variables to refinement predicates over the Horn variables' parameters. An assignment *satisfies* a Horn constraint if the result of substituting the Horn variables with their assignments yields a *valid* (Horn-variable free) formula.

Example: Solution for **`abs`** For example σ_1, σ_2 and σ_3 are three possible assignments for κ:

$$\sigma_1(\kappa)(z_1, z_2) \doteq z_1 \leq z_2 \tag{5.3}$$
$$\sigma_2(\kappa)(z_1, z_2) \doteq 0 < z_2 \tag{5.4}$$
$$\sigma_3(\kappa)(z_1, z_2) \doteq 0 \leq z_2 \tag{5.5}$$

The assignment σ_1 (5.3) does not satisfy the Horn constraint (5.2) as substitution yields the following VC whose last conjunct is invalid:

$$\begin{array}{llllll}
 & \forall x, c, v. & (c \Leftrightarrow 0 \leq x) & \Rightarrow c \quad \Rightarrow v = x & \Rightarrow x \leq v & (\checkmark) \\
 & & & \wedge \neg c \quad \Rightarrow v = 0 - x & \Rightarrow x \leq v & (\checkmark) \\
\wedge & \forall y, z, c, b. & \multicolumn{2}{l}{y \leq z \Rightarrow (c \Leftrightarrow 0 \leq z) \Rightarrow (b \Leftrightarrow c)} & \Rightarrow b & (\times)
\end{array}$$

Assignment σ_2 (5.4) also fails to satisfy the constraint (5.2) as substituting it yields the following VC whose first conjunct is invalid:

$$\begin{array}{llllll}
 & \forall x, c, v. & (c \Leftrightarrow 0 \leq x) & \Rightarrow c \quad \Rightarrow v = x & \Rightarrow 0 < v & (\times) \\
 & & & \wedge \neg c \quad \Rightarrow v = 0 - x & \Rightarrow 0 < v & (\checkmark) \\
\wedge & \forall y, z, c, b. & \multicolumn{2}{l}{0 < z \Rightarrow (c \Leftrightarrow 0 \leq z) \Rightarrow (b \Leftrightarrow c)} & \Rightarrow b & (\checkmark)
\end{array}$$

However, assignment σ_3 (5.5) *does* satisfy the Horn constraint (5.2) as substitution produces the *valid* VC

$$\begin{array}{llllll}
 & \forall x, c, v. & (c \Leftrightarrow 0 \leq x) & \Rightarrow c \quad \Rightarrow v = x & \Rightarrow 0 \leq v & (\checkmark) \\
 & & & \wedge \neg c \quad \Rightarrow v = 0 - x & \Rightarrow 0 \leq v & (\checkmark) \\
\wedge & \forall y, z, c, b. & \multicolumn{2}{l}{0 \leq z \Rightarrow (c \Leftrightarrow 0 \leq z) \Rightarrow (b \Leftrightarrow c)} & \Rightarrow b & (\checkmark)
\end{array}$$

Thus, we can fill the refinement-holes with the satisfying assignment to infer signatures that yield a well-typed program. For example, plugging σ_3 into the template for `abs` yields the "hand-written" signature

$$\texttt{abs} \ : \ x{:}\texttt{int} \rightarrow \texttt{int}\{v{:}0 \leq v\} \tag{5.6}$$

that let us verify `main`. □

Predicates	p	$::=$	$\ldots$	*from Fig. 2.1*
		$\mid$	$\kappa(\overline{x})$	*Horn application*
Refinements	r	$::=$	$\{\nu : p\}$	*known*
		$\mid$	$\{\star\}$	*hole*

Figure 5.1: Syntax of Predicates and Refinements

5.2 Types and Terms

Next, we formalize our three-step strategy in λ_κ whose syntax is summarized in Fig. 5.1.

Predicates We extend the grammar of refinement predicates (Fig. 2.1) to include Horn applications of the form $\kappa(\overline{x})$ where $\overline{x}$ abbreviates a sequence of variables $x_1, \ldots, x_n$. A Horn application $\kappa(\overline{x})$ denotes an *unknown* predicate (or relation) over the variables $\overline{x}$.

Refinements Thus, λ_κ has two kinds of refinements. The first are *known* refinements (or just, refinements) $\{\nu : p\}$, made up of predicates p as before. The second are refinement *holes* (or just, holes) $\{\star\}$ which can appear in type annotations, and which denote an unknown refinement that the programmer has chosen to elide.

Holes vs. Horn applications In λ_κ, Horn-applications do not appear in the *external* surface syntax, *i.e.* in type annotations. Instead, the programmer elides refinements in annotations using holes. During type checking we will replace all holes with Horn applications. That is, dually, holes do not appear in the *internal* typing derivations.

5.3 Declarative Typing

Next, let's see how a term e that may be annotated with refinement holes $\{\star\}$ can be verified to have a type t. The vast majority of the rules, in particular, the rules for well-formedness, subtyping and entailment, remain unchanged from λ_β. However, we will introduce a new *instantiation* judgment that stipulates how holes can be filled by refinements. We will then use the instantiation judgment to eliminate holes in the two rules that pertain to type annotations.

Hole Instantiation $\boxed{s \triangleright t}$

$$\frac{}{b\{\star\} \triangleright b\{v\!:\!p\}}\textsc{Ins-Hole} \qquad \frac{}{b\{v\!:\!p\} \triangleright b\{v\!:\!p\}}\textsc{Ins-Conc}$$

$$\frac{s_1 \triangleright s_2 \quad t_1 \triangleright t_2}{x\!:\!s_1 \to t_1 \triangleright x\!:\!s_2 \to t_2}\textsc{Ins-Fun}$$

Figure 5.2: Hole Instantiation

5.3.1 Instantiation

The instantiation judgment $s \triangleright t$ states that a type s can be *instantiated to* a type t by replacing the refinement holes in s with suitable concrete refinements. This intuition is formalized by the three rules summarized in Fig. 5.2: Ins-Hole, which describes how a single hole is instantiated, Ins-Conc, which states the concrete refinements are left unmodified, and Ins-Fun, which describes the component-wise instantiation of function types.

Let's focus on two important aspects of the instantiation judgment. First, the rules ensure that if $s \triangleright t$ then there are no holes left in t. Second, the judgment is *declarative*, it does not tell us *how* to find suitable concrete refinements. Instead the rules tell us *what* valid concrete refinements should look like for the program to be well-typed.

***Example: Instantiating holes in* `abs`** The above rules establish that

$$x\!:\!\texttt{int} \to \texttt{int}\{\star\} \quad \triangleright \quad x\!:\!\texttt{int} \to \texttt{int}\{v\!:\!0 \leq v\}$$

i.e. the partial type signature for `abs` from (5.1) can be instantiated to the concrete type in (5.6). □

5.3.2 Checking

We need only alter the *checking* rules in one place: the Chk-Rec judgment which deals with the annotations for recursive binders **`let rec`** $x_1 = e_1 : s_1$ **`in`** e_2. The updated Chk-Rec rule is shown in Fig. 5.3. (All the other checking rules from λ_β, shown in Fig. 4.2, carry over to λ_κ.) Instead

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\frac{s_1 \triangleright t_1 \qquad \Gamma \vdash t_1 : k_1 \qquad \Gamma; x{:}t_1 \vdash e_1 \triangleleft t_1 \qquad \Gamma; x{:}t_1 \vdash e_2 \triangleleft t_2}{\Gamma \vdash \textbf{let rec } x = e_1 : s_1 \textbf{ in } e_2 \triangleleft t_2}\textsc{Chk-Rec}$$

Type Synthesis $\boxed{\Gamma \vdash e \triangleright t}$

$$\frac{s \triangleright t \qquad \Gamma \vdash t : k \qquad \Gamma \vdash e \triangleleft t}{\Gamma \vdash e{:}s \triangleright t}\textsc{Syn-Ann}$$

Figure 5.3: Bidirectional Checking and Synthesis: other rules from λ_β (Fig. 4.2)

of using the user-specified annotation s_1 which may contain holes, we use t_1, which is an instantiation of s_1 that is guaranteed to be free of holes.

5.3.3 Synthesis

Similarly, we need only modify the one *synthesis* rule that deals with type annotations, namely, rule Syn-Ann which types the annotation terms $e{:}s$. The new rule is shown in Fig. 5.3. (All the other synthesis rules from λ_β, summarized in Fig. 4.2, apply unchanged, to λ_κ.) Again, as the annotated type s may have holes, we first instantiate it to t and then proceed, as in λ_β, pretending that the annotation was t all along.

5.4 Verification Conditions

The declarative typing rules require an oracle to magically *guess* refinements for the holes, and then *verify* those guesses. Next, let's see how Horn constraints let us *automate* the process of guessing suitable instantiations. This approach, introduced by Rondon *et al.*, 2008, has three steps.

1. **Templates:** First, we generate *templates* with *Horn applications* $\kappa(x)$ that represent the unknown refinements;

2. **Horn Constraints:** When the type annotations contain templates, the VC generation procedure returns *Horn constraints* that circumscribe the possible concrete refinements that would make the program well-typed;

3. **Horn Solving** Finally, we solve the Horn constraints to find either a suitable instantiation for the holes that demonstrates the program is well-typed, or otherwise reject the program as ill-typed, if no such solution can be found.

Next, let's formalize each of these steps.

5.4.1 Instantiating Holes with Templates

Recall from Fig. 5.1, that in λ_κ the language of predicates includes Horn applications $\kappa(\overline{x})$.

Well-formedness A predicate p is well-formed if p has no Horn-applications, or is of the form $p' \wedge \kappa(\overline{x})$ where p' is well-formed. This particular syntactic requirement on predicates ensures that the resulting constraints are indeed Horn constraints, and hence can be solved to determine typeability.

Templates A *template* is a type where all the refinements are well-formed predicates, *i.e.* whose refinements are all of the form $p \wedge_i \kappa_i(x_i)$ where p has no Horn applications.

Instantiation Instantiation arises in two crucial rules: Chk-Rec and Syn-Ann. In both cases, we require that if $s \triangleright t$ then the instantiated t be well-formed in the environment Γ. The procedure fresh summarized in Fig. 5.4 captures this requirement in an algorithmic manner: $\mathsf{fresh}(\Gamma,\ s)$ returns a *template* t such that for every *assignment* for the Horn variables, the type obtained by applying the assignment to t is guaranteed to be well-formed under Γ.

The first case (corresponding to Ins-Hole) instantiates a hole $\{\star\}$ with a Horn application $\kappa(\vec{,}\overline{x})$ where ν is a fresh symbol denoting the value being refined, and κ is a fresh Horn variable denoting an (unknown) relation over the variables in the the environment Γ and ν. The second case (corresponding to Ins-Conc) returns the concrete

fresh	:	$(\Gamma \times T) \rightarrow T$
$\mathsf{fresh}(\Gamma,\ b\{\star\})$	$\doteq$	$b\{v : \kappa(\bar{\cdot}\,\overline{x})\}$
where		
κ	$=$	fresh Horn variable of sort $b \times \overline{t}$
v	$=$	fresh binder
$\overline{x : t}$	$=$	Γ
$\mathsf{fresh}(\Gamma,\ b\{v : p\})$	$\doteq$	$b\{v : p\}$
$\mathsf{fresh}(\Gamma,\ x : s \rightarrow t)$	$\doteq$	$x : s' \rightarrow t'$
where		
s'	$=$	$\mathsf{fresh}(\Gamma,\ s)$
t'	$=$	$\mathsf{fresh}(\Gamma; x : s,\ t)$

Figure 5.4: Generating fresh templates

refinement unmodified, and the third case (corresponding to Ins-Fun) recurses on the function's input and output types, adding the input binder to the environment used to instantiate the output, to let the output refinement *depend upon* the input. For example, to generate a template from the partial type annotation for `abs`, we would invoke:

$$\mathsf{fresh}(\emptyset,\ x : \mathtt{int} \rightarrow \mathtt{int}\{\star\})$$

which would then return the template from (5.1)

$$x : \mathtt{int} \rightarrow \mathtt{int}\{v : \kappa(x, v)\}$$

5.4.2 Horn Constraints

In Fig. 5.5 we extend the VC generation procedure to use fresh to generate templates from partial types.

Checking Procedure check is modified only for the case where it handles annotated terms **let rec** $x = e_1 : s_1$ **in** e_2. Now, we use fresh to generate a template t_1 for the annotation s_1, after which we generate constraints as in λ_β (Fig. 4.2) assuming the annotation was t_1.

Synthesis Similarly, procedure synth is modified only for the case

check	:	$(\Gamma \times E \times T) \to C$
check$(\Gamma,\ $ **let rec** $x = e_1 : s_1$ **in** $e_2,\ t)$	$\doteq$	$c_1 \wedge c_2$
where		
c_1	$=$	check$(\Gamma_1,\ e_1,\ t)$
c_2	$=$	check$(\Gamma_1,\ e_2,\ t)$
Γ_1	$=$	$\Gamma; x : t_1$
t_1	$=$	fresh$(\Gamma,\ s_1)$
synth	:	$(\Gamma \times E) \to (C \times T)$
synth$(\Gamma,\ e : s)$	$\doteq$	$(c,\ t)$
where		
c	$=$	check$(\Gamma,\ e,\ t)$
t	$=$	fresh$(\Gamma,\ s)$

Figure 5.5: Horn Verification Condition Generation for λ_κ, extends cases of Fig. 4.5

where it handles annotated terms $e : s$. Again, we use `fresh` to get a template t for the annotation s, and then proceed as in λ_β (Fig. 4.4).

We encourage the reader to confirm that when run on the code with `abs` and `main` and with the annotation x:`int` $\to$ `int{`$\star$`}` for `abs`, the VC generation procedure `check` returns the Horn constraint eq. (5.2).

5.5 Solving Horn Constraints

Finally, to determine whether the program is typable, we need to *solve* the Horn constraints produced by VC generation.

5.5.1 Constraint Satisfaction

Assignments Recall that a Horn *assignment* σ is a mapping of Horn variables κ to SMT predicates (relations) over the Horn variables' parameters. We *apply* an assignment σ to a predicate and constraint by *replacing* each Horn application $\kappa(y)$ with its solution $\sigma(\kappa)[\overline{x} := \overline{y}]$ where $\overline{x}$ are the parameters of the Horn variable κ.

Satisfaction An assignment σ *satisfies* a Horn constraint c if applying the assignment to the constraint yields a *valid* Horn-variable free formula, *i.e.* if $\mathsf{SmtValid}(\sigma(c))$. A Horn constraint c is *satisfiable* if there *exists* an assignment σ that satisfies c.

If the VC generation procedure yields a satisfiable constraint, then the program is well-typed.

Proposition 5.1. If $\mathsf{check}(\Gamma,\ e,\ t)$ is Horn satisfiable, then $\Gamma \vdash e \triangleleft t$.

5.5.2 Computing Satisfiability via Predicate Abstraction

The λ_κ verifier uses its own Horn constraint solver that is based on predicate abstraction (Graf and Saidi, 1997), in particular, the Houdini algorithm (Flanagan and Leino, 2001), extended with optimizations that enable precise *local* type inference (Rondon *et al.*, 2008; Cosman and Jhala, 2017). This technique is summarized as the procedure solve shown in Fig. 5.6. $\mathsf{solve}(c, \mathbb{Q})$ takes as input a Horn constraint c and set of candidate atomic predicates or *qualifiers* $\mathbb{Q}$. The procedure returns SAT iff there exists an satisfying assignment for c that maps each κ to a *conjunction* of atomic predicates from $\mathbb{Q}$, and satisfies c. The procedure has two essential elements, summarized in Fig. 5.7.

1. Flatten First, we convert the Horn constraint c into a set of *flat* constraints cs each of which is of the form $\forall \overline{x:t}.\ p \Rightarrow p'$ where the *head* p' is either: (1) a single Horn *application* $\kappa(\overline{y})$, or, (2) a Horn-variable free *concrete* predicate. The subset of cs that have (resp. do not have) Horn applications in the head are gathered into the set cs_κ (resp. cs_p). We then invoke a the procedure fixpoint to find a solution σ for the application constraints cs_κ, and solve returns SAT iff σ also satisfies the concrete constraints cs_p.

2. Fixpoint The assignment that maps each κ to the relation *true* suffices to satisfy the application constraints, but of course, this assignment may not satisfy the concrete constraints. Instead, we start with an *initial* solution σ_0 that maps each Horn variable κ to the conjunction of *all* the candidate predicates in $\mathbb{Q}$. *Any* solution that that maps Horn variables to conjunctions over $\mathbb{Q}$ is trivially *weaker* than σ_0. Next, fixpoint iteratively *weakens* the candidate solution σ by (1) *choosing*

solve	:	$(C \times [P]) \rightarrow \textsc{Sat} + \textsc{Unsat}$
solve$(\mathbb{Q}, c)$	$\doteq$	if $\mathsf{SmtValid}(\sigma(cs_p))$ then $\textsc{Sat}$ else $\textsc{Unsat}$
where		
cs	$=$	$\mathsf{flat}(c)$
cs_κ	$=$	$\{c \mid c \in cs, c \equiv \forall \overline{x}:\overline{t}.\ p \Rightarrow \kappa(y)\}$
cs_p	$=$	$\{c \mid c \in cs, c \not\equiv \forall \overline{x}:\overline{t}.\ p \Rightarrow \kappa(y)\}$
σ_0	$=$	$\lambda\kappa. \wedge \{q \mid q \in \mathbb{Q}\}$
σ	$=$	$\mathsf{fixpoint}(cs_\kappa, \sigma_0)$

Figure 5.6: A procedure to solve a Horn constraint c using a set of qualifiers $\mathbb{Q}$.

some constraint c not satisfied by σ, *i.e.* where $\sigma(c)$ is is *not valid*, and (2) *removing* qualifiers from the κ at the head c, and (3) *iterating* the above process until all the application constraints cs_κ are satisfied.

The σ computed by fixpoint is guaranteed to be the *strongest* conjunction of candidate qualifiers that satisfies the application constraints cs_κ. Hence, if this σ *also* satisfies the concrete constraints cs_p then it satisfies c and solve returns $\textsc{Sat}$. Instead, if σ does not satisfy cs_p, we can be sure there is no satisfying assignment for c over conjunctions of $\mathbb{Q}$, and so solve returns $\textsc{Unsat}$ (Rondon *et al.*, 2008).

Proposition 5.2. If $\mathsf{solve}(\mathbb{Q},\ \mathsf{check}(\Gamma,\ e,\ t)) = \mathrm{SAT}$, then $\Gamma \vdash e \lessdot t$.

5.6 Discussion

Pierce, 2003 observes that "the more interesting your types get, the less fun it is to write them down." In this chapter, we saw how the programmer can elide refinement annotations and instead, let the type checker carry out the tedious task of writing them down. To do so, we introduced *Horn variables* to represent the unknown refinements, converting the Verification Conditions into *Horn constraints* whose satisfying assignments show the program is well-typed.

Tools for Horn Constraint Satisfiability In addition to the simple algorithm based on predicate abstraction (Jhala *et al.*, 2018) shown above, there is a rich literature on techniques for solving Horn constraints

flat	:	$C \to [C]$
$\mathsf{flat}(c)$	$\doteq$	$\{\mathsf{simpl}(\emptyset, \mathit{true}, c') \mid c' \in \mathsf{split}(c)\}$
$\mathsf{split}(p)$	$\doteq$	$\{p\}$
$\mathsf{split}(c_1 \wedge c_2)$	$\doteq$	$\mathsf{split}(c_1) \cup \mathsf{split}(c_2)$
$\mathsf{split}(\forall x{:}t.\ p \Rightarrow c)$	$\doteq$	$\{\forall x{:}t.\ p \Rightarrow c' \mid c' \in \mathsf{split}(c)\}$
$\mathsf{simpl}(\overline{x{:}t}, p, \forall x{:}t.\ q \Rightarrow c)$	$\doteq$	$\mathsf{simpl}(\overline{x{:}t}; x{:}t, p \wedge q, c)$
$\mathsf{simpl}(\overline{x{:}t}, p, q)$	$\doteq$	$\forall \overline{x{:}t}.\ p \Rightarrow q$
$\mathsf{fixpoint}$	:	$([C] \times \Sigma) \to \Sigma$
$\mathsf{fixpoint}(cs, \sigma)$	$\doteq$	case $\{c \mid c \in cs,$ not $\mathsf{SmtValid}(\sigma(c))\}$ of $\emptyset \to \sigma$ $c : \_ \to \mathsf{fixpoint}(cs, \mathsf{weaken}(\sigma, c))$
$\mathsf{weaken}(\sigma, \forall \overline{x{:}t}.\ p \Rightarrow \kappa(y))$	$\doteq$	$\sigma[\kappa := qs']$
where		
qs'	$=$	$\{q \mid q \in \sigma(\kappa)$ s.t. $\mathsf{keep}(q)\}$
$\mathsf{keep}(q)$	$\doteq$	$\mathsf{SmtValid}(\forall \overline{x{:}t}.\ \sigma(p) \Rightarrow q(y))$

Figure 5.7: Procedures that respectively flatten Horn constraints and computed a least fixed point solution.

that is comprehensively surveyed by Bjørner *et al.*, 2015. Many of these ideas are based on the notion of iteratively refining solutions using refutations and are implemented in different tools including Spacer (Komuravelli *et al.*, 2016), Eldarica (Hojjat and Rümmer, 2018), and even directly within some SMT solvers like Z3 (Hoder and Bjørner, 2012; Gurfinkel and Bjørner, 2019). Zhu *et al.*, 2018 describes a Horn constraint solver that combines the refutation-guided approach with machine learning over sample data-values that either belong within or without the constrained relations. Many of the above solvers are publicly available and compete regularly in an open competition (Ruemmer, 2021) that aims to benchmark and improve the solvers.

Finding Candidate Qualifiers In practice we have found predicate abstraction to be particularly effective. Though the other approaches are fully automatic, *i.e.* do not require qualifiers or templates, the general problem of inferring solutions is undecidable of course, and the solvers can easily diverge when searching for suitable predicates or relations, thereby making the end-to-end verification quite brittle (Jhala and McMillan, 2006). In contrast, the solve algorithm is parameterized by the set of qualifiers $\mathbb{Q}$ which should be thought of as a set of *candidate* fragments from which refinements should be synthesized, thereby bounding the space of candidate refinements, and ensuring that the solver quickly *terminates*, which is essential for predictable verification. One natural source of candidates are all the atomic predicate fragments that appear inside type annotations written by the programmer. Experience shows that this simple heuristic suffices to automatically infer refinements in practice (Vazou *et al.*, 2014b).

No Principal Types In some settings, inference can produce ideal results, such as the *principal* types of (Hindley, 1969; Damas and Milner, 1982b). Unfortunately (with apologies to Pierce), the more interesting your types get the less principal they become. That is, we do not know of any reasonable definition of an *ideal* refinement type for a function. The reason is that refinements are expressive *contracts* about how functions *should* be used. One can imagine many different and *incomparable* contracts that *e.g.* restrict the space of possible inputs to provide more precise guarantees about the *outputs* of a function. Thus, really the only *ideal* specification would be an explicit enumeration of all the inputs and their respective outputs, which is both incomputable and entirely defeats the purpose of logical specification in the first place!

Intra-Module Inference Consequently, we advocate moderation in the use of inference. In particular, because preconditions of a function are inferred based on the function's clients, it is impossible to deduce *preconditions* describing the all inputs of library functions, *e.g.* the public or exported functions of a module. Instead, inference is best used judiciously, in an *intra-modular* fashion (Lahiri *et al.*, 2009). Here, the *programmer* specifies the types for all exported functions, and the *verifier* uses those specifications, and the code of the module to infer all contracts for internal (private) functions.

This recipe offers several benefits. First, specifications on public functions provide useful documentation. Second, the method allows modular analysis in that the verifier can analyze each module in complete isolation from the others. We have found that intra-modular inference reduces the overhead of using "interesting" types (Rondon *et al.*, 2008; Vazou *et al.*, 2014b), by eliminating explicit annotations during polymorphic instantiation, as we shall see next.

6

Type Polymorphism

Next, let's look at λ_α which adds support for *type* polymorphism.

6.1 Examples

The main challenge with polymorphic signatures is to *instantiate* them appropriately at usage. Consider the following specification and implementation of the `max` function:

```
val max : forall 'a:Base. 'a => 'a => 'a
let max = (x, y) => {
  if (x < y) { y } else { x }
};
```

Comparison is permitted for any base type as illustrated below:

```
val (<) : forall 'a:Base. 'a => 'a => Bool
```

How can we verify the following `client` of `max` ?

```
val client: () => int[v|0 < v]
let client = () => {
  let r = max(0, 5);
  r + 1
};
```

Problem: Instantiation Intuitively, at its usage in `client` the `max` function behaves as if it takes and returns non-negative numbers. Thus, to verify `client` we must *instantiate*, the type variable `'a` with the equivalent of the refinement: `int`$\{\nu\!:\!0 \leq \nu\}$. But how shall we determine appropriate instance refinements?[1]

Solution: Decouple Type and Refinement Inference The key idea in λ_α is to *decouple* the inference of *types* from those of *refinements*. The former, *i.e.* type instances, can be determined by classical methods *e.g.* Hindley-Milner style unification (Damas and Milner, 1982a), or its modern variants as seen in Haskell (Peyton-Jones *et al.*, 2006; Schrijvers *et al.*, 2009), or local inference methods with support for subtyping (Pierce and Turner, 1998; Sulzmann *et al.*, 1997) or higher-rank polymorphism (Dunfield and Krishnaswami, 2013). The latter, *i.e.* refinement instances, can be obtained via Horn constraint solving as shown in λ_κ.

Phase 1: Type Elaboration In the first phase, λ_α uses classical unification (Pierce, 2002) to *elaborate* the source program to

```
val max : forall 'a:Base. 'a => 'a => 'a
let max = Λ 'a:Base. (x, y) => {
  if (x < y) { y } else { x }
};

val client: () => int[v|0 < v]
let client = () => {
  let r = (max[int[★]])(0, 5);
  r + 1
};
```

Each instantiation site is elaborated to an explicit *type application* that annotates the polymorphic function (`max`) with the instance (`int[★]`) for each type variable (`'a`). Crucially, the instances are just the (unrefined) types with *holes* for the as yet unknown refinements.

Phase 2: Refinement Inference In the second phase, we will generate and solve Horn constraints to infer suitable refinement instances.

[1]Of course, we *could* specify that `max` returns *one of* its two inputs, via the type: $x\!:\!\texttt{int} \rightarrow y\!:\!\texttt{int} \rightarrow \texttt{int}\{\nu\!:\!\nu = x \vee \nu = y\}$ after which we would be able to verify the clients as in λ_β. However, for the sake of exposition, let's assume this option is unavailable.

$$
\begin{array}{rcll}
\textbf{\textit{Environments}} & \Gamma & ::= & \dots & \textit{from } \lambda_\phi \textit{ Fig. 3.1} \\
& & \mid & \Gamma; \alpha : k & \textit{type variables} \\
\textbf{\textit{Base Types}} & b & ::= & \dots & \textit{from } \lambda_\kappa \textit{ Fig. 4.1} \\
& & \mid & \alpha & \textit{type variables} \\
\textbf{\textit{Types}} & t & ::= & \dots & \textit{from Fig. 4.1} \\
& & \mid & \forall \alpha : k.t & \textit{type polymorphism} \\
\textbf{\textit{Bare Types}} & \tau & ::= & b\{\star\} & \textit{bare base} \\
& & \mid & x : \tau \to \tau & \textit{dependent function} \\
& & \mid & \forall \alpha : k.\tau & \textit{type polymorphism} \\
\textbf{\textit{Terms}} & e & ::= & \dots & \textit{from Fig. 4.1} \\
& & \mid & \Lambda \alpha : k.e & \textit{type abstraction} \\
& & \mid & e[\tau] & \textit{type application}
\end{array}
$$

Figure 6.1: λ_α: Syntax of Types and Terms

As in λ_κ, we will create a new Horn variable κ for each hole, so the above type application becomes $\texttt{max}[\texttt{int}\{\nu{:}\kappa(\nu)\}]$ which has type

$$\texttt{int}\{\nu{:}\kappa(\nu)\} \to \texttt{int}\{\nu{:}\kappa(\nu)\} \to \texttt{int}\{\nu{:}\kappa(\nu)\}$$

Next, the VC from λ_κ yields the Horn constraint

$$
\begin{array}{lll}
& \forall \nu.\ \nu = 0 & \Rightarrow\ \kappa(\nu) \\
\wedge & \forall \nu.\ \nu = 5 & \Rightarrow\ \kappa(\nu) \\
\wedge & \forall r.\ \kappa(r) & \Rightarrow\ \forall \nu.\ \nu = r + 1\ \Rightarrow\ 0 < \nu
\end{array}
\tag{6.1}
$$

Which has the satisfying assignment σ such that $\sigma(\kappa)(z)\ \doteq\ 0 \le z$ which verifies that the code is well-typed.

