UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EXECUTABLE REFINEMENT TYPES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Kenneth L. Knowles

March 2014

The dissertation of Kenneth L. Knowles
is approved:

Professor Cormac Flanagan, Chair

Professor Luca de Alfaro

Professor Ranjit Jhala

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Executable Refinement Types

by

Kenneth L. Knowles

Precise specifications are integral to effective programming practice. Existing specification disciplines such as structural type systems, dynamic contracts, and extended static checking all suffer from limitations such as imprecision, false positives, false negatives, or excessive manual proof burden. New ways of expressing and enforcing program specifications are needed.

Towards that end, this dissertation introduces *executable refinement types* and establishes their metatheory and accompanying implementation techniques. Executable refinement types enrich structural type systems with basic types refined by semi-decidable predicates. Through the lens of executable refinement types, we also address the broader problem of theory and implementation for undecidable type systems.

To establish a firm foundation for the study of executable refinement types, this dissertation presents a full formal account of their metatheory. Type checking for executable refinement types is undecidable. Nonetheless, they fulfill standard metatheoretical correctness criteria including type soundness and extensional equivalence.

To perform type checking for executable refinement types we introduce *hybrid type checking*, a type enforcement strategy broadly applicable to undecidable type systems. Hybrid type checking enforces specifications via static analysis where possible and

dynamic type casts where necessary. We prove that for any decidable approximation of executable refinement types, either: (1) Hybrid type checking catches some errors *statically* which the decidable approximation would miss, or (2) the decidable approximation rejects some correct program which hybrid type checking would accept.

To perform type reconstruction for executable refinement types, we radically revise the usual notion of type reconstruction. Typeability is undecidable because it subsumes type checking. Instead, we propose a more precise definition of type reconstruction as a typeability-*preserving* transformation. For decidable type systems, our definition coincides with the previous. Using our generalized notion of type reconstruction, we demonstrate that type reconstruction for executable refinement types is decidable even though type checking is not! We show this by providing a syntactic type reconstruction algorithm reminiscent of strongest postcondition calculation.

To enlarge the class of programs for which type checking *is* decidable, we formalize the notion of *compositional reasoning* for types systems. Because standard dependent types perform non-compositional reasoning, type checking is undecidable even when all types appearing in a program fall in a decidable specification language. We present a variant of dependent types which uses existential types to achieve compositional reasoning. Even restricted to compositional reasoning, our type system is *exact*: It can give any term a type that completely classifies that term up to contextual equivalence. When reasoning compositionally, if all the annotations in a program fall into a decidable language, then type checking is decidable. We show this with a type checking algorithm for such programs.

Atop these theoretical foundations we implement Sage, a language blending executable refinement types, dynamic typing, and first-class types. Sage's implementation includes standard type-checking machinery, compile-time computation, automatic theorem proving, dynamic contract checking, and a database of run-time failures which inform the hybrid type checker for future runs. Preliminary experiments indicate that Sage is effective at verifying many common examples statically in a reasonable amount of time. Moreover, every run-time failure in Sage can occur at most once: From then onwards it becomes a compile time failure.

Acknowledgments

I entered graduate study in search of a community of intellectuals, and I found it at UC Santa Cruz. I have never encountered such a spirited and surprising group of scholars, nor any with as broad an engagement with mathematics, computing, and philosophy, as I found at UCSC. For my years with them, I am immeasurably grateful.

Particularly for their helpful attention to this work, I thank my advisor, Cormac Flanagan, and my reading committee members, Luca de Alfaro and Ranjit Jhala.

Chapters of this dissertation include expository reorganizations of work by the author and colleagues, updated to unify notation and theory, engage in more explanation and discussion than possible in a terse conference paper, provide greater detail in proofs, and survey subsequent related work. I would like to thank my coauthors Cormac Flanagan, Stephen Freund, Aaron Tomb, and Jessica Gronski for their contributions to Gronski et al. [2006], Knowles and Flanagan [2007], Knowles and Flanagan [2009], and Knowles and Flanagan [2010].

Beyond my the intellectual engagement of my colleagues, I also owe completion of this dissertation to support offered by my family and friends, so I would like to especially thank my mother, my father, and Meg.

Summary of Metavariables

e, d, f, g, q, p	term
x, y, z	variable
k	primitive constant or function
c	constructor supporting pattern matching
b	boolean constant
m, n	natural number or natural number constant
δ	semantic function for constants
S, T, U	type
B	basic type
Q, R	restricted type with limited predicate language
E, F	augmented type with covariant existentials
a, b	three-valued certainty (one of $\times, \surd, ?$)
Γ	typing environment
Δ	typing environment with augmented types
θ	capture-avoiding substitution function
σ, γ	closing substitution
ζ	approximation function to decidable refinements
ψ	term placeholder
π	*non*-capture-avoiding type replacement
ρ	*non*-capture-avoiding placeholder replacement
$\mathcal{C}$	term context
$\mathcal{D}$	type context
$\mathcal{E}$	call-by-value evaluation context
$\mathcal{S}$	unfolding evaluation context
$\mathcal{W}$	head-normal form evaluation context

Summary of Notations

$\lambda x\!:\!T.\, e$	function term
$f\ e$	function application
$\texttt{case}\ e\ \texttt{of}\ \overline{c \Rightarrow f}$	case discrimination term
$\langle T \triangleleft S \rangle$	type cast
$\langle T \rangle$	single-type cast
$\langle \{x\!:\!B \mid p\}, q, k \rangle$	cast in progress
$S \to T$	function type
$x\!:\!S \to T$	dependent function type
$\{x\!:\!B \mid e\}$	refinement type
$\exists x\!:\!S.\, T$	existential type
$\lfloor T \rfloor, \lfloor e \rfloor$	shape of type or term
$[\![T]\!]$	denotation of type
$\times, \surd, ?$	certainty values of "no", "yes", "maybe"
$[x \mapsto e]$	substitution mapping x to e
$[x \mapsto e : T]$	substitution mapping x to e with type annotation
$\theta \cdot \alpha$	delayed substitution applied to a type variable
$\theta \cdot \psi$	delayed substitution applied to a placeholder
$\mathcal{C}[e]$, $\mathcal{D}[e]$, $\mathcal{E}[e]$, $\mathcal{S}[e]$, $\mathcal{W}[e]$	context application

Summary of Relations

$e_1 \curvearrowright e_2$	single-step redex reduction
$e_1 \rightsquigarrow e_2$, $S \rightsquigarrow T$	single-step term or type reduction
$e_1 \rightsquigarrow^+ e_2$, $S \rightsquigarrow^+ T$	transitive closure of single-step reduction
$e_1 \rightsquigarrow^* e_2$, $S \rightsquigarrow^* T$	reflexive-transitive closure of single-step reduction
$e_1 \leftrightsquigarrow e_2$	equivalence closure of single-step reduction
$e_1 \equiv e_2$	contextual equivalence of terms
$S \curlywedge T$	wedge product of types
$\sigma \curlywedge \gamma$	wedge product of closing substitutions

Summary of Judgments

$\Gamma \vdash e : T$	typing
$\Gamma \vdash T$	type well-formedness
$\vdash \Gamma$	typing environment well-formedness
$\Gamma \vdash_{\mathrm{wf}} \theta$	delayed substution well-formedness
$\Gamma \vdash S <: T$	subtyping
$\vdash \sigma : \Gamma$	closing substitution typing
$\Gamma \vdash e_1 \sim e_2 : T$	extensional equivalence of terms
$\vdash \sigma \sim \gamma : \Gamma$	extensional equivalence of closing substitutions
$\Gamma \vdash e \in T$	semantic typing
$\Gamma \vdash S \subseteq T$	semantic subtyping
$\Gamma \vdash e_1 \Rightarrow e_2$	implication
$\Gamma \models p$	validity
$\Gamma \Vdash d \hookrightarrow e : T$	cast insertion
$\Gamma \Vdash d \hookrightarrow e \downarrow T$	cast insertion and checking
$\Gamma \Vdash S \hookrightarrow T$	cast insertion in types
$\Gamma \Vdash^a S <: T$	three-valued algorithmic subtyping
$\Gamma \Vdash^a S \Rightarrow T$	three-valued algorithmic implication
$\Gamma \Vdash e : E$	decidable algorithmic typing
$\Gamma \Vdash R$	decidable algorithmic type well-formedness
$\Gamma \Vdash R <: E$	decidable algorithmic subtyping

Chapter 1

Introduction

A program is, of necessity, a completely unambiguous document. To manage the enormous detail that this entails, Parnas [1972a] articulates a modular development strategy based on *information hiding* where trusted and precisely specified abstractions of the sections of a program – now known as *interfaces* – reduce the amount of detail that must be considered at one time, enabling cooperation amongst programmers over space and time.

In practice, however, interfaces are often only informally or partially specified. Consider the following increasingly-precise specifications for the argument to a procedure for inverting a matrix:

1. The argument can be any value expressible in the programming language.

2. The argument must be an array of arrays of numbers.

3. The argument must be a *matrix*, that is, a rectangular (non-ragged) array of arrays

of numbers.

4. The argument must be a square matrix.

5. The argument must be a square matrix with nonzero determinant.

All of these specifications may be useful for various purposes, levels of assurance, and moments in the development of a program. Simpler specifications are common during rapid prototyping, at the cost of proliferation of partial functions (functions which are undefined on portions of their purported domain). More precise specifications provide stronger correctness guarantees and better documentation, and simplify function invocation by reducing or removing the possibility of passing a "bad" input.

The first specification corresponds approximately to what are called *dynamically typed* (also *unityped* or *single-sorted*) programming languages. The second item is a likely specification in *statically typed* languages such as Java, C++, C#, OCaml, or Haskell, which automatically distinguish sorts of program values. The last three specifications remain the stuff of research, including this dissertation. Often such specifications, if they are checked at all, are dynamically checked by ad-hoc hand-written methods as part of a program's execution.

As the precision afforded by a specification language – its *expressiveness* – increases, so does the computational complexity of determining whether a program meets a specification. Specifications written in less expressive languages, such as traditional type systems (or no language at all in the case of unityped languages), can be checked *statically*: There is an algorithm which, given the text of a program, either accepts that

the program obeys the interface discipline or rejects the program as ill-formed. In other words, the question of whether a program meets a specification is decidable.

More precise specifications, such as those written in formal logics including recursion/induction, often cannot be checked statically: They are written in a language expressive enough that no algorithm can exist that correctly accepts or rejects all programs. This does not preclude the existence of useful approximate algorithms, but every one must include false negatives (programs which do not meet their specification but are accepted) or false positives (programs which do meet their specification but are rejected) or both.[1]

1.1 Specification Disciplines

Generally, *static* checking reasons about the text of a program prior to its execution, thus implicitly reasons about all possible inputs. *Dynamic* checking avoids this universal quantification by reasoning only about the values that arise during a particular execution. Many properties that are difficult to establish statically are easy or trivial to ensure dynamically, for example that a pointer is non-null or that a binary tree is balanced.

[1] The distinction between whether acceptance or rejection is the positive case is arbitrary. In program verification spurious errors are usually considered false positives, while programs that may crash are considered false negatives.

1.1.1 Dynamic Contracts

Dynamic *contracts* are a programming construct for organizing and composing hand-written routines that dynamically enforce interface specifications and for isolating the root cause of a violation during program execution. Many common specifications can be expressed as executable contracts,[2] such as:

- Subranges: The function `squareRoot` requires its argument to be a non-negative number.

- Aliasing restrictions: The function `swap` requires that its arguments are distinct reference cells.

- Ordering restrictions: The function `binarySearch` requires that its argument is a sorted array.

- Size specifications: The function `serializeMatrix` takes as input a matrix of size n by m, and returns a one-dimensional array of size $n \times m$.

- Arbitrary executable predicates: an interpreter (or code generator) for a typed language (or intermediate representation [Tarditi et al. 1996]) might naturally require that its input be well-typed, *i.e.*, that it satisfies the programmer-written (and likely optimized, hence obfuscated) predicate `wellTyped` : `Expr` $\rightarrow$ `Bool`.

Contracts have been incorporated into many languages, including Turing [Holt and Cordy 1988], Anna [Luckham 1990], Eiffel [Meyer 1992], Sather [Stoutamire and

[2]Notable classes of properties that are not easily checked dynamically are concurrency properties such as race-freedom or atomicity and properties requiring (potentially) infinite quantification such as associativity or injectivity.

Omohundro 1996], Blue [Kölling and Rosenberg 1996], Scheme [Findler and Felleisen 2002], Spec# [Barnett et al. 2005], JML [Burdy et al. 2005; Leavens and Cheon 2006], and Haskell [Hinze et al. 2006; Chitil 2012]. In these languages, functions may have pre-condition checks (executed before the body of the function is evaluated), postcondition checks (executed after the result of the function has been computed), and invariants (executed when a data structure is modified, to ensure its integrity).

However, dynamic checking suffers from two limitations. First, it incurs a computational cost during execution, often running checks repeatedly and redundantly, and in the worst case is expensive enough that it is feasible only during testing. For example constraining a binary tree by the contract `isBinarySearchTree` will require a full traversal of the tree with every insertion, changing the complexity of the operation from $O(\log n)$ to $\Omega(n)$.[3] More seriously, the very property of dynamic checking that makes it simple – that specifications are only checked on data values and code paths of actual executions – often results in incomplete and late (possibly post-deployment) detection of defects. For example, consider the function `abs` for computing the absolute value of an integer, subject to the contract `Int` $\rightarrow$ `NonNeg` which states that `abs` requires an integer input and returns a non-negative result. If we mistakenly simply return the input as the output, then the function violates its contract for all negative inputs. Yet executions such as (`abs` 4) will leave this defect undiscovered.

[3]The "solution" to this problem is to write only those contracts which can be efficiently executed, and to make sure they are only checked when necessary, via clever programming.

1.1.2 Refinement Types

The term *refinement type*[4] is not an agreed-upon mathematical term, but an intuitive label applied to type systems where the basic types of a language may be "refined" to classify a more precise subset of the terms of the language. For example, if the type of linked lists in a structural type system were `List`, then one may consider refining this type to carry information about the length of the list, writing `List` n to describe lists of length n. This style of refinement type, now commonplace, is called an *indexed type* where n is the index a refinement type.

Dependent ML [Xi and Pfenning 1999] includes a type system with indexed types that can express and enforce properties involving linear inequalities between integer variables, and by this express subranges, ordering restrictions, and array size constraints. Other refinement type systems, each enhancing a structural type system with some decidable logic, include Hayashi [1993], Denney [1998], Xi and Pfenning [1999], Xi [2000], and Davies and Pfenning [2000], Mandelbaum et al. [2003], and Dunfield and Pfenning [2004].

Refinement types improve on dynamic contract checking in terms of completeness, because once a program is accepted by the type system, it is certain to satisfy all preconditions and postconditions for all possible executions, but the specification language is intentionally restricted to ensure that specifications can always be checked statically and efficiently so that the analysis can be incorporated into an iterative work-

[4]The term *refinement type* originated with Freeman and Pfenning [1991] but is now used much more broadly.

flow.

In addition to limitations to ensure decidability, properties enforced by indexed types are expressed in an indirect style. A more direct way of expressing `List` n using standard set comprehension notation is $\{x{:}\texttt{List} \mid \texttt{length}(x) = n\}$. More complex types are even less direct, for example the type of lists of length greater than k in indexed style is

$$\exists n > k.\, \texttt{List}\ n$$

where a more direct transliteration of the specification would be

$$\{x{:}\texttt{List} \mid \texttt{length}(x) > k\}$$

For more sophisticated data structures and specifications, such as red-black trees, the indices and methods for expressing the desired specification are even more complex.

1.1.3 Extended Static Checking

If expressivity is more desirable than efficiency but complete checking is still desired, as it was for Parnas [1972b] and Dijkstra [1976], then *extended static checking* may be attempted via a tool such as PVS [Rushby et al. 1998] or ESC/Java [Flanagan et al. 2002]. Extended static checking refers to static verification of specifications written in a very expressive language of preconditions and postconditions. This combines the benefits of complete checking and expressive specifications, but results in large numbers of false positives – programs where all specifications are satisfied, but the analysis cannot verify that this is so. To direct the analysis towards successful verification, the

programmer must add many complex annotations or, in the extreme, provide a manual proof of correctness.

1.2 Thesis Overview

Synthesizing the properties of the specification systems discussed so far, some desirable properties of a specification system are:

- No false positives: Programs which meet their specification are accepted.
- No false negatives: Programs which do not meet their specification are rejected. (This subsumes complete checking, in which a program must meet its specification for *all* inputs or it is rejected.)
- Efficient checking: It can be efficiently and automatically determined whether a program meets its specification. (This subsumes decidable checking and often reduces to to efficient specification inclusion, in which it can be automatically determined whether one specification is strictly more precise than another.)
- Low proof burden: A human need not manually prove complex properties.
- Specification reconstruction: When some specifications are omitted, they can be automatically deduced. (This is similar to "low proof burden" if one considers programming-supplied specifications as a lightweight form of manual proof.)
- Compositionality: The specification of a program term is a sufficient summary for all further analyses.

- Readable specifications: The specification of a program is a self-explanatory mathematical statement about the program.

- Expressivity: A given set of program values can be described in the specification language.

The open (and open-ended) problem that this dissertation addresses is the search for practical combinations of *complete checking* and *expressive specifications* that nonetheless enjoy as many of these desirable properties as may be achieved.

In particular, we narrow the search by working with the theoretical machinery of type systems, while removing any restrictions to ensure decidability or efficiency of type checking as conventionally construed. To illustrate the key idea common to these type systems, consider the following type rule for function application with subtyping.

$$\frac{f : T \to U \qquad e : S \qquad S <: T}{f\ e : U}$$

The antecedent $f : T \to U$ states that the program term f is a function that takes any value of type T as input and returns a value of type U. The antecedent $e : S$ states that e is a program term of type S. The question of whether f may be applied to e hinges on whether a member of type S is necessarily also a member of type T, formally posed as "is S a subtype of T?" and written $S <: T$. If the type checker can prove this subtyping relation, then this application is accepted. Conversely, if the type checker can prove that this subtyping relation does not hold, then the program is rejected. In a conventional, decidable type system, one of these two cases always holds.

However, for type languages that are sufficiently expressive that type checking is not decidable, the type checker will encounter situations where its algorithms can neither prove nor refute the subtype judgment $S <: T$ (particularly within the time bounds imposed by interactive compilation). When subtyping is undecidable, then type checing is undecidable, and type reconstruction as conventionally defined is undecidable. How shall one proceed?

This dissertation defines such a type system: *executable refinement types*. Executable refinement types use the language of dynamic contracts, so that any predicate that can be written in a programming language may refine a type. Thus in addition to traditional types such as `Int`, `Bool`, and $\texttt{Int} \to \texttt{Bool}$, the type system includes types such as $\{x{:}\texttt{Int} \mid x > 0\}$ (the type of positive integers) and $x{:}\texttt{Bool} \to \{y{:}\texttt{Bool} \mid x \Leftrightarrow y\}$ (the type of functions from booleans to booleans where the return value is `true` if and only if the input value is `true`). The techniques gleaned from a close engagement with executable refinement types are broadly applicable, because executable refinement types include *all* semidecidable specifications.

The contributions of this dissertation are organized into chapters as follows.

- Chapter 2 provides references to material containing adequate background to contextualize its contribution to the field of programming languages.

- Chapter 3 presents a core calculus $\lambda_{\mathcal{H}}$ which will form the theoretical basis of the following chapters and establishes its basic correctness properties using a simple, but novel, application of denotational semantics to provide a foundation for our

refinement type system. In addition to the usual soundness criteria, this chapter introduces an original variation of the technique of *logical relations* to establish the correctness of our notion of "function" as well as provide a powerful tool for proving program equivalences.

- Chapter 4 introduces *hybrid type checking* (HTC), a general technique for combining a static type system with dynamic checks, and describes the correctness properties any hybrid type checking system should have. We then illustrate hybrid type checking for the core calculus of Chapter 3 and prove its correctness using the proof tools of the prior chapter. Finally, we compare hybrid type checking with static type checking, concluding that any decidable restriction always rejects a well-typed program which HTC accepts or accepts an ill-typed program which HTC rejects.

- Chapter 5 presents an original generalization of the type reconstruction problem that is applicable to undecidable type systems, then presents a decision procedure for full type reconstruction of $\lambda_{\mathcal{H}}$.

- Chapter 6 examines the special case where all specifications are written in a decidable language, uncovering a critical flaw in the unrestricted use of dependent types in this context. This flaw is resolved by the use of standard type-theoretic techniques, and a decision procedure for type checking is provided and proved sound.

- Chapter 7 presents SAGE, an implementation of executable refinement types fea-

turing hybrid type checking. In addition to refinement types, SAGE includes dynamic types and first-class computation over types, demonstrating the breadth of interesting undecidable type systems to which our techniques apply. This chapter presents some preliminary empirical validation that hybrid type checking decides most type checking queries statically.

- Chapter 8 surveys related work developed in parallel or subsequent to the material presented in this dissertation, recapitulating citations from substantive chapters into coherent research trends.

- Chapter 9 concludes and discusses future directions.

Chapters 1-3 should be read prior to any other chapters, but then all of Chapters 4-7 may be read mostly independently before the concluding discussion of Chapters 8 and 9.

Chapter 2

Preliminaries

This dissertation is best enjoyed with a strong background in programming languages, especially type systems. The references and mathematical content of this chapter are intended to prepare most readers for the remainder of the dissertation, or at least to allow a reader to ascertain the degree of overlap between their experience and the content herein. Please feel free to skip this chapter on first reading and return to it as needed.

As it does not constitute part of the novel contribution of this dissertation (except as the particulars of presentation may differ from others) the material is presented somewhat abruptly and without extensive motivation or discussion.

2.1 Preparatory Readings

The fundamentals of programming language theory that form the basic mathematical setting for this dissertation include: operational semantics, denotational se-

mantics, axiomatic semantics, type systems (particularly dependent type systems), and logical relations.

For operational semantics and type systems, Pierce [2002] provides a thorough introduction in a textbook setting, using notation and terminology similar to that of this dissertation. Its sequel, Pierce [2004], covers advanced topics such as dependent types and logical relations.

From axiomatic semantics [Floyd 1967; Hoare 1969; Dijkstra 1976] we draw upon many descriptive terms for predicates that may be true of program fragments such as *preconditions* (predicates that must hold before a procedure is begun), *postconditions* (predicates that must hold after a procedure completes), *invariants* (predicates that must always hold), *weakest preconditions* (the weakest predicate that allows a fragment to complete successfully), and *strongest postconditions* (the strongest predicate a fragment ensures). Winskel [1993] is a suitable textbook for a unified presentation.

2.2 The Simply-Typed λ-Calculus

The starting point of our metatheoretical work is the simply-typed λ-calculus [Church 1940] augmented with built-in types such as `Int` and `Bool`, and primitives such as boolean constants (`true` and `false`), integer constants (1, 4, -12, etc), conditionals (if), and recursion (in the form of a fixed-point operator $\texttt{fix}_T$ for each type T) We call this variant *STLC*.[1] The material of this section is thoroughly treated by any introductory text, such as Pierce [2002], but we present it here because later developments refer

[1]This system may also be considered a variant of PCF. [Scott 1969]

Figure 2.1: Syntax of STLC

$e, d, f, g, q, p ::=$	*Terms:*
k	primitive
x	variable
$\lambda x : S.\, e$	abstraction
$f\ e$	application
$v ::=$	*Values: ($\subset$ Terms)*
k	primitive
$\lambda x : S.\, e$	abstraction
$S, T, U ::=$	*Types:*
$S \to T$	function type
B	basic type

directly to specific details. This section also may serve to familiarize the reader with our notational conventions.

2.2.1 Syntax of STLC

The syntax for STLC described in this section is summarized via recursive grammars in Figure 2.1 using our notation and metavariables. Many metavariables are reserved for terms and types so that they may be varied for legibility.

- The type $S \to T$ denotes the type of functions from S to T.
- The type B stands for any of the built-in basic types. We freely assume that these include familiar types such as `Bool`, `Int`, and `Unit` (the type with just one element), while introducing others as necessary.

- The term k denotes any of the built-in primitives, which may be functions. We freely assume that these include primitives for arithmetic (*e.g.* 0, 3, +, ×, >=), conditionals (*e.g.* `true`, `false`, $\texttt{if}_T$ for each type T), and recursion ($\texttt{fix}_T$ for each type T).

- The term $\lambda x : S.\, e$ denotes a function taking a parameter x of type S and having a result calculated according to the body of the function e. The particular name chosen for the parameter x is of no consequence. For example, the term $\lambda x\!:\!\texttt{Int}.\, x + 3$ is considered equal to $\lambda y\!:\!\texttt{Int}.\, y + 3$ by definition. Variable binding and renaming (α-equivalence) is covered thoroughly by the preparatory reading of Section 2.1.

- The term x denotes a variable reference – variable references are called *bound* if they occur within the scope of a λ-abstraction of the same variable, and otherwise are called *free*. The set of free variables of a term e is written $fv(e)$. Again, a thorough discussion of free and bound variables is relegated to the background reading of Section 2.1.

- The term $f\ e$ denotes the application of the term f to the term e. Note that f and e are simply metavariables chosen for legibility; either the function or argument may be any term.

Figure 2.2: Operational Semantics of STLC

Redex Evaluation $\boxed{d \curvearrowright e}$

$$(\lambda x{:}S.\, e)\ d \curvearrowright [x \mapsto d]\, e \qquad [\text{E-}\beta]$$
$$k\ v \curvearrowright \delta(k, v) \qquad [\text{E-}\delta]$$

Contextual Evaluation $\boxed{d \rightsquigarrow e}$

$$\mathcal{C}[d] \rightsquigarrow \mathcal{C}[e] \quad \text{if } d \curvearrowright e \qquad [\text{E-Ctx}]$$

Contexts $\boxed{\mathcal{C},\, \mathcal{D}}$

$$\mathcal{C} ::= \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C}$$

2.2.2 Operational Semantics of STLC

Figure 2.2 gives formal meaning to STLC terms via a standard small-step operational semantics. The relation $d \curvearrowright e$ expresses evaluation of *redexes*, short for "reducible expressions". It means that *by definition* we consider d and e to be equivalent terms, and the notation is chosen to correspond to the intuition that a program represented by d may be transformed to a program represented by e in a "single step" of computation. What constitutes a step of computation here is not suitable for algorithmic analysis, but only as a formal tool for discussing equivalent programs.

The two redex evaluation rules in Figure 2.2 concern function application for λ-abstractions and primitive functions:

[E-β]: An application of a function $\lambda x{:}S.\, e$ to argument d is reduced by replacing the

formal parameter x by the actual parameter d. The notation $[x \mapsto d]$ denotes the standard capture-avoiding substitution of x for d; it is not a program term but the following operation carried out (by induction) on the syntax of e:

$$\begin{aligned}
[x \mapsto d]\, k &\stackrel{\text{def}}{=} k \\
[x \mapsto d]\, y &\stackrel{\text{def}}{=} y \qquad (y \neq x) \\
[x \mapsto d]\, x &\stackrel{\text{def}}{=} d \\
[x \mapsto d]\, (f\ e) &\stackrel{\text{def}}{=} ([x \mapsto d]\, f)\ ([x \mapsto d]\, e) \\
[x \mapsto d]\, (\lambda y\!:\!S.\, e) &\stackrel{\text{def}}{=} \lambda y\!:\!S.\, [x \mapsto d]\, e \qquad (y \neq x \text{ and } y \text{ not free in } d)
\end{aligned}$$

In the last equation, the bound variable y may always be chosen (via α-equivalence) to satisfy the side condition. Capture-avoiding substitutions may be composed like any other (meta-)function and are named using metavariables θ, σ, and γ.

[E-δ]: Application of a primitive function is resolved by a presumed partial function

$$\delta : \mathit{Prim} \times \mathit{Term} \rightharpoonup \mathit{Term}$$

which is parameter of the system constrained (Requirement 2.1 on page 21) to be total for appropriately typed arguments and to map them to appropriately typed results.

We assume standard meanings for boolean and arithmetic connectives, conditionals, and the fixed point operator. For example, the following equations

(based on presumed primitives) should all hold for any definition of δ.

$$\begin{aligned}
\delta(\texttt{not}, \texttt{true}) &= \texttt{false} \\
\delta(\texttt{not}, \texttt{false}) &= \texttt{true} \\
\delta(\delta(+, 3), 7) &= 10 \\
\delta(\texttt{if}_T, \texttt{true}) &= \lambda x{:}T.\, \lambda y{:}T.\, x \\
\delta(\texttt{if}_T, \texttt{false}) &= \lambda x{:}T.\, \lambda y{:}T.\, y \\
\delta(\texttt{fix}_T, e) &= e\ (\texttt{fix}_T\ e)
\end{aligned}$$

The single-step evaluation relation $d \rightsquigarrow e$ denotes redex evaluation of an arbitrary subterm of d. Specifically, redex evaluation takes place within a context $\mathcal{C}$ that represents a term with exactly one occurrence of the symbol $\bullet$ (a "hole") where a term may be placed, such as $(\lambda x{:}S.\, f\ \bullet)\ d$. The notation $\mathcal{C}[e]$ denotes the resulting term when the e is placed in the unique hole in $\mathcal{C}$. This operation is *not* capture-avoiding; e may have free variables that become bound once it is placed in $\mathcal{C}$. The reflexive-transitive closure of $\rightsquigarrow$ is written $\rightsquigarrow^*$ and may be read intuitively as "running the program for an arbitrary number of steps".

2.2.3 The STLC Type System

Figure 2.3 presents the typing rules for STLC in full, along with the definition of typing environments Γ which bind variables to their declared types. Bindings in the environments are ordered but environments may be freely treated as partial functions from variables to types or sets of bindings when order is insignificant. The typing

Figure 2.3: Typing Rules for STLC

$$\Gamma ::= \quad \textit{Environments:}$$
$$\emptyset \quad \text{empty environment}$$
$$\Gamma, x : T \quad \text{environment extension}$$

STLC Typing $\boxed{\Gamma \vdash_{\text{STLC}} e : T}$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash_{\text{STLC}} x \ : \ T} \ [\text{STLC-Var}] \qquad \frac{}{\Gamma \vdash_{\text{STLC}} k \ : \ ty(k)} \ [\text{STLC-Prim}]$$

$$\frac{\Gamma, x{:}S \vdash_{\text{STLC}} e \ : \ T}{\Gamma \vdash_{\text{STLC}} \lambda x{:}S.\, e \ : \ S \to T} \ [\text{STLC-Fun}] \qquad \frac{\Gamma \vdash_{\text{STLC}} f \ : \ S \qquad \Gamma \vdash_{\text{STLC}} e \ : \ S \to T}{\Gamma \vdash_{\text{STLC}} f \ e \ : \ T} \ [\text{STLC-App}]$$

judgment

$$\Gamma \vdash_{\text{STLC}} e \ : \ T$$

reads "Assuming the bindings in Γ, the term e may be assigned STLC type T." As with the syntax and semantics, these typing rules are completely standard, and the following narrative descriptions are provided as a reference for the notation.

[STLC-Var]: A variable x is assigned its bound type in the environment Γ.

[STLC-Prim]: A primitive k is given a type by the function ty, which is constrained to maintain type soundness along with the δ function. Any definition of ty

includes such mappings as:

$$\begin{aligned}
\texttt{true} &: \texttt{Bool} \\
\texttt{false} &: \texttt{Bool} \\
\Leftrightarrow &: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \\
\texttt{not} &: \texttt{Bool} \to \texttt{Bool} \\
n &: \texttt{Int} \\
+ &: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
\texttt{if}_T &: \texttt{Bool} \to T \to T \to T \\
\texttt{fix}_T &: (T \to T) \to T
\end{aligned}$$

Primitives with function types may be called *primitive functions* and primitives of basic type may be called *primitive constants*.

[STLC-Fun]: For a λ-abstraction $(\lambda x\!:\!S.\, e)$ the body e of the function is checked in an environment extended by x bound with type S.

[STLC-App]: For an application $(f\ e)$, if the term f in function position has the function type $S \to T$ and is applied to an argument of type S, then the resulting type of the application is T.

The soundness of this calculus is well-established and we do not present the proofs here, but introduce the requirement on primitives upon which this system is parameterized, and state the classical results.

Requirement 2.1 (Types of STLC Primitives)

For each $k \in Prim$, if k is a primitive function then it cannot get stuck and its behavior is compatible with its type, i.e. *if $\emptyset \vdash_{\text{STLC}} k\ v : T$ then $\delta(k, v)$ is defined and $\emptyset \vdash_{\text{STLC}} \delta(k, v) : T$.*

The *preservation* lemma proved upon Requirement 2.1 ensures that for any well-typed term e that in any evaluation sequence proceeding

$$e \rightsquigarrow e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots$$

each e_i is well-typed with the same type as e.

Lemma 2.2 (Preservation for STLC) *For any term e, if $\Gamma \vdash_{\text{STLC}} e : T$ and $e \rightsquigarrow e'$, then $\Gamma \vdash_{\text{STLC}} e' : T$.*

Since terms that reduce to or from each other are definitionally equivalent, this indicates that typing is well-defined for the full equivalence class. However, preservation may be vacuous if no such reduction sequence exists, such as for terms like 3 (4 $-$ $+$) (deliberately constructed to be nonsensical). If the type system assigned this term type `Bool`, preservation would still hold, so the *progress* lemma proves that the only well-typed terms with empty evaluation sequences are those in the known set of values.

Lemma 2.3 (Progress for STLC) *For any term e, if $\emptyset \vdash_{\text{STLC}} e : T$ then either e is a value or there exists e' such that $e \rightsquigarrow e'$.*

Type soundness is such a simple corrollary of preservation and progress that it is often not even stated, but since the remaining chapters prove type soundness in a few different ways, we state the full result to complete the example metatheoretical development.

Theorem 2.4 (Type Soundness for STLC) *For any term e, if $\emptyset \vdash_{\text{STLC}} e : T$ then either e is nonterminating or e evaluates to some value v such that $\emptyset \vdash_{\text{STLC}} v : T$*

Chapter 3

Executable Refinement Types

This chapter describes the metatheoretical core of this dissertation: *executable refinement types*. This is a type system with the expressivity of dynamic contracts. Basic built-in types such as `Int` and `Bool` may be refined by arbitrary predicates written in the programming language at hand. These refined base types are combined via dependent function types. Because these types share their design with higher-order dependent contracts, they can be enforced at run-time using the same techniques.

Before delving into formal definitions, consider these illustrative examples. The type of integers greater than zero may be written as

$$\{x{:}\texttt{Int} \mid x > 0\}$$

and the type of binary search trees may be written as

$$\{x{:}\texttt{Tree} \mid \texttt{isBinarySearchTree}(x)\}$$

where `isBinarySearchTree` is a hand-coded integrity checking routine. The operation

to add an integer x to a set y may be given the type

$$x{:}\texttt{Int} \to \texttt{Set} \to \{z{:}\texttt{Set} \mid x \in z\}$$

Any semi-decidable predicate over base types is expressible, including undecidable predicates. One such undecidable predicate is "integers representing the binary encoding of a turing machine which halts on empty input" which might appear as $\{x{:}\texttt{Int} \mid \texttt{haltsOnEmptyInput}(x)\}$. While such precise types make type checking undecidable, the type system still enjoys the usual notion of type soundness ("well typed programs don't go wrong"). Thus, we can use it as a reference point from which to explore pragmatic techniques for working with expressive types. Rather than building a full programming language suitable for general use, the type system is presented as part of a small calculus, in order to focus on the theoretical aspects of the type system.

Chapter Outline

The metatheoretical contributions of this chapter are organized as follows:

- Section 3.1 presents the constructs of the calculus $\lambda_{\mathcal{H}}$.
- Section 3.2 gives formal meaning to the terms of $\lambda_{\mathcal{H}}$ via a small-step operational semantics.
- Section 3.3 formalizes the intuition that refinement types express subsets of types of the simply-typed lambda calculus, providing a semantic foundation for the syntactic formalisms of the $\lambda_{\mathcal{H}}$ type system.

- Section 3.4 presents the type system of $\lambda_{\mathcal{H}}$ as a set of inference rules for assigning types to terms and proving subtyping between $\lambda_{\mathcal{H}}$ types.

- Section 3.5 establishes type soundness for $\lambda_{\mathcal{H}}$ (any well-typed term is either non-terminating or results in a value of the same type) via the syntactic approach of proving lemmas for *progress* (any well-typed term is either a value or can evaluate further) and *preservation* (a term's type remains unchanged by evaluation).

- Section 3.6 establishes that $\lambda_{\mathcal{H}}$ terms respect *extensionality* (functions which map equal inputs to equal output are observably equivalent) via a new *logical relation* [Statman 1985]. This provides a powerful technique for proving the observable equivalence of $\lambda_{\mathcal{H}}$ terms.

- Section 3.7 discusses other systems with similar approaches to specifying or reasoning about programs.

3.1 The Language $\lambda_{\mathcal{H}}$

This section defines our calculus $\lambda_{\mathcal{H}}$, an enhancement of STLC with more precise types to model programming languages with types that can express the same specifications as dynamic contracts. Figure 3.1 summarizes the syntax of $\lambda_{\mathcal{H}}$ for reference. The grammar of terms is unchanged from STLC; it is the type language of $\lambda_{\mathcal{H}}$ that forms the core of this dissertation: Basic types refined by executable predicates and dependent function types.

Figure 3.1: Syntax of $\lambda_{\mathcal{H}}$

$e, d, f, g, q, p ::=$	*Terms:*
k	constant
x	variable
$\lambda x\!:\!S.\, e$	abstraction
$f\ e$	application
$v ::=$	*Values: ($\subset$ Terms)*
k	constant
$\lambda x\!:\!S.\, e$	abstraction
$S, T, U ::=$	*Types:*
$x\!:\!S \to T$	***dependent function type***
$\{x\!:\!B \mid p\}$	***refined basic type***

- A dependent function type $x\!:\!S \to T$ classifies functions of domain S and codomain T.[1] However, x may appear in T, so the type of the result depends on the input. If it does not occur free in T, yielding the simple function type syntax $S \to T$.

- The basic types B of STLC are fairly coarse and cannot denotes types of interest that are common in contracts, such as integer subranges. To overcome this limitation, we introduce *executable refinement types* of the form $\{x\!:\!B \mid p\}$. Here, the variable x (of basic type B) can occur within the boolean term p. Informally, this refinement type denotes the set of primitive constants k of type B that satisfy the predicate p. Thus, $\{x\!:\!B \mid p\}$ denotes a refined subtype of B. Occasionally, the predicate p will not contain any free occurrences of x, and we will

[1] We use the notation from Cayenne [Augustsson 1998] $x : S \to T$ in preference to the equivalent syntax $\Pi x\!:\!S.\ T$ for dependent function types. The use of Π indicates that $\Pi x\!:\!S.T$ may be considered to be the type of S-indexed products of types $T[x]$.

write $\{B \mid p\}$. We also write simply B as an abbreviation for the trivial refinement type $\{B \mid \texttt{true}\}$. These refinement types are inspired by prior work on decidable refinement type systems discussed in Section 1.1.2. However, our refinement predicates are arbitrary boolean expressions, so every recursively enumerable subset of the integers is actually a $\lambda_{\mathcal{H}}$ type.