6.2 Types and Terms

Let's formalize the above intuition in λ_α, which extends λ_κ with polymorphic types, extends the terms to include type abstraction and application as summarized in Fig. 6.1.

Types We extend the language of types to include type *variables* α of a kind k which can be quantified to get *polymorphic* types $\forall \alpha{:}k.t$. The set of *bare* types τ are those where *all* refinements are holes $\{\star\}$.

Well-formedness $\boxed{\Gamma \vdash t : k}$

$$\frac{\alpha : B \in \Gamma \qquad \Gamma; x : \alpha \vdash p}{\Gamma \vdash \alpha\{x : p\} : B}\textsc{Wf-Var-Base}$$

$$\frac{\alpha : k \in \Gamma}{\Gamma \vdash \alpha\{x : \mathit{true}\} : k}\textsc{Wf-Var} \qquad \frac{\Gamma; \alpha : k \vdash t : k_t}{\Gamma \vdash \Lambda\alpha : k.t : \star}\textsc{Wf-All}$$

Figure 6.2: λ_α: Rules for Well-formedness

Terms As the procedure for determining type instances is classical (Pierce, 2002; Dunfield and Krishnaswami, 2020), we assume that the language of terms is already elaborated with an explicit *type abstraction* form $\Lambda\alpha : k.e$ and a *type application* form $e[\tau]$. Crucially, neither form involves refinements: the former just has type variables, and the latter uses bare types where all the refinements are holes.

6.3 Declarative Typing

To account for type polymorphism, we must extend well-formedness and subtyping to quantified types, and then add rules for checking type abstraction terms and synthesizing types for the type application terms.

Well-formedness We associate each type variable with a *kind*, and use the well-formedness rules in Fig. 6.2 to ensure that only base-kinded type variables are refined. The rule Wf-Var-Base checks well-formedness of refined type variables of base kind by checking that the type variable is bound in the typing environment and that the refinement predicate is well formed. The rule Wf-Var ensures that type variables of non-base types are trivially refined with true to avoid unsoundness. The rule Wf-All ensures that polymorphic types are well-formed with a star kind, if their body is well-formed in an environment extended with the bound variable.

Example: Refining Non-base Variables is Unsound Consider the following polymorphic function that returns a *false* refined `int`.

$$
\begin{array}{rcll}
(\forall\alpha\!:\!k.\tau)[\alpha \mapsto t_\alpha] & \doteq & \forall\alpha\!:\!k.\tau & \\
(\forall\alpha'\!:\!k.\tau)[\alpha \mapsto t_\alpha] & \doteq & \forall\alpha'\!:\!k.(\tau[\alpha \mapsto t_\alpha]), & \alpha' \neq \alpha \\
(x\!:\!t_x \to t)[\alpha \mapsto t_\alpha] & \doteq & x\!:\!(t_x[\alpha \mapsto t_\alpha]) \to (t[\alpha \mapsto t_\alpha]) & \\
(\alpha\{x\!:\!\mathit{true}\})[\alpha \mapsto t_\alpha] & \doteq & t_\alpha & \\
(\alpha\{r\})[\alpha \mapsto b\{r_\alpha\}] & \doteq & b\{r \wedge r_\alpha\} & \\
(\alpha\{r\})[\alpha \mapsto t_\alpha] & \doteq & \bot & \text{undefined case} \\
(b\{r\})[\alpha \mapsto t_\alpha] & \doteq & b\{r\}, & b \neq \alpha
\end{array}
$$

Figure 6.3: Type Variable Instantiation

```
val dead : forall a:Base. a[v|false] => int[v|false]
let dead = (x) => { 0 };
```

For the type of `dead` to be well-formed, the kind of `a` has to be base as `a` is refined with a non-trivial predicate. The precondition ensures that the function `dead` type checks. However, the precondition also effectively prohibits the function from being called at run-time. Suppose that we allowed a call to `dead` with a non-base argument, *e.g.* `id`

```
val unsound : int[v|false]
let unsound = {
  let id = (x) => {x};
  deadcode(id)
};
```

If the above call type checked, our system would *unsoundly* prove that `0` has the type `int[v|false]`. Fortunately, we can ensure that the above definition is rejected by restricting how refined type variables (like `a`) are instantiated. (We could simply prohibit refinements on type variables, but this would preclude many useful specifications § 7.) □

Type Variable Instantiation We use two functions to instantiate (substitute) variables in types. The function $t[\alpha := b]$ *substitutes* the type variable α with the base type b in a standard way. The function $t[\alpha \mapsto t_\alpha]$, on the other hand, *instantiates* the type variable α with the type t_α by strengthening refinement predicates, as defined in Fig. 6.3. Note that definition of $s[\alpha \mapsto t]$ is partial: it is not defined when $s \equiv \alpha\{r\}$ for r different than *true* and t is not a base type, to prevent unsoundness as described above.

Subtyping The rule Sub-All shown in Fig. 6.4 formalizes subtyping

Subtyping $\boxed{\Gamma \vdash t_1 <: t_2}$

$$\frac{\Gamma; \alpha_1 : k \vdash t_1 <: t_2[\alpha_2 := \alpha_1]}{\Gamma \vdash \forall \alpha_1 : k.t_1 <: \forall \alpha_2 : k.t_2}\textsc{Sub-All}$$

Figure 6.4: λ_α: Rules for Subtyping

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\frac{\Gamma; \alpha : k \vdash e \triangleleft t \qquad \Gamma \vdash \forall \alpha : k.t : \star}{\Gamma \vdash \Lambda \alpha : k.e \triangleleft \forall \alpha : k.t}\textsc{Chk-TLam}$$

Type Synthesis $\boxed{\Gamma \vdash e \triangleright t}$

$$\frac{\Gamma \vdash e \triangleright \forall \alpha : k.s \qquad \tau \triangleright t \qquad \Gamma \vdash t : k}{\Gamma \vdash e[\tau] \triangleright s[\alpha \mapsto t]}\textsc{Syn-TApp}$$

Figure 6.5: λ_α: Rules for Checking and Synthesis

for quantified types by renaming the type variables and checking that the types being quantified over belong to the subtyping relation, in an environment extended with the type variable. For example, the rule derives

$$\emptyset \vdash \forall \alpha : k.x : \alpha \to \alpha\{v : v = x\} <: \forall \beta : k.x : \beta \to \beta$$

because, after substituting β with α the above reduces to

$$\alpha \vdash x : \alpha \to \alpha\{v : v = x\} <: x : \alpha \to \alpha$$

which follows from the rules Sub-Base and Sub-Fun.

Checking The rule Chk-TLam in Fig. 6.5 checks type-abstraction terms $\Lambda\alpha : k.e$ against quantified types $\forall\alpha : k.t$ by checking the inner expression e against the t in an environment containing α, and checking the well-formedness of the polymorphic type.

***Example: Implementation of* `max`** The specification and implementation of the max function from section 6.1 are elaborated to:

$$t_{\mathsf{max}} \doteq \forall\alpha{:}B.\alpha \to \alpha \to \alpha$$
$$e_{\mathsf{max}} \doteq \Lambda\alpha{:}B.\lambda x, y.\ \mathbf{if}\ x < y\ \mathbf{then}\ y\ \mathbf{else}\ x$$

The checking rules for λ- and branching terms establish that

$$\alpha{:}B \vdash\ \lambda x, y.\ \mathbf{if}\ x < y\ \mathbf{then}\ y\ \mathbf{else}\ x \Leftarrow \alpha \to \alpha \to \alpha$$

after which the rule Chk-TLam lets us conclude $\emptyset \vdash e_{\mathsf{max}} \triangleleft t_{\mathsf{max}}$. □

Synthesis The rule Syn-TApp in Fig. 6.5 synthesizes the type for a type-application term $e[\tau]$. Recall that the elaboration process inserts the application annotations using one of many standard approaches, but the applied type is *bare* in that every refinement is a *hole* $\{\star\}$ as a standard elaborator is unaware of refinements. Instead, similar to the Chk-Rec and Syn-Ann from Fig. 5.3 which handle type annotations with holes, the rule Syn-TApp first guesses a suitable instantiation $\tau \triangleright t$ such that t is well-formed in the given context. The rule then *substitutes* the concrete t for the type variable α quantified over in the signature for e. To prevent unsoundness, this substitution is partial: type synthesis fails in cases that it is not defined.

Example: Uses of **max** The function application term (max 0 5) inside client from section 6.1 is elaborated to (max[int{★}] 0 5). In the context Γ with the signature for max,

$$\Gamma \doteq \mathsf{max}{:}t_{\mathsf{max}}$$

we have

$$\Gamma \vdash \mathsf{max} \Rightarrow \forall\alpha{:}B.\alpha \to \alpha \to \alpha \quad \text{and} \quad \mathsf{int}\{\star\} \triangleright \mathsf{nat} \quad \text{and} \quad \Gamma \vdash \mathsf{nat} : B$$

and so, using Syn-TApp we conclude that

$$\Gamma \vdash \mathsf{max}[\mathsf{int}\{\star\}] \Rightarrow \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$$

after which the application rule Syn-App Fig. 3.6 yields

$$\Gamma \vdash \mathsf{max}[\mathsf{int}\{\star\}]\ 0\ 5 \Rightarrow \mathsf{nat}$$

□

sub	$:$	$(T \times T) \to C$
$\mathsf{sub}(\forall\alpha_1\!:\!k.t_1,\ \forall\alpha_2\!:\!k.t_2)$	$\doteq$	$\mathsf{sub}(t_1,\ t_2[\alpha_2 := \alpha_1])$
check	$:$	$(\Gamma \times E \times T) \to C$
$\mathsf{check}(\Gamma,\ \Lambda\alpha\!:\!k.e,\ \forall\alpha\!:\!k.t)$	$\doteq$	$\mathsf{check}(\Gamma;\alpha,\ e,\ t)$
synth	$:$	$(\Gamma \times E) \to (C \times T)$
$\mathsf{synth}(\Gamma,\ e[\tau])$	$\doteq$	$(c, s[\alpha := t])$
where		
$(c,\ \forall\alpha\!:\!k.s)$	$=$	$\mathsf{synth}(\Gamma,\ e)$
t	$=$	$\mathsf{fresh}(\Gamma,\ \tau)$

Figure 6.6: Verification Conditions for λ_α, extends cases of Fig. 5.5

6.4 Verification Conditions

Fig. 6.6 summarizes how we extend the Horn Verification Condition generation algorithm to account for type polymorphism. In essence, we add new cases to the procedures sub, check and synth that respectively generate Horn Constraint for the subtyping, checking and synthesis modes to account for the new derivation rules shown in Fig. 6.5.

Subtyping The subtyping constraint for two polymorphic types is generated by recursing on the underlying types, after unifying the type variables.

Checking Similarly, to check a type abstraction, we recursively invoke check on the inner expression using a suitably extended context. We also check well-formedness of the provided type to ensure the kind given to the abstracted type variable is correct.

Synthesis The heavy lifting is done by synth, which synthesizes a type and a constraint for a type application term $e[\tau]$. However, we treat this analogous to synthesizing the type of a type-annotation. Instead of "guessing" a type as in the declarative Syn-TApp (Fig. 6.5), we use τ to generate a fresh *template* for the instantiated t and then substitute the template t for the type variable α to get the template for the $e[\tau]$.

***Example:* `client`** Let us see how the above works on `client` from section 6.1. Let

$$\begin{aligned}
\Gamma &\doteq \mathtt{max}{:}\forall\alpha{:}k.\alpha \rightarrow \alpha \rightarrow \alpha \\
e_0 &\doteq \mathtt{max}[\mathtt{int}\{\star\}] \\
e_1 &\doteq e_0\ 0\ 5 \\
e_2 &\doteq \mathbf{let}\ \mathsf{r} = e_1\ \mathbf{in}\ \mathsf{r} + 1
\end{aligned}$$

Now, it it easy to check that as it simply returns the type of `max` in Γ,

$$\mathsf{synth}(\Gamma,\ \mathtt{max}) \doteq (\mathit{true}, \forall\alpha{:}k.\alpha \rightarrow \alpha \rightarrow \alpha)$$

Thus, the instance of `max` synthesizes the type

$$\mathsf{synth}(\Gamma,\ e_0) \doteq (\mathit{true}, \mathtt{int}\{\nu{:}\kappa(\nu)\} \rightarrow \mathtt{int}\{\nu{:}\kappa(\nu)\} \rightarrow \mathtt{int}\{\nu{:}\kappa(\nu)\})$$

by substituting all occurrences of α with the *fresh* template $\mathtt{int}\{\nu{:}\kappa(\nu)\}$ generated from $\mathtt{int}\{\star\}$. Consequently, the subsequent applications in e_1 synthesize the constraint and type

$$\mathsf{synth}(\Gamma,\ e_1) \doteq (c_1 \wedge c_2,\ \mathtt{int}\{\nu{:}\kappa(\nu)\})$$

where the synthesized type is the (instance) function's *output* and c_1 and c_2 are constrain 0 and 5 to be subtypes of the function's *input*

$$\begin{aligned}
c_1 &\doteq \forall\nu.\ \nu = 0 \Rightarrow\ \kappa(\nu) \\
c_2 &\doteq \forall\nu.\ \nu = 5 \Rightarrow\ \kappa(\nu)
\end{aligned}$$

The result, and hence, output type is bound to `r` and so writing

$$\Gamma' \doteq \Gamma; \mathsf{r}{:}\mathtt{int}\{\nu{:}\kappa(\nu)\}$$

then we get $\mathsf{check}(\Gamma',\ \mathsf{r}+1,\ \mathtt{int}\{\nu{:}0 < \nu\}) \doteq\ c_3$ where

$$c_3 \doteq \forall r.\ \kappa(r)\ \Rightarrow\ \forall\nu.\ \nu = r + 1\ \Rightarrow\ 0 < \nu$$

Hence, checking the body e_2 of `client` against its output type yields

$$\mathsf{check}(\Gamma,\ e_2,\ \mathtt{int}\{\nu{:}0 < \nu\})\ \doteq\ c_1 \wedge c_2 \wedge c_3$$

which is exactly the constraint (6.1). □

6.5 Discussion

Thus, the mechanism for refinement inference introduced for λ_κ chapter 5 makes using polymorphic functions very pleasant. The main problem here is to figure out how to *instantiate* a polymorphic signature at a particular instantiation site. The key idea is to *first* use classical methods to instantiate the *unrefined* ("bare") part of the type, leaving *holes* for the unknown refinements, after which the Horn constraint based method from λ_κ can be applied to infer suitable refinements.

Other approaches to Polymorphic Instantation There are several other possible ways to account for type polymorphism.

- ***Annotations*** One approach is to have the programmer *explicitly specify* the instance refinements. However, this is most unpalatable as polymorphism is ubiquitous in modern code (Pierce and Turner, 1998) and placing explicit annotations would get tedious quickly.
- ***Defaults*** Another approach would be to *default* to some refinement such as *true* as done in Refined Racket (Kent *et al.*, 2016) or the Stainless verifier (Hamza *et al.*, 2019). Sadly, this method is rather conservative, as it precludes the verification of `client` as the type checker has no information about the value returned by `max` other than it is *some* `int`.
- ***Unification*** A third approach, deployed by the F* system (Swamy *et al.*, 2011), is to try to *unify* the types of the inputs or outputs to determine suitable instance refinements. Unfortunately, the interaction with the refinement logic makes unification brittle: for example, in `client` it is unclear how to unify the types of the two inputs `0` and `5` to obtain the instance type `int`$\{\nu{:}0 \leq \nu\}$.

Polymorphism and HOFs Finally, support for type polymorphism is essential for being able to easily use Higher-Order functions. For example, consider the `fold` function below that accumulates some value over the integers between 0 and n:

```
val fold : ('a => int => 'a) => 'a => int => 'a
let fold = (f, acc, n) => {
```

```
    let rec loop = (i, acc) => {
      if (i < n) {
        loop(i+1, f(acc, i))
      } else {
        acc
      }
    };
    loop(0, acc)
  }
```

We can use `fold` to sum the integers from `0` to `m` as:

```
  val sumTo: m:nat => nat
  let sumTo = (m) => {
    let add = (x, y) => {x + y};
    fold(add, 0, 0, m)
  }
```

Readers familiar with the classical Floyd-Hoare proof rule for loops might notice its similarity to the type signature of `fold`:

$$\forall \alpha : k.(\alpha \to \texttt{int} \to \alpha) \to \alpha \to \texttt{int} \to \alpha$$

The type variable α is analagous to the loop *invariant*; the accumulation function's type $\alpha \to \texttt{int} \to \alpha$ says that it *preserves* the invariant, *i.e.* if the input accumulated value satisfies the invariant then so does the output; the initial value of the accumulator must satisfy the invariant α; and hence, "by induction", the final value, regardless of how many accumulation steps is guaranteed to satisfy the invariant α. Hence, the VC generation mechanism of λ_α let the checker infer that within `sumTo`, the type parameter α is instantiated to `nat`, *i.e.* that `nat` is an invariant of the accumulator. Consequently, the value returned by `fold`, and hence, `sumTo` must also be a `nat`.

This ability to automatically infer refinements in the presence of polymorphism will prove especially useful with user-defined *data types*, as we shall see next.

7

Data Types

There is only so much one can do with `int` and `bool` values: programs get much more interesting once we start adding *data types*. Next, let's look at λ_δ which extends λ_α with support for precisely specifying and verifying properties of (algebraic) *user-defined data types*.

7.1 Examples

As usual, let's begin with a bird's eye view of the different kinds of specifications we might write for data types.

7.1.1 Properties of Data

The simplest, but perhaps most ubiquitous and useful examples, pertain to properties of the data stored within polymorphic *containers* like the `list` type defined as:

```
type list('a) =
  | Nil
  | Cons('a, list('a))
```

The type declaration introduces the type `list('a)` which has two constructors:

```
val Nil  : list('a)
val Cons : 'a => list('a) => list('a)
```

Let's use `Nil` and `Cons` to write a function `range` that returns the sequence of `int` values between a lower bound `lo` and upper bound `hi`.

```
val range: lo:int => hi:int =>
           list(int[v|lo <= v && v < hi])
let range = (lo, hi) => {
  if (lo < hi) {
    let rest = range(lo + 1, hi);
    Cons(lo, rest)
  } else {
    Nil
  }
}
```

The signature for `range` specifies that *every* element in the output `list` is in the interval between `lo` and `hi`.

7.1.2 Relationships between Data

The previous example showed how we can capture properties of *individual* datum by refining the type parameter of `list`. What if we want to relate the values of *multiple* data across a structure? For example, here's a data type definition that specifies an *ordered list* of non-decreasing values:

```
type olist('a) =
  | ONil
  | OCons (x:'a, xs:olist('a[v|x<=v]))
```

The type definition endows the constructors with refined signatures

```
val ONil : olist('a)
val OCons: x:'a => olist('a[v|x<=v]) => olist('a)
```

The signature for `OCons` says that the head `x` must be smaller than *each* element of the tail. Thus, the type checker will *accept* the term

```
let okList = OCons(0, OCons(1, OCons(2, ONil)));
```

but will *reject* the term

```
let badList = OCons(0, OCons(2, OCons(1, ONil)));
```

```
val insert : 'a => olist('a) => olist('a)
let rec insert = (x, ys) => {
  switch (ys) {
    | ONil =>
        let tl = ONil;
        OCons(x, tl)
    | OCons(y, ys') =>
        if (x <= y) {
          let tl = OCons(y,ys');
          OCons(x,tl)
        } else {
          let tl = insert(x,ys');
          OCons(y,tl)
        }
  }
};
```

Figure 7.1: A function to `insert` a value `x` into an ordered list `ys`.

That is, the constructor's signature ensures that *illegal* values do not *inhabit* the type. More interestingly, we can specify and verify the function in Fig. 7.1 that `inserts` a value `x` into an ordered list `ys`. We can use `insert` to implement an *insertion-sort* function and verify that it always returns an ordered list:

```
val isort : list('a) => olist('a)
let rec isort = (xs) => {
  switch (xs){
    | Nil         => ONil
    | Cons (h, t) => insert(h, isort(t))
  }
};
```

7.1.3 Properties of Structure

A third class of useful specifications are *aggregate* properties of the entire structure, for example, the *height* of a tree, or the *multi-set* of elements of a list. Next, let's see how these can be specified by refining the *output* types of the constructors with *ghost* functions that specify the aggregate properties via two steps.

1. Defining Measures To specify the *length* of a list, we introduce a function such that `len`(xs) represents the length of the list xs.

```
measure len: list('a) => nat
```

To ensure decidable VC validity checking we ensure that `len` is *uninterpreted* in the refinement logic, *i.e.* the SMT solver only knows that `len` satisfies the *axiom of congruence*

$$\forall xs, ys.\ xs = ys \ \Rightarrow\ \texttt{len}(xs) = \texttt{len}(ys)$$

2. Refining Constructors We use measures to specify the structure's properties, by appropriately refining the type of the constructors' output.

```
type list('a) =
 | Nil                      => [v|len(v) = 0]
 | Cons(x:'a,xs:list('a)) => [v|len(v) = 1+len(xs)]
```

In the definition above, the output for `Nil` says that it constructs a list of length 0; the output for `Cons` says that it constructs a list whose length is one greater than the tail.

3. Using Measures We can now use `len` in refinements in various ways. First, to specify pre-conditions on *partial* functions. For example, refinement checking ensures that due to the precondition — which will be checked at uses of `head` — the `assert(false)` never fails at run-time:

```
val head : list('a)[v|0 < len(v)] => 'a
let head = (xs) => {
  switch(xs){
    | Cons(h, t) => h
    | Nil        => assert(false)
  }
};
```

Second, to specify post-conditions *e.g.* on the result of list concatenation

```
val append: xs:list('a) => ys:list('a) =>
            list('a)[v|len(v) = len(xs)+len(ys)]
let rec append = (xs, ys) => {
  switch (xs) {
    | Nil        => ys
    | Cons(h, t) => let rest = append(t, ys);
                    Cons(h, rest)
  }
};
```

Measures are Ghost Code Measures only exist at the level of the *specification*: they cannot be used in the implementation.[1] However, it is easy to connect measures to run-time values, via functions like

```
val length: xs:list('a) => int[v|v = len(xs)]
let rec length = (xs) => {
  switch(xs){
    | Nil        => 0
    | Cons(h, t) => 1 + length(t)
  }
}
```

We can now use the result of `length` to determine whether it is safe to compute the `head` of a `list`

```
val safeHead : 'a => list('a) => 'a
let safeHead = (default, xs) => {
  let nonEmpty = 0 < length(xs);
  if (nonEmpty) { head(xs) } else { default }
};
```

Refinement typing establishes that when `nonEmpty` is `true`, we indeed have `0 < len(xs)` thereby verifying the call `head(xs)`.

7.2 Types and Terms

From the examples in section 7.1 one might get the impression that λ_δ must have multiple extensions over λ_α. In fact, all three flavors of specifications — reasoning about individual data, about relationships between data, and reasoning about structure — are supported by a single pillar: *refined data constructors*. Thus, the only mechanisms we need are a way to "apply" the refined type when *constructing* new data and to "unapply" the type when *destructing* the data by pattern matching. Next, let's see how these two ideas are formalized in λ_δ, whose syntactic additions are summarized in Fig. 7.2.

Datatypes First, we assume there is a set of *type constructors* C (*e.g.* `list`) and *data constructors* D (*e.g.* `Nil`, `Cons`). The *polarity* p captures

[1]Systems like LiquidHaskell allow the programmer to specify the measure as a function that satisfies certain syntactic constraints, and then automatically *generate* the constructor's refined types. However, that is merely a convenience: conceptually, a measure exists only for specification.

$$
\begin{array}{rlll}
\textbf{\textit{Data Constructors}} & D & ::= & D_1, D_2, D_3, \ldots \\
\textbf{\textit{Type Constructors}} & C & ::= & C_1, C_2, C_3, \ldots \\
\textbf{\textit{Polarity}} & p & ::= & + \mid - \mid \pm \mid \epsilon \\
\textbf{\textit{Datatypes}} & \delta & ::= & \langle C,\ \overline{\alpha:k/p},\ \overline{D:t}\rangle \\
\textbf{\textit{Environments}} & \Gamma & ::= & \ldots \qquad \textit{from Fig. 6.1} \\
 & & \mid & \Gamma;\delta \qquad \textit{type definitions} \\
\textbf{\textit{Base Types}} & b & ::= & \ldots \qquad \textit{from Fig. 6.1} \\
 & & \mid & C[\overline{t}] \qquad \textit{datatypes} \\
\textbf{\textit{Alternatives}} & a & ::= & D(\overline{x}) \to e \qquad \textit{switch alternative} \\
\textbf{\textit{Terms}} & e & ::= & \ldots \qquad \textit{from Fig. 6.1} \\
 & & \mid & D \qquad \textit{data constructor} \\
 & & \mid & \texttt{switch}\ x\ \{\overline{a}\} \qquad \textit{data destructor}
\end{array}
$$

Figure 7.2: λ_δ: Syntax of Types and Terms

the position that type variables appear in definitions of data types and can be positive $(+)$, negative $(-)$, both $(\pm)$, or neither (ϵ). A *datatype* δ is a triple $\langle C,\ \overline{\alpha:k/p},\ \overline{D:t}\rangle$ comprising a type constructor C, the list of type variables over which the datatype is parameterized together with their kind and polarity $\overline{\alpha:k/p}$, and a set of data constructors and their refinement types $\overline{D:t}$.

Example: Ordered Lists Suppose we wrote the following ordered list type, refined with a `len` measure tracking the list's size

```
type ol('a) =
 | ON                          => [v|len(v)=0]
 | OC('a,xs:ol('a[v|x<=v])) => [v|len(v)=1+len(xs)]
```

We would represent the above as $\delta_{\texttt{OL}} \doteq \langle \texttt{OL},\ \{\alpha:B/+\},\ \{\texttt{ON}:t_N;\texttt{OC}:t_C\}\rangle$ The type variable α appears in one positive position and since α elements are compared it is of base kind. The types of the "nil" and "cons" constructors are respectively:

$$
\begin{aligned}
t_N &\doteq \forall\alpha:B.\texttt{OL}[\alpha]\{v:\texttt{len}(v) = 0\} \\
t_C &\doteq \forall\alpha:B.x:\alpha \to xs:\texttt{OL}[\alpha\{v:x \leq v\}] \to \texttt{OL}[\alpha]\{v:\texttt{len}(v) = 1 + \texttt{len}(xs)\}
\end{aligned} \tag{7.1}
$$

That is, the type t_N says that "nil" returns an ordered list of length 0; and the type t_C says that "cons" takes as input a head x of type α and a tail xs *each* of whose elements is an α larger than x, and returns an ordered list whose size is one more than that of the tail xs. □

Environments We extend the environments to include all the data type definitions δ. For simplicity, we will assume that all the data type definitions and measure names are *global*, that is, they belong in the top-level environment used for type checking and synthesis.