To illustrate the precision of this type language, let us adjust the ty function for $\lambda_{\mathcal{H}}$ to give primitives more precise types, such as

$$\begin{aligned}
\texttt{true} &: \{b{:}\texttt{Bool} \mid b\} \\
\texttt{false} &: \{b{:}\texttt{Bool} \mid \texttt{not}\ b\} \\
\Leftrightarrow &: b_1{:}\texttt{Bool} \to b_2{:}\texttt{Bool} \to \{b{:}\texttt{Bool} \mid b \Leftrightarrow (b_1 \Leftrightarrow b_2)\} \\
\texttt{not} &: b{:}\texttt{Bool} \to \{b'{:}\texttt{Bool} \mid b \Leftrightarrow \texttt{not}\ b\} \\
n &: \{m{:}\texttt{Int} \mid m = n\} \\
+ &: n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
\texttt{if}_T &: \texttt{Bool} \to T \to T \to T \\
\texttt{fix}_T &: (T \to T) \to T
\end{aligned}$$

In particular, we assume that each primitive constant of boolean and integer type is assigned a singleton type that denotes exactly that constant. For example, the type of an integer n denotes the singleton set $\{n\}$. This reflects the fact that a literal constant itself, being present in the code, need not be abstracted to any larger type.

The types for primitive functions may also be quite precise: The type

$$+ : n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\}$$

exactly specifies that this function performs addition. That is, the term $n + m$ has the type $\{z{:}\texttt{Int} \mid z = n + m\}$ denoting the singleton set $\{n + m\}$. Note that even though the type of "+" is defined in terms of "+" itself, this does not cause any problems in our technical development, since the semantics of + do not depend on its type.[2]

The constant $\texttt{fix}_T$ is the fixpoint constructor of type T, and enables the definition of recursive functions. For example, the factorial function can be defined as:

$$
\begin{array}{l}
\texttt{fix}_{\texttt{Int}\to\texttt{Int}} \\
\quad \lambda f{:}(\texttt{Int} \to \texttt{Int}). \\
\qquad \lambda n{:}\texttt{Int}. \\
\qquad\quad \texttt{if}_{\texttt{Int}}\ (n = 0) \\
\qquad\qquad\quad 1 \\
\qquad\qquad\quad (n * (f\ (n - 1)))
\end{array}
$$

The types of $\lambda_{\mathcal{H}}$ can express many other precise specifications, such as the following (where we assume that `Unit`, `Array`, and `RefInt` are additional basic types, and the primitive function $\texttt{sorted} : \texttt{Array} \to \texttt{Bool}$ identifies sorted arrays.)

- $\texttt{printDigit} : \{x{:}\texttt{Int} \mid 0 \leq x \wedge x \leq 9\} \to \texttt{Unit}$.

- $\texttt{swap} : x{:}\texttt{RefInt} \to \{y{:}\texttt{RefInt} \mid x \neq y\} \to \texttt{Bool}$.

- $\texttt{binarySearch} : \{a{:}\texttt{Array} \mid \texttt{sorted}\ a\} \to \texttt{Int} \to \texttt{Bool}$.

3.2 Operational Semantics of $\lambda_{\mathcal{H}}$

We next give formal meaning to $\lambda_{\mathcal{H}}$ terms via small-step operational semantics, as we did for STLC in Section 2.2.2. In fact, redex evaluation for $\lambda_{\mathcal{H}}$ is identical to that

[2]The definition of a primitive in terms of itself is, however, unsatisfying. In the case of functions, we may hope that the type is an abstraction of the value. In Chapter 6 we explore some of the consequences of giving such precise types to functions.

Figure 3.2: Operational Semantics of $\lambda_{\mathcal{H}}$

Redex Evaluation $\boxed{d \curvearrowright e}$

$$(\lambda x{:}S.\, d)\ e \ \curvearrowright\ [x \mapsto e]\, d \qquad [\text{E-}\beta]$$
$$k\ v \ \curvearrowright\ \delta(k, v) \qquad [\text{E-}\delta]$$

Contextual Evaluation $\boxed{d \rightsquigarrow e,\ S \rightsquigarrow T}$

$$\mathcal{C}[d] \ \rightsquigarrow\ \mathcal{C}[e] \quad \text{if } d \curvearrowright e \qquad [\text{E-Compat}]$$
$$\mathcal{D}[d] \ \rightsquigarrow\ \mathcal{D}[e] \quad \text{if } d \curvearrowright e \qquad [\text{E-TyCompat}]$$

Contexts $\boxed{\mathcal{C},\ \mathcal{D}}$

$$\mathcal{C} \ ::= \ \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C} \mid \lambda x{:}\mathcal{D}.\, e$$
$$\mathcal{D} \ ::= \ x{:}\mathcal{D} \to T \mid x{:}S \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\}$$

of STLC. However, $\lambda_{\mathcal{H}}$ types may contain terms, so we extend evaluation to types via additional contexts in a natural way. Figure 3.2 presents the operational semantics in full, for easy reference.

Since types contain terms, evaluation is also defined for types. When a term d reduces to another term e, then the two terms are definitionally equivalent, so the corresponding types $\mathcal{D}[d]$ and $\mathcal{D}[e]$ should also be equivalent. This is proven directly once the appropriate notion of type equivalence is defined. (Lemma 3.2 on page 38)

Figure 3.3: Shape of $\lambda_{\mathcal{H}}$ types and terms

$$
\begin{array}{rcl}
\lfloor \{x{:}B \mid p\} \rfloor & \stackrel{\text{def}}{=} & B \\
\lfloor x{:}S \to T \rfloor & \stackrel{\text{def}}{=} & \lfloor S \rfloor \to \lfloor T \rfloor \\
 & & \\
\lfloor k \rfloor & \stackrel{\text{def}}{=} & k \\
\lfloor x \rfloor & \stackrel{\text{def}}{=} & x \\
\lfloor \lambda x{:}S.\, e \rfloor & \stackrel{\text{def}}{=} & \lambda x{:}\lfloor S \rfloor.\, \lfloor e \rfloor \\
\lfloor f\ e \rfloor & \stackrel{\text{def}}{=} & \lfloor f \rfloor\ \lfloor e \rfloor
\end{array}
$$

3.3 Denotation of $\lambda_{\mathcal{H}}$ types to sets of $STLC$ terms

Intuitively, $\lambda_{\mathcal{H}}$ types describe subsets of the basic types of $STLC$ and functions upon those subsets. To validate this intuition and provide a solid foundation for $\lambda_{\mathcal{H}}$, as well as demonstrate how to do so for future systems, we "bootstrap" the type system by using a denotational interpretation into STLC. This corresponds with the intuition – also supported by the identical redex evaluation rules – that a program with executable refinement types is essentially an STLC program with a more precise specification.

Underlying every $\lambda_{\mathcal{H}}$ type T and term e is an STLC type or term, respectively, which we call its *shape* and write $\lfloor T \rfloor$ or $\lfloor e \rfloor$. Figure 3.3 shows the formal definition of the function $\lfloor - \rfloor : \lambda_{\mathcal{H}} \to STLC$.[3]

We give each type T an interpretation $[\![T]\!]$ (defined by induction on $\lfloor T \rfloor$) that is the set of terms e such that $\lfloor e \rfloor$ is of type $\lfloor T \rfloor$ and e obeys the contractual aspect of

[3]This mapping is similar in spirit to *erasure* which maps a term of the typed λ-calculus to the corresponding term of the untyped λ-calculus, but "shape" is more concise than "refinement erasure".

Figure 3.4: Denotation from $\lambda_{\mathcal{H}}$ types to sets STLC terms

$$[\![\{x{:}B \mid p\}]\!] \stackrel{\text{def}}{=} \{e \mid \vdash_{\text{STLC}} \lfloor e \rfloor : B \ \wedge \ (e \rightsquigarrow^* k \text{ implies } [x \mapsto k]\, p \rightsquigarrow^* \texttt{true})\}$$

$$[\![x{:}S \to T]\!] \stackrel{\text{def}}{=} \{f \mid \vdash_{\text{STLC}} \lfloor f \rfloor : \lfloor S \rfloor \to \lfloor T \rfloor \ \wedge \ \forall e \in [\![S]\!],\ f\ e \in [\![[x \mapsto e]\, T]\!]\}$$

T. Figure 3.4 contains the formal definition.

- A refined basic type $\{x{:}B \mid p\}$ is interpreted as the set of closed terms e of type B for which the predicate p holds, in the sense that whenever $e \rightsquigarrow^* k$ for some constant k, then $[x \mapsto k]\, p \rightsquigarrow^* \texttt{true}$. Note that we cannot insist on the simpler requirement that $[x \mapsto e]\, p \rightsquigarrow^* \texttt{true}$ because that would forbid assigning types to divergent terms, but we intend that any type T is populated by at least the divergent term $\texttt{fix}_T\ (\lambda x : T.\, x)$. In other terminology, our types specify *partial* correctness, not *total* correctness.[4]

- A dependent function type $x : S \to T$ is interpreted as the set of closed terms of simple type $\lfloor S \rfloor \to \lfloor T \rfloor$ that give output in $[\![[x \mapsto e]\, T]\!]$ whenever their input e is in $[\![S]\!]$.

With this denotation defined, we can proceed to define the $\lambda_{\mathcal{H}}$ type system.

[4] An alternate fomulation used by Xu et al. [2009] also included divergent terms in each type but only required for convergent e that $[x \mapsto e]\, p \not\rightsquigarrow^* \texttt{false}$. In other words, the semantics of a predicate included all terms e for which membership in $\{x{:}B \mid p\}$ cannot be effectively refuted. The treatment of the case where $[x \mapsto k]\, p$ diverges is subtle and is concerned primarily with blame for dynamic contracts. See Dimoulas et al. [2011] for a detailed discussion.

3.4 The $\lambda_{\mathcal{H}}$ Type System

Figure 3.5 and Figure 3.9 present the type system of $\lambda_{\mathcal{H}}$ via rules defining the following collection of judgments, where a typing environment Γ maps variables to types.

- The typing judgment $\Gamma \vdash e : T$ states that, assuming the bindings in environment Γ, term e has type T. (Figure 3.5)

- The type well-formedness judgment $\Gamma \vdash T$ states that T is a well formed type under the assumptions in environment Γ. (Figure 3.6)

- The environment well-formedness judgment $\vdash \Gamma$ states that Γ is a well formed environment. (Figure 3.7)

- The subtyping judgment $\Gamma \vdash S <: T$ states that S is a subtype of T under the assumptions in environment Γ. (Figure 3.8)

- The implication judgment $\Gamma \vdash p \Rightarrow q$ states that the predicate p implies q under the assumptions in Γ. (Figure 3.9)

- The closing substitution judgment $\vdash \sigma : \Gamma$ states that the substitution σ is consistent with environment Γ. Defining this appropriately is somewhat subtle and is the subject of Section 3.4.5. (Figure 3.9)

Figure 3.5: Typing Rules for $\lambda_{\mathcal{H}}$

Type rules $\boxed{\Gamma \vdash e : T}$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x \,:\, T}\ [\text{T-Var}] \qquad \frac{}{\Gamma \vdash k \,:\, ty(k)}\ [\text{T-Prim}]$$

$$\frac{\Gamma \vdash S \qquad \Gamma, x : S \vdash e \,:\, T}{\Gamma \vdash (\lambda x{:}S.\, e) \,:\, (x{:}S \to T)}\ [\text{T-Fun}] \qquad \frac{\Gamma \vdash f \,:\, (x{:}S \to T) \qquad \Gamma \vdash e \,:\, S}{\Gamma \vdash f\ e \,:\, [x \mapsto e]\, T}\ [\text{T-App}]$$

$$\frac{\Gamma \vdash e \,:\, S \qquad \Gamma \vdash S <: T \qquad \Gamma \vdash T}{\Gamma \vdash e \,:\, T}\ [\text{T-Sub}]$$

3.4.1 Typing for $\lambda_{\mathcal{H}}$

The inference rules in Figure 3.5 defining the typing judgment

$$\Gamma \vdash e \,:\, T$$

are analogous to those for STLC with the addition of a *subsumption* rule ([T-Sub]) that allows subtyping to be applied at any point in a typing derivation. As will always be the case, we assume that variables are bound at most once in an environment and implicitly utilize α-renaming of bound variables to maintain this assumption and to ensure substitutions are capture-avoiding.

[T-Var]: A variable x is assigned the type it is bound to in the environment Γ.

[T-Prim]: A constant k is given a type by the function ty, as updated for $\lambda_{\mathcal{H}}$ in Section 3.1.

[T-Fun]: For a λ-abstraction $(\lambda x : S.\, e)$, if the domain type S is well-formed and the body e has type T under the additional assumption $x : S$, then the function is given the dependent function type $x : S \to T$. Note again that x may appear free in T.

[T-App]: For an application $(f\ e)$, if the term f in function position has the dependent function type $x : S \to T$, then the resulting type of the application is $[x \mapsto e]\, T$, where free occurences of the formal argument x in T have been replaced by the actual argument e.

[T-Sub]: Whenever a term e has type S and S is a subtype of T the that term also has type T.

3.4.2 Type well-formedness for $\lambda_{\mathcal{H}}$

In STLC any syntactically valid type is well-formed, but in $\lambda_{\mathcal{H}}$ types may contain terms (predicates) that may themselves be ill-typed. The type well-formedness judgment

$$\Gamma \vdash T$$

ensures that each predicate contained anywhere within a type has type `Bool`.

[WT-Arrow]: A function type $x : S \to T$ is well-formed if the domain type is well-formed and the range type is well-formed with the added assumption of $x : S$.

Figure 3.6: Type Well-formedness for $\lambda_{\mathcal{H}}$

Well-formed types $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma \vdash S \qquad \Gamma, x : S \vdash T}{\Gamma \vdash x{:}S \to T}\ [\text{WT-ARROW}] \qquad \frac{\Gamma, x : B \vdash p : \texttt{Bool}}{\Gamma \vdash \{x{:}B \mid p\}}\ [\text{WT-BASE}]$$

[WT-BASE]: A refined basic type $\{x{:}B \mid p\}$ is well-formed if the predicate p is a well-typed boolean term under the additional assumption $x : B$.

3.4.3 Environment well-formedness for $\lambda_{\mathcal{H}}$

As types may be ill-formed, so an arbitrary environment built from the syntactic definition may contain ill-formed types. The judgment

$$\vdash \Gamma$$

ensures that an environment is well-formed by checking that each type is well-formed using only variables bound prior to that types introduction into the environment. The typing rules maintain well-formedness of the environment, but when discussing an arbitrary environment it is necessary to be able to distinguish those that make sense from those that do not.

Figure 3.7: Environment Well-formedness for $\lambda_{\mathcal{H}}$

Well-formed environment $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ [\text{WE-Empty}] \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash T}{\vdash \Gamma, x : T}\ [\text{WE-Ext}]$$

3.4.4 Subtyping for $\lambda_{\mathcal{H}}$

The subtyping judgment

$$\Gamma \vdash S <: T$$

defines when terms of a type S may be considered to have type T. The judgment has no analogue in STLC because STLC does not support subtyping but requires types to match exactly.[5] This judgment may be invoked via occurrences of the typing rule [T-Sub] at any point within a typing derivation.

[S-Arrow]: Subtyping for function types is completely standard, adjusted for dependent types: A dependent function type $x : S_1 \to S_2$ is a subtype of $x : T_1 \to T_2$ if the $T_1 <: S_1$ (contravariant, as always) and the result types are subtypes in a context with x bound to T_1. Note that T_1 must be used in preference to the coarser argument type S_1 because T_2 may not be well-formed in an environment binding x to S_1.

[5] Equivalently, one may consider STLC a system with a degenerate subtyping relation that relates each type only to itself, and thus the subsumption rule is simply elided.

Figure 3.8: Subtyping for $\lambda_{\mathcal{H}}$

Subtyping $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, x : T_1 \vdash S_2 <: T_2}{\Gamma \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)}\ [\text{S-Arrow}]$$

$$\frac{\Gamma, x : B \vdash p \Rightarrow q}{\Gamma \vdash \{x{:}B \mid p\} <: \{x{:}B \mid q\}}\ [\text{S-Base}]$$

[S-Base]: Since a refined basic type $\{x{:}B \mid p\}$ roughly denotes the set of constants k of type B for which $[x \mapsto k]\, p$ is valid, subtyping between refinement types reduces to implication between their refinement predicates. As an example, the rule [S-Base] states that the subtyping judgment

$$\emptyset \vdash \{x{:}\texttt{Int} \mid x > 7\} <: \{x{:}\texttt{Int} \mid x > 0\}$$

follows from the validity of the implication:

$$x : \texttt{Int} \vdash (x > 7) \Rightarrow (x > 0)$$

Basic properties that should hold of any subtyping relationship can now be stated, though their proofs depend on further formalisms from Section 3.5. The first property is that subtyping should be reflexive – the narrative reading of the subtyping judgment $\Gamma \vdash T <: T$ is the tautology "any term of type T has type T". Secondly

subtyping should be transitive, both to respect the intuition that it is an enriched form of containment and also since two applications of the rule [T-Sub] express transitivity in another way.

Lemma 3.1 (Subtyping is a Preorder)

1. *If* $\Gamma \vdash T$ *then* $\Gamma \vdash T <: T$

2. *If* $\Gamma \vdash S <: T$ *and* $\Gamma \vdash T <: U$ *then* $\Gamma \vdash S <: U$

Proof:

1. By induction on the derivation of $\Gamma \vdash T$, since implication is a preorder as well (Lemma 3.3 on page 41).

2. By induction on the derivation of $\Gamma \vdash S <: T$ (followed by inversion on $\Gamma \vdash T <: U$) using Lemma 3.9 for the bindings in function types, and the transitivity of implication (Lemma 3.3 on page 41). □

This establishes that subtyping for $\lambda_{\mathcal{H}}$ obeys the expected properties of any subtyping system. But since terms appear within $\lambda_{\mathcal{H}}$ types, the more novel condition of type invariant under reduction is particular to dependent type systems.

Lemma 3.2 (Type Equivalence under Reduction) *If* $\Gamma \vdash S$ *and* $S \rightsquigarrow T$ *then* $\Gamma \vdash S <: T$ *and* $\Gamma \vdash T <: S$.

Proof: By induction on the derivation of S. In the base case of [WT-Base] invoking the definition of implication. □

Figure 3.9: Implication for $\lambda_{\mathcal{H}}$

Implication $\boxed{\Gamma \vdash p \Rightarrow q}$

$$\frac{\forall \sigma.\ \text{if } \vdash \sigma : \Gamma \text{ and } \sigma(p) \rightsquigarrow^* \texttt{true} \text{ then } \sigma(q) \rightsquigarrow^* \texttt{true}}{\Gamma \vdash p \Rightarrow q}\ [\textsc{Imp}]$$

Closing Substitution $\boxed{\vdash \sigma : \Gamma}$

$$\frac{\forall (x : T) \in \Gamma,\ \sigma(x) \in [\![\sigma(T)]\!]}{\vdash \sigma : \Gamma}\ [\textsc{Subst}]$$

Just as confluence of reduction ensures that reduction produces well-defined equivalence classes of terms, type equivalence under reduction ensures the same for types and that these equivalence classes correspond to the equivalence closure of subtyping.