Types As hinted in the discussion for constructors above, the language of *base* types is extended to include a *type constructor application* form $C[\overline{t}]$, where the type constructor C is applied to the type arguments $\overline{t}$. Intuitively one can think of the above as the type obtained by instantiating the type parameters $\overline{\alpha}$ of C with the actual type arguments $\overline{t}$. As before, these base types can be refined, so $\texttt{OL[nat]}\{v{:}3 \leq \texttt{len}(v)\}$ would correspond to the type of ordered lists of non-negative `int` values comprising three or more elements.

Alternatives Each *alternative* $D(\overline{x}) \to e$ comprises a *pattern* $D(\overline{x})$ and the term e to be evaluated if the scrutinee matches the pattern.

Terms We add the data constructors D to the language of terms so that polymorphic instantiation $e[t]$ (from λ_α) and function application $e\ x$ (from λ_ϕ) can be combined to *construct* values of user-defined types. To *destruct* values of user-defined types, we introduce a pattern-match form $\textbf{switch}\ y\ \{\overline{a}\}$ where the value bound to y is *scrutinized* by each of the alternatives in $\overline{a}$.

7.3 Declarative Typing

Next, let's see how the rules for well-formedness, subtyping, checking and synthesis are extended to account for constructors and destructors.

7.3.1 Well-formedness

The rule Wf-Data shown in Fig. 7.3 formalizes well-formedness of datatypes. The rule checks that the refinement of the type is well-formed, that each of the type arguments has the proper kind, and that the type

Well-formedness $\boxed{\Gamma \vdash t : k}$

$$\frac{\overline{k} = \mathsf{kinds}(\delta,\ C) \quad \Gamma \vdash t_i : k_i \text{ for each } 1 \le i \le |\overline{k}| \quad \Gamma; x : C[\overline{t}] \vdash p}{\Gamma \vdash C[\overline{t}]\{x : p\} : B}\ \textsc{Wf-Data}$$

Figure 7.3: λ_δ: Rule for Well-formedness

Subtyping $\boxed{\Gamma \vdash t_1 <: t_2}$

$$\frac{\begin{array}{c}\Gamma \vdash s_i <: t_i \text{ for each } i.p_i \in \{+, \pm\} \\ \Gamma \vdash t_i <: s_i \text{ for each } i.p_i \in \{-, \pm\} \\ \Gamma; v_1 : \{C[\overline{s}] : p_1\} \vdash p_2[v_2 := v_1] \qquad \overline{p} = \mathsf{polarities}(\delta,\ C)\end{array}}{\Gamma \vdash C[\overline{s}]\{v_1 : p_1\} <: C[\overline{t}]\{v_2 : p_2\}}\ \textsc{Sub-Data}$$

Figure 7.4: λ_δ: Rules for Subtyping

constructor is fully applied. The premises use the function $\mathsf{kinds}(\delta,\ C)$ that retrieves the kinds of C from the data environment δ:

$$\mathsf{kinds}(\delta,\ C) \doteq \overline{k} \quad \text{if } \langle C,\ \overline{\alpha : k/p},\ \overline{D : t}\rangle \in \delta$$

7.3.2 Subtyping

The rule Sub-Data shown in Fig. 7.4 formalizes subtyping between datatypes. In an environment Γ, the type $C[\overline{s}]\{v_1 : p_1\}$ is a subtype of $C[\overline{t}]\{v_2 : p_2\}$, if the base refinement p_1 entails p_2, and *each* of the component types s_i is a subtype of the corresponding component t_i. Subtyping of the components is checked using $\mathsf{polarities}(\delta,\ C)$ which retrieves the polarity information from the data environment δ

$$\mathsf{polarities}(\delta,\ C) = \overline{p} \text{ if } \langle C,\ \overline{\alpha : k/p},\ \overline{D : t}\rangle \in \delta$$

For each component with positive polarity (resp. negative) polarity the rule uses covariant (resp. contravariant) subtyping.

***Example: Subtyping in* `insert`** Consider the environment

$$\Gamma'_{\le} \ \doteq\ \alpha : B;\ x : \alpha;\ y : \alpha;\ ys' : \mathsf{OL}[\alpha\{v : y \le v\}];\ ys : \mathsf{OL}^{+}[\alpha, ys'];\ x \le y$$

Type Checking $\boxed{\Gamma \vdash e \lhd t}$

$$\frac{\Gamma \mid y \vdash a_i \lhd t \text{ for each } i}{\Gamma \vdash \textbf{switch}\ y\ \{\overline{a}\} \lhd t}\textsc{Chk-Swt}$$

Checking Alternatives $\boxed{\Gamma \mid y \vdash a \lhd t}$

$$\frac{s = \mathsf{ctor}(\Gamma,\ D,\ y) \qquad \Gamma' = \mathsf{unapply}(\Gamma,\ y,\ \overline{z},\ s) \qquad \Gamma' \vdash e \lhd t}{\Gamma \mid y \vdash D(\overline{z}) \to e \lhd t}\textsc{Chk-Alt}$$

Figure 7.5: λ_δ: Rules for Type Checking

and the alias $\mathtt{OL}^+[\alpha, z]$ that denotes ordered lists of type α whose size is one more than that of z. As the following entailment is valid

$$\forall x, y, v.\ x \leq y \ \Rightarrow\ y \leq v \ \Rightarrow\ x \leq v$$

the rule Sub-Data lets us conclude

$$\Gamma'_{\leq}\ \vdash\ \mathtt{OL}[\alpha\{v{:}y \leq v\}] <:\ \mathtt{OL}[\alpha\{v{:}x \leq v\}] \tag{7.2}$$

As the following entailment is valid

$$\forall ys,\ ys',\ tl.\ \mathtt{len}(ys) = 1 + \mathtt{len}(ys') \ \Rightarrow\ \mathtt{len}(tl) = 1 + \mathtt{len}(ys') \ \Rightarrow$$
$$\mathtt{len}(v) = 1 + \mathtt{len}(tl) \ \Rightarrow\ \mathtt{len}(v) = 1 + \mathtt{len}(ys)$$

the rule Sub-Data lets us conclude

$$\Gamma'_{\leq}; tl{:}t_y\ \vdash\ \mathtt{OL}^+[\alpha, tl]\ <:\ \mathtt{OL}^+[\alpha, ys] \tag{7.3}$$

where $t_y \doteq \mathtt{OL}^+[\alpha_x, ys']$ □

7.3.3 Checking

The rule Chk-Swt shown in Fig. 7.5 describes how to *check* that a switch expression has a given type t, by checking that each alternative of the switch produces a value of type t.

Checking an Alternative The judgment $\Gamma \mid y \vdash D(\overline{z}) \to e \lhd t$ states that in the *environment* Γ when the *scrutinee* y matches the *pattern*

unapply	:	$(\Gamma \times X \times X^* \times T) \rightarrow \Gamma$
$\mathsf{unapply}(\Gamma,\ y,\ z;\overline{z},\ x\!:\!s \rightarrow t)$	$\doteq$	$\mathsf{unapply}(\Gamma; z\!:\!s,\ y,\ \overline{z},\ t[x := z])$
$\mathsf{unapply}(\Gamma,\ y,\ \emptyset,\ t)$	$\doteq$	$\Gamma; y\!:\!\mathsf{meet}(\Gamma(y), t)$
ctor	:	$(\Gamma \times D \times X) \rightarrow T$
$\mathsf{ctor}(\Gamma,\ D,\ y)$	$\doteq$	$s[\overline{\alpha} := \overline{t}]$
where		
$C[\overline{t}]$	$=$	$\Gamma(y)$
$\forall \overline{\alpha}\!:\!k.s$	$=$	$\Gamma(D)$

Figure 7.6: Meta-functions for Type Checking Switch Alternatives

meet	:	$(T \times T) \rightarrow T$
$\mathsf{meet}(b\{v_1\!:\!p_1\}, b\{v_2\!:\!p_2\})$	$\doteq$	$b\{v_1\!:\!p_1 \wedge p_2[v_2 := v_1]\}$
$\mathsf{meet}(x_1\!:\!s_1 \rightarrow t_1, x_2\!:\!s_2 \rightarrow t_2)$	$\doteq$	$x_1\!:\!\mathsf{meet}(s_1, s_2) \rightarrow \mathsf{meet}(t_1, t_2[x_2 := x_1])$
$\mathsf{meet}(\forall \alpha_1\!:\!k.t_1, \forall \alpha_2\!:\!k.t_2)$	$\doteq$	$\forall \alpha_1\!:\!k.\mathsf{meet}(t_1, t_2[\alpha_2 := \alpha_1])$

Figure 7.7: Conjoining Types

$D(\overline{z})$ the evaluated *result* e has type t. The rule CHK-ALT establishes this judgment in three steps.

1. We use $\mathsf{ctor}(\Gamma,\ D,\ y)$ summarized in Fig. 7.6 to get s, the monomorphic *instantiation* of the polymorphic type of constructor D. In other words, s is the type of D at *this* particular match-instance.

2. We invoke $\mathsf{unapply}(\Gamma,\ y,\ \overline{z},\ s)$ summarized in Fig. 7.6 to obtain the environment Γ' which is Γ extended with the types for the pattern match bindings $\overline{z}$ and also, with additional refinements for the scrutinee y that are revealed by matching against this particular pattern.

3. We check that result e has the type t in environment Γ'.

Unapply At destruction sites we use $\mathsf{unapply}(\Gamma,\ y,\ \overline{z},\ s)$ summarized in Fig. 7.6. The function unapply can be viewed as the dual of function application Given the output type of the constructed value (s), we want to (1) determine the types that the inputs ($\overline{z}$) must have had, and to then (2) add those bindings to get the environment used to check the alternative's body e. unapply does so by recursively "zipping" together the match-binders $\overline{z}$ with the input binders of the constructor's (function) type s. If the sequence of binders is *non-empty* ($z;\overline{z}$) and the constructor type is $x{:}s \rightarrow t$ then we extend Γ with the binding $z{:}s$, and recurse on the extended environment and the remaining binders $\overline{z}$ and the "rest" of the constructor type, *i.e.* its output t after substituting the formal x with the "actual" z. Once the sequence of binders is *empty* ($\emptyset$) then constructor type t is exactly the result of $D(\overline{z})$. Crucially, t can have extra information about the scrutinee y that holds under this particular pattern match, and so we *strengthen* the type of y by using meet, shown in Fig. 7.7, to conjoin the old type $\Gamma(y)$ with the pattern-match result t, and return the extended environment as the final result (that is used to check the alternative's body e.)

***Example: Checking in* `insert`** Let's see how the declarative rules let us *check* the implementation of the `insert` function from Fig. 7.1. Suppose our goal is to check that `insert` implements the type

$$x:\alpha \rightarrow ys:\mathtt{OL}[\alpha] \rightarrow \mathtt{OL}^{+}[\alpha, ys]$$

i.e. that `insert` returns an ordered list with one more element than the input list ys. The Fig. 7.8 shows a fragment of the definition of `insert` elaborated with explicit type applications at constructor application sites. Rule Chk-Lam reduces type checking to the following judgment that checks the body of `insert` against the specified output type in the context Γ comprising the bindings for the inputs x and ys:

$$\Gamma \ \vdash\ \textbf{switch}\ ys\ \{\ \mathtt{OC}(y, ys') \rightarrow e' \ \mid\ \ldots\} \Leftarrow\ \mathtt{OL}^{+}[\alpha, ys]$$

The rule Chk-Alt reduces the above into obligations for each alternative. For the `OC` case, we get:

$$\Gamma\ \mid\ ys\ \vdash\ \mathtt{OC}(y, ys') \rightarrow e' \Leftarrow\ \mathtt{OL}^{+}[\alpha, ys]$$

$$
\begin{array}{l}
\lambda x,\ ys.\ \textbf{switch}\ (ys) \\
\qquad\qquad\qquad \mid \texttt{OC}(y, ys') \ \rightarrow \\
\qquad\qquad\qquad\quad \textbf{if}\ x \le y\ \textbf{then} \\
\qquad\qquad\qquad\qquad \textbf{let}\ tl = \texttt{OC}[\alpha_x]\ y\ ys'\ \textbf{in} \\
\qquad\qquad\qquad\qquad \texttt{OC}[\alpha]\ x\ tl \\
\qquad\qquad\qquad\quad \textbf{else}\ \ldots \\
\qquad\qquad\qquad \mid \texttt{ON} \ \rightarrow \ldots
\end{array}
$$

Figure 7.8: Definition of `insert` (Fig. 7.1) with elaborated type applications.

By the definition of unapply the above judgment reduces to

$$\Gamma' \ \vdash\ e' \Leftarrow\ \texttt{OL}^{+}[\alpha, ys]$$

where Γ' is Γ extended with the bindings computed by unapply

$$\Gamma' \ \doteq\ \Gamma;\ y\!:\!\alpha;\ ys'\!:\!\texttt{OL}[\alpha\{v\!:\!y \le v\}];\ ys\!:\!\texttt{OL}^{+}[\alpha, ys']$$

As e' is $\textbf{if}\ x \le y\ \textbf{then}\ e'_{\le}\ \textbf{else}\ \ldots$ by rule Chk-If, eliding the else-branch, and recalling that $e'_{\le} \doteq \textbf{let}\ tl = e_y\ \textbf{in}\ e_x$ (Fig. 7.9) the above judgment reduces to

$$\Gamma';\ x \le y\ \vdash\ \textbf{let}\ tl = e_y\ \textbf{in}\ e_x \Leftarrow\ \texttt{OL}^{+}[\alpha, ys] \tag{7.4}$$

In the sequel, we will show how the synthesis rules establish

$$\Gamma'_{\le}\ \vdash\ e_y \Rightarrow t_y \tag{7.5}$$

$$\Gamma'_{\le}; tl\!:\!t_y\ \vdash\ e_x \Rightarrow \texttt{OL}^{+}[\alpha, tl] \tag{7.6}$$

which Chk-Let and Chk-Syn combine with the subtyping judgment

$$\Gamma'_{\le}; tl\!:\!t_y\ \vdash\ \texttt{OL}^{+}[\alpha, tl] <:\ \texttt{OL}^{+}[\alpha, ys] \quad \text{previously derived in (7.3)}$$

to yield the goal (7.4). □

7.3.4 Synthesis

As we said at the outset, the key mechanism to supporting datatypes are refined *data constructors*. Their types are applied at construction

$$
\begin{array}{lcl}
t_{\mathtt{ins}} & \doteq & x\!:\!\alpha \rightarrow ys\!:\!\mathtt{OL}[\alpha] \rightarrow \mathtt{OL}^{+}[\alpha, ys] \\
\mathtt{OL}^{+}[\alpha, z] & \doteq & \mathtt{OL}[\alpha]\{v\!:\!\mathtt{len}(v) = 1 + \mathtt{len}(z)\} \\
\Gamma & \doteq & \alpha\!:\!B;\ \mathtt{insert}\!:\!t_{ins};\ x\!:\!\alpha;\ ys\!:\!\mathtt{OL}[\alpha] \\
\Gamma' & \doteq & \Gamma;\ y\!:\!\alpha;\ ys'\!:\!\mathtt{OL}[\alpha\{v\!:\!y \le v\}];\ ys\!:\!\mathtt{OL}^{+}[\alpha, ys'] \\
\Gamma'_{\le} & \doteq & \Gamma';\ x \le y \\
e' & \doteq & \mathbf{if}\ x \le y\ \mathbf{then}\ e^{C}_{\le}\ \mathbf{else}\ \ldots \\
e'_{\le} & \doteq & \mathbf{let}\ tl = e_y\ \mathbf{in}\ e_x \\
e_y & \doteq & \mathtt{OC}[\alpha_x]\ y\ ys' \\
e_x & \doteq & \mathtt{OC}[\alpha]\ x\ tl \\
\alpha_x & \doteq & \alpha\{v\!:\!x \le v\} \\
t_y & \doteq & \mathtt{OL}^{+}[\alpha_x, ys']
\end{array}
$$

Figure 7.9: Definitions for checking and synthesizing types for `insert` Fig. 7.1.

sites in a manner identical to plain function application, where given the input type we synthesize the output. Hence, to synthesize the types for constructor applications we add the rule Syn-Data shown in Fig. 7.10. The rule synthesizes the type for a data constructor D by looking up its type in the environment Γ; after this, the rules for application *i.e.* Syn-App from Fig. 3.5 suffice for constructor applications.

***Example: Synthesis in* `insert`** Let's see how the synthesis rules let us establish the judgments (7.5) and (7.6) that are needed to check the **`then`** branch of the implementation of `insert` from Fig. 7.1.

First, let's establish the type synthesized for the constructor application of 7.5 (where the abbreviations α_x, t_y are summarized in Fig. 7.9):

$$\Gamma'_{\le}\ \vdash\ \mathtt{OC}[\alpha_x]\ y\ ys' \Rightarrow \mathtt{OL}^{+}[\alpha_x, ys']$$

Syn-Data synthesizes the polymorphic type of the constructor `OC` (7.1) and Syn-TApp synthesizes its instantiation to yield

$$\Gamma'_{\le}\ \vdash\ \mathtt{OC}[\alpha_x] \Rightarrow z\!:\!\alpha_x \rightarrow zs\!:\!\mathtt{OL}[\alpha_x\{v\!:\!z \le v\}] \rightarrow \mathtt{OL}^{+}[\alpha_x, zs] \qquad (7.7)$$

The rule Syn-App requires that the types of the first and second arguments y and ys' must be subtypes of the constructor's respective input types. The subtyping for the first argument y follows from the

Type Synthesis $\boxed{\Gamma \vdash e \triangleright t}$

$$\frac{\Gamma(D) = t}{\Gamma \vdash D \triangleright t}\textsc{Syn-Data}$$

Figure 7.10: λ_δ: Rules for Type Synthesis

validity of the entailment

$$\forall x, y, v.\ x \leq y \ \Rightarrow\ v = y \ \Rightarrow\ x \leq v$$

The subtyping for the second argument was established in (7.2). Hence, Syn-App establishes (7.5).

Next, let's synthesize a type for the constructor application (7.6) in the environment $\Gamma'_{\leq}$ extended by binding t_y synthesized in (7.5) to tl:

$$\Gamma'_{\leq}; tl{:}t_y \ \vdash\ \mathtt{OC}[\alpha]\ x\ tl \Rightarrow \mathtt{OL}^{+}[\alpha, tl]$$

Again via Syn-Data and Syn-TApp the constructor's instance type is

$$\Gamma'_{\leq}; tl{:}t_y \ \vdash\ \mathtt{OC}[\alpha] \Rightarrow z{:}\alpha \rightarrow zs{:}\mathtt{OL}[\alpha\{v{:}z \leq v\}] \rightarrow \mathtt{OL}^{+}[\alpha, zs] \qquad (7.8)$$

As $\Gamma(x) = \alpha$ we trivially get that the first argument x is a subtype of the first input type α. Similarly from the binding $tl : t_y$, the second argument tl is a subtype of the second input type $\mathtt{OL}[\alpha\{v{:}x \leq v\}]$, after substituting the parameter z for the actual x. Hence, by Syn-App and the type of the constructor (7.8), we arrive at our destination (7.6). □

7.4 Verification Conditions

The declarative rules for checking destructor types and synthesizing constructor types readily translate to an algorithm for verification condition generation.

Subtyping The subtyping constraint for two polymorphic datatypes types follows the rule Sub-Data. As shown in Fig. 7.11, the constraint is the conjuction of the constraint generated by the top-level refinements of the data types (c_o), the positive (c_p) and negative (c_n) type components.

sub	:	$(T \times T) \to C$
$\mathsf{sub}(C[\overline{s}]\{v_1 : p_1\},\ C[\overline{t}]\{v_2 : p_2\})$	$\doteq$	$c_o \wedge c_p \wedge c_n$
where		
$\overline{p}$	$=$	$\mathsf{polarities}(\delta,\ C)$
c_o	$=$	$\forall v_1 : b.\ p_1 \Rightarrow p_2[v_2 := v_1]$
c_p	$=$	$\bigwedge_{\forall i. p_i \in \{+, \pm\}} \mathsf{sub}(s_i,\ t_i)$
c_n	$=$	$\bigwedge_{\forall i. p_i \in \{-, \pm\}} \mathsf{sub}(t_i,\ s_i)$

Figure 7.11: Subtyping Constraints for λ_δ, extends cases of Fig. 6.6

Checking Recall that the check function takes an environment Γ, term e and type t and returns a Horn constraint c whose satisfiability implies $\Gamma \vdash e \triangleleft t$ (3.3). Fig. 7.12 shows how we extend the function to account for destructor (pattern-match) terms **switch** y $\{\overline{a}\}$ by invoking $\mathsf{checkAlt}(\Gamma,\ y,\ a,\ t)$ which implements rule Chk-Swt by computing the conjunction of the constraints for *each* alternative. Thus, the hard work is done by $\mathsf{checkAlt}(\Gamma,\ y,\ D(\overline{z}) \to e,\ t)$ which implements rule Chk-Alt. To do so, checkAlt first obtains the data constructor's type s at this instance. Next, it uses $\mathsf{unapply}(\Gamma,\ y,\ \overline{z},\ s)$ to obtain the environment Γ' extended with types for the pattern binders. Finally, it recursively invokes $\mathsf{check}(\Gamma',\ e,\ t)$ to recursively compute the constraint for the pattern-expression under the extended environment.

Synthesis The function synth, shown in Fig. 7.12, implements rule Syn-Data by returning the environment's (polymorphic) type for the constructor D. This lets us then handle constructor applications just like plain function applications.

Example: VC for **`insert`** Fig. 7.13 shows a part of the VC generated by invoking checkon the environment Γ, the definition of `insert` from Fig. 7.8 – with the type applications replaced with holes $\alpha\{\star\}$ – and the goal type t_{ins}. (The abbreviations Γ and t_{ins} are shown in Fig. 7.9.) The VC generation recurses into the body, to generate two constraints for the "cons" and "nil" cases respectively. The latter is elided for brevity. The former adds the pattern match binders for y and ys' and adds the

check	$:$	$(\Gamma \times E \times T) \to C$
$\mathsf{check}(\Gamma,\ \mathbf{switch}\ y\ \{\overline{a}\},\ t)$	$\doteq$	$\bigwedge_{a \in \overline{a}}\ \mathsf{checkAlt}(\Gamma,\ y,\ a,\ t)$
$\mathsf{checkAlt}$	$:$	$(\Gamma \times X \times A \times T) \to C$
$\mathsf{checkAlt}(\Gamma,\ y,\ D(\overline{z}) \to e,\ t)$	$\doteq$	$(\overline{z :: s}) \Rightarrow (y :: s') \Rightarrow c$
where		
c	$=$	$\mathsf{check}(\Gamma',\ e,\ t)$
$\Gamma; \overline{z\!:\!s}; y\!:\!s'$ as Γ'	$=$	$\mathsf{unapply}(\Gamma,\ y,\ \overline{z},\ s)$
s	$=$	$\mathsf{ctor}(\Gamma,\ y,\ D)$
synth	$:$	$(\Gamma \times E) \to (C \times T)$
$\mathsf{synth}(\Gamma,\ D)$	$\doteq$	$(\mathit{true},\ \Gamma(D) = t)$

Figure 7.12: Verification Condition Generation for λ_δ, extends cases of Fig. 6.6

hypothesis that the size of ys is one more than that of ys'. We then recurse into the **if**-expression: with one sub-constraint for the **then** branch (with the hypothesis $x \leq y$) and a second for the **else** branch, elided for brevity. Recall that the **then**-expression is

$$\mathbf{let}\ tl = e_y\ \mathbf{in}\ e_x$$

Where e_y and e_x are the two constructor applications in Fig. 7.9. The **then**-constraint has a sub-constraint c_y that arises from calling synth on e_y and which also produces the hypothesis that the size of tl is one more than that of ys'. Similarly, we then again invoke synth on e_x to get c_x and the result type: a list v whose length is one more than tl which must then imply the post-condition, that the output v's size is in fact, one greater than the input ys.

Next, let's look at the constraint c_y shown in Fig. 7.13, returned by synth on the constructor application term

$$e_y \ \doteq\ \mathtt{OC}[\alpha\{\star\}]\ y\ ys'$$

As described in section 6.4, we generate a fresh template $\alpha\{v\!:\!\kappa_y(v)\}$ for the hole, which means that at this site, the constructor OC has the

$$
\begin{aligned}
c \ &\doteq \forall x,\ ys. \\
&\qquad \forall y,\ ys'.\ \mathtt{len}(ys) = 1 + \mathtt{len}(ys') \ \Rightarrow \\
&\qquad\quad x \le y \ \Rightarrow c_y \ \wedge \\
&\qquad\qquad \forall tl.\ \mathtt{len}(tl) = 1 + \mathtt{len}(ys') \ \Rightarrow \ c_x \ \wedge \\
&\qquad\qquad\quad \forall v.\ \mathtt{len}(v) = 1 + \mathtt{len}(tl) \Rightarrow \mathtt{len}(v) = 1 + \mathtt{len}(ys) \\
&\qquad \wedge\, x \not\le y \ \Rightarrow \ \ldots \text{(constraint for } \textbf{else} \text{ branch)} \\
&\quad\ \wedge \ldots \text{(constraint for } \mathtt{ON} \text{ case)} \\
c_y \ &\doteq \forall v.\ v = y \ \Rightarrow \ \kappa_y(v) \ \wedge \\
&\quad\ \forall v.\ y \le v \ \Rightarrow \ (\kappa_y(v) \wedge y \le v) \\
c_x \ &\doteq \forall v.\ v = x \ \Rightarrow \ \kappa_x(v) \ \wedge \\
&\quad\ \forall v.\ \kappa_y(v) \ \Rightarrow \ (\kappa_x(v) \wedge x \le v)
\end{aligned}
$$

Figure 7.13: Verification Condition for definition of `insert` (Fig. 7.1).

following type (analogous to that shown in (7.7), except with $\kappa_y(v)$ instead of the to-be-inferred refinement $x \le v$:)

$$z{:}\alpha\{v{:}\kappa_y(v)\} \rightarrow zs{:}\mathtt{OL}[\alpha\{v{:}\kappa_y(v) \wedge z \le v\}] \rightarrow \mathtt{OL}^{+}[\alpha\{v{:}\kappa_y(v)\}, zs]$$

Thus, c_y has two conjuncts, one each for the subtyping obligations for the inputs y, ys'. The first conjunct ensures that the first argument y is indeed a subtype of the constructor's first input. The second conjunct checks that the second input ys' whose type, per unapply is $\mathtt{OL}[\alpha\{v{:}y \le v\}]$ (as shown in $\Gamma'_{\le}$ in Fig. 7.9) is a subtype of the second input of the constructor.

Finally, let's consider the constraint c_x shown in Fig. 7.13 that is returned by synth on the constructor application term

$$e_x \ \doteq \ \mathtt{OC}[\alpha\{\star\}]\ x\ tl$$

Again, we generate a fresh template for the hole $\alpha\{v{:}\kappa_x(v)\}$ which means that at this site, the constructor OC has the type

$$z{:}\alpha\{v{:}\kappa_x(v)\} \rightarrow zs{:}\mathtt{OL}[\alpha\{v{:}\kappa_x(v) \wedge z \le v\}] \rightarrow \mathtt{OL}^{+}[\alpha\{v{:}\kappa_x(v)\}, zs]$$

Hence, the constructor application yields the constraint c_x with one conjunct for each of the input arguments x and tl. As the output type of the *first* constructor application, and so the type of tl is $\mathtt{OL}^+[\alpha\{v{:}\kappa_y(v)\}, ys']$ the second conjunct of c_x – which represents the subtyping obligation for the input argument tl – has the antecedent $\kappa_y(v)$.