3.4.5 Implication and Closing Substitutions

Subtyping for $\lambda_{\mathcal{H}}$ is determined by the implication judgment

$$\Gamma \vdash p \Rightarrow q$$

defined by the single rule [Imp] which informally reads "for all closing substitutions consistent with the types in the environment, if the now-closed predicate p holds, then so must q". However, defining closing substitutions in such a way as to yield a consistent implication relation is new to this research, and merits discussion.

One approach taken by Flanagan [2006][6] is to define closing substitutions in terms of the typing judgment, as follows:

$$\frac{\forall x \in dom(\Gamma),\ \ \vdash \sigma(x) \ : \ \Gamma(x)}{\vdash \sigma \ : \ \Gamma}$$

This approach leads to mutual recursion between the typing, subtyping, implication, and closing substitution judgments. Unfortunately, the implication rule [Imp] from Figure 3.5 refers to the closing substitution relation in a contravariant position, so standard monotonicity arguments are not sufficient to show that the resulting collection of mutually recursive inference rules converge to a fixed point, and it is not obvious if there are interesting type systems that satisfy these rules.

An alternative approach to defining implication is to axiomatize its logic [Denney 1998; Ou et al. 2004; Gronski et al. 2006], but the underlying problem remains, in that it is still not obvious if there are interesting implication relations, and hence type systems, that satisfy the axioms.

A third approach, taken by Belo et al. [2011], is to restrict refinements by insisting they be well-typed predicates in an empty environment, so that closing substitutions are not necessary. Unfortunately, refinements then cannot express even simple facts such as that the integer stored in y is greater than the integer stored in x!

Instead, in rule [Subst] we have leveraged the denotational interpretation of $\lambda_{\mathcal{H}}$ types from Section 3.3 to define closing substitutions in a direct manner with no circularity problems: A substitution $\sigma : Var \rightarrow Term$ is a *closing substition* for environment

[6] Corrected by Knowles and Flanagan [2010] to use the technique described here.

Γ if for all bindings $(x : T) \in \Gamma$ we have $\sigma(x) \in [\![\sigma(T)]\!]$.

Lemma 3.3 (Implication is a Preorder)

1. *For any* Γ *and* p *we have* $\Gamma \vdash p \Rightarrow p$.

2. *If* $\Gamma \vdash p_1 \Rightarrow p_2$ *and* $\Gamma \vdash p_2 <: p_3$ *then* $\Gamma \vdash p_1 <: p_2$

PROOF: Both of these statements reduce to tautologies by inverting [IMP].

3.5 Type Soundness for $\lambda_{\mathcal{H}}$

We prove soundness via the usual syntactic "progress" and "preservation" lemmas. However, due to the semantic nature of implication, many of the steps along the way are new, thus routine lemmas usually omitted from a formal development (because practitioners have seen them so many times) are less obvious and are included here. Throughout this section, environments are assumed to be well formed, to elide many uninteresting antecedents.

As a preliminary measure, let us formally state additional expectations on the relationship between a primitives's operational semantics and the type assigned to it, analogous to Requirement 2.1 for STLC primitives.

Requirement 3.4 (Types of $\lambda_{\mathcal{H}}$ Primitives)

For each primitive k*:*

1. *The type of* k *is closed and well-formed,* i.e. $\emptyset \vdash ty(k)$.

2. *If k is a primitive function then it cannot get stuck and its behavior is compatible with its type,* i.e. *if $\emptyset \vdash k\ v : T$ then $\delta(k, v)$ is defined and $\emptyset \vdash \delta(k, v) : T$*

3. *If k is a primitive constant then it is a member of its type, which is a singleton type,* i.e.*if $ty(k) = \{x{:}B \mid p\}$ then $[x \mapsto k]\, p \rightsquigarrow^*$ `true` and $\forall k' \neq k.\ [x \mapsto k']\, p \not\rightsquigarrow^*$ `true`.*

Because we define closing substitutions semantically, many of our proofs connect the semantic relation with our syntactic type system. First, we define the *semantic subtyping* relation $\Gamma \vdash S \subseteq T$ and the *semantic typing* relation $\Gamma \vdash e \in T$ induced by our semantic notion of types, following Frisch et al. [2008]:

$$\Gamma \vdash e \in T \quad \stackrel{\text{def}}{=} \quad \forall \sigma,\ \vdash \sigma : \Gamma \text{ implies } \sigma(e) \in [\![\sigma(T)]\!]$$

$$\Gamma \vdash S \subseteq T \quad \stackrel{\text{def}}{=} \quad \forall \sigma,\ \vdash \sigma : \Gamma \text{ implies } [\![\sigma(S)]\!] \subseteq [\![\sigma(T)]\!]$$

Our formal subtype system is sound with respect to this model:

Lemma 3.5 (Semantic typing soundness)

1. *If $\Gamma \vdash S <: T$ then $\Gamma \vdash S \subseteq T$*

2. *If $\Gamma \vdash e : T$ then $\Gamma \vdash e \in T$*

Proof:

1. Proceed by induction on the derivation of $\Gamma \vdash S <: T$, considering the final rule applied. If S and T are refined basic types, then semantic and formal subtyping have identical definitions. If S and T are function types, then the result follows by induction.

2. Proceed by induction on the derivation of $\Gamma \vdash e : T$ considering the final rule applied. If the final rule is [T-Sub] then part 1 maps subtyping to semantic sutyping. □

In fact, semantic soundness is essentially a form of type soundness that leverages soundness of STLC: Because of type soundness of STLC, a well-typed type is either nonterminating or terminates in a value of the proper shape, and the additional semantic conditions on the denotational definition ensure that the value satisfies its refinements. However, to illustrate other desirable properties of the syntactic typing discipline, we proceed with syntactic proofs.

The semantic substitution properties for these relations mirror the usual syntactic properties, but have a completely different basis. Note, for example, the contravariance of the substitution lemma for closing substitutions. In the following proofs, we use juxtoposition of substitutions to indicate composition or application in order to reduce the number of extraneous symbols; no ambiguity arises.

Lemma 3.6 (Semantic Substitution)

Suppose $\Gamma \vdash d \in S$. *Let* $\theta = [x \mapsto d]$, *and* σ_Γ *represent* σ *limited to the domain of* Γ.

1. *If* $\vdash \sigma : \Gamma, \theta\Gamma'$ *then* $\vdash \sigma_\Gamma \theta \sigma_{\Gamma'} : \Gamma, x : S, \Gamma'$
2. *If* $\Gamma, x : S, \Gamma' \vdash S \subseteq T$ *then* $\Gamma, \theta\Gamma' \vdash \theta S \subseteq \theta T$
3. *If* $\Gamma, x : S, \Gamma' \vdash e \in T$ *then* $\Gamma, \theta\Gamma' \vdash \theta e \in \theta T$

Proof:

1. By definition of $\Gamma \vdash d \in S$ we know $\sigma_\Gamma(d) \in [\![\sigma_\Gamma(S)]\!]$

2 and 3. For each lemma, fix a closing substitution σ such that $\vdash \sigma : \Gamma, \theta\Gamma'$. Then $\vdash \sigma_\Gamma\theta\sigma_{\Gamma'} : \Gamma, x : S, \Gamma'$ by part 1. Since $\sigma S = \sigma_\Gamma\theta\sigma_{\Gamma'}S$ and $\sigma T = \sigma_\Gamma\theta\sigma_{\Gamma'}T$ and $\sigma e = \sigma_\Gamma\theta\sigma_{\Gamma'}e$ each conclusion follows by set-theoretic calculation.

Corollary 3.7 *All the conclusions of Lemma 3.6 hold if* $\Gamma \vdash d : S$.

A key notion of health for any inference system is that additional assumptions should never lessen the conclusions that may be drawn. The act of adding additional assumptions to the environment is known as *weakening*.

Lemma 3.8 (Weakening)

Let $\Gamma = \Gamma_1, \Gamma_2$ *and* $\Gamma' = \Gamma_1, x : U, \Gamma_2$.

1. *If* $\Gamma \vdash p : \texttt{Bool}$ *and* $\Gamma \vdash q : \texttt{Bool}$ *and* $\Gamma \vdash p \Rightarrow q$ *then* $\Gamma' \vdash p \Rightarrow q$
2. *If* $\Gamma \vdash T$ *and* $\Gamma \vdash S$ *and* $\Gamma \vdash S <: T$ *then* $\Gamma' \vdash S <: T$.
3. *If* $\Gamma \vdash e : T$ *then* $\Gamma' \vdash e : T$.
4. *If* $\Gamma \vdash T$ *then* $\Gamma' \vdash T$.

Proof:

1. The quantification over closed terms after weakening is the same as the quantification before weakening, as the new binding is irrelevant.

2. By induction on the derivation of $\Gamma \vdash S <: T$, using part 1 in the case for rule [S-Base].

3 and 4. By mutual induction on the derivations of $\Gamma \vdash e : T$ and $\Gamma \vdash T$, using part 2 in the case of rule [T-Sub] $\square$

Likewise, shrinking the domain of a quantification should only increase the available conclusions, as there are fewer potential counterexamples to any conclusion. For a subtyping system, this operation of *narrowing* indicates that subtyping behaves according to this intuition.

Lemma 3.9 (Narrowing) *Suppose* $\Gamma \vdash S <: T$. *Then*

1. *If* $\Gamma, x : T, \Gamma' \vdash p \Rightarrow q$ *then* $\Gamma, x : S, \Gamma' \vdash p \Rightarrow q$.
2. *If* $\Gamma, x : T, \Gamma' \vdash U_1 <: U_2$ *then* $\Gamma, x : S, \Gamma' \vdash U_1 <: U_2$.
3. *If* $\Gamma, x : T, \Gamma' \vdash e : U$ *then* $\Gamma, x : S, \Gamma' \vdash e : U$.

Proof:

1. By Lemma 3.5 (Soundness of Semantic Typing) we know $\Gamma \vdash S \subseteq T$ so the quantification over closed terms in the conclusion is contained in that of the premise. $\square$
2. By induction over the derivation of $\Gamma, x : T, \Gamma' \vdash U_1 <: U_1$ using part 1.
3. By induction over the derivation of $\Gamma, x : T, \Gamma' \vdash e : U$, using part 2 as needed and applying subsumption immediately when x is pulled from the environment. $\square$

Via Lemma 3.8 and Lemma 3.9 the reflexivity and transitivity of subtyping may be established. The reader interested in formalism may now wish to revisit Section 3.4.4 and review the these properties and their proofs.

The central lemma for proving preservation is the so-called *substitution lemma*, which states that a variable of a certain type may be soundly replaced by any term of the same type. Each of the judgments of the $\lambda_{\mathcal{H}}$ enjoys a substitution property.

Lemma 3.10 (Substitution) *Suppose* $\Gamma \vdash e : S$,

1. *If* $\Gamma, x : S, \Gamma' \vdash p \Rightarrow q$ *then* $\Gamma, [x \mapsto e]\,\Gamma' \vdash [x \mapsto e]\,p \Rightarrow [x \mapsto e]\,q$

2. *If* $\Gamma, x : S, \Gamma' \vdash T <: U$ *then* $\Gamma, [x \mapsto e]\,\Gamma' \vdash [x \mapsto e]\,T <: [x \mapsto e]\,U$

3. *If* $\Gamma, x : S, \Gamma' \vdash d : T$ *then* $\Gamma, [x \mapsto e]\,\Gamma' \vdash [x \mapsto e]\,d : [x \mapsto e]\,T$

4. *If* $\Gamma, x : S, \Gamma' \vdash T$ *then* $\Gamma, [x \mapsto e]\,\Gamma' \vdash [x \mapsto e]\,T$

PROOF:

1. Immediately follows from Lemma 3.6 (Semantic Substitution).

2. By induction on the derivation of $\Gamma, x : S, \Gamma' \vdash T <: U$. In the case of [S-BASE] invoking part 1.

3. and 4. By mutual induction on the derivations of $\Gamma, x : S, \Gamma' \vdash d : T$ and $\Gamma, x : S, \Gamma' \vdash T$, invoking part 2 in the case of [T-SUB].□

Leveraging these lemmas, progress and preservation are routine and ensure type soundness for $\lambda_{\mathcal{H}}$ in the usual way.

Lemma 3.11 (Preservation) *If* $\vdash \Gamma$ *and* $\Gamma \vdash d : T$ *and* $d \rightsquigarrow e$ *then* $\Gamma \vdash e : T$

PROOF: By induction on the typing derivation $\Gamma \vdash d : T$, invoking Lemma 3.10 when evaluation proceeds via [E-β] and Requirement 3.4 when evaluation proceeds via [E-δ]. $\square$

Lemma 3.12 (Progress) *If $\emptyset \vdash e : T$ then either e is a value, or there exists some e' such that $e \rightsquigarrow e'$.*

PROOF: Proceed by induction on the derivation of $\vdash e : T$, considering the final rule applied. The entire proof is standard and routine $\square$

Theorem 3.13 (Type Soundness for $\lambda_{\mathcal{H}}$) *For any term e, if $\emptyset \vdash e : T$ then either e is nonterminating or e evaluates to some value v such that $\emptyset \vdash v : T$*

PROOF: This is a corollary of progress and preservation.

3.6 Extensional Equivalence for $\lambda_{\mathcal{H}}$

Type soundness assures that the $\lambda_{\mathcal{H}}$ type system relates to the semantics in the intended manner, but is not particularly useful for reasoning about meaning of terms. The most fundamental relationship between terms is that of equality, and the defining trait of equality is substitutability: If a term e is equal to d, then replacing e with d never affects any conclusion. Relative to the semantics of a calculus, the authoritative definition of equivalence between terms is *contextual equivalence* (sometimes called *observable equivalence*) which states that no program context can distinguish the two terms.

Definition 3.14 (Contextual Equivalence)

Two terms e_1 *and* e_2 *are* contextually equivalent*, written* $e_1 \equiv e_2$*, if and only if:*

Given any context $\mathcal{C}$ *such that* $\emptyset \vdash \mathcal{C}[e_1] : B$ *and* $\emptyset \vdash \mathcal{C}[e_2] : B$*,*

$$\mathcal{C}[e_1] \rightsquigarrow^* k \;\Leftrightarrow\; \mathcal{C}[e_2] \rightsquigarrow^* k$$

Reasoning about contextual equivalence between terms is notoriously difficult. The equivalence closure of reduction is much weaker than contextual equivalence, so establishing equality between terms that are not obviously syntactically related often involves complex syntactic techniques such as bisimulation (of which many varities exist) which are extremely sensitive to the form (not content) of a calculus.

In contrast, when reasoning about mathematical functions, extensional equivalence – that equal functions map equal arguments to equal results – is the definition of equality. For a computational calculus, this sort of equality is known as a *logical relation* [Statman 1985]. That such a form of reasoning is applicable confirms that what we call a "function" in a calculus does, indeed, define a function relative to contextual equivalence.

The logical relation we establish is extensional equivalence of terms of a particular type under reduction, written $\Gamma \vdash e_1 \sim e_2 : T$ and defined in Figure 3.10. Two terms e_1 and e_2 of basic type are (extensionally) equivalent if, for any closing substitution σ, whenever $\sigma(e_1)$ evaluates to a constant k so does $\sigma(e_2)$, and vice versa. Two functions f_1 and f_2 of type $x : S \to T$ are equivalent if they yield equivalent output when given equivalent arguments e_1 and e_2.

Figure 3.10: Extensional Equivalence Under Reduction for $\lambda_{\mathcal{H}}$

$\boxed{\Gamma \vdash e_1 \sim e_2 : T}$ (defined by induction on $\lfloor T \rfloor$)

$$\Gamma \vdash e_1 \sim e_2 : \{x{:}B \mid p\} \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \Gamma \vdash e_1 \in \{x{:}B \mid p\}$$
$$\Gamma \vdash e_2 \in \{x{:}B \mid p\}$$
$$\forall \sigma \text{ such that } \vdash \sigma : \Gamma,\ (\sigma(e_1) \rightsquigarrow^* k) \Leftrightarrow (\sigma(e_2) \rightsquigarrow^* k)$$

$$\Gamma \vdash f_1 \sim f_2 : (x{:}S \to T) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \forall e_1, e_2 \text{ such that } \Gamma \vdash e_1 \sim e_2 : S,$$
$$\Gamma \vdash f_1\ e_1 \sim f_2\ e_2 : [x \mapsto e_1]\,T \curlywedge [x \mapsto e_2]\,T$$

$\boxed{S \curlywedge T}$

$$\{x{:}B \mid p\} \curlywedge \{x{:}B \mid q\} \quad \stackrel{\text{def}}{=} \quad \{x{:}B \mid p \wedge q\}$$

$$(x{:}S_1 \to S_2) \curlywedge (x{:}T_1 \to T_2) \quad \stackrel{\text{def}}{=} \quad x{:}(S_1 \curlywedge T_1) \to (S_2 \curlywedge T_2)$$

A question that immediately arises is whether these two applications $f_1\ e_1$ and $f_2\ e_2$ should have type $[x \mapsto e_1]\,T$ or $[x \mapsto e_2]\,T$. In some sense it does not matter, since e_1 and e_2 are extensionally equivalent by assumption. But this assumes the premise: that $\lambda_{\mathcal{H}}$ operational semantics respects extensional equivalence. So, since this is not yet demonstrated, the definition requires that $f_1\ e_1$ and $f_2\ e_2$ are extensionally equivalent at type

$$[x \mapsto e_1]\,T \curlywedge [x \mapsto e_2]\,T$$

where we combine both types via the *wedge product* ($\curlywedge$), also defined in Figure 3.10. For basic types, wedge product is type intersection, *i.e.*, conjunction of refinement predicates. For function types, the wedge product is covariant in both the function domain and the range. This covariance is an illusion: we we only ever combine equivalent types with the wedge product, so co- or contravariance is a formality: If the wedge product were made more "intuitively" contravariant the [T-App] case of Theorem 3.19 would not be possible to prove as stated. Since extensional equivalence is uncommon in type systems research exposition, we present the following proofs in great detail.

To work fluidly with this wedge product, we note that it is a commutative and associative operator with respect to the equivalence closure of subtyping, and it distributes through subtyping.

Lemma 3.15 *Assuming the underlying shapes of all mentioned types are equal, and each type is well-formed in the environment,*

1. $\Gamma \vdash S \curlywedge T <: T \curlywedge S$

2. $\Gamma \vdash S \curlywedge (T \curlywedge U) <: (S \curlywedge T) \curlywedge U$

3. If $\Gamma \vdash S_1 <: T_1$ *and* $\Gamma \vdash S_2 <: T_2$ *then* $\Gamma \vdash S_1 \curlywedge S_2 <: T_1 \curlywedge T_2$

PROOF: By induction on the underlying shape of the types. □

We require that primitive functions do not violate extensionality, and so each primitive function k is extensionally equivalent to itself.

Requirement 3.16

For any primitive k*,*

$$\emptyset \vdash k \sim k : ty(k)$$

Equivalence is preserved under subtyping, allowing free manipulation of $\curlywedge$ in the type assigned to an equivalence.

Lemma 3.17 (Extensional equivalence under subtyping)

If $\Gamma \vdash e_1 \sim e_2 : S$ *and* $\Gamma \vdash S <: T$ *then* $\Gamma \vdash e_1 \sim e_2 : T$

PROOF: By induction on $\lfloor S \rfloor$ (which equals $\lfloor T \rfloor$), with proof cases elaborated by inversion of subtyping rules.