Solution We encourage the reader to verify that the assignment

$$\begin{aligned} \kappa_y(v) &\doteq x \leq v \\ \kappa_x(v) &\doteq \mathit{true} \end{aligned}$$

satisfies the Horn constraint in Fig. 7.13, by yielding, after substitution, a valid VC, thereby verifying that `insert` implements $t_{\mathtt{ins}}$. □

7.5 Discussion

To sum up, in this chapter we saw how to extend the language to support refinements on user defined algebraic data-types. We saw how to encode three classes of properties – invariants of individual datum, properties relating multiple data, and properties of the structure itself – using a single mechanism: a refined type for the data constructors. This approach lets us reuse the rules for function *application* to *generalize* properties when *constructing* the structure. Dually, it lets us define a notion of *un-application* to let us *instantiate* properties when *destructing* the structure.

Refinements on Data vs. Type Constructors Our approach is using refinements of data constructors. An alternative approach would be to refine type constructors. For example, the ordered lists of section 7.1.2 could be alternatively defined as `list int {l: isOrdered l}`, where `isOrdered` is a boolean function that checks is the input list is ordered. Sekiyama *et al.*, 2015 compare the two alternative approaches and conclude that refinements of type constructors are easier for the programmer to *specify*, but refinements on data constructors are much easier to (automatically) *verify*.

Types as a Decision Procedure The ease of verification arises from the fact that refined data constructors give rise to a simple syntax-directed approach to establish invariants of complex heap-allocated data structures. Crucially, this approach does not require the expensive blur and materialize mechanisms of shape analysis (Sagiv *et al.*, 2002), or the undecidable (universal) quantifier-based reasoning used by Floyd-Hoare logic based deductive verifiers like Dafny (Leino, 2010). Instead, *subtyping* provides a syntactic proof system for checking entailment between quantfied assertions. Subtyping lets us prove the (quantified) assertion that if every element of a list is positive then every element is non-negative, by proving the (quantifier-free) assertion that if v is positive then v is non-negative. Similarly, the constructor application is a syntactic heuristic for generalizing facts about individual elements into facts about the whole structures: *e.g.* if `x` is a `nat` and `xs` is a `list(nat)`, then `Cons(x, xs)` is a `list(nat)`. Finally, the destructor sites provide a syntactic heuristic for instantiating quantified facts about the whole structure: *e.g.* if `xs` is a `list(nat)` and is destructed to `Cons(x, xs')` then `x` is a `nat`.

Refinements for Pointer-based Structures While this chapter develops this ideas for a purely functional language, related ideas have been proposed for imperative languages. Notable examples include (Qiu *et al.*, 2013) which introduces the idea of "Natural Proofs" which shows how separation-logic based verifiers need only fold and unfold recursive predicates (which encode the same information as our recursive types) at the equivalent of constructor application and destruction sites, and (Bakst and Jhala, 2016) which shows how to add refinements to an alias type system (Walker and Morrisett, 2000) that precisely tracks locations and aliasing, yielding an expressive and automated way to reason about invariants of pointer-based data structures.

8

Refinement Polymorphism

Modern programming languages allow code and data to abstract over the concrete types that they work with. For example, it is commonplace to define polymorphic arrays or hash-tables that can, *e.g.* hold `int` or `string` or other values, and to write type-agnostic functions that operate over these structures. In § 6 we saw how polymorphic types could be instantiated at different use-sites to precisely track invariants in a context-sensitive manner.

However, we also often write monomorphic functions and data that nevertheless abstract over the refinements satisfied by the data they manipulate. This abstraction is often implicit, *i.e.* not captured in the function's type specification, which makes it hard to establish the invariants at usage sites. Hence, next, let's develop a means for specifying signatures that abstract over the refinements, see how to automatically verify these signatures, and learn how to instantiate them automatically at client sites, all while keeping verification efficiently decidable.

8.1 Examples

Let's crack our knuckles with some illustrative examples.

8.1.1 Problem: Picking the Right Specification

Consider `maxI(x,y)` which returns the larger of `x` and `y`.

```
val maxI : int => int => int
let maxI = (x, y) => {
  if (x < y) { y } else { x }
};
```

Suppose that we want to verify that `maxI(a, b)` must be non-negative if `a` and `b` are non-negative.

```
val bigger: nat => nat => nat
let bigger = (a, b) => { maxI(a, b) };
```

Unfortunately, the above `bigger` will *fail* as the specified type is silent about what properties hold of the `int` returned by `maxI`!

Premature Specialization We could try to type `maxI` as

```
val maxI : nat => nat => nat
```

This signature would let us verify `bigger` but would prematurely restrict `maxI` to non-negative values. For example, if we had a refinement for valid 8-bit `int`s

```
type int8 = int[v|0 <= v && v < 256]
```

we would end up *rejecting* the perfectly correct function below

```
val brighter : int8 => int8 => int8
let brighter = (c1, c2) => {
  maxI(c1, c2)
}
```

Instead, we might specify that the output

```
val maxI: x:int => y:int => int[v|v=x || v=y]
```

This would let us verify `bigger` and `brighter`. This approach suffices when there are just two `int` inputs but does not scale up to unbounded collections, as in `maxL` which computes the largest element of a *list*

```
let rec maxL = (default, xs) => {
  fold_right(maxI, default, xs)
};
```

Since the list is unbounded we have no way to say the output is *one of* the elements of xs. We could use an *existential quantifier* but that would lead to verification conditions that are outside the boundaries of decidable SMT based validity checking. Now suppose we use maxL with different kinds of values, for example:

```
val biggerL : list(nat) => nat
let biggerL = (xs) => { maxL(0, xs) }

val brighterL : list(int8) => int8
let brighterL = (xs) => { maxL(0, xs) }
```

How can we verify that maxList returns a nat or int8 values when invoked with lists of nat and int8 values respectively?

8.1.2 Solution: Abstracting Over Refinements

If we take a step back, we might notice that the fact that maxI returns *one of* its two inputs x and y can be rephrased as follows: if there is some property p that *both* inputs satisfy, then the output must also satisfy p. That is, we can make the refinement p itself be a *parameter* in the specification of maxI

```
val maxI: int[v|p v] => int[v|p v] => int[v|p v]
```

The above specification says that *for any* predicate p over int values, maxI takes two inputs x and y which respectively satisfy p x and p y and returns an output v that satisfies p v.

It is only useful to parameterize over refinements if there is some convenient way to *instantiate* the parameters. We will extend the machinery for inferring refinements for holes $\{\star\}$ to automatically instantiate the refinement parameters at the usage-sites — *e.g.* in bigger and brighter — where p will be instantiated to the concrete refinements

$$p_{\mathtt{nat}} \doteq \lambda v.\ 0 \leq v \tag{8.1}$$

$$p_{\mathtt{int8}} \doteq \lambda v.\ 0 \leq v \wedge v < 256 \tag{8.2}$$

thereby verifying those two client functions.

The above method scales up to unbounded lists. We can specify

```
val maxL: int[v|p v] => list(int[v|p v]) => int[v|p v]
```

which says that when `maxL` is given a *default* `int` and a `list(int)` all of which satisfy the predicate `p`, the output is also an `int` that satisfies `p`. Once again, we can verify the clients `biggerL` and `brighterL` by instantiating `p` as p_{nat} and p_{int8} at the respective call-sites.

Preserving Decidability At first glance, it may appear that these abstract predicate variables `p` have taken us into the realm of higher-order logics, and that we must leave decidable SMT based checking at the door. Fortunately, that is not the case. We will see how to encode abstract refinements `p` as *uninterpreted functions* in the refinement logic, which will allow us to continue with SMT based Horn VC checking.

8.1.3 Abstracting Refinements over Data Types

Refinement parameters are a natural fit for *datatype* definitions.

Dependent Pairs For example, we can specify a `pair` datatype where there is some relationship between the first and second component, by parameterizing the datatype definition with a refinement parameter `p`:

```
type pair('a, 'b)(p : 'a => 'b => bool) =
  | MkPair(x:'a, y:'b[v|p x v])
```

The definition is parameterized by a binary predicate `p` that relates the two elements of the `pair`. Hence, we can define the set of `int` pairs where the second component exceeds the first as:

```
type incPair = pair(int,int)((x, y) => x < y)
```

The refinement parameter is *asserted* when the pair is *constructed*. Hence, `okPair` will verify but `badPair` will be rejected:

```
val okPair: n:int => incPair
let okPair = (n) => { MkPair(n, n+1) };

val badPair: x:int => incPair
let badPair = (n) => { MkPair(n, n-1) };
```

Dually, the refinement is *assumed* when the pair is *destructed*, and so the below verifies:

```
val chkPair : incPair => nat
let chkPair = (p) => {
  switch (p) {
```

```
    MkPair(x1, x2) => x2 - x1
  }
};
```

Abstracting Refinements over Lists The combination of recursion and refinement parameters allows us to compactly specify various interesting properties of collections without baking them into the datatype's definition. Recall the definition of ordered lists of non-decreasing values from section 7.1

```
type olist('a) =
  | ONil
  | OCons (x:'a, xs:olist('a[v|x<=v]))
```

Order is established by the signature for `OCons` that requires that every element of the tail `xs` is greater than the head `x`. Refinement parameters let us abstract the particular relation: we can specify a generic `list` as

```
type list('a)(p:'a =>'a => bool) =
  | Nil
  | Cons(x:'a, xs:list('a[v|p x v])((y,z) => p y z))
```

The definition is parameterized by a binary predicate `p` that relates *every* element of the tail with the head `x` at each application of the constructor `Cons`. That is, the specification says that if

$$\texttt{Cons}(x_1, \texttt{Cons}(x_2, \ldots \texttt{Cons}(x_n, \texttt{Nil}))) : \texttt{list}(\alpha)(p)$$

then for each $1 \leq i < j \leq n$ we have $p(x_i, x_j)$.

Ordered Lists We can now specify non-decreasing lists by instantiating the refinement parameter `p` appropriately:

```
type incList('a) = list('a)((x1, x2) => x1<=x2)

val checkInc : (int) => incList(int)
let checkInc = (x) => {
  Cons(x, Cons(x+1, Cons(x+2, Nil)))
};
```

Similarly, we can define non-increasing lists

```
type decList('a)  = list('a)((x1,x2) => x1>=x2)

val checkDec : (int) => decList(int)
```

```
let checkDec = (x) => {
  Cons(x+2, Cons(x+1, Cons(x, Nil)))
};
```

or *duplicate-free* lists, simply by changing the relation:

```
type uniqList('a) = list('a)((x1,x2) => x1!=x2)

val checkUnique : (int) => uniqList(int)
let checkUnique = (x) => {
  Cons(x+2, Cons(x, Cons(x+1, Nil)))
};
```

We can omit the refinement instantiation to write just `list('a)` to denote the trivially (un)refined instance `list('a)((x1,x2)=> true)`. Now, the machinery developed for reasoning about datatypes in section 7.1 let us verify properties of code manipulating (abstractly) refined lists, *e.g.* that the following insertion-sort produces ordered lists

```
val isort : list('a) => incList('a)
let isort = (xs) => {
  foldr(insert, Nil, xs);
};
```

8.2 Types and Terms

Fig. 8.1 summarizes how we extend the language of types to include abstraction over refinements, and the language of terms to include instantiation of refinement variables.

Refinement Variables We write ρ to denote refinement variables that abstract over concrete or specific refinements. Refinement variables ρ are of the form $\kappa : \overline{b} \rightarrow \texttt{bool}$, *i.e.* represent a `bool` valued *predicates* over some (non-empty) sequence of base types.

Abstracting over Refinements We abstract over refinements either in function signatures of the form $\forall \rho.\ t$ or in data type definitions $\langle C,\ \overline{\alpha : k / p},\ \overline{\rho / p},\ \overline{D : t} \rangle$ which are now parameterized by a (possibly empty) sequence of refinement variables and their respective polarities in addition to type variables from Fig. 7.2. For example, we might assign

Ref. Var.	ρ	$::=$	$\kappa : \overline{b} \rightarrow \texttt{bool}$	
Abs. Refine.	φ	$::=$	$\lambda \overline{x : b}.\ p$	
Datatypes	δ	$::=$	$\langle C,\ \overline{\alpha : k / p},\ \overline{\rho / p},\ \overline{D : t} \rangle$	
Base Types	b	$::=$	$\ldots$	*from Fig. 7.2*
		$\mid$	$C[\overline{t}](\overline{\varphi})$	*datatypes*
Types	t	$::=$	$\ldots$	*from λ_δ Fig. 7.2*
		$\mid$	$\forall \rho.\ t$	*ref. polymorphism*
Terms	e	$::=$	$\ldots$	*from λ_δ Fig. 7.2*
		$\mid$	$e[\star]$	*refinement application*

Figure 8.1: λ_δ: Syntax of Types and Terms

`maxI` the type

$$t_{\texttt{maxI}} \doteq \forall \kappa : \texttt{int} \rightarrow \texttt{bool}.\ \texttt{int}\{v : \kappa(v)\} \rightarrow \texttt{int}\{v : \kappa(v)\} \rightarrow \texttt{int}\{v : \kappa(v)\} \tag{8.3}$$

which captures the intuition that when given two $\texttt{int}\{v : \kappa(v)\}$ inputs, the function is guaranteed to return an $\texttt{int}\{v : \kappa(v)\}$ output, for any refinement κ on `int` values. In the sequel, we will elide the refinement variables' sort when it is clear from the context.

Implicit Refinement Application We instantiate refinements either in *refinement application* terms of the form $e[\star]$ or in *type-constructor application* types of the form $C[\overline{t}](\overline{\varphi})$, where φ is a *concrete refinement* $\lambda \overline{x : b}.\ p$ which is a boolean valued predicate over a set of variables $\overline{x : b}$. We leave the refinement instances *implicit* for terms (denoted by $\star$) as, like type instances, these are ubiquitous, and hence, should be automatically synthesized. For example, `brighter` is defined via the following term in which the refinement variable in the signature of `maxI` is implicitly instantiated at the usage site:

$$\textbf{let}\ \texttt{brighter} = \lambda x, y.\ \texttt{maxI}[\star]\ x\ y \tag{8.4}$$

Explicit Refinement Application However, we allow *explicit* refinement instances for data types as we often want to specify signatures where the constructor is refined with a particular concrete refinement. For example, the definitions of `incPair`, `incList`, and `decList` from sec-

Well-formedness $\boxed{\Gamma \vdash t : k}$

$$\frac{\Gamma; \kappa : \overline{b} \to \mathtt{bool} \vdash t : k}{\Gamma \vdash \forall \kappa : \overline{b} \to \mathtt{bool}.\ t : \star}\textsc{Wf-RAbs}$$

Figure 8.2: λ_ρ: Rules for Well-formedness

tion 8.1 are, respectively:

$$\textbf{type}\ \mathtt{incPair} = \mathtt{pair(int, int)}(\lambda x_1, x_2.\ x_1 < x_2) \tag{8.5}$$

$$\textbf{type}\ \mathtt{incList} = \mathtt{list(int)}(\lambda x_1, x_2.\ x_1 \leq x_2) \tag{8.6}$$

$$\textbf{type}\ \mathtt{decList} = \mathtt{list(int)}(\lambda x_1, x_2.\ x_1 \geq x_2) \tag{8.7}$$

8.3 Declarative Typing

Next, let's look at how the declarative typing rules are extended to accomodate abstract refinements.

8.3.1 Well-formedness

Rule Wf-RAbs shown in Fig. 8.2 states that a refinement polymorphic type is well-formed with kind $\star$, if the underlying (quantified) type is also well-formed.

8.3.2 Subtyping

Fig. 8.3 summarizes the new rules for subtyping. The rule Sub-CRef shows the rule for checking subsumption between two concrete refinements $\lambda \overline{x_1 : b}.\ p_1$ and $\lambda \overline{x_2 : b}.\ p_2$, by checking that p_1 is subsumed by p_2 after suitably renaming the bound variables. The second rule Sub-Con shows how to extend the rule for checking subtyping between two refined instances $C[\overline{s}](\overline{\phi})\{v_1 : p_1\}$ and $C[\overline{t}](\overline{\varphi})\{v_2 : p_2\}$ of a type constructor C, by checking the subsumption holds between the corresponding concrete refinements ϕ and φ of the two instances according to their polarity using Sub-CRef, and checking the subsumption of the base refinements p_1 and p_2 and type components as before.

Abs. Refinement Implication $\boxed{\Gamma \vdash \varphi_1 \prec: \varphi_2}$

$$\frac{\Gamma; \overline{x_1 : b} \vdash p_1 \Rightarrow p_2[\overline{x_2} := \overline{x_1}]}{\Gamma \vdash \lambda \overline{x_1 : b}.\ p_1 \prec: \lambda \overline{x_2 : b}.\ p_2}\textsc{Sub-CRef}$$

Subtyping $\boxed{\Gamma \vdash t_1 \prec: t_2}$

$$\frac{\begin{array}{c} \Gamma \vdash s_i \prec: t_i \text{ for each } i.p_i \in \{+, \pm\} \\ \Gamma \vdash t_i \prec: s_i \text{ for each } i.p_i \in \{-, \pm\} \\ \Gamma \vdash \phi_i \prec: \varphi_i \text{ for each } i.p_{ri} \in \{+, \pm\} \\ \Gamma \vdash \varphi_i \prec: \phi_i \text{ for each } i.p_{ri} \in \{-, \pm\} \\ \Gamma; v_1 : \{C[\overline{s}] : p_1\} \vdash p_2[v_2 := v_1] \qquad (\overline{p}, \overline{p_r}) = \mathsf{polarities}(\delta,\ C) \end{array}}{\Gamma \vdash C[\overline{s}](\overline{\phi})\{v_1 : p_1\} \prec: C[\overline{t}](\overline{\varphi})\{v_2 : p_2\}}\textsc{Sub-Con}$$

Figure 8.3: λ_ρ: Rules for Subtyping

***Example: Subtyping in* `incPair`** As an example, let

$$\begin{aligned} \Gamma &\doteq a : \mathtt{int} \\ \varphi &\doteq \lambda x, y.\ x = a \wedge y = a + 1 \\ \phi &\doteq \lambda x, y.\ x < y \end{aligned}$$

and consider the subtyping obligation

$$\Gamma \vdash \ \mathtt{pair}(\mathtt{int}, \mathtt{int})(\varphi) \ \prec: \ \mathtt{pair}(\mathtt{int}, \mathtt{int})(\phi)$$

that could arise in the course of checking that

$$\Gamma \vdash \ \mathtt{MkPair}(a, a + 1) \Leftarrow \mathtt{incPair}$$

where `incPair` is defined in 8.5. Rule Sub-Con reduces the subtyping obligation to the following concrete refinement subsumption

$$\Gamma \vdash \ \varphi \ \prec: \ \phi$$

and then Sub-CRef reduces the above to the entailment

$$\Gamma, x, y : \mathtt{int} \vdash \ (x = a \wedge y = a + 1) \ \Rightarrow \ (x < y)$$

that is readily validated by the SMT solver. □

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\frac{\rho = \kappa : \overline{b} \rightarrow \mathtt{bool} \quad \varphi = \lambda \overline{x}.\ f_\kappa(\overline{x}) \qquad \Gamma; f_\kappa : \overline{b} \rightarrow \mathtt{bool} \vdash e[\rho := \varphi] \triangleleft s[\rho := \varphi]}{\Gamma \vdash e \triangleleft \forall \rho.\ s}\ \textsc{Chk-RAbs}$$

Figure 8.4: λ_ρ: Rules for Type Checking

8.3.3 Checking

The rule Chk-RAbs checks that a term e implements the abstractly refined signature $\forall \rho.\ s$. The key idea is to use the refinement variable $\rho = \kappa : \overline{b} \rightarrow \mathtt{bool}$ to (1) generate a fresh *uninterpreted function symbol* f_κ, (2) substitute all occurrences of κ with f_κ in the term e and the type s, to respectively obtain $e[\kappa := f_\kappa]$ and $s[\kappa := f_\kappa]$ and then (3) perform the check on the substituted type and term, in an environment extended with a binding for the uninterpreted function f_κ.

Refinement Instantiation We replace all occurrences of the refinement variable $\rho \doteq \kappa : \cdot$ with an uninterpreted function f_κ via an operation $s[\rho := \varphi]$ shown in Fig. 8.6 which *instantiates* (or *substitutes*) a concrete refinement φ for a refinement variable ρ in a signature s. The operation traverses s to replace all occurrences of $\kappa(\overline{x})$ with $p[\overline{y} := \overline{x}]$ when $\rho \doteq \kappa : \cdot$ and $\varphi \doteq \lambda \overline{y}.\ p$, *i.e.* we replace the *parameters* of the refinement ($\overline{y}$) with the *arguments* ($\overline{x}$) in the concrete refinement p.

Example: Checking **`maxI`** Recall the implementation and specification of `maxI` (§ 8.1)

$$e_{\mathtt{maxI}} \doteq \lambda x, y.\ \mathbf{if}\ x < y\ \mathbf{then}\ y\ \mathbf{else}\ x \tag{8.8}$$

$$t_{\mathtt{maxI}} \doteq \forall \kappa.\ \mathtt{int}\{v : \kappa(v)\} \rightarrow \mathtt{int}\{v : \kappa(v)\} \rightarrow \mathtt{int}\{v : \kappa(v)\} \tag{8.9}$$

Let's consider the goal of verifying that the above implementation checks against its specification (8.3)

$$\emptyset \vdash\ e_{\mathtt{maxI}} \Leftarrow t_{\mathtt{maxI}} \tag{8.10}$$

The rule Chk-RAbs reduces the above to

$$\emptyset \vdash\ e_{\mathtt{maxI}} \Leftarrow \mathtt{int}\{v : f_\kappa(v)\} \rightarrow \mathtt{int}\{v : f_\kappa(v)\} \rightarrow \mathtt{int}\{v : f_\kappa(v)\}$$

ctor	:	$(\Gamma \times D \times X) \rightarrow T$
$\mathsf{ctor}(\Gamma,\ D,\ y)$	$\doteq$	$s[\overline{\alpha} := \overline{t}][\overline{\rho} := \overline{\varphi}]$
where		
$C[\overline{t}](\overline{\varphi})$	=	$\Gamma(y)$
$\forall\overline{\alpha}\!:\!k.\forall\overline{\rho}.\ s$	=	$\Gamma(D)$

Figure 8.5: Meta-functions for Checking Alternatives; unapply as in Fig. 7.6

which rules Chk-Lam, Chk-If and Syn-Var reduce to two goals

$$\Gamma \vdash\ \mathtt{int}\{v\!:\!v = x\}\ \prec:\ \mathtt{int}\{v\!:\!f_\kappa(v)\}$$
$$\Gamma \vdash\ \mathtt{int}\{v\!:\!v = y\}\ \prec:\ \mathtt{int}\{v\!:\!f_\kappa(v)\}$$

where Γ has bindings for the refinement and value parameters

$$\Gamma \doteq\ f_\kappa\!:\!\mathtt{int} \rightarrow \mathtt{bool}, x\!:\!\mathtt{int}\{v\!:\!f_\kappa(v)\}, y\!:\!\mathtt{int}\{v\!:\!f_\kappa(v)\}$$

The two goals above check that the specified output type is indeed returned in each branch. Both the above reduce to the entailment

$$\Gamma, v\!:\!\mathtt{int} \vdash\ v = y\ \Rightarrow\ f_\kappa(v)$$

that is easily confirmed by the SMT solver. □

Checking Case Alternatives We continue to check case alternatives using Chk-Alt from Fig. 7.5. However, we must extend the definition of $\mathsf{ctor}(\Gamma,\ D,\ y)$ — which defines the monomorphic instantiation of the polymorphic type of the data constructor D corresponding the case-scrutinee y — to also account for refinement polymorphism. This extension is summarized by the definition in Fig. 8.5, where we obtain both the monomorphic types $\overline{t}$ and additionally the concrete refinements $\overline{\varphi}$ from the environment Γ signature of y, and use those to respectively instantiate the type ($\overline{\alpha}$) and refinement ($\overline{\rho}$) variables in the signature of the data constructor D, where the latter is done via the refinement instantiation mechanism described above.

***Example: Checking* `chkPair`** Recall `chkPair` from § 8.1

$$e \doteq\ \lambda p.\ \mathbf{switch}\ p\ \{\mathtt{MkPair}(x, y) \rightarrow y - x\}$$

$$
\begin{array}{llcl}
\textbf{\textit{Predicates}} & \kappa(\overline{x})[\rho := \varphi] & \doteq & p[\overline{y} := \overline{x}] \text{ if } \rho = \kappa : \cdot,\ \ \varphi = \lambda\overline{y}.\ p \\
 & p_1 \bowtie p_2[\rho := \varphi] & \doteq & p_1[\rho := \varphi] \bowtie p_2[\rho := \varphi] \\
 & p[\rho := \varphi] & \doteq & p \\
\textbf{\textit{Abstr. Refs}} & (\lambda\overline{x : b}.\ p)[\rho := \varphi] & \doteq & \lambda\overline{x : b}.\ (p[\rho := \varphi]) \\
\textbf{\textit{Base Types}} & C[\overline{t}](\overline{\phi})[\rho := \varphi] & \doteq & C[\overline{t[\rho := \varphi]}](\overline{\phi[\rho := \varphi]}) \\
 & b[\rho := \varphi] & \doteq & b \\
\textbf{\textit{Types}} & b\{v : p\}[\rho := \varphi] & \doteq & b[\rho := \varphi]\{v : p[\rho := \varphi]\} \\
 & (x : s \rightarrow t)[\rho := \varphi] & \doteq & x : s[\rho := \varphi] \rightarrow t[\rho := \varphi]
\end{array}
$$

Figure 8.6: Instantiating a refinement variable ρ with a concrete refinement φ.

Let's see how the rules establish that

$$\emptyset \vdash\ e \Leftarrow \mathtt{incPair} \rightarrow \mathtt{nat}$$

First, Chk-Lam reduces the above to

$$p : \mathtt{incPair} \vdash\ \mathbf{switch}\ p\ \{\mathsf{MkPair}(x, y) \rightarrow y - x\} \Leftarrow \mathsf{nat}$$

Via Chk-Alt (Fig. 7.5) we extend the environment with the pattern-binders derived from $s \doteq\ \mathsf{ctor}(p : \mathtt{incPair},\ \mathtt{MkPair},\ p)$ *i.e.* the type of `MkPair` *at* this instance. As

$$\mathtt{MkPair} :: \ \forall\alpha, \beta : k.\forall\kappa.\ a : \alpha \rightarrow b : \beta\{\kappa(a, b)\} \rightarrow \mathtt{pair}[\alpha, \beta](\lambda a, b.\ \kappa(a, b))$$

as $p : \mathtt{incPair}$, we get

$$s \doteq\ a : \mathtt{int} \rightarrow b : \mathtt{int}\{a < b\} \rightarrow \mathtt{pair}[\mathtt{int}, \mathtt{int}](\lambda a, b.\ a < b)$$

Thus, $\mathsf{unapply}$ (Fig. 7.6) extends the environment with pattern binders

$$\Gamma' \doteq\ \Gamma,\ x : \mathtt{int},\ y : \mathtt{int}\{x < y\}$$

to obtain the goal for the case-alternative

$$\Gamma' \vdash\ y - x \Leftarrow \mathtt{nat}$$

which, Syn-Var and Syn-App reduce to the entailment

$$\Gamma, x, y, v : \mathtt{int} \vdash\ (x < y) \Rightarrow (v = y - x) \rightarrow (0 \leq v)$$

that is readily verified by the SMT solver. □

Type Synthesis $\boxed{\Gamma \vdash e \triangleright t}$

$$\frac{\rho = \kappa : \overline{b}, b \rightarrow \texttt{bool} \quad \Gamma \vdash e \triangleright \forall \rho.\ s \qquad b\{\star\} \triangleright b\{x : p\} \quad \Gamma; \overline{x : b} \vdash b\{x : p\} : B}{\Gamma \vdash e[\star] \triangleright s[\rho := \lambda \overline{x : b}, x : b.p]} \textsc{Syn-RApp}$$

Figure 8.7: λ_ρ: Rules for Type Synthesis

8.3.4 Synthesis

To make refinement polymorphism ergonomic, we need a way to automatically synthesize appropriate concrete refinements for each term $e[\star]$ corresponding to *uses* of terms e whose types abstract over the refinements. Otherwise, the programmer would have to bear the burden of writing concrete refinements at the ubiquitous usage sites, and worse, the resulting code would be very difficult to read! Thus, refinement synthesis is analogous to how we usually want to relieve the programmer of the burden of providing (monomorphic) instances at uses of a polymorphic signature. Next, let's see how this instantiation can be achieved by the refinement synthesis machinery introduced in λ_κ (§ 5).

Instantiating Refinement Variables The rule Syn-RApp shown in Fig. 8.7 shows how to synthesize types at refinement application sites $e[\star]$. The rule says if we synthesize for e a type $\forall \rho.\ s$ which uses an abstract refinement variable $\rho \equiv \kappa : \overline{b}, b \rightarrow \texttt{bool}$ then we can *instantiate* ρ in s with any *well-formed* concrete refinement $\lambda \overline{x : b}, x : b.\ p$ whose sort is compatible with that of ρ, *i.e.* that is parameterized by the sorts compatible with the inputs of κ.

Observe that Syn-RApp mimics Chk-Rec from Fig. 5.3 which uses the hole instantiation judgment $s \triangleright t$ from Fig. 5.2 to declaratively *guess* a suitable way to replace the holes $\{\star\}$ in s with concrete refinements. In essence, Syn-RApp views the refinement variable $\rho \equiv \kappa : \overline{b}, b \rightarrow \texttt{bool}$ which denotes a relation over values of types $\overline{b}, b$ as an *unknown* refinement over the base type $b\{\star\}$ and requires that we fill the hole with any concrete refinement p that is well-formed under an environment extended with (*i.e.* can refer to) binders $\overline{x}$ for the other parameters $\overline{b}$.

***Example: Synthesis in* `brighter`** Let's use Syn-RApp to verify `brighter` from § 8.4. Let

$$\begin{aligned}\texttt{int8} &\doteq \texttt{int}\{v{:}0 \le v < 256\}\\ \Gamma &\doteq \texttt{maxI}{:}t_{\texttt{maxI}}, x{:}\texttt{int8}, y{:}\texttt{int8}\end{aligned}$$

where $t_{\texttt{maxI}}$ is from (8.3). Using Syn-Var we have

$$\Gamma \vdash \texttt{maxI} \Rightarrow t_{\texttt{maxI}}$$

Hence, as $\texttt{int}\{\star\} \triangleright \texttt{int8}$ and $\Gamma \vdash \texttt{int8} : B$ Syn-RApp let us instantiate the refinement variable κ quantifying the signature of $t_{\texttt{maxI}}$ with the concrete refinement $\lambda x.\ 0 \le x < 256$ to obtain

$$\Gamma \doteq \texttt{maxI}[\star] \Rightarrow \texttt{int8} \rightarrow \texttt{int8} \rightarrow \texttt{int8}$$

Now, the function application rule Syn-App lets us establish

$$\Gamma \doteq \texttt{maxI}[\star]\ x\ y \Rightarrow \texttt{int8}$$

which, via rules Chk-Syn and Chk-Lam establishes that the implementation of `brighter` $\lambda x, y.\ \texttt{maxI}[\star]\ x\ y$ indeed checks against its specified type $\texttt{int8} \rightarrow \texttt{int8} \rightarrow \texttt{int8}$. □

8.4 Verification Conditions

The declarative rules simply guess suitable concrete refinements at each instantiation site. Next, let's see how to implement those rules via the method of Horn constraints introduced in § 5, as summarized in Fig. 8.8.

Checking To extend the `check` function to implement Chk-RAbs we add a case for checking a term e against a type of form $\forall\rho.\ s$ under environment Γ. To this end, we substitute the refinement variable ρ with a concrete refinement φ corresponding to an uninterpreted function application $\lambda\overline{x}.\ f_\kappa(\overline{x})$ and then return the Horn constraint c corresponding to the VC for checking the (substituted) term e' against the (substituted) type s'.

***Example: VC for* `maxI`** Let's see the VC $\texttt{check}(\emptyset,\ e_{\texttt{maxI}},\ t_{\texttt{maxI}})$ generated to verify that the implementation (8.8) of `maxI` adheres to its specification (8.9). The VC is obtained by substituting the refinement variable κ with the uninterpreted function symbol f_κ and then recursively invoking `check` on the substituted body and signature, which produces the VC:

$$\forall f_\kappa, x, y, v.\ f_\kappa(x) \Rightarrow f_\kappa(y) \Rightarrow ((v = x \Rightarrow f_\kappa(v)) \wedge (v = y \Rightarrow f_\kappa(v)))$$

which mirrors the entailments for the checking judgment (8.10). □

Synthesis Finally, in Fig. 8.8 we extend `synth` to account for refinement instantiation terms $e[\star]$ via an algorithmic implementation of SynRApp. First, we recursively invoke `synth` to As in the declarative rule, we recursively invoke `synth` to synthesize the refinement-polymorphic signature $\forall\rho.\ s$ for e. Next, as in Fig. 5.5 we implement the declarative $b\{\star\} \triangleright b\{x\!:\!p\}$ by using $\mathsf{fresh}(\Gamma; \overline{x\!:\!b},\ b\{\star\})$ to obtain a new *template* $b\{x\!:\!p\}$ where p contains refinement (Horn) variables for the unknown concrete refinements to be used for instantiation. We use p to build a concrete refinement that is substituted for ρ in s to return the synthesized type for $e[\star]$, together with the VC c obtained for e.

***Example: Checking of* `brighter`** Let's see how the above procedure computes a Horn constraint (VC) whose satisfiability implies that `brighter` (8.4) implements the specification `int8` → `int8` → `int8`. Let

$$\begin{aligned}\Gamma &\doteq \mathtt{maxI}\!:\!t_{\mathtt{maxI}}\\ e_{\mathsf{br}} &\doteq \lambda x, y.\ \ \mathtt{maxI}[\star]\ x\ y\\ t_{\mathsf{br}} &\doteq \mathtt{int8} \rightarrow \mathtt{int8} \rightarrow \mathtt{int8}\end{aligned}$$

respectively be the environment, implementation and specification for `brighter`, where $t_{\mathtt{maxI}}$ is from (8.3). Let

$$\Gamma' \doteq \Gamma,\ x\!:\!\mathtt{int8},\ y\!:\!\mathtt{int8}$$

By consulting Γ', $\mathsf{synth}(\Gamma',\ \mathtt{maxI})$ returns $(\mathit{true},\ t_{\mathtt{maxI}})$ where

$$t_{\mathtt{maxI}} \doteq \forall\kappa.\ \mathtt{int}\{v\!:\!\kappa(v)\} \rightarrow \mathtt{int}\{v\!:\!\kappa(v)\} \rightarrow \mathtt{int}\{v\!:\!\kappa(v)\}\}$$

Hence, we invoke $\mathsf{fresh}(\Gamma, \mathtt{int}\{\star\},)$ to obtain the template $\mathtt{int}\{\nu\!:\!\kappa_{\mathsf{br}}(\nu)\}$ with a new Horn variable κ_{br} denoting the unknown concrete refinement at this instantiation site. Upon substituting the corresponding concrete refinement for the refinement variable we get the template (eliding the trivial Horn constraint *true*)

$$\mathsf{synth}(\Gamma',\ \mathtt{maxI}[\star]) \doteq \mathtt{int}\{\nu\!:\!\kappa_{\mathsf{br}}(\nu)\} \to \mathtt{int}\{\nu\!:\!\kappa_{\mathsf{br}}(\nu)\} \to \mathtt{int}\{\nu\!:\!\kappa_{\mathsf{br}}(\nu)\}$$

Then, via the usual cases for function application, we get the VC

$$\begin{array}{lcl}
\mathsf{check}(\Gamma,\ e_{\mathsf{br}},\ t_{\mathsf{br}}) & \doteq & \forall x.(0 \leq x < 256) \Rightarrow \\
 & & \quad \forall y.(0 \leq y < 256) \Rightarrow \\
 & . & \qquad\quad \forall \nu.(\nu = x) \Rightarrow \kappa_{\mathsf{br}}(\nu) \\
 & & \qquad \wedge \ \ \forall \nu.(\nu = y) \Rightarrow \kappa_{\mathsf{br}}(\nu) \\
 & & \qquad \wedge \ \ \forall \nu.\kappa_{\mathsf{br}}(\nu) \Rightarrow (0 \leq v < 256)
\end{array}$$

The first two conjuncts check that the arguments x and y respectively satisfy the input types (preconditions) of $\mathtt{maxI}[\star]$, and the last conjunct stipulates that the output type (postcondition) of the call is a valid `int8` value, all assuming the values x and y inhabit `int8` as specified by the input type of `brighter`. The above VC can be satisfied by assigning $\kappa_{\mathsf{br}} \doteq \lambda a.\ 0 \leq a < 256$, and hence we verify that `brighter` correctly implements its specification. □

8.5 Discussion

In λ_ρ we saw how we often want to specify signatures that abstract over refinements, how these signatures can be checked using uninterpreted functions, and how we can extend the Horn-constraint based inference method from § 5 to automatically instantiate abstract refinements at usage sites. The method was introduced by Vazou *et al.*, 2013 who illustrated a variety of applications in establishing invariants of data structures. Gordon *et al.*, 2017 demonstrated that abstract refinements could be used to encode a form of concurrent rely-guarantee reasoning, enabling the verification of implementations of lock-free data structures. Subsequent work by Polikarpova *et al.*, 2016 showed how refinement polymorphism allows writing compact specifications from which implementations can be automatically *synthesized*.

check	:	$(\Gamma \times E \times T) \to C$
$\mathsf{check}(\Gamma,\ e,\ \forall\rho.\ s)$	$\doteq$	$(f_\kappa :: \overline{t} \to \mathtt{bool}) \Rightarrow c$
where		
c	$=$	$\mathsf{check}(\Gamma',\ e[\rho := \varphi],\ s[\rho := \varphi])$
Γ'	$=$	$\Gamma; f_\kappa : \overline{t} \to \mathtt{bool}$
φ	$=$	$\lambda\overline{x}.\ f_\kappa(\overline{x})$
$\kappa : \overline{t} \to \mathtt{bool}$	$=$	ρ
synth	:	$(\Gamma \times E) \to (C \times T)$
$\mathsf{synth}(\Gamma,\ e[\star])$	$\doteq$	$(c, s[\rho := \lambda\overline{x:b}, x:b.p])$
where		
$(c, \forall\rho.\ s)$	$=$	$\mathsf{synth}(\Gamma,\ e)$
$\kappa : \overline{b}, b \to \mathtt{bool}$	$=$	ρ
$\overline{x}$	$=$	fresh variables of sort $\overline{b}$
$b\{x:p\}$	$=$	$\mathsf{fresh}(\Gamma; \overline{x:b},\ b\{\star\})$

Figure 8.8: Algorithmic Checking for λ_ρ, extends cases of Fig. 7.12

Bounded Quantification The abstract refinements in λ_ρ were completely unconstrained. However, we could imagine a form of *bounded quantification* for refinement variables analogous to type variables (Canning *et al.*, 1989), which would restrict instantiating refinements to those satisfying particular conditions. Such an extension was explored by Vazou *et al.*, 2015 who showed how the programmer can use Horn constraints to express bounds which allow specifying expressive signatures whilst preserving decidable and automatic verification. These extensions enable, for example, encoding a monadic information flow control (IFC) mechanism, purely within refinement types (Polikarpova *et al.*, 2020) enabling the construction of web applications adhering to expressive data-sensitive security policies (Lehmann *et al.*, 2020).

9

Termination

The problem of verifying that the execution of a certain piece of code always terminates is perhaps one of the oldest in computing, going back all the way to Turing, 1936's work on the undecidability of the Halting Problem. Of course, just because a problem is undecidable, doesn't mean that it goes away! Indeed, as Rice, 1953 pointed out *all* non-trivial semantic properties of programs are undecidable. That is, it is just as undecidable to guarantee that, *e.g.* a program will not attempt to add `int` and `bool` values. Yet, we have developed syntactic disciplines that have turned verifying the absence of errors like adding `int` and `bool` values into into a routine part of compiling code. Next, let's see how refinements allow us to develop a simple and practical discipline for verifying, at compile time, that functions always terminate.

9.1 Examples

Let's see some examples that illustrate how to verify termination.

Well-founded Metrics Consider the function `sum` which adds up the numbers from `0` to `n`:

```
val sum : n:nat => nat
let rec sum = (n) => {
```

```
    if (n < 1) {
      0
    } else {
      n + sum(n-1)
    }
  }
```

Why *does* `sum n` terminate? First, notice that `sum` will not terminate if invoked on a negative number, *e.g.* `sum(-3)` will diverge. We eliminate this possibility by requiring the precondition that `n:nat` *i.e.* the inputs be non-negative. Now, when `n` equals `zero`, the procedure simply returns the result `0`. Otherwise, it recurses on a *strictly* smaller input, until, ultimately, it reaches `0`, at which point it terminates.

Proving Termination by Induction Thus we can, somewhat more formally, prove termination *by induction* on `n`.

- *Base case* `sum` terminates for inputs $k = 0$.
- *Induction Hypothesis* Assume `sum` terminates on all $k \leq n$.
- *Inductive Step* Check that `sum(n+1)` only recursively invokes `sum(n)` which satisfies the induction hypothesis and hence terminates.

This reasoning suffices to convince ourselves that `sum(n)` terminates for every non-negative `n`. That is, we have shown that `sum` terminates because a *well-founded* metric: here the non-negative `n` is *strictly decreasing* at each recursive call.

Proving Termination with Types We can capture the above reasoning via the type system as follows. First we require that `sum` only be called with non-negative `nat` values, which were defined as

```
type nat = int[v|0 <= v]
```

Second, we to ensure that the recursion is on *strictly smaller* values, we need only typecheck the *implementation* of `sum` in an environment that requires `sum` only be called with inputs smaller than `n`, *i.e.* we check the body in a environment of the form

$$\Gamma_{\mathsf{sum}} \doteq n{:}\mathsf{nat},\ \mathsf{sum}{:}n'{:}\mathsf{int}\{0 \leq n' < n\} \rightarrow \mathsf{nat}$$

The above ensures that any (recursive) call in the body only calls `sum` with inputs smaller than the "current" parameter `n`. Notice that if we had not required `n` to be non-negative, then the parameter `n-1` passed in at the recursive call would be smaller than `n` but would not be non-negative, and hence, would fail the strengthened precondition for `sum`, as indeed it should, as such a computation does not terminate!

Recursion on Multiple Parameters The above method works even when there are multiple parameters, as long as there is *some* `nat`-valued parameter that is used to limit the recursion. Consider the following tail-recursive variant of `sum`

```
let rec sumT = (total, n) => {
  if (n == 0) {
    total
  } else {
    sumT(total + n, n - 1)
  }
}
```

The function `sumT(total, n)` takes two parameters: `n` as before, and `total` which holds the accumulated summation of the "previously seen" seen. That is, `sumT(0,3)` evaluates as follows:

```
sumT(0,3) ⇢ sumT(3,2) ⇢ sumT(5,1) ⇢ sumT(6,0) ⇢ 6
```

Specifying Termination Metrics The accumulation parameter `total` is not strictly decreasing. However, the parameter `n` *is* decreasing and non-negative and serves to witness that `sumT` always terminates. But how might the type-checker *guess* that it should use `n` instead of `total`? While one can imagine a variety of pragmatic and effective heuristics to make such guesses, for our purposes, we shall simply give the type checker an explicit *termination metric*

```
val sumT: total:nat => n:nat => nat / n
```

In the above, we end the type signature for `sumT` with `/ n` to denote that the value `n` should be used as the termination metric. The typechecker will verify that the value of the metric `n` is indeed well-founded: *i.e.* non-negative and strictly decreasing at each recursive call, and if so, will deem the function terminating.

Metric Expressions Metrics generalize to situations where no single parameter is decreasing, but some *expression* over the parameters is. For example, consider the function `range(i, j)` which returns the list of `int`egers between `i` and `j`

```
val range : i:int => j:int => list(int) / j - i
let rec range = (i, j) => {
  if (i < j) {
    Cons(i, range(i+1, j))
  } else {
    Nil
  }
}
```

In the above, neither argument is decreasing: `i` *increases* at each call, and `j` is unchanged. Nevertheless, the function terminates as the *gap* between `i` and `j` diminishes at each recursive call, and the function terminates when that gap reaches `0`. We can make this intuition precise via the termination metric `/ j-i`. Armed with this information, the type checker ensures that at each recursive call in the body, the value of `j-i` is decreasing and non-negative. That is, we will check the implementation of `range` in an environment

$$\Gamma_{\texttt{range}} \doteq i, j\mathpunct{:}\texttt{int},\ \texttt{range}\mathpunct{:}i'\mathpunct{:}\texttt{int} \rightarrow j'\mathpunct{:}\texttt{int}\{0 \leq j' - i' < j - i\} \rightarrow \texttt{int} \tag{9.1}$$

that stipulates that recursive calls to `range` must have strictly smaller, non-negative gaps, to verify that `range` indeed always terminates.

Lexicographic Metrics Sometimes, it is convenient to split up the termination metric across *multiple* smaller metrics. For example consider Ackermann's function

```
val ack : m:nat => n:nat => nat / m, n
let rec ack = (m, n) => {
  if (m == 0) { n + 1 } else {
    if (n == 0) { ack (m - 1, 1) } else {
      ack (m - 1, ack (m, n - 1))
    }
  }
}
```

Why does `ack` terminate? At each iteration either the *first* parameter `m` decreases, or `m` remains the same and the *second* parameter `n` decreases.

Each time that `n` reaches `0`, it cannot decrease further so `m` must decrease. Hence, `m` will eventually reach `0` and `ack` will terminate. In other words, the *pair* `(m, n)` decreases in the *lexicographic order* on pairs, which is a well-ordering that has no infinite descending chains (Baader and Nipkow, 1998).

Specifying Lexicographic Orders via Types We can extend our notion of metrics to account for lexicographic orders by allowing the user to write a *sequence* of metrics. For example, we type `ack` with the signature with the termination metric `/ m, n` and then we will use the sequence to check the implementation of `ack` in environment Γ_{ack}

$$m, n : \mathsf{nat},\ \mathsf{ack} : m' : \mathsf{nat} \to n' : \mathsf{nat}\{m' < m \vee (m' = m \wedge n' < n)\} \to \mathsf{nat}$$

The signature for `ack` limits recursive uses of `ack` to parameters that satisfy the lexicographic ordering to ensure that `ack` terminates.

Structural Recursion Often the recursion is over the elements of a datatype like a list or a tree. For example, consider the function that appends two lists `xs` and `ys`

```
val append : xs:list('a) => ys:list('a) => list('a)
           / len(xs)
let rec append = (xs, ys) => {
  switch (xs) {
    | Nil          => ys
    | Cons (x, xs') => Cons (x, append (xs', ys))
  }
}
```

The function `append` recurses on the *tail* of the first list `xs` and stops when that list is empty *i.e.* equal to `Nil`. This is a form of *structural recursion* where each recursive call is over sub-structures (*e.g.* the tail) of some input parameter. We can verify the termination of structurally recursive functions by using *measures* to specify suitable metrics. For `append` we specify the metric `/ len(xs)` which tells the type checker to limit recursive calls in the implementation of `append` to lists whose length is smaller than `xs`. That is, we check the implementation of `append` in an environment where $xs, ys : \mathtt{list}(\alpha)$, and `append` is limited to

$$xs' : \mathtt{list}(\alpha)\{\mathtt{len}(xs') < \mathtt{len}(xs)\} \to ys' : \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$$

Metric	m	$::=$	p	*decreasing expression*
		$\mid$	p, m	*lexicographic metric*
Terms	e	$::=$	$\ldots$	*from Fig. 8.1*
		$\mid$	$\texttt{let rec}\ x = e_1 : t/m\ \texttt{in}\ e_2$	*recursive binder*

Figure 9.1: λ_τ: Syntax of Types and Terms

Non-Structural Recursion Finally, the notion of metrics over measures scales up to account for more general scenarios where the recursion over the datatypes is not structural. For example, consider the function `braid` which takes two lists $x_1, \ldots$ and $y_1, \ldots$ and returns the list $x_1, y_1, \ldots$

```
val braid : xs:list('a) => ys:list('a) => list('a)
          / len(xs) + len(ys)
let rec braid = (xs, ys) => {
  switch (xs) {
    | Nil => ys
    | Cons (x, xs') => Cons(x, braid(ys, xs'))
  }
}
```

The recursion in `braid` is not structural: the recursive call flips the order of the lists to ensure the values alternate in the output. However, in this case, the *sum* of the lengths of the two input lists shrinks. We specify this via the metric `/ len(xs) + len(ys)` which tells the type checker to check the body of `braid` under the following environment

$$xs,\ ys{:}\texttt{list}(\alpha),\ \texttt{braid}{:}xs'{:}\texttt{list}(\alpha) \rightarrow ys'{:}\texttt{list}(\alpha)\{p\} \rightarrow \texttt{list}(\alpha)$$

where the refinement

$$p \ \doteq\ 0 \leq \texttt{len}(xs') + \texttt{len}(ys') < \texttt{len}(xs) + \texttt{len}(ys)$$

limits recursive calls to parameters the sum of whose lengths are decreasing, to verify that `braid` terminates.

9.2 Types and Terms

As demonstrated by the examples, λ_τ requires two small extensions to its syntax: a way to specify termination metrics, and a means of specifying metrics in type signatures, as summarized in Fig. 9.1.

Metric Well-formedness $\boxed{\Gamma \vdash m}$

$$\frac{\Gamma \vdash p : \mathtt{int}}{\Gamma \vdash p}\textsc{Wfm-Base} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash m}{\Gamma \vdash p, m}\textsc{Wfm-Lex}$$

Figure 9.2: Rules for Checking Metric Well-formedness

Metrics A *termination metric* (or just metric in brief) is either a single *decreasing expression* p which is an `int`-sorted term from the refinement logic (Fig. 2.1), or a *lexicographic metric* comprising a sequence of decreasing expressions.

Recursive Signatures In λ_τ we require that recursive `rec` binders be annotated with signatures that also specify a termination metric m. For example, we would type ack as

$$\mathbf{let\ rec}\ \mathsf{ack} = \lambda m, n.\ e_{\mathsf{ack}} : t_{\mathsf{ack}}\ \mathbf{in}\ \ldots$$

where the signature t_{ack} specifies the lexicographic metric with the sequence of decreasing expressions m, n

$$t_{\mathsf{ack}} \doteq m{:}\mathsf{nat} \rightarrow n{:}\mathsf{nat} \rightarrow \mathsf{nat}\ /\ m, n \tag{9.2}$$

9.3 Declarative Typing

As we saw with the examples in section 9.1 termination checking reduces quite directly to plain refinement checking after *limiting* recursive applications within the implementation to types that are strengthened with special refinements that ensure that the recursion is *well-founded*.

Metric Well-formedness The judgment $\Gamma \vdash m$ says that a termination metric m is *well-formed* in an environment Γ. The judgment is established by the rules Wfm-Base and Wfm-Lex which, in concert, check that each decreasing expression in the metric, shown in Fig. 9.2, can be typed as an `int`-valued term under Γ.

Well-foundedness Refinements The procedure $\mathsf{wfr}(m^*, m')$ shown in Fig. 9.3 takes as input two termination metrics, and returns as output a predicate that guarantees the metric demonstrates well-founded

wfr	$:$	$(M \times M) \to P$
$\mathsf{wfr}(p^*, p')$	$\doteq$	$0 \leq p' \wedge p' < p^*$
$\mathsf{wfr}(p^*; m^*, p'; m')$	$\doteq$	$0 \leq p' \wedge (p' < p^* \vee r)$
where		
r	$=$	$p' = p^* \wedge \mathsf{wfr}(m^*, m')$

Figure 9.3: Computing Well-foundedness Refinements

(terminating) recursion. The inputs m^* and m' respectively denote the values of a function's termination metric over the *original* and *recursive* call parameters. The output is a *well-foundedness refinement* corresponding to the precondition that must hold at each recursive call in order for the metric to demonstrate the function terminating.

For `range`, we would use the termination metric $/\ j - i$ to compute the well-foundedness refinement

$$\mathsf{wfr}(j' - i', j - i) \doteq 0 \leq j' - i' < j - i \tag{9.3}$$

Similarly, for `ack`, we would use the termination metric $/\ m, n$ to compute a well-foundedness refinement

$$\mathsf{wfr}((m', n'), (m, n)) \doteq 0 \leq m' \wedge (m' < m \vee (m' = m \wedge 0 \leq n' < n)) \tag{9.4}$$

Type Limiting The procedure $\mathsf{lim}(\Gamma, m, t)$ shown in Fig. 9.4 computes a type t' which strengthens the input types (preconditions) to require that all (recursive) calls be *limited* to values that are allowed by the metric m. The real work is done by the helper $\mathsf{lim}^*(\Gamma, m^*, m, t)$ which takes the *original* metric m^* and the *recursive* metric m where all the input binders (x) are replaced with "primed" versions (x') and returns a version of t where (1) the inputs are renamed with their primed variants and (2) the well-foundedness refinement is used to strengthen the first input refinement where it is well-formed, *i.e.* where all the binders appearing in the refinement are in scope. To this end, the procedure recurses over the structure of the (function) type t, adding the binders to Γ and renaming the inputs m with their primed versions, *until* it has added enough binders to Γ for m^* to be well-formed, at which

$$
\begin{array}{lcl}
\hline
\mathsf{lim} & : & (\Gamma \times M \times T) \to T \\
\hline
\mathsf{lim}(\Gamma, m, t) & \doteq & \mathsf{lim}^*(\Gamma, m, m, t) \\
\hline
\mathsf{lim}^* & : & (\Gamma \times M \times M \times T) \to T \\
\mathsf{lim}^*(\Gamma, m^*, m, x\!:\!b\{p\} \to t) & & \\
\quad \mid \Gamma; x\!:\!b \vdash m & \doteq & x'\!:\!b\{p'\} \to t' \\
\quad \text{where} & & \\
\qquad p' & = & p[x := x'] \wedge \mathsf{wfr}(m^*, m') \\
\qquad m' & = & m[x := x'] \\
\qquad t' & = & t[x := x'] \\
\mathsf{lim}^*(\Gamma, m^*, m, x\!:\!s \to t) & \doteq & x'\!:\!s' \to t'' \\
\quad \text{where} & & \\
\qquad t'' & = & \mathsf{lim}^*(\Gamma; x\!:\!s, m^*, m', t') \\
\qquad m' & = & m[x := x'] \\
\qquad s' & = & s[x := x'] \\
\qquad t' & = & t[x := x'] \\
\mathsf{lim}^*(\Gamma, m^*, m, \forall\alpha\!:\!k.t) & \doteq & \forall\alpha\!:\!k.\mathsf{lim}^*(\Gamma, m^*, m, t) \\
\hline
\end{array}
$$

Figure 9.4: Checking termination by type limiting

point, it strengthens the current parameter's refinement p with the well-foundedness refinement $\mathsf{wfr}(m^*, m')$.

For simplicity, the definition of lim^* and the rule Chk-Term do not support refinement polymorphism. It is straightforward to remove this restriction by extending the definitions to allow for types abstracted over refinements.