Case $\mathrm{S} = \{x{:}B \mid p\}$: Then we have a derivation

$$\frac{\Gamma \vdash p \Rightarrow q}{\Gamma \vdash \{x{:}B \mid p\} <: \{x{:}B \mid q\}}\ \text{[S-BASE]}$$

By definition of $\Gamma \vdash e_1 \sim e_2 : \{x{:}B \mid p\}$, both e_1 and e_2 are semantically members of $\{x{:}B \mid p\}$ hence also semantic members of $\{x{:}B \mid q\}$, which yields $\Gamma \vdash e_1 \sim e_2 : \{x{:}B \mid q\}$.

Case $\mathrm{S} = x{:}S_1 \to S_2$: Then we have a derivation

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, x : T_1 \vdash S_2 <: T_2}{\Gamma \vdash x{:}S_1 \to S_2 <: x{:}T_1 \to T_2}\ [\text{S-Arrow}]$$

Fix d_1 and d_2 such that $\Gamma \vdash d_1 \sim d_2 : T_1$. By induction, $\Gamma \vdash d_1 \sim d_2 : S_1$.

By definition of e_1 and e_2 being extensionally equivalent,

$$\Gamma \vdash e_1\ d_1 \sim e_2\ d_2 : [x \mapsto d_1]\, S_2 \curlywedge [x \mapsto d_2]\, S_2$$

By Lemma 3.10 (Substitution),

$$\Gamma \vdash [x \mapsto d_1]\, S_2 <: [x \mapsto d_1]\, T_2 \quad \text{and} \quad \Gamma \vdash [x \mapsto d_2]\, S_2 <: [x \mapsto d_2]\, T_2$$

By Lemma 3.15(3) the above two subtyping relationship combine with the wedge product to yield

$$\Gamma \vdash [x \mapsto d_1]\, S_2 \curlywedge [x \mapsto d_2]\, S_2 <: [x \mapsto d_1]\, T_2 \curlywedge [x \mapsto d_2]\, T_2$$

By applying induction to the above unfolding of the definition of extensional equivalence,

$$\Gamma \vdash e_1\ d_1 \sim e_2\ d_2 : [x \mapsto d_1]\, T_2 \curlywedge [x \mapsto d_2]\, T_2$$

Hence $\Gamma \vdash e_1 \sim e_2 : (x{:}T_1 \to T_2)$. □

Next, we show that extensional equivalence of reducts implies extensional equivalence of the original terms.

Lemma 3.18 (Extensional equivalence under reduction)

If $\Gamma \vdash e_1' \sim e_2' : U$ *and* $e_1 \rightsquigarrow^* e_1'$ *and* $e_2 \rightsquigarrow^* e_2'$ *then* $\Gamma \vdash e_1 \sim e_2 : U$

Proof: By induction on $\lfloor U \rfloor$.

Case $\{x{:}B \mid p\}$: If $e_1' \rightsquigarrow^* k$ then $e_1 \rightsquigarrow^* e_1' \rightsquigarrow^* k$; likewise for e_2 and e_2'.

Case $x{:}S \to T$: Fix d_1, d_2 such that

$$\Gamma \vdash d_1 \sim d_2 : S.$$

By definition

$$\Gamma \vdash e_1'\ d_1 \sim e_2'\ d_2 : [x \mapsto d_1]\,T \curlywedge [x \mapsto d_2]\,T$$

By applying an evaluation context,

$$e_1\ d_1 \rightsquigarrow^* e_1'\ d_1 \quad \text{and} \quad e_2\ d_2 \rightsquigarrow^* e_2'\ d_2$$

By induction,

$$\Gamma \vdash e_1\ d_1 \sim e_2\ d_2 : [x \mapsto d_1]\,T \curlywedge [x \mapsto d_2]\,T$$

Hence, $\Gamma \vdash e_1 \sim e_2 : (x{:}S \to T)$. $\square$

To prove the fundamental soundness theorem for extensional equivalence, we extend the notion of extensional equivalence to closing substitutions.

$$\vdash \sigma \sim \gamma : \Gamma \quad \stackrel{\text{def}}{=} \quad \forall x \in dom(\Gamma),\ \emptyset \vdash \sigma(x) \sim \gamma(x) : (\sigma \curlywedge \gamma)(\Gamma(x))$$

For convenience, we overload the wedge product operator for closing substitutions, so $(\sigma \curlywedge \gamma)$ maps a type T to the wedge product of all possible choices of which variables to use from σ and which from γ. Note that we are only interested in closing substitutions for the same environment, so they always have the same domain.

$$(\sigma \curlywedge \gamma)(T) \quad \stackrel{\text{def}}{=} \quad \bigwedge_{A \in \mathcal{P}(dom(\sigma))} (\sigma|_A \circ \gamma|_{dom(\sigma)\setminus A})(T)$$

Lemma 3.19 (Extensional equivalence under substitution)

If $\Gamma \vdash e : T$ *then for any closing substitutions* σ *and* γ *such that* $\vdash \sigma \sim \gamma : \Gamma$*, we have*

$$\emptyset \vdash \sigma(e) \sim \gamma(e) : (\sigma \curlywedge \gamma)(T)$$

PROOF: By induction on the height of the derivation of $\Gamma \vdash e : T$, rebinding metavariables for each case.

Case [T-VAR]: Then $(x : T) \in \Gamma$, so by definition of equivalent substitutions $\emptyset \vdash \sigma(x) \sim \gamma(x) : (\sigma \curlywedge \gamma)(T)$

Case [T-PRIM]: Given by Requirement 3.16.

Case [T-APP]: Then the derivation concludes with

$$\frac{\Gamma \vdash f : x{:}S \to T \qquad \Gamma \vdash e : S}{\Gamma \vdash f\ e : [x \mapsto e]\, T}$$

Fix σ and γ. By induction,

$$\emptyset \vdash \sigma(f) \sim \gamma(f) : (\sigma \curlywedge \gamma)(x{:}S \to T)$$

$$\emptyset \vdash \sigma(e) \sim \gamma(e) : (\sigma \curlywedge \gamma)(S)$$

Applying the definition of extensional equivalence,

$$\begin{aligned} \emptyset \vdash \sigma(f)\ \sigma(e) \sim \gamma(f)\ \gamma(e) \quad &: \quad [x \mapsto \sigma(e)]\, (\sigma \curlywedge \gamma)(T) \\ &\curlywedge \quad [x \mapsto \gamma(e)]\, (\sigma \curlywedge \gamma)(T) \end{aligned}$$

which simplifies to the conclusion for this case:

$$\emptyset \vdash \sigma(f\ e) \sim \gamma(f\ e) : (\sigma \curlywedge \gamma)([x \mapsto e]\, T)$$

Case [T-Lam]: The the derivation concludes with

$$\frac{\Gamma, x : S \vdash e \ : \ T}{\Gamma \vdash \lambda x{:}S.\, e \ : \ x{:}S \to T}$$

Fix σ and γ, then fix d_1 and d_2 such that $\emptyset \vdash d_1 \ \sim \ d_2 \ : (\sigma \curlywedge \gamma)(S)$. The definition of extensional equivalence of substitution yields

$$\vdash (\sigma \circ [x \mapsto d_1]) \ \sim \ (\gamma \circ [x \mapsto d_2]) : (\Gamma, x : S)$$

By induction,

$$\emptyset \vdash (\sigma \circ [x \mapsto d_1]) \ e \ \sim \ (\gamma \circ [x \mapsto d_2]) \ e : ((\sigma \circ [x \mapsto d_1]) \curlywedge (\gamma \circ [x \mapsto d_2]))(T)$$

By Lemma 3.18, we perform β-expansion and rearrange some substitutions to yield

$$\begin{aligned} \emptyset \vdash \sigma(\lambda x{:}S.\, e) \ d_1 \ \sim \ \gamma(\lambda x{:}S.\, e) \ d_2 \quad &: \quad [x \mapsto d_1]\,(\sigma \curlywedge \gamma)(T) \\ &\curlywedge \quad [x \mapsto d_2]\,(\sigma \curlywedge \gamma)(T) \end{aligned}$$

which simplifies to the conclusion of this case,

$$\emptyset \vdash \sigma(\lambda x{:}S.\, e) \ \sim \ \gamma(\lambda x{:}S.\, e) : (\sigma \curlywedge \gamma)(x{:}S \to T)$$

Case [T-Sub]: The derivation concludes with

$$\frac{\Gamma \vdash e \ : \ S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e \ : \ T}$$

Fix σ and γ. By induction,

$$\emptyset \vdash \sigma(e) \ \sim \ \gamma(e) : (\sigma \curlywedge \gamma)(S)$$

Applying the substitution lemma and distributing $\curlywedge$ through the subtyping,

$$\emptyset \vdash (\sigma \curlywedge \gamma)(S) <: (\sigma \curlywedge \gamma)(T)$$

Thus by Lemma 3.17 we conclude with

$$\emptyset \vdash \sigma(e) \sim \gamma(e) : (\sigma \curlywedge \gamma)(T)$$

This concludes the case-by-case analysis. □

As a corollary, we have contextual equivalence of related terms.

Theorem 3.20 (Soundness of extensional equivalence) *If* $\Gamma \vdash d \sim e : T$ *then* $d \equiv e$.

PROOF: From the definition of $d \equiv e$, fix any $\mathcal{C}$ such that $\emptyset \vdash \mathcal{C}[d] : B$ and $\emptyset \vdash \mathcal{C}[e] : B$. Apply Lemma 3.19 to the typing $x : T \vdash \mathcal{C}[x] : B$ with the related substitutions $\vdash [x \mapsto d] \sim [x \mapsto e] : (x : T)$ to yield $\emptyset \vdash \mathcal{C}[d] \sim \mathcal{C}[e] : B$, which is exactly the definition of $d \equiv e$.

3.7 Related Work

Research on refinement type systems discussed in Section 1.1.2 has influenced our choice of how to express program invariants, with the major difference that none of the referenced systems include arbitrary executable refinement predicates.

Liquid types [Rondon et al. 2008] also refines basic types with predicates, but the research agenda is more aligned with the question "how can we develop usable and effective static verification of modern higher-order programs?" rather than ours of "how shall we proceed into the realm of undecidable type systems?". Thus the metatheory of Liquid Types is a model of an implementation for OCaml (subsequently for C [Rondon et al. 2012], Javascript [Chugh et al. 2012a], and Haskell [Vazou et al. 2013]) rather than

a foundational calculus, resulting in different choices for formalization: The meaning of validity is given via a direct mapping to the The refinement predicates of Liquid Types, while written in the programming language at hand, are given meaning by a mapping to the logic of Satisfiability Modulo Theories (SMT). Evaluation is that of the host language (call-by-value for most implementations, and call-by-need for Haskell) rather than a general theory of β-equivalence, thus proof techniques for program equivalence are largely out of scope.

ESC/Haskell [Xu 2006; Xu et al. 2009] features a similar language for expressing specifications, but is concerned with the practicalities of statically checking contracts for Haskell (hence the name analogous to its predecessor ESC/Java) and takes a different formal approach not primarily concerned with providing a foundational calculus. A program meeting its specification is *defined* by symbolic execution, versus our presentation as a type system later demonstrated sound by relating it to the semantics of a program. However, ESC/Haskell gives a much more thorough treatment to the notion of blame, which is integral to a complete contract system.

Manifest contracts [Greenberg et al. 2012] is a term synonymous with executable refinement types, chosen to emphasize that predicates from contracts are made "manifest" in the type system, versus being only "latent" properties of the program as it executes. The work shares with this chapter a foundation built upon from Knowles and Flanagan [2010], hence uses a nearly identical approach to defining subtyping between contracts. Subsequent to this research, Belo et al. [2011] extended manifest to include polymorphism and apply refinements to any type, not just basic types. However as

a foundation it is somewhat lacking due to the restriction that refinements may only refer to the value being refined and may not have other free variables to relate different portions of a program. It also does not include subtyping, but simply ensures types are compatible and then defers contract enforcement to runtime, much like gradual typing. Subtyping between function types is the principle remaining challenge for allowing refinements of arbitrary types.

None of the above systems provide a robust method for reasoning about extensional equivalence of terms.

Chapter 4

Hybrid Type Checking

With $\lambda_{\mathcal{H}}$ as a sound foundation for the study of executable refinement type systems, this chapter presents *hybrid type checking*, a new technique for combining static type checking with dynamic contract checking.

To illustrate the key idea of hybrid type checking in general, before developing a system specialized to $\lambda_{\mathcal{H}}$, consider again this possible type rule for function application:

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash f\ e : U}$$

The situation we focus on is when the antecedent $\Gamma \vdash S <: T$ can be neither definitively proven nor refuted.

- *Statically rejecting* such programs would cause the compiler to reject some programs that, on deeper analysis, could be shown to be well-typed. Flanagan et al. [2002] illustrates that this approach is brittle in practice because there are many

such programs and it is difficult to predict which programs the compiler would accept.

- *Statically accepting* such programs (based on the optimistic assumption that the unproven subtype relations actually hold) may result in specifications being violated at run time.

Instead, hybrid type checking is the general approach of accepting such programs on a provisional basis, but inserting sufficient dynamic checks to ensure that specification violations never occur at run time. Of course, checking that $\Gamma \vdash S <: T$ at run time is still a difficult problem and would violate the principle of *phase distinction* [Cardelli 1988]. Instead, our hybrid type checking approach transforms the above application into the code

$$f\ (\langle T \triangleleft S \rangle\ e)$$

where the additional *cast* (or *coercion* [Breazu-Tannen et al. 1991]) $\langle T \triangleleft S \rangle$ dynamically enforces that e adheres to T.

Figure 4.1 illustrates the behavior of hybrid type checking on various kinds of programs. Every program is either ill-typed or well-typed, but it is not always possible to make this classification statically. However, a particular algorithm may still identify some (hopefully many) clearly ill-typed programs, which are rejected, and similarly can identify some clearly well-typed programs, which are accepted unchanged.

For the remaining programs, dynamic type casts are inserted to check any unverified correctness properties at run time. We refer to this category of programs as

Figure 4.1: Behavior of hybrid type checking for various categories of programs

<table>
<tr><th colspan="2">Ill-typed programs</th><th colspan="2">Well-typed programs</th></tr>
<tr><td>Clearly ill-typed</td><td colspan="2">Subtle programs</td><td>Clearly well-typed</td></tr>
<tr><td>Rejected by type checker</td><td colspan="2">Accepted
with casts</td><td>Accepted without casts</td></tr>
<tr><td></td><td>Casts
may
fail</td><td>Casts
never
fail</td><td></td></tr>
</table>

subtle.

Definition 4.1 *A program is called* subtle *relative to a hybrid type checker if the checker cannot statically prove or refute its well-typedness.*

If the original program is actually well-typed, these casts are redundant and will never fail. Conversely, if the original program is ill-typed in a manner that is not demonstrable at compile time, the inserted casts may fail. As static analysis technology improves, the category of subtle programs in Figure 4.1 will shrink, as more ill-typed programs are rejected and more well-typed programs are fully verified at compile time.

Hybrid type checking may facilitate the evolution and adoption of advanced static analyses, by allowing software engineers to experiment with sophisticated specification strategies that cannot (yet) be verified statically or efficiently. Such experiments can then motivate and direct static analysis research.

For example, if a hybrid type checker fails to verify or refute a subtyping query, it could send that query back to the compiler writer. Similarly, if a cast $\langle T \triangleleft S \rangle\ v$ fails, the value v is a witness that refutes an undecided subtyping query $S <: T$, and such witnesses could also be sent back to the compiler writer. This information would provide concrete and quantifiable motivation for subsequent improvements in the type checker's analysis. In Chapter 7 we explore the design of such an implementation.

Just as different compilers (or different configurations of a single compiler) for the same language may yield object code of varying quality, we might imagine a variety of hybrid type checkers with different trade-offs between static and dynamic checks. Fast interactive hybrid compilers might perform only limited static analysis to detect obvious type errors, while production compilers could perform deeper analyses to detect more defects statically and to generate improved code with fewer dynamic checks.

Hybrid type checking is inspired by prior work on soft typing [Fagan 1991; Wright and Cartwright 1994; Aiken et al. 1994; Flanagan et al. 1996], but it extends soft typing by rejecting many ill-typed programs, in the spirit of static type checkers. The interaction between static typing and dynamic checks has also been studied in the context of type systems with the type `Dynamic` [Thatte 1990; Abadi et al. 1991], and in systems that combine dynamic checks with dependent types [Ou et al. 2004]. Hybrid type checking extends these ideas to support more precise specifications.

The general approach of hybrid type checking appears applicable to a variety of programming languages and specification languages. This chapter develops the design and correctness criteria for such a system using $\lambda_{\mathcal{H}}$ as a representative core calculus.

The combination of $\lambda_{\mathcal{H}}$ and HTC enjoys the following benefits:

1. It supports precise interface specifications, which facilitate modular development of reliable software.

2. As many defects as possible and practical are detected at compile time (and we expect this set will increase as static analysis technology evolves).

3. All well-typed programs are accepted by the checker, and their semantics are unchanged.

4. Due to decidability limitations, the hybrid type checker may statically accept some *subtly ill-typed* programs, but it will insert sufficient dynamic casts to guarantee that specification violations are always detected, either statically or dynamically.

5. The output of the hybrid type checker is always a well-typed program (and so, for example, type-directed optimizations are applicable).

6. If the source program is well-typed, then the inserted casts are guaranteed to succeed, and so the source and output programs are behaviorally equivalent.

7. If the subtyping algorithm can decide all reflexive queries, then cast insertion is idempotent: All necessary casts are inserted in a single pass.

Chapter Outline

The remainder of this chapter is organized as follows

- Section 4.1 introduces the extensions to $\lambda_{\mathcal{H}}$ to support hybrid type checking.

- Section 4.2 describes the operational semantics of dynamic casts.

- Section 4.3 presents the new typing rules for casts.

- Section 4.4 formally defines hybrid type checking.

- Section 4.5 walks through a sizable example in depth.

- Section 4.4.4 introduces the correctness properties of HTC beyond those of traditional type systems, and proves them for our context.

- Section 4.7 surveys work most closely related to HTC.

4.1 The Language $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$

In order to perform hybrid type checking augment $\lambda_{\mathcal{H}}$ with terms for casts, yielding $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$. The extension includes two new forms, one for casts and one for refinement checks in progress:

$$e \ ::= \ \cdots \ \mid \ \langle T \triangleleft S \rangle \ \mid \ \langle \{x\!:\!B \mid p\}, q, k \rangle$$

For easy reference, Figure 4.2 includes the full syntax of the augmented language with new forms highlighted.

- The term $\langle T \triangleleft S \rangle$ denotes a cast from S to T as described in the introduction to this chapter.

- The term $\langle \{x\!:\!B \mid p\}, q, k \rangle$ denotes a cast-in-progress where a determination of whether k has type $\{x\!:\!B \mid p\}$ is underway. If q eventually evaluates to `true`, then

Figure 4.2: Syntax for $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$

$e, d, f, g, q, p ::=$	*Terms:*
$\quad k$	constant
$\quad x$	variable
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad f\ e$	application
$\quad \langle T \triangleleft S \rangle$	***type cast***
$\quad \langle \{x\!:\!B \mid p\}, q, k \rangle$	***cast in progress***
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	constant
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad \langle T \triangleleft S \rangle$	***type cast***
$S, T, U ::=$	*Types:*
$\quad x\!:\!S \to T$	dependent function type
$\quad \{x\!:\!B \mid p\}$	refined basic type

Figure 4.3: Operation Semantics of $\lambda_{\mathcal{H}}^{\text{HTC}}$

Redex Evaluation $\boxed{d \curvearrowright e}$

$$(\lambda x{:}S.\, e)\ d \curvearrowright [x \mapsto d]\, e \qquad \text{[E-}\beta\text{]}$$

$$k\ v \curvearrowright \delta(k, v) \qquad \text{[E-}\delta\text{]}$$

$$\langle x{:}T_1 \to T_2 \lhd x{:}S_1 \to S_2 \rangle\ f \curvearrowright \lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \rangle\ (f\ (\langle S_1 \lhd T_1 \rangle\ x)) \qquad \textbf{[E-Cast-F]}$$

$$\langle \{x{:}B \mid p\} \lhd \{x{:}B \mid q\} \rangle\ k \curvearrowright \langle \{x{:}B \mid p\}, [x \mapsto k]\, p, k \rangle \qquad \textbf{[E-Cast-Begin]}$$

$$\langle \{x{:}B \mid q\}, \texttt{true}, k \rangle \curvearrowright k \qquad \textbf{[E-Cast-End]}$$

Contexts $\boxed{\mathcal{C},\ \mathcal{D}}$

$$\mathcal{C} ::= \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\mathcal{C} \mid \lambda x{:}\mathcal{D}.\, e \mid \langle T, \mathcal{C}, k \rangle \mid \langle \mathcal{D}, e, k \rangle \mid \langle T \lhd \mathcal{D} \rangle \mid \langle \mathcal{D} \lhd S \rangle$$

$$\mathcal{D} ::= x{:}\mathcal{D} \to T \mid x{:}S \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\}$$

the cast will succeed.

4.2 Operational Semantics of $\lambda_{\mathcal{H}}^{\text{HTC}}$

Figure 4.3 recapitulates the redex reduction rules and evaluation contexts for $\lambda_{\mathcal{H}}^{\text{HTC}}$ with additions to include casts used in hybrid type checking. The new evaluation rules governing the meaning of casts are:

[E-Cast-F] Casting a function f of type $x : S_1 \to S_2$ to the type $x : T_1 \to T_2$ yields a new function

$$\lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \rangle\ (f\ (\langle S_1 \lhd T_1 \rangle\ x))$$

Though large, this term of type $x : T_1 \to T_2$ is actually simple: Consider applying it to an argument e of type T_1, and β-reduce to the following:

$$\langle [x \mapsto e]\, T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ e]\, S_2 \rangle\ \ (f\ \ (\langle S_1 \lhd T_1 \rangle\ e))$$

First the argument e is cast to type S_1, then it is passed to the original function f, yielding a result of type $[x \mapsto \langle S_1 \lhd T_1 \rangle\ e]\, S_2$; call it e'. The cast inserted into the return type is necessary as e may not actually have type S_1, so substituting it directly would make the resulting type ill-formed and could cause actual errors not caught by run-time checks. Then the result e' is cast to type $[x \mapsto e]\, T_2$ as required. Thus, higher-order casts are performed a lazy fashion – the new casts $\langle T_2 \lhd S_2 \rangle$ and $\langle T_2 \lhd T_1 \rangle$ are performed at every application of the resulting function, in a manner reminiscent higher-order coercions [Thatte 1990] or higher-order contracts [Findler and Felleisen 2002].[1]

[E-Cast-Begin] and [E-Cast-End] A basic constant k is cast to a base refinement type $\{x{:}B \mid p\}$ via multiple evaluation steps. Via rule [E-Cast-Begin] a cast application $\langle \{x{:}B \mid p\} \lhd \{x{:}B \mid q\} \rangle\ k$ evaluates to a cast-in-progress $\langle \{x{:}B \mid p\}, [x \mapsto k]\, p, k \rangle$. The instantiated predicate $[x \mapsto k]\, t$ then evaluates via the closure rule [E-Ctx],

[1] We ignore the issue of blame assignment in the event of a run-time cast failure – see [Gronski and Flanagan 2007] for a detailed analysis.

either diverging or terminating in a value v. If v is `true` then the cast-in-progress evaluates to k, as it has been dynamically verified that k has type $\{x{:}B \mid p\}$, otherwise the cast-in-progress is "stuck" and we call it a failed cast.

Definition 4.2 (Failed Cast) *A* failed cast *is a term of the form* $\langle\{x{:}B \mid p\}, q, k\rangle$ *where* q *cannot evaluate further, yet* $q \neq$ `true`.

Note that these casts involve only familiar dynamic operations: tag checks, predicate checks, and creating checking wrappers for functions. Thus, our approach adheres to the principle of phase separation [Cardelli 1988], in that there is no type checking of actual program syntax at run time.

4.3 The Type System of $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$

The typing rules for casts are shown in Figure 4.4. Together with the collective typing rules of $\lambda_{\mathcal{H}}$ (Figure 3.5, Figure 3.6, Figure 3.8, Figure 3.7, Figure 3.9, pages 33 - 39), these define the type system of $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$.

[T-Cast] A cast $\langle T \triangleleft S\rangle$ is simply assigned the function type $S \to T$ provided both S and T are well-formed types of the same shape.

[T-Checking] A cast-in-progress $\langle\{x{:}B \mid p\}, q, k\rangle$ requires that the refinement type $\{x{:}B \mid p\}$ is well-formed and that the predicate q is a boolean term. Moreover, since this cast succeeds whenever q evaluates to `true`, we require that q actually implies the property being checked, $[x \mapsto k]\, p$. In practice, this always

Figure 4.4: Type rules for $\lambda^{\mathrm{HTC}}_{\mathcal{H}}$

Additional Type rules $\boxed{E \vdash e : T}$

$$\frac{\Gamma \vdash S \qquad \Gamma \vdash T \qquad \lfloor S \rfloor = \lfloor T \rfloor}{\Gamma \vdash \langle T \triangleleft S \rangle \ : \ S \to T} \text{ [T-Cast]}$$

$$\frac{\Gamma \vdash \{x{:}B \mid p\} \qquad \Gamma \vdash k \ : \ B \qquad \Gamma \vdash q \ : \ \texttt{Bool} \qquad \Gamma \vdash q \Rightarrow [x \mapsto k]\, p}{\Gamma \vdash \langle \{x{:}B \mid p\}, q, k \rangle \ : \ \{x{:}B \mid p\}} \text{ [T-Checking]}$$

holds because the evaluation rules ensure that q is arrived at by evaluating $[x \mapsto k]\, p$.

A cast in a program may be considered unnecessary in two different ways: via the operational semantics or via the type system. Operationally, a cast is not useful if it can never fail. The type system indicates a cast $\langle T \lhd S \rangle$ is redundant if $S <: T$. We take the term *redundant* to formally indicate this latter.

Definition 4.3 (Redundant Cast) *A* redundant cast*, or* upcast*, is a cast* $\langle T \lhd S \rangle$ *in a term* $\mathcal{C}[\langle T \lhd S \rangle]$ *such that* $\Gamma \vdash \mathcal{C}[\langle T \lhd S \rangle] : U$ *by a derivation*

$$\dfrac{\dfrac{\vdots}{\Gamma' \vdash \langle T \lhd S \rangle \ : \ S \to T}}{\vdots}$$
$$\Gamma \vdash \mathcal{C}[\langle T \lhd S \rangle] \ : \ U$$

and $\Gamma' \vdash S <: T$.

The fundamental correctness property for casts is that any redundant cast can never fail. In a higher-order setting, where casts are evaluated lazily, defining "can never fail" is an interesting problem with multiple approaches.

Wadler and Findler [2009] present a syntactic approach to this definition for a simpler calculus without dependent type or refinement types, but with careful tracking of blame for failed casts. Through the blame-based analogues of "progress" (if a term e cannot blame a principal A then it is not stuck blaming A) and "preservation" (if

a term e cannot blame A and it evaluates one step to e' then neither can e' blame A). Together these prove that an upcast (in their case, a contract with blame labels mediating between a simply-typed language and a dynamically-typed language) can never blame a principal that provides well-typed input.

Instead of attempting a translation of their results to our dependently-typed setting, we prove a stronger result using the extensional definition of equivalence: A redundant cast is always contextually equivalent to the identity function.[2]

The definition of extensional equivalence for $\lambda_{\mathcal{H}}$ in Figure 3.10 on page 49 applies unmodified to $\lambda_{\mathcal{H}}$ with hybrid type checking, and all the theorems surrounding it extend without difficulty. Thus correctness of the operational semantics of casts reduces to the following theorem:

Theorem 4.4 *If* $\Gamma \vdash S <: T$ *then* $\Gamma \vdash \langle T \triangleleft S \rangle \sim (\lambda x{:}S.\, x) : (S \to T)$

PROOF: Proceed by induction on the shape of S (which is necessarily also the shape of T) proceeding by considering the outermost syntactic form.

Case $S = \{x{:}B \mid p\}$ and $T = \{x{:}B \mid q\}$

To test equivalence of $\langle \{x{:}B \mid q\} \triangleleft \{x{:}B \mid p\} \rangle$ and $(\lambda x : \{B \mid q\}.\, x)$, we apply them to two terms d and e such that $\Gamma \vdash d \sim e : \{x{:}B \mid q\}$. For any closing substitution σ, we are guaranteed that $\sigma(d) \rightsquigarrow^* k$ if and only if $\sigma(e) \rightsquigarrow^* k$, so it suffices to consider applying the cast to such a k. By the semantic definition of implication and the assumption that constants are assigned appropriate types,

[2]This result was actually proved prior to the publication of their work.

$\sigma([x \mapsto k]\, q) \rightsquigarrow^*$ `true` so the cast succeeds, behaving as the identity function on k.

Case $S = x{:}S_1 \to S_2$ and $T = x{:}T_1 \to T_2$

To show that this cast is equivalent to the identity function, fix a term f and apply the cast, yielding

$$\lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \rangle\ (f\ (\langle S_1 \lhd T_1 \rangle\ x))$$

which must be shown equivalent to f. So now fix an argument an argument e of type T_1 and β-reduce to

$$\langle [x \mapsto e]\, T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ e]\, S_2 \rangle\ \ (f\ \ (\langle S_1 \lhd T_1 \rangle\ e))$$

By induction, $\langle S_1 \lhd T_1 \rangle\ e$ is equivalent to e, so the whole term is equivalent to

$$\langle [x \mapsto e]\, T_2 \lhd [x \mapsto e]\, S_2 \rangle\ \ (f\ e)$$

Again by induction (and the substitution property of subtyping) this is equivalent to $(f\ e)$. □

Corollary 4.5 *For a derivation*

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \langle T \lhd S \rangle\ :\ S \to T}\ \text{[T-Cast]}}{\dfrac{\vdots}{\emptyset \vdash \mathcal{C}[\langle T \lhd S \rangle\ e]\ :\ U}}$$

if the cast leads to a failure (a formal definition involving labels is beyond the scope of this chapter) then $\Gamma \not\vdash S <: T$ *and* e *is a* counterexample*, a term such that* $\Gamma \vdash e : S$ *but* $\Gamma \not\vdash e : T$.

To conclude this section, before treating the hybrid type checking algorithm, we note that all the lemmas of the preceding chapter extend in routine manner to casts, and we review just the key lemmas of progress and preservation. The preservation lemma holds as stated, with four new cases to consider.

Lemma 4.6 (Preservation) *If* $\Gamma \vdash d : T$ *and* $d \rightsquigarrow e$ *then* $\Gamma \vdash e : T$.

PROOF: The proof is by straightforward induction as before; we discuss only the cases involving casts. Note the free rebinding of metavariables, which is immaterial to the argument.

Case [T-CAST]: Evaluation within types does not affect typing.

Case [T-APP] for a function cast: We have

$$\langle x{:}T_1 \to T_2 \lhd x{:}S_1 \to S_2 \rangle \ f$$

evaluating to

$$\lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle \ x]\, S_2 \rangle \ (f \ (\langle S_1 \lhd T_1 \rangle \ x))$$

and we invite the reader to verify that preservation holds in this case.

Case [T-APP] for a cast to basic type: We have

$$\langle \{x{:}B \mid q\} \lhd \{x{:}B \mid p\} \rangle \ k$$

evaluating to

$$\langle \{x{:}B \mid q\}, [x \mapsto k]\, q, k \rangle$$

All the antecedents of [T-CHECKING] are satisfied obviously since the implication judgment is reflexive.

Case [T-Checking] still in progress: Evaluation of the predicate does not affect the implication judgment.

Case [T-Checking] completed: The implication antecedent of [T-Checking] ensures that k can be assigned the necessary type by subsumption. $\square$

In this setting, the Progress theorem holds only modulo failed casts; a term may be "justifiably" stuck because a cast has failed.

Lemma 4.7 (Progress) *If $\emptyset \vdash e : T$ then either e is a value, e contains a failed cast, or there is some e' such that $e \rightsquigarrow e'$.*

Proof: By induction on the derivation of $\emptyset \vdash e : T$ as before. $\square$

4.4 Hybrid Type Checking for $\lambda_{\mathcal{H}}^{\mathbf{HTC}}$

We now describe how to perform hybrid type checking. We work in the specific context of the language $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$, but have also demonstrated that the general approach extends to other languages with similarly expressive type systems, such as Sage of Chapter 7.

The key judgments constituting our hybrid type checking system utilize a three-element lattice of certainty, where "$\surd$" means a judgement certainly holds, "$\times$" means a judgment has been refuted, and "?" means that a judgment can be neither proven nor refuted. In the following, $a \in \{\surd, \times, ?\}$ and $\times < ? < \surd$.

- The implication algorithm $\Gamma \Vdash^a p \Rightarrow q$ soundly approximates implication, yield a certainty a about whether the implication holds. (This is a parameter to the system, hence not in any figure)

- The subtyping algorithm $\Gamma \Vdash^a S <: T$ soundly approximates subtyping. (Figure 4.5)

- Cast insertion and checking $\Gamma \Vdash d \hookrightarrow e \downarrow T$ inserts casts as into d as necessary to enforce type T, producing a new term e. (Figure 4.6)

- Cast insertion and synthesis $\Gamma \Vdash d \hookrightarrow e : T$ traverses d to produce a new term e and its type T. (Figure 4.7)

- Cast insertion on types $\Gamma \Vdash S \hookrightarrow T$ traverses the type S and inserts necessary casts to ensure that T is a well-formed type. (Figure 4.8)

The difference between the two cast insertion judgments of one of "polarity", *i.e.* which arguments of the judgment are considered inputs and which are considered outputs. The defining rules for these judgments make the distinction clear and rigorous. Each judgment is discussed in depth below.

4.4.1 Implication Algorithm

The implication algorithm is a parameter of the hybrid type checking system. Since implication is undecidable, it is intended that our theories adapt to embrace the latest technology in best-effort automated proving, so it is most effective to provide

the specification to which any such implementation must adhere. For any implication $\Gamma \vdash p \Rightarrow q$, the algorithmic judgment

$$\Gamma \Vdash^{a} p \Rightarrow q$$

always must hold with $a \in \{\surd, \times, ?\}$:

- The judgment $\Gamma \Vdash^{\surd} p \Rightarrow q$ means the algorithm finds a proof that $\Gamma \vdash p \Rightarrow q$.
- The judgment $\Gamma \Vdash^{\times} p \Rightarrow q$ means the algorithm finds a proof that $\Gamma \nvdash p \Rightarrow q$.
- The judgment $\Gamma \Vdash^{?} p \Rightarrow q$ means the algorithm terminates without either discovering a proof of either $\Gamma \vdash p \Rightarrow q$ or $\Gamma \nvdash p \Rightarrow q$.

The requirement placed upon this result is the subject of Requirement 4.8:

Requirement 4.8 (Soundness of $\Gamma \Vdash^{a} p \Rightarrow q$)

1. *If* $\Gamma \Vdash^{\surd} p \Rightarrow q$ *then* $\Gamma \vdash p \Rightarrow q$.
2. *If* $\Gamma \Vdash^{\times} p \Rightarrow q$ *then* $\Gamma \nvdash p \Rightarrow q$.

This algorithm cannot ever be complete. An extremely naive algorithm could simply return $\Gamma \Vdash^{?} p \Rightarrow q$ in all cases.

4.4.2 Subtyping Algorithm

The implication-checking algorithm $\Gamma \Vdash^{a} p \Rightarrow q$ naturally induces a 3-valued algorithmic subtyping judgment:

$$\Gamma \Vdash^{a} S <: T$$

Figure 4.5: Algorithmic Subtyping for $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$

Subtyping Algorithm $\boxed{\Gamma \Vdash^{a} S <: T}$

$$\frac{\Gamma \Vdash^{a} T_1 <: S_1 \qquad \Gamma, x : T_1 \Vdash^{b} S_2 <: T_2}{\Gamma \Vdash^{a \sqcap b} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)}\ [\text{SA-Arrow}]$$

$$\frac{\Gamma, x : B \Vdash^{a} p \Rightarrow q}{\Gamma \Vdash^{a} \{x{:}B \mid p\} <: \{x{:}B \mid q\}}\ [\text{SA-Base}]$$

as shown in Figure 4.5.

[SA-Base]: Algorithmic subtyping between refined basic types $\{x{:}B \mid p\}$ and $\{x{:}B \mid q\}$ reduces to a corresponding three-valued implication judgment between p and q.

[SA-Arrow]: Subtyping between function types reduces to subtyping between corresponding contravariant domain and covariant range types. This rule uses the standard meet (greater lower bound) operation $\sqcap$ on the three-element lattice

of possible implication results:

$\sqcap$	$\surd$	?	$\times$
$\surd$	$\surd$	?	$\times$
?	?	?	$\times$
$\times$	$\times$	$\times$	$\times$

If the subtyping relation can be proven between the domain and range components then the subtyping relation holds between the function types. If the appropriate subtyping relation does not hold between either the domain or range components, then the subtyping relation does not hold between the function types. Otherwise, in the uncertain case, subtyping *may* hold between the function types.

A consequence of the required soundness of the implication algorithm is that the algorithmic subtyping judgment $\Gamma \Vdash^a S <: T$ is also sound.

Lemma 1 (Soundness of $\Gamma \Vdash^a S <: T$) *Suppose* $\vdash \Gamma$.

1. *If* $\Gamma \Vdash^{\surd} S <: T$ *then* $\Gamma \vdash S <: T$.

2. *If* $\Gamma \Vdash^{\times} S <: T$ *then* $\Gamma \nvdash S <: T$.

Proof: By induction on the derivation of $\Gamma \Vdash^a S <: T$ using Requirement 4.8. $\square$

Lemma 1 should also be read as a requirement for future hybrid systems: It may not always be that algorithmic subtyping statically reduces to algorithmic implication. From a broader perspective, the subtyping algorithm itself is a parameter to

hybrid type checking, but this fact should still hold of the algorithmic approximation to subtyping.

4.4.3 Cast Insertion

Hybrid type checking uses this subtyping algorithm to type check the source program, and to simultaneously insert dynamic casts to compensate for any indefinite answers returned by the subtyping algorithm. The judgment which translates subtyping results to appropriate casts is the *cast insertion and checking judgment*

$$\Gamma \Vdash d \hookrightarrow e \downarrow T$$

which intuitively takes Γ, d and T as inputs and provides e either with casts or without. There are only two defining rules, presented in Figure 4.6, which are mutually recursive with the *cast insertion and synthesis* judgment described next:

[CC-Ok]: If the subtyping algorithm succeeds in proving that S is a subtype of T (*i.e.*, $\Gamma \Vdash^{\surd} S <: T$), then e is clearly of the desired type T, and so is returned unchanged.

[CC-Chk]: In the uncertain case where $\Gamma \Vdash^{?} S <: T$, the cast $\langle T \triangleleft S \rangle$ is inserted to dynamically ensure that e is actually of the desired type T.