Example: Type of **range** Recall that range has type

$$t_{\mathsf{range}} \doteq i\!:\!\texttt{int} \to j\!:\!\texttt{int} \to \texttt{list(int)} \ / \ j - i \tag{9.5}$$

We will check its body with the metric limited type

$$\mathsf{lim}(\emptyset,\ j - i,\ i\!:\!\texttt{int} \to j\!:\!\texttt{int} \to \texttt{list(int)})$$

which is defined as

$$= \mathsf{lim}^*(\emptyset,\ j - i,\ j - i,\ i\!:\!\texttt{int} \to j\!:\!\texttt{int} \to \texttt{list(int)})$$

Termination Checking $\boxed{\Gamma \vdash x = e : s/m \triangleright t}$

$$\dfrac{t = \mathsf{fresh}(\Gamma,\ s) = \forall \overline{\alpha : k}.\overline{y : s} \to t^* \quad e = \Lambda \overline{\alpha} : k.\lambda \overline{y}.\ e^* \qquad \Gamma; \overline{\alpha : k}; \overline{y : s}; x : \mathsf{lim}(\Gamma, m, t) \vdash e^* \triangleleft t^*}{\Gamma \vdash x = e : s/m \triangleright t}\ \textsc{Chk-Term}$$

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\dfrac{\Gamma \vdash x = e_1 : s_1/m \triangleright t_1 \qquad \Gamma; x : t_1 \vdash e_2 \triangleleft t_2}{\Gamma \vdash \textbf{let rec}\ x = e_1 : s_1/m\ \textbf{in}\ e_2 \triangleleft t_2}\ \textsc{Chk-Rec}$$

Figure 9.5: Bidirectional Checking: Other rules from λ_ρ (Fig. 8.4)

as the metric $j - i$ is not well-formed, we rename i to i' and recurse

$$= i' : \mathtt{int} \to \mathsf{lim}^*(i : \mathtt{int},\ j - i,\ j - i',\ j : \mathtt{int} \to \mathtt{list(int)})$$

now, since $i : \mathtt{int}, j : \mathtt{int} \vdash j - i$ the above is

$$= i' : \mathtt{int} \to j' : \mathtt{int}\{\mathsf{wfr}(j - i, j' - i')\} \to \mathtt{list(int)}$$

which, after substituting the well-foundedness refinement (eq. (9.3))

$$= i' : \mathtt{int} \to j' : \mathtt{int}\{0 \le j' - i' < j - i\} \to \mathtt{list(int)}$$

□

Termination Checking We introduce a new *termination checking* judgment $\Gamma \vdash x = e : s/m \triangleright t$ which *guarantees* that in an environment Γ, the (recursive) definition $x = e$ is terminating with the metric m, and that downstream definitions can *assume* that x behaves as t. The rule Chk-Term shown in Fig. 9.5 establishes this judgment by

1. ***Instantiating*** the *holes* in s to obtain the complete signature t,
2. ***Splitting*** the definition e into its input *binders* $\overline{y : s}$ and *body* e^*,
3. ***Checking*** the body e^* in an environment containing the input binders $\overline{y : s}$ which name the *current* input, *and* where x is bound to its *metric limited type* $\mathsf{lim}(\Gamma, m, t)$.

That is, the key change is to check the body of the recursive binder in an environment that limits the recursion using the specified metric.

Checking Recursive Definitions We can now use the termination checking judgment in the updated rule CHK-REC shown in Fig. 9.5. To check that $\mathbf{let\ rec}\ x = e_1 : s_1/m\ \mathbf{in}\ e_2$ has type t_2, the new rule requires that the binder x terminates with m, and then, (as before), checks e_2 against t_2 in the environment extended with x.

Example: Checking **range** Let's see how CHK-TERM checks the term

$$e_{\mathsf{r}} \doteq \mathbf{let\ rec}\ \mathsf{range} = (\lambda i, j.\ e_{\mathsf{range}}) :\ t_{\mathsf{range}}\ \mathbf{in}\ \ldots \tag{9.6}$$

where

$$\begin{aligned} e_{\mathsf{range}} &\doteq \mathbf{if}\ i < j\ \mathbf{then}\ \mathsf{Cons[int]}\ i\ (\mathsf{range}\ (i+i)\ j)\ \mathbf{else}\ \mathsf{Nil[int]} \\ t_{\mathsf{range}} &\doteq i{:}\mathtt{int} \rightarrow j{:}\mathtt{int} \rightarrow \mathtt{list(int)}\ /\ j - i \end{aligned}$$

CHK-TERM stipulates that we use the metric to limit the type and check the body e_{range} against the specified output `list(int)` in environment Γ_{range} from eq. (9.1)

$$\Gamma_{\mathsf{range}} \ \vdash\ e_{\mathsf{range}} \Leftarrow \mathtt{list(int)}$$

Rule CHK-IF reduces the above to checks on each branch. Eliding the trivial `else` case, we get

$$\Gamma_{\mathsf{range}}; i < j \ \vdash\ \mathsf{Cons[int]}\ i\ (\mathsf{range}\ (i+i)\ j) \Leftarrow \mathtt{list(int)}$$

which SYN-APP splits into a check that

$$\Gamma_{\mathsf{range}}; i < j \ \vdash\ \mathsf{range}\ (i+i)\ j \Leftarrow \mathtt{list(int)}$$

Again, SYN-APP splits the above into checks that ensure each *input* to `range` satisfies the environment's (limited) input type

$$\begin{aligned} \Gamma_{\mathsf{range}}; i < j \ &\vdash\ \mathtt{int}\{i' : i' = i + 1\}\ \ <:\ \ \mathtt{int} \\ \Gamma_{\mathsf{range}}; i < j; i' : \{i' = i + 1\} \ &\vdash\ \mathtt{int}\{j' : j' = j\}\ \ <:\ \ \mathtt{int}\{j' : 0 \leq j' - i' < j - i\} \end{aligned}$$

the first of those is trivial; the second reduces to the entailment

$$i < j; i' = i + 1; j' = j \ \vdash\ 0 \leq j' - i' < j - i$$

that is readily verified by the SMT solver. □

checkT	:	$(\Gamma \times x \times E \times T \times M) \to (T \times C)$
$\mathsf{checkT}(\Gamma,\ x,\ e,\ s,\ m)$	$\doteq$	$(t,\ (\overline{y :: s}) \Rightarrow (x :: t') \Rightarrow c)$
where		
c	$=$	$\mathsf{check}(\Gamma; \overline{\alpha}; \overline{y\!:\!s}; x\!:\!t',\ e^*,\ t^*)$
t'	$=$	$\mathsf{lim}(\Gamma, m, t)$
$\forall \overline{\alpha}\!:\!k.\overline{y\!:\!s} \to t^*$ as t	$=$	$\mathsf{fresh}(\Gamma,\ s)$
$\Lambda\overline{\alpha}\!:\!k.\lambda\overline{y}.\ e^*$	$=$	e

Figure 9.6: Algorithmic Termination Checking

9.4 Verification Conditions

The algorithmic VC generation procedure for λ_τ extends how check generates Horn constraints for **let rec** $x = e_1 : s_1$ **in** e_2, as summarized in Fig. 9.7. The implementation mirrors the declarative formulation Chk-Rec, where we first check that the recursive definition e_1 is terminating, and then use its type t_1 to check e_2.

Algorithmic Termination Checking We implement the termination checking judgment with a procedure checkT summarized in Fig. 9.6, which takes as input a recursive definition $x = e$ and the type s and metric m ascribed to the definition and returns as output a pair comprising the Horn VC c whose satisfiability indicates that the definition is well-typed and terminating, and the type t that subsequent binders can assume for x. First, (as in Fig. 5.5) we use fresh to instantiate the holes in the specification s with new Horn variables. Second, we obtain the body e^* and input binders $\overline{y\!:\!s}$. Finally, we invoke check to compute the VC c for the body in an environment containing the input binders and the metric-limited type t'. The following states the correspondence between the algorithmic and declarative versions of termination checking:

Proposition 9.1. If $\mathsf{checkT}(\Gamma,\ x,\ e,\ s,\ m) = (t, c)$ and c is Horn satisfiable then $\Gamma \vdash x = e : s/m \triangleright t$.

Following the same reasoning as the declarative checking, *i.e.* generating a VC for the body in the metric-limited environment, $\mathsf{check}(\emptyset,\ e_\mathsf{r},\ \ldots)$

check	:	$(\Gamma \times E \times T) \rightarrow C$
$\mathsf{check}(\Gamma,\ \textbf{let rec}\ x = e_1 : s_1/m\ \textbf{in}\ e_2,\ t_2)$	$\doteq$	$c_1 \wedge (x :: t_1) \Rightarrow c_2$
where		
(t_1, c_1)	$=$	$\mathsf{checkT}(\Gamma,\ x,\ e_1,\ s_1,\ m)$
c_2	$=$	$\mathsf{check}(\Gamma; x : t_1,\ e_2,\ t_2)$

Figure 9.7: Algorithmic Checking for λ_τ, extends cases of Fig. 8.8

generates the following VC for the term from eq. (9.6):

$$\forall i, j, i', j' : \mathtt{int}.\ i < j \Rightarrow i' = i + 1 \Rightarrow j' = j \Rightarrow 0 \le j' - i' < j - i$$

The above VC is valid, which proves that `range` terminates.

9.5 Discussion

To recap, in λ_τ we saw how to extend the type system to also check that (recursive) functions terminate. We started with functions like `sum` and `sumT` that recurse on some natural number `n` that directly demonstrates termination. Then, we saw how to generalize the above idea to decreasing expressions like the one that we used to demonstrate `range` terminates. We saw how the idea of expressions can be generalized to sequences to yield termination metrics which demonstrate termination via well-founded lexicographic ordering. Finally, we saw how the notion of measures generalizes the above to functions that work on datatypes. The key idea in all of the above, is simply to check the body of the recursive call with a strengthened type that limits (recursive) inputs to be well-founded, thereby enforcing termination.

Incompleteness Of course, thanks to the Halting problem there are terminating functions that cannot be proven terminating via the approach shown above, because there is no algorithmic procedure to find termination metrics, and because the metrics themselves may, in general, fall outside the SMT solver's decidable theories. For example, it would be a major result to find a suitable terminating metric for the `collatz` function (*Collatz Conjecture* 2021).

Termination in Practice Nevertheless, there is evidence that the mechanisms shown here with simple extensions, *e.g.* to account for mutually recursive functions, suffice to verify termination in the vast majority of programs that arise in practice. For example, the metric-based approach extended with simple heuristics to generate default metrics associated with particular datatypes, suffices to verify 96% of recursive functions on a corpus of more than 10,000 lines of widely used Haskell libraries, whilst requiring only about 1.7 metrics per 100 lines of code (Vazou *et al.*, 2014a). Other SMT-based verifiers like F* (Swamy *et al.*, 2011) and Dafny (Leino, 2010) use similar strategies to check termination very effectively.

Other Strategies for Proving Termination There is a vast literature on techniques for proving termination all of which ultimately find their roots in the notion of well-founded metrics, introduced by Turing, 1949. Jones and Bohr, 2004 and Sereni and Jones, 2005 embody this idea via the "size-change principle" that they use to verify termination of recursive functions, and which, can be rephrased as a *contract* to enable *dynamic* termination checking Nguyen *et al.*, 2019. Proof assistants like Coq (Bertot and Castéran, 2004) and Isabelle (Wenzel, 2016) employ *structural* termination checks wherein recursive calls can only be made on strict sub-structures of the inputs (*e.g.* the tail of an input list.) Hughes *et al.*, 1996 and Barthe *et al.*, 2004 show how to generalize this idea via *sized types* wherein the bodies of recursive functions are checked under metric limited environments. An alternative approach formulated by Giesl *et al.*, 2011 is to reduce termination for functional programs to termination of term rewriting systems. Podelski and Rybalchenko, 2004 show how to generalize and unify the notion of termination metrics and program invariants, via the notion of *transition invariants*, which also allow the use of abstract interpretation based methods to automatically synthesize suitable metrics (or "ranking functions"), which was the basis of the Terminator tool (Cook *et al.*, 2011) which verifies the termination of device drivers written in C.

Our formulation for λ_τ is inspired by the method introduced by Xi, 2001 to encode sized types in a refinement setting. Refinements provide the advantage of *unifying* reasoning about invariants of data with reasoning about termination. This unification is crucial for large real-world code bases, where termination requires functions only be called under certain pre-conditions (*e.g.* `int` valued inputs are non-negative), or require specific post-conditions (*e.g.* the `split` function in a merge-sort routine returns output lists that are strictly smaller than the input), or let us specify arithmetic metrics like those in `range` where termination depends crucially on the path-sensitive reasoning performed by the rest of the type checking.

10

Programs as Proofs

We have been rather timid in what we allow specifications to *say*, limiting them mostly to facts about integers or sets or ordering extended with uninterpreted measures that describe properties of algebraic data, to ensure that the VCs are SMT decidable. Next, let's see how to break out of this shell, to allow users to write specifications over user defined functions and then prove theorems about those functions by supplying proofs structured as programs. We will do so by *reflecting* the implementation of the user-defined function into its output type, thereby converting its type signature into an exact description of the function's behavior. Reflection has a profound consequence: at *uses* of the function, the standard rule Syn-App for function application turns into a means of explicating how the function behaves at the given input, which lets us encode the function's behavior at the (refinement) type level. The above idea, coupled with a small set of combinators, lets us write sophisticated proofs simply as programs.

10.1 Examples

Let's start with an overview of how refinement reflection works by seeing how it lets us write paper-and-pencil-style proofs as programs.

10.1.1 Propositions as Types

Refinements let us encode propositions as types. For example, a unit type can be refined with a logical proposition that encodes that `1 + 1 = 2`

```
type one_plus_one_eq_two = ()[v|1 + 1 = 2]
```

As the `v` and `()` are unimportant, we will elide them and just write

```
type one_plus_one_eq_two = [1 + 1 = 2]
```

As another example, here is the proposition that `int` addition is commutative, *i.e.* $\forall x, y : \texttt{int}. x + y = y + x$:

```
type plus_comm = x:int => y:int => [x + y = y + x]
```

Programs as Proofs Notice that we can represent universal quantification as a function type, following the Curry-Howard correspondence (Howard, 1980; Wadler, 2015). Thus, following the correspondence, any term e whose type corresponds to a proposition P can be viewed as a *proof* of that proposition. Here is a trivial "proof" of the proposition `one_plus_one_eq_two`

```
val thm_one_plus_one_eq_two : one_plus_one_eq_two
let thm_one_plus_one_eq_two = ()
```

Note that the VC generation procedure we outlined in chapter 3 would verify the above program by checking the validity of the VC

$$1 + 1 = 2$$

which is validated by the SMT solver. Here is a proof for `plus_comm`

```
val thm_plus_comm : plus_comm
let thm_plus_comm = (x, y) => ()
```

The VC generation procedure from chapter 3 generates the VC

$$\forall x, y.\ x + y = y + x$$

which the SMT solver validates via the theory of linear arithmetic, giving us our "theorem". These two propositions fell squarely within decidable theories and hence had trivial proofs, with the SMT solver doing all the work.

10.1.2 Refinement Reflection

Next, let's extend the language of refinements with user-defined functions, and write propositions and proofs over those functions.

Step 1: Propositions over User-defined Functions First, λ_π introduces a way to define functions, *e.g.* to `sum` numbers from `0` to `n`

```
val sum : n:nat => nat / n
def sum = (n) => {
  if (n == 0) {
    0
  } else {
    n + sum(n-1)
  }
}
```

The **`def`**-bound functions are just like the usual **`rec`** binders — the metric `/ n` ensures they terminate — except that we can refer to them in refinements as the *uninterpreted* function *sum*. Consequently, we can now specify the proposition

$$\forall i, j.\ i = j \Rightarrow sum(i) = sum(j) \tag{10.1}$$

and verify it via a trivial proof

```
val sum_eq : i:nat => j:nat[i=j] => [sum(i) = sum(j)]
let sum_eq = (i, j) => ()
```

The above program generates the VC corresponding directly to the proposition eq. (10.1) which the SMT solver automatically validates via congruence closure (Nelson, 1980).

Step 2: Refinement Reflection λ_π imbues **`def`** binders with a second crucial property: we strengthen their user-specified types with a refinement that exactly *reflects the function's implementation.* That is, let $\Delta_{\mathsf{sum}}(n)$ be an abbreviation for the refinement

$$\Delta_{\mathsf{sum}}(n) \doteq \textit{if}\ n = 0\ \textit{then}\ 0\ \textit{else}\ n + sum(n-1)$$

where *sum* is the *uninterpreted* function representing `sum` in the refinement logic. Now, we assign `sum` the type

$$\mathsf{sum} : n{:}\mathsf{nat} \to \mathsf{nat}\{v{:}v = sum(n) \wedge v = \Delta_{\mathsf{sum}}(n)\}$$

which says that the output value v equals to the logical representation $sum(n)$ which itself equals the value of the reflected body $\Delta_{\mathsf{sum}}(n)$.

Step 3: Proofs using Function Applications As we have reflected the function's definition into its output type, each *application* of `sum` *instantiates* its definition at the given input. For example, here is a proof that $sum(2) = 3$

```
val sum_2_eq_3 : () => [sum(2) = 3]
let sum_2_eq_3 = () => {
  let t0 = sum(0);
  let t1 = sum(1);
  let t2 = sum(2);
  ()
}
```

The usual rules Chk-Let and Syn-App yield the VC

$$\begin{array}{l}\forall t_0.\ t_0 = sum(0) = (\mathit{if}\ 0 = 0\ \mathit{then}\ 0\ \mathit{else}\ 0 + sum(0-1)) \Rightarrow \\ \quad \forall t_1.\ t_1 = sum(1) = (\mathit{if}\ 1 = 0\ \mathit{then}\ 0\ \mathit{else}\ 1 + sum(1-1)) \Rightarrow \\ \quad\quad \forall t_2.\ t_2 = sum(2) = (\mathit{if}\ 2 = 0\ \mathit{then}\ 0\ \mathit{else}\ 2 + sum(2-1)) \Rightarrow \\ \quad\quad\quad sum(2) = 3\end{array}$$

Intuitively, each application $\mathsf{sum}(i)$ *instantiates* the definition of `sum` at the input i, after which the SMT solver's theories for equality, congruence and arithmetic kick in to internally simplify the above VC to the following valid formula

$$\begin{array}{l}sum(0) = 0 \Rightarrow \\ \quad sum(1) = 1 + sum(0) \Rightarrow \\ \quad\quad sum(2) = 2 + sum(1) \Rightarrow \\ \quad\quad\quad sum(2) = 3\end{array}$$

We need *all* the instances for 0, 1 and 2 to prove the goal. A proof term that omitted the binding t_1 would be *rejected* as it yields the VC

$$\begin{array}{l}sum(0) = 0 \Rightarrow \\ \quad sum(2) = 2 + sum(1) \Rightarrow \\ \quad\quad sum(2) = 3\end{array}$$

which is *invalid* as $sum(1)$ is unconstrained.

10.1.3 Structuring Proofs via Combinators

Writing proofs like `sum_2_eq_3` can be tedious: how are we to divine *which* terms to instantiate the definition of `sum` at?

Equational Proofs We can solve this problem by structuring proofs to follow the style of calculational or equational reasoning (Bird, 1989; Dijkstra, 1976) and implemented in Agda (Mu *et al.*, 2009), and Dafny (Leino and Polikarpova, 2016), via an *equality-chaining* combinator

```
val (===) : x:'a => y:'a[y == x] => [v=x && v=y]
let (===) = (x, y) => y
```

The combinator's type specification says that e_1 `===` e_2 is a *proof* that e_1 and e_2 are equal, and further, a term that equals e_1 and e_2. We can use equality-chaining to rewrite `sum_2_eq_3` in a way that might mirror a pencil-and-paper proof

```
val sum_2_eq_3 : _ -> [sum 3 = 6] @-}
let sum_2_eq_3 = () => {
  sum(2)
    === 2 + sum(1)
    === 2 + 1 + sum(0)
    === 3
}
```

The precondition of `(===)` checks that each *intermediate* equality holds, simply by using the *applied* instance of `sum` at the respective call. The postcondition of `(===)` lets us chain the equalities together to prove the goal. Thus, if we skip a step, *e.g.* if we write

```
let sum_2_eq_3' = () => {
  sum(2)
    === 2 + 1 + sum(0)
    === 3
}
```

the precondition for the first equality-chain will fail, and so type checking pinpoints where information is needed to complete the proof.

Functions as Lemmas Suppose we want to verify the proposition `sum(3) == 6`. We could, of course, repeat all the calculations we did to prove `sum_2_eq_3` but instead, it would be nice to *reuse* the proposition

Proof	Program
Theorem	Function
Apply Theorem	Call Function
Case Split	Branch
Induction	Recursion

Figure 10.1: Correspondence between Proofs and Programs.

that we have already proved as a *lemma* to prove the new goal. We do so by introducing a *because* combinator that conjoins propositions

```
val (?) : x:'a[p x] => y:'b[q y] => 'a[v| p v && q y]
let (?) = (x,_) => x
```

We can now reuse `sum_2_eq_3` as a lemma, simply by applying it as

```
val sum_3_eq_6 : _ => [sum(3) = 6]
let sum_3_eq_6 = () => {
  sum(3)
    === 3 + sum(2)
      ? sum_2_eq_3 ()
    === 6
}
```

The types of our combinators ensure that the above yields a VC like

$$sum(3) = \Delta_{\mathsf{sum}}(3) \Rightarrow sum(2) = 3 \Rightarrow sum(3) = 6$$

where the fact establishing the value of $sum(2)$ is established by applying, and hence, obtaining the output (post-condition) of the function `sum_2_eq_3`.

10.1.4 Proofs as Programs

The above proofs are quite unremarkable: they merely confirm what a computation evaluates to. However, they introduce the building blocks of more interesting examples that illustrate the correspondence between proofs and programs summarized in Fig. 10.1.

Induction on Numbers Let's write and prove the proposition

$$\forall n.\ \sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$$

We can specify the proposition as a type and then provide a proof as:

```
val thm_sum : n:nat => [2 * sum(n) = n * (n+1)]
let thm_sum = (n) => {
  switch (n) {
  | 0 => {    2 * sum(0)
          === 0 * (0+1)
          }
  | n => {    2 * sum(n)
          === 2 * (n + sum(n-1))
          === 2 * n + 2 * sum(n-1)
            ? thm_sum(n-1)
          === n * (n+1)
          }
  }
}
```

The above proof mirrors the classic proof *by induction*. We split cases on `n` via a **`switch`** that branches on the value of `n`. In the *base* case we prove the equality via a calculation on `sum(0)`. In the *inductive* case we prove the equality by *recursively* applying `thm_sum` at `(n-1)` which effectively allows us to use the *induction hypothesis* for a smaller value of `n`. The *termination* check — made possible by the metric `/ n` — crucially ensures that the recursion (*i.e.* induction) is well-founded, and hence, that the proof is not circular.

Induction on Data We will see how *measures* (as defined in chapter 7) let us define *selectors* that let us reflect functions on data types like lists into the refinement logic. This lets us write theorems over datatypes. Here is a function that appends lists:

```
val app : list('a) => list('a) => list('a)
def app = (xs, ys) => {
  switch (xs) {
  | Nil          => ys
  | Cons(x, xs') => Cons(x, app(xs', ys))
  }
}
```

Let's verify that app is *associative*, *i.e.*

$$\forall xs, ys, zs.\ \mathsf{app}(\mathsf{app}(xs,\ ys),\ zs)\ =\ \mathsf{app}(xs,\ \mathsf{app}(ys,\ zs))$$

by writing a recursive proof:

```
val app_assoc : xs:list('a) => ys:list('a) =>
  [app(app(xs, ys), zs) = app(xs, app(ys, zs))]
  / len(xs)

let app_assoc = (xs, ys, zs) => {
  switch (xs) {
  | Nil => {
          app (app(Nil, ys), zs)
      === app(ys, zs)
      === app(Nil, app(ys, zs))
    }
  | Cons(x, xs') => {
          app(app(Cons(x,xs'), ys), zs)
      === app(Cons(x, app(xs', ys)), zs)
      === Cons(x, app(app(xs', ys), zs))
        ? app_assoc(xs', ys, zs)
      === Cons(x, app(xs', app(ys, zs)))
      === app(Cons(x, xs'), app(ys, zs))
    }
  }
}
```

This time, the induction is on xs and is shown well-founded due to the metric len(xs). As before, we split on the base case where xs is Nil, and the inductive case where xs is Cons(x, xs'). In either case, we prove the respective goal via a calculation. In the inductive case, we get to invoke the induction hypothesis by *calling* the theorem app_assoc on the smaller input xs'.

10.2 Types and Terms

Fig. 10.2 summarizes the extensions in λ_π needed to support reflection and proofs. The syntax of types remains unchanged. The terms are extended with a form **def** $x = e_1 : t/m$ **in** e_2 which are just like **rec**-binders except that we will strengthen their output types using reflection.

Terms	e	$::=$	$\ldots$	*from Fig. 9.1*
		$\mid$	$\mathbf{def}\ x = e_1 : t/m\ \mathbf{in}\ e_2$	*reflected binder*

Figure 10.2: λ_π: Syntax of Types and Terms

reflect	:	$(X \times E \times T) \to T$
$\mathsf{reflect}(f, e, s)$	$\doteq$	$\forall \overline{\alpha}:k.\overline{y:s} \to b\{v:p \wedge p'\}$
where		
$\forall \overline{\alpha}:k.\overline{y:s} \to b\{v:p\}$	$=$	s
$\Lambda \overline{\alpha}:k.\lambda \overline{y}.\ e^*$	$=$	e
p'	$=$	$(v = f(\overline{y}) \wedge v = \mathsf{embed}(e^*))$

Figure 10.3: Reflecting Terms into Types

Reflection The workhorse of λ_π is the procedure $\mathsf{reflect}(f, e, s)$ shown in Fig. 10.3, which takes as input a binder f, the binder's definition e and the binder's type s, and returns a variant of s where the *output* type is strengthened with a refinement that says that the returned value *equals* the implementation of the function at the given inputs. The procedure works by first obtaining the parameters $\overline{y}$ and *body* e^* of the definition e. Next, it translates the body e^* into the refinement logic via the procedure embed. Finally, it strengthens the output type with the postcondition p' that says that the output value v equals the function body. For example, $\mathsf{reflect}(add, e, s)$ where $e \doteq \lambda x_1, x_2.\ x_1 + x_2$ and $s \doteq \mathtt{int} \to \mathtt{int} \to \mathtt{int}$ returns as output the type

$$x_1 : \mathtt{int} \to x_2 : \mathtt{int} \to \mathtt{int}\{v : v = add(x_1, x2) \wedge v = x_1 + x_2\}$$

Embedding The hard work in reflect is done by the procedure embed, summarized in Fig. 10.4 which recursively translates *implementation* terms into *logical* expressions. The procedure translates literals like `2` and `false` into the corresponding values in the logic (*i.e.* 2 and *false* respectively); primitive function applications like `x+y` or `a <= b` into the corresponding terms or relations in the logic (*i.e.* $x + y$ and $a \leq b$ respectively); and translates all other function calls into uninterpreted function applications. Let-binders can be translated by substitution

embed	:	$E \rightarrow P$
$\mathsf{embed}(n)$	$\doteq$	n
$\mathsf{embed}(\mathit{true})$	$\doteq$	true
$\mathsf{embed}(\mathit{false})$	$\doteq$	false
$\mathsf{embed}(\mathsf{c}\ x_1\ x_2)$	$\doteq$	$x_1 \bowtie_{\mathsf{c}} x_2$
$\mathsf{embed}((f\ e_1)\ldots e_n)$	$\doteq$	$f(\mathsf{embed}(e_1),\ldots,\mathsf{embed}(e_n))$
$\mathsf{embed}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$	$\doteq$	$\mathsf{embed}(e_2)[x := \mathsf{embed}(e_1)]$
$\mathsf{embed}(\mathbf{if}\ x\ \mathbf{then}\ x_1\ \mathbf{else}\ x_2)$	$\doteq$	$\mathit{if}\ x\ \mathit{then}\ \mathsf{embed}(e_1)\ \mathit{else}\ \mathsf{embed}(e_2)$
$\mathsf{embed}(\mathbf{switch}\ x\ \{\overline{a}\})$	$\doteq$	$\mathsf{embAlts}(x,\overline{a})$

Figure 10.4: Embedding Terms into the Refinement Logic

as type checking ensures that all reflected terms are terminating, and hence, well defined. Branches are translated into ternary choices.

Example: Reflection of **`sum`** Suppose that e_{sum} is the implementation of the `sum` function from section 10.1.2. Then

$$\mathsf{embed}(e_{sum}) \doteq \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ n + \mathit{sum}(n-1) \tag{10.2}$$

where the recursive call is translated into an uninterpreted function application $\mathit{sum}(n-1)$. □

Embedding Case Alternatives Pattern match terms $\mathbf{switch}\ x\ \{a_1;\ldots\}$ are embedded using $\mathsf{embAlts}(x, a_1;\ldots)$ shown in Fig. 10.5, which translates them as nested ternary branches of the form $\mathit{if}\ c_1\ \mathit{then}\ e_1\ \mathit{else}\ \ldots$ where c_1 is a logical predicate that is true when x matches the first constructor and e_1 the embedding of the corresponding result. We translate case alternatives using two operators from the SMT decidable theory of Algebraic Datatypes (Nelson, 1980). First, the *test* predicate $\mathit{is}_D(x)$ determines whether x equals to a term $D(c_1,\ldots,c_k)$. Second, if x equals $D(c_1,\ldots,c_k)$ then the *projection* function $\mathit{proj}^i_D(x)$ equals c_i. Thus, given the scrutinee x, the procedure $\mathsf{embAlts}$ translates each alternative, by using the test predicate to determine if that alternative matches: if

embAlts	:	$(X \times \overline{A}) \rightarrow P$
$\mathsf{embAlts}(x, a; \overline{a})$	$\doteq$	$\mathit{if}\ \mathit{is}_D(x)\ \mathit{then}\ \mathsf{embAlt}(x, a)\ \mathit{else}\ \mathsf{embAlts}(x, \overline{a})$
$\mathsf{embAlts}(x, a)$	$\doteq$	$\mathsf{embAlt}(x, a)$
embAlt	:	$(X \times A) \rightarrow P$
$\mathsf{embAlt}(x, D(\overline{y}) \rightarrow e)$	$\doteq$	$\mathsf{embed}(e)[\overline{y_i} := \overline{\mathit{proj}^i_D(x)}]$

Figure 10.5: Embedding Switch Alternatives into the Refinement Logic

Type Checking $\boxed{\Gamma \vdash e \triangleleft t}$

$$\frac{\Gamma \vdash x = e_1 : s_1/m \triangleright t_1 \quad t_1' = \mathsf{reflect}(x, e_1, t_1) \quad \Gamma; x : t_1' \vdash e_2 \triangleleft t_2}{\Gamma \vdash \mathbf{def}\ x = e_1 : s_1/m\ \mathbf{in}\ e_2 \triangleleft t_2}\ \textsc{Chk-Refl}$$

Figure 10.6: Bidirectional Checking: Other rules from λ_τ (Fig. 9.5)

so, translating its body with the binders replaced by the respective projections, and otherwise, recursing on the remaining alternatives.

***Example: Reflecting* app** Recall the definition of the list append function from section 10.1.4. Let e_{app} denote the body of the implementation, *i.e.* `switch (ys){ ...}`. Then

$$\mathsf{embed}(e_{app}) \doteq \mathit{if}\ \mathit{is}_{Nil}(ys)\ \mathit{then}\ \mathit{Nil}\ \mathit{else}\ \mathit{Cons}(\mathit{head}(xs), \mathit{app}(\mathit{tail}(xs), ys))$$

where *head* and *tail* are the projections for the `Cons` constructor. □

10.3 Declarative Checking

The rule Chk-Refl shown in Fig. 10.6 shows how we use reflect to check reflect-binders $\mathbf{def}\ x = e_1 : s_1/m\ \mathbf{in}\ e_2$. First, we check that the definition $x = e_1$ is terminating with metric m.[1] Next, we reflect the

[1] Non-recursive binders can be accomodated using the trivial metric 0.

check	:	$(\Gamma \times E \times T) \rightarrow C$
$\mathsf{check}(\Gamma,\ \mathbf{def}\ x = e_1 : s_1/m\ \mathbf{in}\ e_2,\ t_2)$	$\doteq$	$c_1 \wedge \forall x : t_1'.\ c_2$
where		
(t_1, c_1)	$=$	$\mathsf{checkT}(\Gamma,\ x,\ e_1,\ s_1,\ m)$
t_1'	$=$	$\mathsf{reflect}(x, e_1, t_1)$
c_2	$=$	$\mathsf{check}(\Gamma; x : t_1',\ e_2,\ t_2)$

Figure 10.7: Algorithmic Checking for λ_π, extends cases of Fig. 9.7

definition of e_1 to strengthen the type t_1 to t_1' which is bound to x in the environment used to check e_2.

***Example: Using reflected* `sum`** Suppose that we wanted to check

$$\Gamma \ \vdash\ \mathbf{def}\ sum = \lambda n.\ e_{sum} : \mathtt{int} \rightarrow \mathtt{int}/n\ \mathbf{in}\ e \triangleleft t$$

Rule Chk-Refl says we should first establish that *sum* terminates

$$\Gamma \ \vdash\ sum = \lambda n.\ e_{sum} : \mathtt{int} \rightarrow \mathtt{int}/n \triangleright \mathtt{int} \rightarrow \mathtt{int}$$

Next, we reflect the definition of *sum* into its type

$$t_{sum} \ \doteq\ n : \mathtt{int} \rightarrow \mathtt{int}\{v : v = sum(n) \wedge v = \mathsf{embed}(e_{sum})\}$$

where $\mathsf{embed}(e_{sum})$ is the embedding of *sum*'s body eq. (10.2). We can then check e in the environment extended by binding *sum* to its reflected type: $\Gamma; sum : t_{sum} \ \vdash e \triangleleft t$. □

10.4 Verification Conditions

Finally, we extend algorithmic VC generation function to account for reflected-binders, as summarized in Fig. 10.7. Following the declarative rule, we first invoke checkT to obtain a constraint c_1 that checks that the reflected definition $x_1 = e_1$ terminates with metric m. We then embed the definition into the returned type to obtain the reflected type t_1' that is used to compute the VC c_2 for e_2.

Example: Checking use of **add** Let's look at the VC generated by $\mathsf{check}(\emptyset,\ e,\ t)$ where

$$e \doteq \ \textbf{def}\ add = e_{add} : t_{add}/0\ \textbf{in}\ add\ 4\ 5$$
$$t \doteq \ \texttt{int}\{v : v = 9\}$$

where the definition and type of *add* are respectively

$$e_{add} \doteq \ \lambda x, y.\ x + y$$
$$t_{add} \doteq \ \texttt{int} \to \texttt{int} \to \texttt{int}$$

First, the termination check $\mathsf{checkT}(\emptyset,\ add,\ e_{add},\ t_{add},\ 0)$ yields the trivial type and constraint

$$(t_1, c_1) \doteq \ (t_{add}, true)$$

as there are no holes or refinements in the specified signature. Next, we reflect the definition into the output type to obtain

$$t_1' \doteq \ x : \texttt{int} \to y : \texttt{int} \to \texttt{int}\{v : v = x + y\}$$

and then invoke $\mathsf{check}(add : t_1',\ add\ 4\ 5,\ t)$ to get the constraint

$$c_2 \doteq \ \forall v. v = 4 + 5 \Rightarrow v = 9$$

which is proved valid by SMT. □

10.5 Discussion

With λ_π we saw how to extend specifications with arbitrary user-defined functions whose definitions are reflected into the function's output type. This lets us prove propositions over those functions, simply by writing programs where each *use* of a function instantiates, via the reflected output, the defintion of the function at the given input. Reflection dramatically expands the range of what refinements can be used for, from enforcing invariants of values or data types, to proving the functional correctness of *e.g.* various parallelism constructs (Vazou *et al.*, 2017), dynamic information flow enforcement (Parker *et al.*, 2019b), and laws governing replicated data types (Liu *et al.*, 2020b).

Axioms Other SMT based verifiers, notably Dafny and F* support specifications over user defined functions by encoding their semantics with universally-quantified *axioms*. Modern SMT solvers have sophisticated heuristics for instantiating these axioms automatically using user specified *triggers* (Detlefs *et al.*, 2005) yielding proofs where the user need not spell out all the intermediate computations. One drawback of the axiomatic approach is that reckless triggering can cause the SMT engine to diverge (Leino and Pit-Claudel, 2016a). Dafny and F* use a notion of *fuel* (Amin *et al.*, 2014) to limit the instantiation to some fixed depth. While fuel can be quite effective in practice, it lacks semantic completeness guarantees which are useful to characterize what kinds of proofs can be successfully automated. Suter *et al.*, 2011, show completeness guarantees for a class of *sufficiently surjective* recursive functions, which, informally, correspond to catamorphisms over algebraic datatypes, *e.g.* functions like the *measures* from λ_δ that compute the length of a list or height of a tree. Unfortunately, this result does not extend to arbitrary (terminating) user-defined functions.

Proof by Logical Evaluation Vazou *et al.*, 2018 observe that much of the verbosity in proofs arises from spelling out long *chains* of computations, for example, the intermediate equalities in the proofs `sum_2_eq_3` (§ 10.1.2) and `thm_sum` and `app_assoc` (§ 10.1.4). Based on this observation, the paper introduces the notion of *proof by logical evaluation* (PLE), an algorithm for strengthening the antecedents in the VCs by automatically unfolding the definition of recursive functions in way that is both terminating and complete for equational chains, effectively enabling a form of *refinement-level computation*. With PLE, we can prove `sum_3_eq_6` simply as

```
val sum_3_eq_6 : () => [sum 3 = 6]
let sum_3_eq_6 = () => ()
```

as the PLE algorithm unfolds the definitions of `sum` three times. PLE greatly simplifies inductive proofs by eliminating the tedious internal steps, allowing the proof to focus on the important case-splitting and induction (recursion). That, is the proof of `thm_sum` is reduced to

```
val thm_sum : n:nat => [2 * sum(n) = n * (n+1)]
let thm_sum = (n) => {
```

```
    switch (n) {
    | 0 => ()
    | n => thm_sum(n-1)
    }
  }
```

which simply spells out the inductive skeleton, yielding a VC:

$$
\begin{array}{lll}
\forall n. 0 \leq n \Rightarrow & & \\
\quad n = 0 \Rightarrow & & \\
\quad\quad 2 \times sum(n) = n \times (n+1) & & \text{(base case)} \\
\wedge \quad n \neq 0 \Rightarrow & & \\
\quad\quad 0 \leq n-1 \wedge n-1 < n & & \text{(metric)} \\
\quad \wedge \quad 2 \times sum(n-1) = (n-1) \times n \Rightarrow & & \text{(ind hyp)} \\
\quad\quad\quad 2 \times sum(n-1) = n \times (n+1) & & \text{(ind case)}
\end{array}
$$

Similarly, app_assoc reduces to the below, where the proof need only include the recursive skeleton, and the recursive call that establishes the induction hypothesis for xs':

```
  val app_assoc : xs:list('a) => ys:list('a) =>
    [app(app(xs, ys), zs) = app(xs, app(ys, zs))]
    / len(xs)
  let app_assoc = (xs, ys, zs) => {
    switch (xs) {
    | Nil          => ()
    | Cons(x, xs') => app_assoc(xs', ys, zs)
    }
  }
```

Note that it is essential that the proof function *terminates*: otherwise we could simply write circular "proofs" for *any* proposition. While the increased automation of PLE or axioms makes the proofs much more concise, much work remains in devising interfaces that can help the programmer structure their proofs, and understand why particular proofs are rejected by the type checker.

11

Related Work

In this article, we saw how to extend a programming language with refinement types, using the techniques used by the LiquidHaskell (LH) system. The approaches followed by other checkers are similar, but of course, tailored to the particular languages or domains they are intended for. Next, we provide an (incomplete) high-level overview of the many other ways to build SMT-based program verifiers and refinement type checkers.

11.1 Program Logic based Verifiers

Refinements are a type-based alternative to assertion-based verification systems, going back to the Stanford Verifier (Nelson, 1980), ESC/-Modula (Leino and Nelson, 1998), ESC/Java (Flanagan *et al.*, 2002), SPEC# (Barnett *et al.*, 2011), Dafny (Leino, 2010), and more recently OpenJML (Cok, 2014) and Prusti (Astrauskas *et al.*, 2019). Like refinement types, these systems start with an existing programming language and automatically discharge assertions with an external solver. (Dafny is an exception as the language was co-designed to facilitate verification.) Unlike refinement types, they use the classical Floyd-Hoare *Axiomatic* approach to directly encode the language's semantics into assertions in

Table 11.1: Features of Refinement Type Systems: ✓* denotes local inference.

	DML	**SAGE**	**F***	**LH**	**Stainless**
Existing PL	✓	×	×	✓	✓
Decidable	✓	✓	×	✓	✓
Static	✓	×	✓	✓	✓
Inference	✓*	×	✓*	✓	✓*
Subtyping	×	✓	✓	✓	×
Polymorphism	×	×	✓	✓	×
Data Types	×	×	×	✓	×

the solver's logic, without exploiting the type system to simplify verification conditions. In contrast, Refinement types use type abstractions in two key ways to simplify verification. First, polymorphism – inherent in refinement type systems – gives "theorems for free" (Wadler, 1989). For example, consider the `append` function:

```
val append: xs:list('a) => ys:list('a) => list('a)
```

Invoking `append` on two lists of integers `xs`, `ys` each of which exceed some value `x`, *i.e.* `xs,ys :: [{v:Int | x < v }]`, returns a list that preserves the same property, *i.e.* `append(xs,ys)` has type `[{v:Int | x < v }]` because of refinement type instantiation of the type variable `'a`. Such reasoning, obtained for free in type-based verification, requires quantification in assertion-based systems that quickly renders the VCs undecidable and the resulting verification unpredictable verification (Leino and Pit-Claudel, 2016b). We invite the reader to see Jhala, 2019 for more concrete examples of the above phenomenon.

11.2 Refinement Type based Verifiers

Table 11.1 summarizes some key features of refinement types and lead systems that implement them. The name refinement types was introduced in 1991 (Freeman and Pfenning, 1991b) which syntactically refines ML (Milner, 1978)'s user-defined data types into more precise subsets.

For example, `singleton` can be defined by the user as a `list` that contains exactly one element. This form of refinements, now known as *datasort refinements* was combined with indexed types (Zenger, 1997) in `DML` (Xi and Pfenning, 1998) which has decidable type checking via linear programming, and was further extended to verify data structure invariants in Stardust (Dunfield, 2017). Importantly, `DML` set the first objectives for practical refinement types: integration within an *existing programming language*, *decidable* type checking that can be automated by an external solver and *inference* so the user need not need explicitly annotate intermediate terms.

The Sage system (Flanagan, 2006) generalized refinements from data types to any base type, introducing the syntax $\{v{:}t \mid r\}$, where t is any base type (*e.g.* `int`, `bool`) and r is any pure, boolean expression of the underlying language. Sage uses semantic, *predicate subtyping* (Rushby *et al.*, 1998) to generate verification conditions and an SMT solver (Simplify (Detlefs *et al.*, 2005)) to check them. In Sage, type checking is undecidable and checked in a *hybrid* manner: partly at compile time using SMTs and partly at run-time via contract checks.

F7 (Bengtson *et al.*, 2011; Bierman *et al.*, 2010), later evolved to F* (Swamy *et al.*, 2011), changed the language r of refinements into expressions of the underling solver to restore static type checking. In F*, refinements can include quantifiers, thus type checking is undecidable. In practice, the solver is helped directed using SMT hints and triggers. However, the system has been used to verify several significant real-world applications, including cryptographic routines of web-browsers (Zinzindohoué *et al.*, 2017; Protzenko *et al.*, 2019). F* now also includes other features, outside the scope of this article, including dependent types (Swamy *et al.*, 2016), effects (Maillard *et al.*, 2019) and meta-programming (Martínez *et al.*, 2019) which together yield a complete environment for theorem proving.

Liquid types (Rondon *et al.*, 2008), now used by LiquidHaskell (Vazou *et al.*, 2014a), made decidable type checking a priority. LiquidHaskell restricts the language r of refinements to logics that can be efficiently decided by SMTs (*e.g.* equality, uninterpreted functions, linear arithmetic, data types, but no quantifiers), for the sake of decidable type inference and predictable verification. To restore expressiveness, Liquid-

HASKELL allows refinement and subtyping of polymorphic types and uses refinement reflection (§ 10) to permit controlled reasoning about pure, terminating functions of the underlying language. LIQUIDHASKELL has been used to verify a wide range of sophisticated properties, including resource usage (Handley *et al.*, 2019), security meta-properties (Parker *et al.*, 2019a), properties of real-world, distributed code (Liu *et al.*, 2020a), and the security of web-applications (Lehmann *et al.*, 2021).

Finally, several groups have worked on applying refinement types to imperative languages. Rondon *et al.*, 2010 and Chugh *et al.*, 2012 respectively verify C and JavaScript programs by refining a low-level language of locations. Kent *et al.*, 2016 integrate refinements within Racket's occurrence based type system. Vekris *et al.*, 2016 describe a checker for TypeScript that integrates refinements with types that track the immutability and ownership of references (Potanin *et al.*, 2013). Kazerounian *et al.*, 2017 integrate refinements in Ruby's type system using just-in-time type checking. Finally, `Stainless` (Hamza *et al.*, 2019) introduces a refinement-type based verifier for higher-order Scala programs.

11.3 Soundness of Refinement Types

Different refinement type systems have different flavors of meta-theoretic guarantees. Data-sort refinements (*i.e.* `DML`-style refinement types) come with strong meta-theoretic principles (Zeilberger, 2016): (Lovas and Pfenning, 2010) defines a logical framework where refinements are proof-irrelevant predicates and (Melliès and Zeilberger, 2015) gives a categorical interpretation of refinement types. F* (Gordon and Fournet, 2010) followed the LCF approach (Plotkin, 1977) to set the principles of refinement types by using an externally defined logic to validate subtyping. However, none of the existing LCF-based approaches allow refinements on type variables or refined data types, features heavily used in practice. Sage came with a novel soundness proof that shows soundness of refinement type systems with respect to operational semantics (Knowles and Flanagan, 2010). The denotation of the refined type $\{v:t \mid r\}$ is defined to be the set of all expressions e, with unrefined type t, for which $r[e/v]$ evaluates, using operational semantics, to true, *i.e.*

$r[e/v] \hookrightarrow^\star$ `true`. Denotational subtyping is also defined via operational semantics over all possible instantiations of the typing environment and is undecidable. In practice, an external SMT solver is used to approximate the undecidable denotational subtyping relation. Soundness in this setting is defined using denotational inclusion: if e has type τ, then e belongs in the denotation of τ. For example, since $2 : \{v\texttt{:int} \mid 0 < v\}$, then $0 < 2 \hookrightarrow^\star$ `true`. This proof methodology provides a deep intuition on the semantics of refinement types and has been extensively used to formalize *gradual* types (Belo *et al.*, 2011; Sekiyama *et al.*, 2015), but is insufficient to formalize the guarantees of completely *static* verification. LiquidHaskell (Vazou *et al.*, 2014a) and `Stainless` (Hamza *et al.*, 2019) used Sage's methodology to formalize its core calculus, *i.e.* a subset of its implementation.

Currently, refinement type systems are designed to be practical, and as such, they leave a big gap between the formalization of core calculi and real implementations that span tens to hundreds of thousands lines of code. Unsoundness bugs are inevitable in such big code bases (in each of these three verifiers approximately five soundness bugs are reported per year.) In contrast, *foundational* theorem provers are constructed around a small trusted kernel that implements a set of deductive rules that are proved consistent by the existence of a consistent mathematical model. For example, Isabelle (Paulson *et al.*, 2019) has a core kernel of 5K lines of ML code that implements HOL whose consistency is established via a set theoretic model (Andrews, 2002). Similarly, Coq's kernel is 14K lines of ML code that implements CIC (Coquand and Huet, 1988; Coquand and Huet, 1985), a calculus also shown consistent by a set theoretic model (Timany and Sozeau, 2018). Still, "on average, one critical bug has been found every year in Coq" (Sozeau *et al.*, 2020), so, one can only imagine how many such bugs lurk within the implementations of refinement type checkers! In future work, it would be interesting to see how to emit *certificates* (Necula, 1997) or devise some other way to carve out a core kernel that can be used to ensure the soundness of verification despite errors in the rest of the checker.

12

Conclusion

In this article we saw a progression of languages that incrementally implement a refinement type checker capable of enforcing a full spectrum of correctness requirements at compile time. We conclude with some remarks on our experience developing and using refinement type checkers over the past decade, and point the way to some interesting and important directions for future work.

12.1 The Good: Types Enable Compositional Reasoning

The great advantage of refinement types is that they *align the abstractions* that the analysis uses with those that the programmer uses. Consequently, they provide a simple syntax-directed way to *decompose* reasoning about complex values like collections and higher-order functions or collections into VCs over simple values like integers.

For example, consider the goal of verifying array-access safety in the following function that sums the squares of an array `x`

```
val sumSquares : array(int) => int
let sumSquares = (x) => {
  sum [get(x, i)^2 for i in 0 .. length(x) - 1]
}
```

The programmer might write the function using for-comprehension syntax, but internally, the function would be translated into

```
val sumSquares : array(int) => int
let sumSquares = (x) => {
  let is   = range(0, length(x) - 1);
  let body = (i) => { get(x, i) ^ 2 };
  let ys   = map(body, is);
  sum(ys)
}
```

which makes use of collections, higher-order functions and polymorphism and collections. Each of these features are problematic for classical program logics or program analysis, but are decomposed away by types.

- ***Collections*** First, we need a way to represent and establish the fact that *every* element of the collection `is` was a valid index for `x`. With program logics, this would require universally quantified invariants which make for brittle SMT solving. With program analysis, this would require tailoring sophisticated abstract domains capable of performing shape analysis (Gopan *et al.*, 2005). In contrast, refinements represent this fact as

  $$\texttt{list}(\texttt{int}\{v\!:\!0 \leq v < \mathit{length}(x)\})$$

  the *refinement* expressing the constraint on a single `int` and the *type constructor* `list` generalizing the constraint over the collection.

- ***Closures*** Next, we need to represent the fact that the closure `body` accesses the array `x` at various indices *supplied by* `map`. Classical (*e.g.* Floyd-Hoare) program logics do not account for closures. Modern logics like *e.g.* Hoare-Type Theory (Nanevski *et al.*, 2008b) or Iris (Jung *et al.*, 2018) do handle them, and tools like Dafny permit reasoning about closures, but with significantly more overhead. In the program analysis world, this problem is a variant of *Control-flow Analysis* (Shivers, 1988) which is complicated by reasoning about properties of free variables (*e.g.* `x`) (Might, 2007), which has resisted a robust solution for several decades. Types make the problem practically disappear: we ascribe the type `i:nat [i < length(x)] => int` to `body` and then (contra-variant) function subtyping naturally ensures that only valid indices are in fact passed into the closure.

- ***Polymorphism*** Finally, functions like `map` are ubiquitous: they are reused in many different sites, and verification requires a way to specify a contract that is both *general* enough to account for all the use-cases, yet *precise* enough to facilitate verification at each site. In the program logic setting, this requires quantification over the possible invariants (Nanevski *et al.*, 2008b) which makes verification less ergonomic, as the programmer must spell out where the quantifier is added (*i.e.* generalized) and removed (*i.e.* instantiated). In the program analysis setting this the problem of *context sensitivity* which remains a notorious source of imprecision (Li *et al.*, 2020). In contrast, type- and refinement- polymorphism provides a natural solution: we need only (automatically) instantiate the type variables α and β in `map`'s type

$$(\alpha \rightarrow \beta) \rightarrow \texttt{list}(\alpha) \rightarrow \texttt{list}(\beta)$$

with suitable refined types to enable context-sensitive verification.

12.2 The Bad: Reasoning about State

A reader who has made it this far is clearly aware of the elephant in the room: this article has hewed closely to *pure* programs, and entirely shied away from discussing the topic of *state*. This is partly for exposition, partly because there is already a substantial literature on the topic that merits its own separate survey and partly because *precise* reasoning about imperative features remains quite difficult.

Invariant References The simplest way to account for state is by introducing a type

```
type ref('a)       /* 'a is co- and contra- variant */
```

denoting pointers to values of type `('a)` and then use the standard API for accessing pointers:

```
val new : 'a => ref('a)                    /* allocate */
val get : ref('a) => 'a                    /* read     */
val set : 'a => ref('a) => ()              /* read     */
```

The machinery described in § 5 scales up to account for such references, but suffers from two problems. First, it is *flow-insensitive*: we end up assigning a *single* type to a reference throughout its lifetime (*e.g.* `nat`) as opposed to different types at different points as the reference is *updated*. Second, there is no way to use references in refinements, as the values referred to might change.

Alias and Ownership Types Smith *et al.*, 2000 introduces a mechanism called *Alias Types* for reasoning about references and aliasing within a type system, essentially by typing each pointer with a singleton *location*, and tracking a separate *store* that holds the values of each location. Rondon *et al.*, 2010 describes a way to combine logical predicates with alias types to obtain a refinement type checker for C programs. Similarly, Chugh *et al.*, 2012 shows how to combine alias types with refinements to obtain a refinement system for JavaScript, and Bakst and Jhala, 2016 shows how to extend the approach to recursive alias types thereby allowing refinements to specify and verify complex invariants of linked data structures. Vekris *et al.*, 2016 shows how *ownership types* (Clarke *et al.*, 2001) used to enforce reference immutability for Java (Potanin *et al.*, 2013) can provide a more lightweight mechanism wherein immutable refinements can be embedded within an imperative language like TypeScript.

Monads None of the above methods scale up to handle the combination of higher-order functions *and* state. Filliâtre, 1998 introduced a method for verifying higher order programs with references, and Nanevski *et al.*, 2008a introduce *Hoare Type Theory* which encapsulates that reasoning within *Hoare Monads* which are the usual state-transformers indexed by pre- and post-conditions. This approach, while expressive, is tricky to use as it lacks a way to algorithmically generate verification conditions whose validity implies correctness. F* elegantly solves this problem via the notion of *Dijkstra Monads* where the monad is indexed by a single *predicate transformer*: a function that computes the (most general) heap pre-conditions under which some desired post-condition will hold.

Crucially, the composition of the transformers yields a mechanism for computing VCs. This method, combined with SMT solvers' native support for McCarthy's axioms for reasoning about arrays via the `select` and `store` operations (McCarthy, 1962), yields a powerful way to verify higher-order stateful programs.

Separation Logic Finally, over the last two decades, Separation Logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002) has transformed how we think about the verification of pointer-manipulating programs, and is the basis for modern mechanized program logics like Iris (Jung *et al.*, 2018) and FCSL (Ley-Wild and Nanevski, 2013) which has been used to verify a range of sophisticated concurrent, pointer manipulating algorithms using the Coq proof assistant. In future work it would be interesting to investigate how refinements can be combined with the monadic approach perhaps in combination with separation logic (Kloos *et al.*, 2015) to yield simpler and easier to use tools for verifying stateful programs.