Note that if $\Gamma \Vdash^{\times} S <: T$ then no rule applies with inputs Γ, d, and T, so the term d is rejected.

The cast insertion and synthesis judgment,

$$\Gamma \Vdash d \hookrightarrow e : T$$

Figure 4.6: Cast insertion and Checking for $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$

Cast insertion and checking $\boxed{\Gamma \Vdash d \hookrightarrow e \downarrow T}$

$$\frac{\Gamma \Vdash d \hookrightarrow e : S \qquad \Gamma \Vdash^{\checkmark} S <: T}{\Gamma \Vdash d \hookrightarrow e \downarrow T}\ [\text{CC-Ok}]$$

$$\frac{\Gamma \Vdash d \hookrightarrow e : S \qquad \Gamma \Vdash^{?} S <: T}{\Gamma \Vdash d \hookrightarrow \langle T \triangleleft S \rangle\ e \downarrow T}\ [\text{CC-Chk}]$$

take Γ and d as inputs and produces e and T as outputs, where T is the type of the resulting term e. Since types contain terms, we extend this cast insertion process to types via the judgment

$$\Gamma \Vdash S \hookrightarrow T$$

which serves only to traverse types and insert casts to the terms contained therein. The rules defining each of these this judgments are presented in Figure 4.7; Most of the rules are straightforward enhancements of the typing rules:

[C-Var] and [C-Const]: Variable references and constants do not require additional casts.

[C-Fun]: An abstraction $(\lambda x : S_1 . d)$ by processed by first inserting casts into the type S_1 to yield T_1 and then processing d to yield a term e of type T_2; the resulting

Figure 4.7: Cast insertion and synthesis for $\lambda^{\mathrm{HTC}}_{\mathcal{H}}$ terms

Cast insertion and synthesis $\boxed{\Gamma \Vdash d \hookrightarrow e : T}$

$$\frac{(x : T) \in \Gamma}{\Gamma \Vdash x \hookrightarrow x : T}\ [\text{C-Var}] \qquad \frac{}{\Gamma \Vdash k \hookrightarrow k : ty(k)}\ [\text{C-Const}]$$

$$\frac{\Gamma \Vdash S_1 \hookrightarrow T_1 \qquad \Gamma, x : T_1, \Vdash d \hookrightarrow e : T_2}{\Gamma \Vdash (\lambda x{:}S_1.\, d) \hookrightarrow (\lambda x{:}T_1.\, e) : (x{:}T_1 \to T_2)}\ [\text{C-Fun}]$$

$$\frac{\Gamma \Vdash f \hookrightarrow g : (x{:}T_1 \to T_2) \qquad \Gamma \Vdash d \hookrightarrow e \downarrow T_1}{\Gamma \Vdash f\ d \hookrightarrow g\ e : [x \mapsto e]\, T_2}\ [\text{C-App}]$$

Figure 4.8: Cast insertion for $\lambda^{\mathrm{HTC}}_{\mathcal{H}}$ types

Cast insertion on types $\boxed{\Gamma \Vdash S \hookrightarrow T}$

$$\frac{\Gamma \Vdash S_1 \hookrightarrow T_1 \qquad \Gamma, x : T_1 \Vdash S_2 \hookrightarrow T_2}{\Gamma \Vdash (x{:}S_1 \to S_2) \hookrightarrow (x{:}T_1 \to T_2)}\ [\text{C-Arrow}]$$

$$\frac{\Gamma, x : B \Vdash p \hookrightarrow q \downarrow \texttt{Bool}}{\Gamma \Vdash \{x{:}B \mid p\} \hookrightarrow \{x{:}B \mid q\}}\ [\text{C-Base}]$$

abstraction $\lambda x{:}T_1.\, e$ has type $x{:}T_1 \to T_2$.

[C-Cast]: If a cast $\langle T \triangleleft S \rangle$ occurs in a program, then either of S or T may contain terms, so casts are inserted into each of these types. In an implementation of hybrid type checking, one might simply forbid casts in input programs, but we include this rule for completeness and to discuss idempotence of cast insertion.

[C-App]: For an application $(f\ d)$, first casts are inserted into f, yielding g of type $x{:}T_1 \to T_2$. Then invokes the cast insertion *and checking* judgment to convert the argument d into a term e of the appropriate argument type T_1.

[C-Arrow]: Casts are inserted into the domain and codomain of a function type $x{:}S \to T$ by induction.

[C-Base]: The refinement predicate p of a refined base type is checked against type `Bool`.

4.4.4 Correctness of Cast Insertion

Since hybrid type checking relies on necessarily incomplete algorithms for subtyping and implication, this section contributes a clear specification for hybrid type checking in general and proves that our description of hybrid type checking for $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$ meets that specification.

A modern typed compiler relies on type safety for a variety of factors, including memory layout and optimizations. So it is important that during hybrid type checking

while the input program may be subtly ill-typed, the output program is well-typed. Thus any terms that would violate guarantees a compiler relied on are safely wrapped.

Theorem 4.9 (Cast insertion produces well-typed terms) *Suppose* $\vdash \Gamma$.

1. *If* $\Gamma \Vdash e \hookrightarrow e' : T$ *then* $\Gamma \vdash e' : T$.

2. *If* $\Gamma \Vdash e \hookrightarrow e' \downarrow T$ *and* $\Gamma \vdash T$ *then* $\Gamma \vdash e' : T$.

3. *If* $\Gamma \Vdash T \hookrightarrow T'$ *then* $\Gamma \vdash T'$.

Proof: Proceeds directly by mutual induction on cast insertion derivations. □

This fact ensures that the cast insertion rules actually do their job of inserting casts everywhere that they are needed. Furthermore, the cast insertion process is idempotent provided algorithmic subtyping meets the most basic requirement of reflexivity.

Theorem 4.10 (Idempotence of Cast Insertion)

Suppose that for any Γ *and* T *such that* $\Gamma \vdash T$, *the subtyping algorithm also determines* $\Gamma \Vdash^{\surd} T <: T$.

1. *If* $\Gamma \Vdash d \hookrightarrow e \downarrow T$ *then* $\Gamma \Vdash e \hookrightarrow e \downarrow T$.

2. *If* $\Gamma \Vdash d \hookrightarrow e : T$ *then* $\Gamma \Vdash e \hookrightarrow e : T$

Proof: By mutual induction on the derivations. □

Converse to the fact that hybrid type checking always produces a well-typed program, it also accepts all well-typed programs. The program may be subtle and thus have some casts added, but these casts are all redundant.

Theorem 4.11 (Cast insertion is the identity on well-typed terms)

1. *If* $\Gamma \vdash d : T$ *then there exists a term* e *and type* S *such that* $\Gamma \Vdash d \hookrightarrow e : S$ *and* $\Gamma \vdash S <: T$ *and* $\Gamma \vdash d \sim e : T$.

2. *If* $\Gamma \vdash d : S$ *then for any* T *such that* $\Gamma \vdash S <: T$ *there exists a term* e *such that* $\Gamma \Vdash d \hookrightarrow e \downarrow T$ *and* $\Gamma \vdash d \sim e : T$.

3. *If* $\Gamma \vdash S$ *then there exists* T *such that* $\Gamma \Vdash S \hookrightarrow T$ *and* $\Gamma \vdash S <: T$ *and* $\Gamma \vdash S <: T$.

Proof: By mutual induction on the derivations of $\Gamma \vdash e : T$ and $\Gamma \vdash T$. Because algorithmic subtyping can only return ? or $\surd$, the output of cast insertion is simply the input with redundant casts inserted. By Theorem 4.4 these are all equivalent to identity functions. □

4.5 An Example of HTC

To illustrate the behavior of the cast insertion algorithm, consider a function `serializeMatrix` that serializes an n by m matrix into an array of size $n \times m$. We extend the language $\lambda_{\mathcal{H}}^{\text{HTC}}$ with two additional base types:

- `Array`, the type of one dimensional arrays containing integers.

- `Matrix`, the type of two dimensional matrices, again containing integers.

The following primitive functions return the size of an array; create a new array of the given size; and return the width and height of a matrix, respectively:

$$\begin{aligned}
\texttt{asize} &: a{:}\texttt{Array} \to \texttt{Int} \\
\texttt{newArray} &: n{:}\texttt{Int} \to \{a{:}\texttt{Array} \mid \texttt{asize}\ a\} = n \\
\texttt{matrixWidth} &: a{:}\texttt{Matrix} \to \texttt{Int} \\
\texttt{matrixHeight} &: a{:}\texttt{Matrix} \to \texttt{Int}
\end{aligned}$$

We introduce the following type abbreviations to denote arrays of size n and matrices of size n by m:

$$\begin{aligned}
\texttt{Array}_n &\stackrel{\text{def}}{=} \{a{:}\texttt{Array} \mid (\texttt{asize}\ a = n)\} \\
\texttt{Matrix}_{n,m} &\stackrel{\text{def}}{=} \{a{:}\texttt{Matrix} \mid \begin{pmatrix} \texttt{matrixWidth}\ \ a = n \\ \wedge\ \texttt{matrixHeight}\ a = m \end{pmatrix}\}
\end{aligned}$$

The shorthand $e\ \texttt{as}\ T$ ensures that the term e has type T by passing e as an argument to the identity function of type $T \to T$:

$$e\ \texttt{as}\ T \stackrel{\text{def}}{=} (\lambda x{:}T.\, x)\ e$$

We now define the function `serializeMatrix` as:

$$\begin{pmatrix} \lambda n{:}\texttt{Int}.\ \lambda m{:}\texttt{Int}.\ \lambda a{:}\texttt{Matrix}_{n,m}. \\ \texttt{let}\ r\ =\ \texttt{newArray}\ e\ \texttt{in}\ \ldots\ ;\ r \end{pmatrix}\ \texttt{as}\ T$$

The elided term $\ldots$ initializes the new array r with the contents of the matrix a, and we will consider several possibilities for the size expression e. The type T is the specification of `serializeMatrix`:

$$T \stackrel{\text{def}}{=} (n{:}\texttt{Int} \to\ m{:}\texttt{Int} \to\ \texttt{Matrix}_{n,m} \to\ \texttt{Array}_{n \times m})$$

For this declaration to type check, the inferred type $\texttt{Array}_e$ of the function's body must be a subtype of the declared return type:

$$n : \texttt{Int},\ m : \texttt{Int} \ \vdash \texttt{Array}_e <: \texttt{Array}_{n \times m}$$

Checking this subtype relation reduces to checking the implication:

$$\begin{aligned} n : \texttt{Int},\ m : \texttt{Int},\ r : \texttt{Array} \ \vdash \quad & (\texttt{asize}\ r = e) \\ \Rightarrow \quad & (\texttt{asize}\ r = (n \times m)) \end{aligned}$$

which in turn reduces to checking the equality:

$$\forall n, m \in \texttt{Int}.\ \ e = n \times m$$

The implication checking algorithm might use an automatic theorem prover to verify or refute such conjectured equalities.

We now consider three possibilities for the expression e.

1. If e is the expression $n \times m$, the equality is trivially true, and no additional casts are inserted (even when using a rather weak theorem prover).

2. If e is $m \times n$ (*i.e.*, the order of the multiplicands is reversed), and the underlying theorem prover can verify

   $$\forall n, m \in \texttt{Int}.\ \ m \times n = n \times m$$

   then again no casts are necessary. Note that a theorem prover that is not complete for arbitrary multiplication might still have a specific axiom about the commutativity of multiplication.

If the theorem prover is too limited to verify this equality, the hybrid type checker will still accept this program. However, to compensate for the limitations of the theorem prover, the hybrid type checker will insert a redundant cast, yielding the function (where we have elided the source type of the cast):

$$\left(\langle T \triangleleft \ldots \rangle \left(\begin{array}{l}\lambda n{:}\texttt{Int}.\ \lambda m{:}\texttt{Int}.\ \lambda a{:}\texttt{Matrix}_{n,m}.\\ \\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ e\ \texttt{in}\ \ldots\ ;\ r\end{array}\right)\right)\ \texttt{as}\ T$$

This term can be optimized, via [E-Beta] and [E-Cast-Fn] steps and via removal of clearly redundant $\langle \texttt{Int} \triangleleft \texttt{Int} \rangle$ casts, to:

$$\begin{array}{l}\lambda n{:}\texttt{Int}.\ \lambda m{:}\texttt{Int}.\ \lambda a{:}\texttt{Matrix}_{n,m}.\\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ (m \times n)\ \texttt{in}\\ \quad\ \ldots\ ;\\ \quad \langle \texttt{Array}_{n\times m} \triangleleft \texttt{Array}_{m\times n} \rangle\ r\end{array}$$

The remaining cast checks that the result value r is of the declared return type $\texttt{Array}_{n\times m}$, which reduces to dynamically checking that the predicate

$$\texttt{asize}\ r\ \ =\ \ n \times m$$

evaluates to `true`, which it does.

3. Finally, if e is erroneously $m \times m$, the function is ill-typed. By performing random or directed [Godefroid et al. 2005] testing of several values for n and m until it finds a counterexample, the theorem prover might reasonably refute the conjectured equality:

$$\forall n, m \in \texttt{Int}.\ \ m \times m = n \times m$$

In this case, the hybrid type checker reports a static type error.

Conversely, if the theorem prover is too limited to refute the conjectured equality, then the hybrid type checker will produce (after optimization) the program:

$$\lambda n{:}\texttt{Int}.\ \lambda m{:}\texttt{Int}.\ \lambda a{:}\texttt{Matrix}_{n,m}.$$
$$\texttt{let}\ r\ =\ \texttt{newArray}\ (m \times m)\ \texttt{in}$$
$$\ldots\ ;$$
$$\langle \texttt{Array}_{n\times m} \triangleleft \texttt{Array}_{m\times m} \rangle\ r$$

If this function is ever called with arguments for which $m \times m \neq n \times m$, then the cast will detect the type error.

Note that prior work on practical dependent types [Xi and Pfenning 1999] could not handle these cases, since the type T uses non-linear arithmetic expressions. In contrast, case 2 of this example demonstrates that even fairly partial techniques for reasoning about complex specifications (*e.g.*, commutativity of multiplication, random testing of equalities) can facilitate static detection of defects. Furthermore, even though catching errors at compile time is ideal, catching errors at run time (as in case 3) is still an improvement over not detecting these errors at all, and getting subsequent crashes or incorrect results.

4.6 Static Checking vs. Hybrid Checking

Given the proven benefits of traditional, purely-static type systems, an important question that arises is how hybrid type checkers relate to static type checkers. To

study this question theoretically, suppose we are given a static type checker that targets a restricted subset of $\lambda_{\mathcal{H}}^{\text{HTC}}$ for which type checking is statically decidable. Specifically, suppose $\mathcal{D}$ is a subset of *Term* such that for all $p, q \in \mathcal{D}$ and for all singleton environments $x : B$, the judgment $x : B \vdash p \Rightarrow q$ is decidable. We introduce a statically-decidable language $\lambda_{\mathcal{S}}$ that is obtained from $\lambda_{\mathcal{H}}^{\text{HTC}}$ by only permitting $\mathcal{D}$ predicates in refinement types. We also assume all types in $\lambda_{\mathcal{S}}$ are closed, to avoid the complications of substituting arbitrary terms into refinement predicates via the rule [T-App] (and hence the above environment $x : B$ suffices). It then follows that subtyping and type checking for $\lambda_{\mathcal{S}}$ are decidable, and we denote this type checking judgment as $\Gamma \vdash_{\mathcal{S}} e : T$.

As an extreme, we could take $\mathcal{D} = \{\texttt{true}\}$, in which case the $\lambda_{\mathcal{S}}$ type language is essentially that of STLC. However, to yield a more general argument, we assume only that $\mathcal{D}$ is a subset of *Term* for which implication is decidable.

Clearly, the hybrid implication algorithm can give precise answers on (decidable) $\mathcal{D}$-terms, and so we assume that for all $p, q \in \mathcal{D}$ and for all environments $x : B$, the judgment $x : B \Vdash^{a} p \Rightarrow q$ holds for some $a \in \{\surd, \times\}$. Under this assumption, hybrid type checking behaves identically to static type checking on (well-typed or ill-typed) $\lambda_{\mathcal{S}}$ programs, formalized as:

Theorem 4.12 *For all $\lambda_{\mathcal{S}}$ terms e, $\lambda_{\mathcal{S}}$ environments Γ, and $\lambda_{\mathcal{S}}$ types T, the following three statements are equivalent:*

1. $\Gamma \vdash_{\mathcal{S}} e : T$
2. $\Gamma \vdash e : T$

3. $\Gamma \vdash e \hookrightarrow e : T$

PROOF: The hybrid implication algorithm is complete on $\mathcal{D}$-terms, and hence the hybrid subtyping algorithm is complete for $\lambda_{\mathcal{S}}$ types. The proof then follows by induction on typing derivations. $\square$

Thus, to a $\lambda_{\mathcal{S}}$ programmer, a hybrid type checker behaves exactly like a traditional static type checker.

We now compare static and hybrid type checking from the perspective of a $\lambda_{\mathcal{H}}^{\text{HTC}}$ programmer. To enable this comparison, we need to map expressive $\lambda_{\mathcal{H}}^{\text{HTC}}$ types into the more restrictive $\lambda_{\mathcal{S}}$ types, and in particular to map arbitrary boolean predicates into $\mathcal{D}$ predicates. We assume the computable function

$$\zeta : \mathit{Term} \to \mathcal{D}$$

performs this mapping. The function *erase* then maps $\lambda_{\mathcal{H}}^{\text{HTC}}$ refinement types to $\lambda_{\mathcal{S}}$ refinement types by using ζ to abstract boolean terms:

$$\mathit{erase}(\{x{:}B \mid p\}) \stackrel{\text{def}}{=} \{x{:}B \mid \zeta(p)\}$$

We extend *erase* in a compatible manner to map $\lambda_{\mathcal{H}}^{\text{HTC}}$ types, terms, and environments to corresponding $\lambda_{\mathcal{S}}$ types, terms, and environments. Thus, for any $\lambda_{\mathcal{H}}^{\text{HTC}}$ program e, this function yields the corresponding $\lambda_{\mathcal{S}}$ program $\mathit{erase}(e)$.

As might be expected, the *erase* function must lose information, and we now explore the consequences of this information loss for programs with complex specifications. As an extreme example, let $\mathit{Halt}_{n,m}$ denote a closed formula that encodes "Turing

machine m eventually halts on input n". Suppose that ζ maps $Halt_{n,m}$ to `false` for all n and m, and consider the collection of programs $P_{n,m}$, which includes both well-typed and ill-typed programs:

$$P_{n,m} \stackrel{\text{def}}{=} (\lambda x{:}\{x{:}\texttt{Int} \mid Halt_{n,m}\}.\, x)\ 1$$

Decidability arguments show that the hybrid type checker will accept some ill-typed program $P_{n,m}$ based on its inability to statically prove that $Halt_{n,m} = \texttt{false}$. In contrast, the static type checker will reject (the erased version of) $P_{n,m}$. Hence:

> *The static type checker statically rejects (the erased version of) an ill-typed program that the hybrid type checker accepts.*

Thus, the static type checker performs better in this situation.

However, this particular static type checker will also reject (the erased version of) many well-typed programs $P_{n,m}$ that the hybrid type checker accepts. The following theorem generalizes this argument, and shows that for any computable mapping ζ there exists some program P such that hybrid type checking of P performs better than static type checking of $erase(P)$.