12.3 The Ugly: Explaining Verification Failures

The more sophisticated a static type system, analysis or program logic, the more difficult it is to *explain* failures. In our experience, the most challenging aspect of using refinement types is that the high degree of automation makes it difficult for beginners to understand verification *failures*, which can arise in several modes.

Problem: The Implementation is Wrong The most common case is when the implementation does not respect the specification. For example, suppose `noDups` is a measure (chapter 7) such that `noDups(xs)` holds when the list `xs` contains no duplicates. Hence we can define a type of *unique lists i.e.* without any duplicates as

```
type ulist('a) = list('a)[v| noDups(v)]
```

The following code fails to verify

```
val append : ulist('a) => ulist('a) => ulist('a)
let append = (xs, ys) => {
  switch (xs) {
  | Nil => ys
```

```
    | Cons(x, xs') => Cons(x, append(xs', ys))
    }
  }
```

Unfortunately the error message will simply say that the result of `Cons (x,...)` is not a unique list and the programmer may be quite puzzled as to why.

These failures are the easiest to explain, as one can augment the type checker with some form of symbolic execution to produce *counterexamples* that describe why the property does not hold (Hallahan *et al.*, 2019). For example, in this case, the programmer could be given a counterexample of the form

```
xs              := Cons(1, Nil)
ys              := Cons(1, Nil)
append(xs, ys)  := Cons(1, Cons(1, Nil))
```

which would demonstrate a situation where the output refinement fails to hold even though the input requirements are met, and hopefully this will provide a hint as to how to modify the specification or the code.

Problem: The Specification is Weak A more vexing situation arises when the code *does* satisfy the specification, in that there are no counterexamples, but where verification fails because the specifications were not enough to *prove* the property. Continuing with the `ulist` example from above, suppose that from the counterexample, the programmer has realized that the output is unique only when the input lists have no common elements. They will specify this extra requirement as:

```
val append: xs:ulist('a)
         => ys:ulist('a)[intersect(xs, ys) = empty]
         => ulist(a')
```

But now, imagine their dismay when the code is *again* rejected by the type checker. Unfortunately, this time, we cannot find a counterexample as indeed the function *does* correctly implement the given specification: the concatatenation of two unique lists with no common elements always yields a unique list.

In this case, verification fails because typechecking is *modular*. The only information that the type checker has about the output of a function application, is whatever was specified in the function's type. Thus, in the above example consider the case

```
| Cons(x, xs') => Cons(x, append(xs', ys))
```

The signature for `append` says that the (recursive) call `append(xs',ys)` can return *any* unique list, including one that may possibly contain `x`, and so the `Cons(x,...)` need not be a duplicate-free list. Of course, this cannot happen *in reality* because the list *output from* `append` can only contain elements from the lists *passed into* `append`, but this information is absent from the type signature, preventing verification.

This classic failure mode — widely known in the verification community as the difference between invariants and *inductive* invariants — is a significant stumbling block for programmers as it is difficult to pinpoint exactly where the extra information is needed, and what that information should be. Hallahan *et al.*, 2019 demonstrate an algorithm for generating *counterfactual counterexamples* that can pinpoint the functions whose types need to be strengthened. In future work it would interesting to see if ideas from the synthesis literature (Gulwani *et al.*, 2017) can be used to suggest hints on how to strengthen the specifications or code to facilitate proof, or more broadly, help the programmer rapidly build a robust mental model of the requirements for formal verification. This would go a long way towards flattening the steep learning curve that remains the most daunting hurdle limiting the broader adoption of formal verification in software development.

References

Amin, N., K. R. M. Leino, and T. Rompf. (2014). "Computing with an SMT Solver". In: *Tests and Proofs*.

Andrews, P. B. (2002). *An Introduction to Mathematical Logic and Type Theory*. Springer. URL: https://www.springer.com/gp/book/9781402007637.

Astrauskas, V., P. Müller, F. Poli, and A. J. Summers. (2019). "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. URL: https://doi.org/10.1145/3360573.

Baader, F. and T. Nipkow. (1998). *Term Rewriting and All That*. USA: Cambridge University Press.

Bakst, A. and R. Jhala. (2016). "Predicate Abstraction for Linked Data Structures". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. *Lecture Notes in Computer Science*. Springer. 65–84. DOI: 10.1007/978-3-662-49122-5_3.

Barnett, M., M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. (2011). "Specification and Verification: The Spec# Experience". In: *Communications of the ACM*. URL: https://doi.org/10.1145/1953122.1953145.

Barrett, C., A. Stump, and C. Tinelli. (2010). "The SMT-LIB Standard: Version 2.0". In: *SMT*.

Barthe, G., M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. (2004). "Type-based termination of recursive definitions". In: *MSCS*.

Belo, J. F., M. Greenberg, A. Igarashi, and B. C. Pierce. (2011). "Polymorphic Contracts". In: *European Symposium on Programming (ESOP)*. URL: https://doi.org/10.1007/978-3-642-19718-5_2.

Bengtson, J., K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. (2011). "Refinement types for secure implementations". *ACM TOPLAS*.

Bertot, Y. and P. Castéran. (2004). *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

Bierman, G., A. Gordon, C. Hriţcu, and D. Langworthy. (2010). "Semantic Subtyping with an SMT Solver". In: *International Conference on Functional Programming (ICFP)*. URL: https://doi.org/10.1145/1863543.1863560.

Lehmann, N., R. Kunkel, D. Stefan, and R. Jhala. (2020). "The Binah Web Framework".

Bird, R. S. (1989). "Algebraic Identities for Program Calculation". In: *The Computer Journal*.

Bjørner, N., A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. (2015). "Horn Clause Solvers for Program Verification". In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte. Vol. 9300. *Lecture Notes in Computer Science*. Springer. 24–51. DOI: 10.1007/978-3-319-23534-9_2.

Canning, P., W. Cook, W. Hill, W. Olthoff, and J. Mitchell. (1989). "Abstract F-Bounded Polymorphism for Object-Oriented Programming". In: DOI: 10.1145/99370.99392.

Cartwright, R. (1976). "User-Defined Data Types as an Aid to Verifying LISP Programs". In: *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*. Ed. by S. Michaelson and R. Milner. Edinburgh University Press. 228–256.

Chugh, R., D. Herman, and R. Jhala. (2012). "Dependent Types for JavaScript". In: *OOPLSA*.

Clarke, D. G., J. Noble, and J. M. Potter. (2001). "Simple Ownership Types for Object Containment". In: *ECOOP 01: European Conference on Object Oriented Programming*. 53–76.

Cok, D. R. (2014). "OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse". In: *Electronic Proceedings in Theoretical Computer Science*. URL: http://dx.doi.org/10.4204/EPTCS.149.8.

"Collatz Conjecture". (2021).

Constable, R. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.

Constable, R. L. (1983). "Mathematics as Programming". In: *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*. Ed. by E. M. Clarke and D. Kozen. Vol. 164. *Lecture Notes in Computer Science*. Springer. 116–128. DOI: 10.1007/3-540-12896-4_359.

Cook, B., A. Podelski, and A. Rybalchenko. (2011). "Proving program termination". *Commun. ACM*.

Coquand, T. and G. Huet. (1985). "Constructions: A higher order proof system for mechanizing mathematics". In: *European Conference on Computer Algebra (EUROCAL)*. URL: https://doi.org/10.1007/3-540-15983-5_13.

Coquand, T. and G. Huet. (1988). "The Calculus of Constructions". In: *Information and Computation*. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

Cosman, B. and R. Jhala. (2017). "Local refinement typing". *PACMPL*. 1(ICFP): 26:1–26:27. DOI: 10.1145/3110270.

Damas, L. and R. Milner. (1982a). "Principal Type-Schemes for Functional Programs." In: *POPL*.

Damas, L. and R. Milner. (1982b). "Principal Type-Schemes for Functional Programs". In: *POPL*.

Detlefs, D., G. Nelson, and J. B. Saxe. (2005). "Simplify: a theorem prover for program checking". *J. ACM*. 52(3): 365–473. DOI: 10.1145/1066100.1066102.

Dewar, R. B. K., G. A. Fisher, E. Schonberg, R. Froehlich, S. Bryant, C. F. Goss, and M. Burke. (1980). "The NYU Ada Translator and Interpreter". *SIGPLAN Not.* 15(11): 194–201. DOI: 10.1145/947783.948659.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Dunfield, J. (2007). "A Unified System of Type Refinements". *PhD thesis*. Pittsburgh, PA, USA: Carnegie Mellon University.

Dunfield, J. (2017). "Extensible Datasort Refinements". In: *European Symposium on Programming (ESOP)*. URL: https://doi.org/10.1007/978-3-662-54434-1_18.

Dunfield, J. and N. Krishnaswami. (2020). "Bidirectional Typing". In:

Dunfield, J. and N. R. Krishnaswami. (2013). "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by G. Morrisett and T. Uustalu. ACM. 429–442. DOI: 10.1145/2500365.2500582.

Filliâtre, J. (1998). "Proof of Imperative Programs in Type Theory". In: *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, March 27-31, 1998, Selected Papers*. Ed. by T. Altenkirch, W. Naraschewski, and B. Reus. Vol. 1657. *Lecture Notes in Computer Science*. Springer. 78–92. DOI: 10.1007/3-540-48167-2_6.

Flanagan, C. (2006). "Hybrid Type Checking". In: *POPL*.

Flanagan, C. and K. Leino. (2001). "Houdini, an Annotation Assistant for ESC/Java". URL: citeseer.ist.psu.edu/flanagan00houdini.html.

Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. (1993). "The Essence of Compiling with Continuations." In: *PLDI*.

Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. (2002). "Extended Static Checking for Java". In: *Programming Language Design and Implementation (PLDI)*. URL: https://doi.org/10.1145/512529.512558.

Floyd, R. (1967). "Assigning meanings to programs". In: *Mathematical Aspects of Computer Science*.

Freeman, T. and F. Pfenning. (1991a). "Refinement Types for ML". In: *PLDI*.

Freeman, T. and F. Pfenning. (1991b). "Refinement Types for ML". In: *Programming Language Design and Implementation (PLDI)*. URL: https://doi.org/10.1145/113445.113468.

Giesl, J., M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. (2011). "Automated termination proofs for Haskell by term rewriting". In: *TPLS*.

Gopan, D., T. W. Reps, and S. Sagiv. (2005). "A framework for numeric analysis of array operations." In: *POPL*. 338–350.

Gordon, A. D. and C. Fournet. (2010). "Principles and Applications of Refinement Types". In: *Logics and Languages for Reliability and Security*. IOS Press. URL: https://doi.org/10.3233/978-1-60750-100-8-73.

Gordon, C. S., M. D. Ernst, D. Grossman, and M. J. Parkinson. (2017). "Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types". *ACM Trans. Program. Lang. Syst.* 39(3): 11:1–11:54. DOI: 10.1145/3064850.

Graf, S. and H. Saidi. (1997). "Construction of abstract state graphs with PVS". In: *CAV. LNCS 1254*. Springer. 72–83.

Gronski, J., K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. (2006). "Sage: Hybrid checking for flexible specifications". In: *Scheme and Functional Programming Workshop*. 93–104.

Gulwani, S., O. Polozov, and R. Singh. (2017). "Program Synthesis". *Found. Trends Program. Lang.* 4(1-2): 1–119. DOI: 10.1561/2500000010.

Gurfinkel, A. and N. Bjørner. (2019). "The Science, Art, and Magic of Constrained Horn Clauses". In: *21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019*. IEEE. 6–10. DOI: 10.1109/SYNASC49474.2019.00010.

Hallahan, W. T., A. Xue, M. T. Bland, R. Jhala, and R. Piskac. (2019). "Lazy counterfactual symbolic execution". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 411–424. DOI: 10.1145/3314221.3314618.

Hamza, J., N. Voirol, and V. Kuncak. (2019). "System FR: formalized foundations for the stainless verifier". *Proc. ACM Program. Lang.* 3(OOPSLA): 166:1–166:30. DOI: 10.1145/3360592.

Handley, M. A. T., N. Vazou, and G. Hutton. (2019). "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3371092.

Hindley, R. (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society*.

Hoare, C. A. R. (1971). "Procedures and parameters: An axiomatic approach". In: *Symposium on Semantics of Algorithmic Languages*.

Hoare, C. (1969). "An Axiomatic Basis for Computer Programming". *Communications of the ACM*. 12: 576–580.

Hoder, K. and N. Bjørner. (2012). "Generalized Property Directed Reachability". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. *Lecture Notes in Computer Science*. Springer. 157–171. DOI: 10.1007/978-3-642-31612-8_13.

Hojjat, H. and P. Rümmer. (2018). "The ELDARICA Horn Solver". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by N. Bjørner and A. Gurfinkel. IEEE. 1–7. DOI: 10.23919/FMCAD.2018.8603013.

Howard, W. A. (1980). "The formulae-as-types notion of construction". In: *Essays on Combinatory Logic, Lambda Calculus and Formalism*.

Hughes, J., L. Pareto, and A. Sabry. (1996). "Proving the Correctness of Reactive Systems Using Sized Types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '96*. St. Petersburg Beach, Florida, USA: Association for Computing Machinery. 410–423. DOI: 10.1145/237721.240882.

Ishtiaq, S. S. and P. W. O'Hearn. (2001). "BI as an Assertion Language for Mutable Data Structures." In: *POPL*. 14–26.

Jhala, R. and K. McMillan. (2006). "A Practical and Complete Approach to Predicate Refinement". In: *TACAS 06. LNCS 2987*. Springer-Verlag. 298–312.

Jhala, R., A. Podelski, and A. Rybalchenko. (2018). "Predicate Abstraction for Program Verification". In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing. 447–491. DOI: 10.1007/978-3-319-10575-8_15.

Jones, N. D. and N. Bohr. (2004). "Termination Analysis of the Untyped lamba-Calculus". In: *RTA*.

Jung, R., R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. (2018). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". *J. Funct. Program.* 28: e20. DOI: 10.1017/S0956796818000151.

Kazerounian, M., N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak. (2017). "Refinement Types for Ruby". *CoRR*. abs/1711.09281. arXiv: 1711.09281. URL: http://arxiv.org/abs/1711.09281.

Kent, A. M., D. Kempe, and S. Tobin-Hochstadt. (2016). "Occurrence typing modulo theories". In: *PLDI*.

Kloos, J., R. Majumdar, and V. Vafeiadis. (2015). "Asynchronous Liquid Separation Types". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Ed. by J. T. Boyland. Vol. 37. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 396–420. DOI: 10.4230/LIPIcs.ECOOP.2015.396.

Knowles, K. W. and C. Flanagan. (2009). "Compositional and decidable checking for dependent contract types". In: *PLPV*.

Knowles, K. W. and C. Flanagan. (2010). "Hybrid type checking". In: *TOPLAS*.

Komondoor, R., G. Ramalingam, S. Chandra, and J. Field. (2005). "Dependent Types for Program Understanding." In: *TACAS*. 157–173.

Komuravelli, A., A. Gurfinkel, and S. Chaki. (2016). "SMT-based model checking for recursive programs". *Formal Methods Syst. Des.* 48(3): 175–205. DOI: 10.1007/s10703-016-0249-4.

Kroening, D. and O. Strichman. (2008). *Decision Procedures: An Algorithmic Point of View*. Springer.

Lahiri, S. K., S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. (2009). "Intra-module Inference". In: *Computer Aided Verification*. Ed. by A. Bouajjani and O. Maler. Berlin, Heidelberg: Springer Berlin Heidelberg. 493–508.

Lehmann, N., R. Kunkel, J. Brown, J. Yang, N. Vazou, N. Polikarpova, D. Stefan, and R. Jhala. (2021). "STORM: Refinement Types for Secure Web Applications". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association. 441–459. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann.

Leino, K. R. M. and G. Nelson. (1998). "An Extended Static Checker for Modula-3". In: *Compiler Construction*.

Leino, K. R. M. and C. Pit-Claudel. (2016a). "Trigger selection strategies to stabilize program verifiers". In: *CAV*.

Leino, K. R. M. and N. Polikarpova. (2016). "Verified Calculations". In: *VSTTE*.

Leino, K. R. M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. URL: https://doi.org/10.1007/978-3-642-17511-4_20.

Leino, R. and C. Pit-Claudel. (2016b). "Trigger Selection Strategies to Stabilize Program Verifiers". In: *Computer Aided Verification (CAV)*. URL: https://doi.org/10.1007/978-3-319-41528-4_20.

Ley-Wild, R. and A. Nanevski. (2013). "Subjective Auxiliary State for Coarse-Grained Concurrency". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '13*. Rome, Italy: Association for Computing Machinery. 561–574. DOI: 10.1145/2429069.2429134.

Li, Y., T. Tan, A. Møller, and Y. Smaragdakis. (2020). "A Principled Approach to Selective Context Sensitivity for Pointer Analysis". *ACM Trans. Program. Lang. Syst.* 42(2). DOI: 10.1145/3381915.

Liu, Y., J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou. (2020a). "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell". In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. URL: https://doi.org/10.1145/3428284.

Liu, Y., J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou. (2020b). "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell". *Proc. ACM Program. Lang.* 3(OOPSLA).

Lovas, W. and F. Pfenning. (2010). "Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance". In: *Logical Methods in Computer Science*. URL: http://dx.doi.org/10.2168/LMCS-6(4:5)2010.

Maillard, K., D. Ahman, R. Atkey, G. Martínez, C. Hriţcu, E. Rivas, and É. Tanter. (2019). "Dijkstra Monads for All". In: *International Conference on Functional Programming (ICFP)*. URL: https://doi.org/10.1145/3341708.

Martínez, G., D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. (2019). "Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms". In: *European Symposium on Programming (ESOP)*. URL: https://doi.org/10.1007/978-3-030-17184-1%5C_2.

McCarthy, J. (1962). "Towards a Mathematical Science of Computation". In: *IFIP*.

Melliès, P.-A. and N. Zeilberger. (2015). "Functors Are Type Refinement Systems". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/2676726.2676970.

Might, M. (2007). "Logic-flow analysis of higher-order programs". In: *POPL*. 185–198.

Milner, R. (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences*. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

Mu, S. C., H. S. Ko, and P. Jansson. (2009). "Algebra of Programming in Agda: Dependent Types for Relational Program Derivation". In: *J. Funct. Program.*

Nanevski, A., G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. (2008a). "Ynot: dependent types for imperative programs". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by J. Hook and P. Thiemann. ACM. 229–240. DOI: 10.1145/1411204.1411237.

Nanevski, A., J. G. Morrisett, and L. Birkedal. (2008b). "Hoare type theory, polymorphism and separation". *J. Funct. Program.* 18(5-6): 865–911. DOI: 10.1017/S0956796808006953.

Necula, G. C. (1997). "Proof carrying code". In: *POPL 97: Principles of Programming Languages*. ACM. 106–119.

Nelson, C. G. (1980). "Techniques for Program Verification". *PhD thesis*. Stanford University.

Nguyen, P. C., T. Gilray, S. Tobin-Hochstadt, and D. Van Horn. (2019). "Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019*. Phoenix, AZ, USA: Association for Computing Machinery. 845–859. DOI: 10.1145/3314221.3314643.

Nordstrom, B. and K. Petersson. (1983). "Types and Specifications". In: *IFIP*.

Ou, X., G. Tan, Y. Mandelbaum, and D. Walker. (2004). "Dynamic Typing with Dependent Types". In: *IFIP TCS*.

Parker, J., N. Vazou, and M. Hicks. (2019a). "LWeb: Information Flow Security for Multi-Tier Web Applications". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3290388.

Parker, J., N. Vazou, and M. Hicks. (2019b). "LWeb: information flow security for multi-tier web applications". *Proc. ACM Program. Lang.* 3(POPL): 75:1–75:30. DOI: 10.1145/3290388.

Paulson, L. C., T. Nipkow, and M. Wenzel. (2019). "From LCF to Isabelle/HOL". In: *Formal Aspects of Computing*. URL: https://doi.org/10.1007/s00165-019-00492-1.

Peyton-Jones, S. L., D. Vytiniotis, S. Weirich, and G. Washburn. (2006). "Simple unification-based type inference for GADTs". In: *ICFP*.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pierce, B. C. and D. N. Turner. (1998). "Local Type Inference". In: *POPL*.

Pierce, B. (2003). "Types and Programming Languages: The Next Generation". In: *Logic in Computer Science*. IEEE Computer Society. 32. DOI: 10.1109/LICS.2003.1210042.

Plotkin, G. (1977). "LCF considered as a programming language". In: *Theoretical Computer Science*. URL: http://www.sciencedirect.com/science/article/pii/0304397577900445.

Podelski, A. and A. Rybalchenko. (2004). "Transition Invariants". In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. LICS '04*. USA: IEEE Computer Society. 32–41.

Polikarpova, N., I. Kuraj, and A. Solar-Lezama. (2016). "Program synthesis from polymorphic refinement types". In: *PLDI*.

Polikarpova, N., D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama. (2020). "Liquid information flow control". *Proc. ACM Program. Lang.* 4(ICFP): 105:1–105:30. DOI: 10.1145/3408987.

Potanin, A., J. Östlund, Y. Zibin, and M. D. Ernst. (2013). "Immutability". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by D. Clarke, J. Noble, and T. Wrigstad. Vol. 7850. *Lecture Notes in Computer Science*. Springer. 233–269. DOI: 10.1007/978-3-642-36946-9_9.

Protzenko, J., B. Beurdouche, D. Merigoux, and K. Bhargavan. (2019). "Formally Verified Cryptographic Web Applications in WebAssembly". In: *Security and Privacy (SP)*. URL: http://dx.doi.org/10.1109/SP.2019.00064.

Qiu, X., P. Garg, A. Stefanescu, and P. Madhusudan. (2013). "Natural proofs for structure, data, and separation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by H. Boehm and C. Flanagan. ACM. 231–242. DOI: 10.1145/2491956.2462169.

Reynolds, J. C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures." In: *LICS*. 55–74.

Rice, H. G. (1953). "Classes of Recursively Enumerable Sets and Their Decision Problems". *Transactions of the American Mathematical Society*. 74(2): 358–366. URL: http://www.jstor.org/stable/1990888.

Rondon, P., M. Kawaguchi, and R. Jhala. (2008). "Liquid Types". In: *PLDI*.

Rondon, P., M. Kawaguchi, and R. Jhala. (2010). "Low-Level Liquid Types". In: *POPL*.

Ruemmer, P. (2021). "Constrained Horn Clause Solver Competition".

Rushby, J., S. Owre, and N. Shankar. (1998). "Subtypes for Specifications: Predicate Subtyping in PVS". In:

Sagiv, S., T. W. Reps, and R. Wilhelm. (2002). "Parametric shape analysis via 3-valued logic." *ACM Trans. Program. Lang. Syst.* 24(3): 217–298.

Sarkar, D., O. Waddell, and R. K. Dybvig. (2004). "A Nanopass Infrastructure for Compiler Education". *SIGPLAN Not.* 39(9): 201–212. DOI: 10.1145/1016848.1016878.

Schrijvers, T., S. L. Peyton-Jones, M. Sulzmann, and D. Vytiniotis. (2009). "Complete and decidable type inference for GADTs". In: *ICFP*.

Sekiyama, T., Y. Nishida, and A. Igarashi. (2015). "Manifest Contracts for Datatypes". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by S. K. Rajamani and D. Walker. ACM. 195–207. DOI: 10.1145/2676726.2676996.

Sereni, D. and N. Jones. (2005). "Termination analysis of higher-order functional programs". In: *APLAS*.

Shivers, O. (1988). "Control-Flow Analysis in Scheme". In: *PLDI*.

Smith, F., D. Walker, and J. Morrisett. (2000). "Alias Types". In: *ESOP*.

Sozeau, M., S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. (2020). "Coq Coq Correct. Verification of Type Checking and Erasure for Coq, in Coq". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3371076.

Sulzmann, M., M. Odersky, and M. Wehr. (1997). "Type Inference with Constrained Types". In: *FOOL*. URL: citeseer.ist.psu.edu/article/odersky99type.html.

Suter, P., A. S. Köksal, and V. Kuncak. (2011). "Satisfiability Modulo Recursive Programs". In: *SAS*.

Swamy, N., J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. (2011). "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by M. M. T. Chakravarty, Z. Hu, and O. Danvy. ACM. 266–278. DOI: 10.1145/2034773.2034811.

Swamy, N., C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. (2016). "Dependent Types and Multi-Monadic Effects in F*". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/2837614.2837655.

Timany, A. and M. Sozeau. (2018). "Cumulative Inductive Types in Coq". In: *Formal Structures for Computation and Deduction (FSCD)*. URL: https://doi.org/10.4230/LIPIcs.FSCD.2018.29.

Tobin-Hochstadt, S. and M. Felleisen. (2008). "The design and implementation of typed scheme". In: *POPL*.

Turing, A. M. (1936). "On computable numbers, with an application to the Eintscheidungsproblem". In: *LMS*.

Turing, A. (1949). "Checking a Large Routine". In: *The Early British Computer Conferences*. Cambridge, MA, USA: MIT Press. 70–72.

Jhala, R. (2019). "Types vs. Floyd-Hoare".

Vazou, N., A. Bakst, and R. Jhala. (2015). "Bounded refinement types". In: *ICFP*.

Vazou, N., L. Lampropoulos, and J. Polakow. (2017). "A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq". In: *Haskell*.

Vazou, N., P. Rondon, and R. Jhala. (2013). "Abstract Refinement Types". In: *ESOP*.

Vazou, N., E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton-Jones. (2014a). "Refinement Types for Haskell". In: *ICFP*.

Vazou, N., E. L. Seidel, and R. Jhala. (2014b). "LiquidHaskell: Experience with Refinement Types in the Real World". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. Haskell '14*. Gothenburg, Sweden: Association for Computing Machinery. 39–51. DOI: 10.1145/2633357.2633366.

Vazou, N., A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. (2018). "Refinement reflection: complete verification with SMT". *Proc. ACM Program. Lang.* 2(POPL): 53:1–53:31. DOI: 10.1145/3158141.

Vekris, P., B. Cosman, and R. Jhala. (2016). "Refinement types for TypeScript". In: *PLDI*.

Wadler, P. (2015). "Propositions As Types". In: *Communications of the ACM*.

Wadler, P. (1989). "Theorems for Free". In: *Functional Programming Languages and Computer Architecture (FPCA)*. URL: https://doi.org/10.1145/99370.99404.

Walker, D. and J. Morrisett. (2000). "Alias Types for Recursive Data Structures". In: *Types in Compilation (TIC)*.

Wenzel, M. (2016). "The Isabelle System Manual". URL: https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2016-1/doc/system.pdf.

Winant, T., D. Devriese, F. Piessens, and T. Schrijvers. (2014). "Partial Type Signatures for Haskell". In: *Practical Aspects of Declarative Languages (PADL)*. Ed. by M. Flatt and H.-F. Guo. Vol. 8324. *Lecture Notes in Computer Science*. Springer. 17–32. URL: /Research/papers/padl2014.pdf.

Xi, H. (2001). "Dependent Types for Program Termination Verification". In: *LICS*.

Xi, H. and F. Pfenning. (1998). "Eliminating Array Bound Checking Through Dependent Types". In: *PLDI*.

Zeilberger, N. (2016). "Principles of Refinement Types". In: *Oregon Programming Languages Summer School (OPLSS)*. URL: https://www.cs.bham.ac.uk/~zeilbern/oplss16/refinements-notes.pdf.

Zenger, C. (1997). "Indexed Types". *TCS*.

Zhu, H., S. Magill, and S. Jagannathan. (2018). "A Data-Driven CHC Solver". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2018*. Philadelphia, PA, USA: Association for Computing Machinery. 707–721. DOI: 10.1145/3192366.3192416.

Zinzindohoué, J. K., K. Bhargavan, J. Protzenko, and B. Beurdouche. (2017). “HACL*: A Verified Modern Cryptographic Library”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM. 1789–1806. DOI: 10.1145/3133956.3134043.