Theorem 4.13 *For any computable mapping ζ either:*

1. *the static type checker rejects the erased version of some well-typed $\lambda_{\mathcal{H}}^{HTC}$ program, or*

2. *the static type checker accepts the erased version of some ill-typed $\lambda_{\mathcal{H}}^{HTC}$ program for which the hybrid type checker would statically detect the error.*

PROOF: Let Γ be the environment $x : \texttt{Int}$.

By reduction from the halting problem, the judgment $\Gamma \vdash p \Rightarrow \texttt{false}$ for arbitrary boolean terms p is undecidable. However, the implication judgment $\Gamma \vdash \zeta(p) \Rightarrow \zeta(\texttt{false})$ is decidable. Hence these two judgments are not equivalent, *i.e.*:

$$\{t \mid (\Gamma \vdash p \Rightarrow \texttt{false})\} \quad \neq \quad \{t \mid (\Gamma \vdash \zeta(p) \Rightarrow \zeta(\texttt{false}))\}$$

It follows that there must exists some *witness* w that is in one of these sets but not the other, and so one of the following two cases must hold.

1. Suppose:

$$\Gamma \vdash w \Rightarrow \texttt{false}$$

$$\Gamma \nvdash \zeta(w) \Rightarrow \zeta(\texttt{false})$$

We construct as a counter-example the program P_1:

$$P_1 \quad \stackrel{\text{def}}{=} \quad \lambda x{:}\{x{:}\texttt{Int} \mid w\}.\,(x \texttt{ as } \{x{:}\texttt{Int} \mid \texttt{false}\})$$

From the assumption $\Gamma \vdash w \Rightarrow \texttt{false}$ the subtyping judgment

$$\emptyset \vdash \{x{:}\texttt{Int} \mid w\} <: \{x{:}\texttt{Int} \mid \texttt{false}\}$$

holds. Hence, P_1 is well-typed, and (by Theorem 4.11 is accepted by the hybrid type checker. However, from the assumption $\Gamma \nvdash \zeta(w) \Rightarrow \zeta(\texttt{false})$ the erased version of the subtyping judgment does not hold:

$$\emptyset \nvdash \mathit{erase}(\{x{:}\texttt{Int} \mid w\}) <: \mathit{erase}(\{x{:}\texttt{Int} \mid \texttt{false}\})$$

Hence $\mathit{erase}(P_1)$ is ill-typed and rejected by the static type checker.

2. Conversely, suppose:

$$\Gamma \not\vdash w \Rightarrow \texttt{false}$$

$$\Gamma \vdash \zeta(w) \Rightarrow \zeta(\texttt{false})$$

From the first supposition and by the definition of the implication judgment, there exists integers n and m such that

$$[x \mapsto n]\, w \rightsquigarrow^{m} \texttt{true}$$

We now construct as a counter-example the program P_2:

$$P_2 \stackrel{\text{def}}{=} \lambda x{:}\{x{:}\texttt{Int} \mid w\}.\,(x\ \texttt{as}\ \{x{:}\texttt{Int} \mid \texttt{false} \wedge (n = m)\})$$

In the program P_2, the term $n = m$ has no semantic meaning since it is conjoined with `false`. The purpose of this term is to serve only as a "hint" to the following rule for refuting implications (which we assume is included in the reasoning performed by the implication algorithm). In this rule, the integers a and b serve as hints, and take the place of randomly generated values for testing if p ever evaluates to `true`.

$$\frac{[x \mapsto a]\, p \rightsquigarrow^{b} \texttt{true}}{\Gamma \Vdash^{\times} p \Rightarrow (\texttt{false} \wedge a = b)}$$

This rule enables the implication algorithm to conclude that:

$$\Gamma \Vdash^{\times} w \Rightarrow \texttt{false} \wedge (n = m)$$

Hence, the subtyping algorithm can conclude:

$$\Vdash^{\times} \{x{:}\texttt{Int} \mid w\} <: \{x{:}\texttt{Int} \mid \texttt{false} \wedge (n = m)\}$$

Therefore, the hybrid type checker correctly rejects P_2, which by Theorem 4.11 is therefore ill-typed.

We next consider how the static type checker behaves on the program $\mathit{erase}(P_2)$. We consider two cases, depending on whether the following implication judgment holds:

$$\Gamma \vdash \zeta(\texttt{false}) \Rightarrow \zeta(\texttt{false} \wedge (n = m))$$

(a) If this judgment holds then by the transitivity of implication and the assumption $\Gamma \vdash \zeta(w) \Rightarrow \zeta(\texttt{false})$ we have that:

$$\Gamma \vdash \zeta(w) \Rightarrow \zeta(\texttt{false} \wedge (n = m))$$

Hence the subtyping judgment

$$\emptyset \vdash \{x{:}\texttt{Int} \mid \zeta(w)\} <: \{x{:}\texttt{Int} \mid \zeta(\texttt{false} \wedge (n = m))\}$$

holds and the program $\mathit{erase}(P_2)$ is incorrectly accepted by the static type checker:

$$\emptyset \vdash \mathit{erase}(P_2) \; : \; \{x{:}\texttt{Int} \mid \zeta(w)\} \rightarrow \{x{:}\texttt{Int} \mid \zeta(\texttt{false} \wedge (n = m))\}$$

(b) If the above judgment does not hold then consider as a counter-example the program P_3:

$$P_3 \stackrel{\text{def}}{=} \lambda x{:}\{x{:}\texttt{Int} \mid \texttt{false}\}.\, (x \texttt{ as } \{x{:}\texttt{Int} \mid \texttt{false} \wedge (n{=}m)\})$$

This program is well-typed, from the subtype judgment:

$$\emptyset \vdash \{x{:}\texttt{Int} \mid \texttt{false}\} <: \{x{:}\texttt{Int} \mid \texttt{false} \wedge (n = m)\}$$

However, the erased version of this subtype judgment does not hold:

$$\emptyset \not\vdash \mathit{erase}(\{x{:}\texttt{Int} \mid \texttt{false}\}) <: \mathit{erase}(\{x{:}\texttt{Int} \mid \texttt{false} \wedge (n{=}m)\})$$

Hence, $\mathit{erase}(P_3)$ is rejected by the static type checker. □

4.7 Related Work

Many systems feature a combination of static and dynamic checking of specifications. The details of what languages the specifications are written in (if any), which checks are performed during which phase of a program's lifecycle, and how this decision is made, differ considerably.

4.7.1 Static structural types, dynamic contracts

Widespread languages such as Java and C# enforce structural types statically, while enforcing the implicit contracts on array bounds dynamically. The programming language Eiffel [Meyer 1988] also includes a structural type system while making dynamic contracts explicit; the languages of statically checked specifications and dynamically checked specifications are completely different.

All implementations of "contracts as a library" such as Hinze et al. [2006], Chitil [2012] fall into the same classification as Eiffel, with types forming the language of statically checked specifications and boolean terms of the language at hand expressing dynamically checked specifications.

Having multiple specification languages is somewhat awkward since it requires

the programmer to factor each specification into its static and dynamic components. Furthermore, the factoring is too rigid, since the specification needs to be manually refactored to exploit improvements in static checking technology. Contracts implemented as a library have the additional drawback that there is not possible to take advantage of any formal correspondence between the static and dynamic specifications languages.

4.7.2 Optimizing implied dynamic checks

The limitations of purely-static and purely-dynamic approaches have also motivated other work on hybrid analyses. For example, CCured [Necula et al. 2005] is a sophisticated hybrid analysis for preventing the ubiquitous array bounds violations in the C programming language. Unlike our proposed approach, it does not detect errors statically - instead, the static analysis is used to optimize the run-time analysis.

Soft typing Fagan [1991] performs best-effort static type checking in order to eliminate casts which are implicit in the semantics of any dynamically typed programming language. Henglein [1994] characterized the *completion process* of inserting the necessary coercions, and presented a rewriting system for generating minimal completions Thatte [1990] developed a similar system in which the necessary casts are implicit.

Any of these analyses could be viewed as hybrid type systems where the subtyping algorithm never rejects a query. Rather, they start from dynamically typed languages which are equivalent to a subtyping algorithm that always returns "?" and enhance it to sometimes return "$\surd$". Optionally, they may be enhanced to determine

that "×" is the correct answer, but will only issue warnings since rejection of a program is not within their purview.

4.7.3 Dynamic typing in a statically-typed language

Many authors have proposed systems that combine static structural type systems and dynamic typing, using casts to mediate between the dynamically typed portions of a program and the statically typed portions. Abadi et al. [1991] extended a static type system with a type `Dynamic` that could be explicitly cast to and from any other type (with appropriate run-time checks). Siek and Taha [2006] independently developed a system dubbed *gradual typing* that combines dynamic and static typing via casts similar to ours. However, they do not address dependent types or refinement predicates, so it remains to be investigated whether the approaches from gradual typing can shed additional light on our more advanced setting. A longer discussion of subsequent work on gradual typing is deferred to Chapter 8.

Ou et al. [2004] developed a type system similar to $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$, adding an interface between untyped code and code with refinement and dependent types. Unlike the hybrid type checking approach, their type system restricts refinement predicates to ensure decidability and it supports mutable data. In addition, whereas hybrid type checking uses dynamic checks to circumvent decidability limitations, their system uses dynamic checks to permit interoperability between precisely typed code fragments (that use refinement types) and more coarsely typed code fragments (that do not), and so it permits the introduction of refinement predicates into a large program in an incremental

manner.

These systems are intended to support looser type specifications, blending classic simple type systems with a dynamically typed programming style. In contrast, our work uses similar, automatically-inserted casts to support more precise type specifications. Chapter 7 combines both approaches to support a large range of specifications, from `Dynamic` at one end to precise hybrid-checked specifications at the other.

Chapter 5

Type Reconstruction

We have addressed the problem of type-checking for a language like $\lambda_{\mathcal{H}}$, given complete type annotation by the programmer. But, even for small examples, writing explicitly typed terms can be tedious and would become truly onerous for larger programs. To reduce the annotation burden, many typed languages – such as ML, Haskell, and their variants – perform type reconstruction,[1] often stated as: *Given a program containing type variables, find a replacement for those variables such that the resulting program is well typed.* If there exists such a replacement, the program is said to be *typeable*. Under this definition, type reconstruction subsumes type checking. Hence, for expressive and undecidable type systems, such as that of $\lambda_{\mathcal{H}}$, type reconstruction is clearly undecidable.

Instead of surrendering to undecidability, we separate type reconstruction from

[1]Type reconstruction is sometimes referred to as "type inference", especially in mainstream programming practice where it is often an inextricable part of a type checker. But type checking itself is an inference process, and has also been called "type inference", so for clarity and precision we carefully distinguish the terminology.

type checking, and define the type reconstruction problem as: *Given a program containing type variables, find a replacement for those variables such that typeability is preserved.* In a decidable type system, this definition coincides with the previous one, since the type checker can decide if the resulting explicitly typed program is well typed. Our generalized definition also extends to undecidable type systems, since alternative techniques, such as hybrid type checking of Chapter 4, can be applied to the resulting program. In particular, type reconstruction for $\lambda_{\mathcal{H}}$ is decidable!

Our approach to inferring refinement predicates is inspired by techniques from axiomatic semantics, most notably the strongest postcondition (SP) transformation [Back 1988]. This transformation supports arbitrary predicates in some specification logic, and computes the most precise correctness predicate for each program point. It is essentially syntactic in nature, deferring all semantic reasoning to a subsequent theorem-proving phase. For example, looping constructs in the program are expressed simply as fixed point operations in the specification logic.

In the richer setting of $\lambda_{\mathcal{H}}$, which includes higher order functions with dependent types, we must infer both the structural shape of types and also any refinement predicates they contain. We solve the former using traditional type reconstruction techniques and the latter using a syntactic, SP-like, transformation. Like SP, our algorithm infers the most precise predicates possible. The calculated higher-order strongest postconditions may contain complex terms that cannot be feasibly checked dynamically due to fixed-point operations, existential quantification, and parallel boolean connectives, that are only partially executable, but such execution will never be necessary as the

predicates are all correct by construction.

Though the most obvious novelty in type reconstruction for $\lambda_{\mathcal{H}}$ lies in higher-order strongest postcondition calculation, we present an end-to-end solution for the full type reconstruction problem, including pieces that resemble traditional type reconstruction. There are two reasons for this:

1. This full presentation demonstrates exactly how ordinary type reconstruction is separable from the higher-order postcondition calculation.

2. The metatheoretical approach is of key importance for guiding future work in how to provide generalized type reconstruction in other contexts.

Chapter Outline

The organization of this chapter is as follows

- Section 5.1 formally defines our generalized notion of type reconstruction.

- Section 5.2 describes the first phase of type reconstruction, *constraint generation*, which results in a provisional typing derivation and a set of subtyping constraints of the form $\Gamma \vdash S <: T$ (the same as the subtyping judgment) that must be satisfied for this typing derivation to be valid.

- Section 5.3 describes the second phase of type reconstruction, *shape reconstruction*, which converts the subtyping constraints to implication constraints, each of the form $\Gamma \vdash p \Rightarrow q$ (the same as the implication judgment).

- Section 5.4 describes the final phase of type reconstruction, the calculation of *strongest refinements* (SR), which solves all remaining implication constraints.

- Section 5.5 establishes the end-to-end correctness of the phases of type reconstruction.

- Section 5.6 surveys closely related work.

5.1 Type Reconstruction for $\lambda_{\mathcal{H}}^{\mathbf{Recon}}$

To explore type reconstruction for $\lambda_{\mathcal{H}}$ we work with an extended calculus $\lambda_{\mathcal{H}}^{\text{Recon}}$, which the sections of this chapter will extend as needed to present further aspects of type reconstructions. The input to type reconstruction is described by the syntax of Figure 5.1, which extends $\lambda_{\mathcal{H}}$ with *type variables* α, representing types omitted in an input program.

$$T \ ::= \ \cdots \ \mid \ \alpha$$

A type variable may appear anywhere a type is expected. A candidate solution to type reconstruction is a *type replacement* π that maps type variables to types. Application of a type replacement is lifted compatibly to all syntactic sorts, and is not capture avoiding.

Definition 5.1 (Typeability) *A closed term e containing type variables in place of some or all type annotations is* typeable *if there is some type T and type replacement π from type variables to types such that $\emptyset \vdash \pi(e) : \pi(T)$.*

Figure 5.1: Syntax for $\lambda_{\mathcal{H}}^{\text{Recon}}$ with type variables

$e, d, f, g, q, p ::=$	*Terms:*
$\quad k$	constant
$\quad x$	variable
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad f\ e$	application
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	constant
$\quad \lambda x\!:\!S.\, e$	abstraction
$R, S, T, U ::=$	*Types:*
$\quad x\!:\!S \to T$	dependent function type
$\quad \{x\!:\!B \mid p\}$	refined basic type
$\quad \alpha$	***type variable***

Definition 5.2 (The Type Reconstruction Problem) *Given a term e containing type variables in place of some or all type annotations, find a type replacement π such that e is typeable if and only if $\pi(e)$ is typeable, and $\pi(e)$ contains no type variables.* [2]

Note that for the type reconstruction problem, we consider only the basic language of $\lambda_{\mathcal{H}}$, omitting casts from Chapter 4 since they add little interest to type reconstruction.

[2]Note that the condition requiring the replacement of *all* type variables could be relaxed to allow partial type reconstruction, in the event that even this generalized notion of type reconstruction were not decidable or feasible.

5.2 Constraint Generation

The first phase of type reconstruction is constraint generation, which essentially mirrors type checking but collects subtyping constraints instead of enforcing them. Figure 5.2 presents the syntax of constraints and extensions to $\lambda_{\mathcal{H}}^{\text{Recon}}$ necessary.

- A *subtyping constraint* has the same form as the subtyping judgment, $\Gamma \vdash S <: T$. Constraint generation results in a set of such constraints that essentially record the flow of data through the program.

- A *type well-formedness constraint* has the same form as the type well-formedness judgment, $\Gamma \vdash T$. Constraint generation results in type well-formedness constraints that record the environment in which a type variable must be well-formed, constraining the variables that may appear free in T.

Definition 5.3 (Constraint Satisfaction) *A type replacement* π satisfies *a subtyping constraint* $\Gamma \vdash S <: T$ *(respectively, type well-formedness constraint* $\Gamma \vdash T$*) if* $\pi\Gamma \vdash \pi S <: \pi T$ *holds as a subtyping judgment (respectively* $\pi\Gamma \vdash \pi S$ *holds as a type well-formedness judgment). A type replacement* π *satisfies a constraint set* C *if it satisfies all the constraints in* C*.*

To facilitate our development, we require that the type language be closed under substitution. But a substitution cannot immediately be applied to a type variable, so each type variable α is equipped with a *delayed substitution* θ.

$$T ::= \cdots \mid \theta \cdot \alpha$$

If the substitution θ is empty, then $\theta \cdot \alpha$ may simply be written as α. The usual definition of capture-avoiding substitution is then extended to type variables, which simply delay that substitution:

$$[x \mapsto e : T]\ (\theta \cdot \alpha)\ =\ ([x \mapsto e : T] \circ \theta) \cdot \alpha$$

Here we use the notation $[x \mapsto e : T]$ for a delayed substitution, adding a type annotation that we use for later syntactic processing – it does not affect the semantics of substitution. When a type replacement is applied to a type variable α with a delayed substitution θ, the substitution $\pi(\theta)$ is immediately applied to $\pi(\alpha)$:

$$\pi(\theta \cdot \alpha)\ =\ \pi(\theta)(\pi(\alpha))$$

Constraint generation is defined by the judgments in Figure 5.3. Each rule is derived from the corresponding type rule in Figure 3.5 (page 33) with subsumption distributed throughout the derivation to make the rules syntax-directed.

- The judgment $\Gamma \Vdash e : T \ \&\ C$ states that under environment Γ the term e has type T, subject to constraint set C. (Figure 5.3)

- The judgment $\Gamma \Vdash T \ \&\ C$ states that under environment Γ the type T is well-formed, subject to constraint set C. (Figure 5.4)

5.2.1 Constraint Generation for $\lambda_{\mathcal{H}}^{\mathbf{Recon}}$ terms

The constraint generation judgment for $\lambda_{\mathcal{H}}^{\mathrm{Recon}}$ terms,

$$\Gamma \Vdash e : T \ \&\ C$$

Figure 5.2: Syntax for $\lambda_{\mathcal{H}}^{\text{Recon}}$ constraint generation

$e, d, f, g, q, p ::=$	*Terms:*
$\quad k$	constant
$\quad x$	variable
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad f\ e$	application
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	constant
$\quad \lambda x\!:\!S.\, e$	abstraction
$R, S, T, U ::=$	*Types:*
$\quad x\!:\!S \to T$	dependent function type
$\quad \{x\!:\!B \mid p\}$	refined basic type
$\quad \theta \cdot \alpha$	***type variable with substitution***
$\theta ::=$	*Delayed substitution*
$\quad [\,]$	empty substitution
$\quad [x \mapsto e : T] \circ \theta$	substitution extension
$\Gamma \vdash S <: T$	*Subtyping constraint*
$\Gamma \vdash T$	*Type well-formedness constraint*

Figure 5.3: Constraint Generation Rules for $\lambda_{\mathcal{H}}^{\text{Recon}}$ terms

Constraint Generation rules $\boxed{\Gamma \Vdash e : T \,\&\, C}$

$$\frac{(x : T) \in \Gamma}{\Gamma \Vdash x : T \,\&\, \emptyset}\ [\text{CG-Var}] \qquad \frac{}{\Gamma \Vdash k : ty(k) \,\&\, \emptyset}\ [\text{CG-Const}]$$

$$\frac{\Gamma \Vdash S \,\&\, C_1 \qquad \Gamma, x : S \Vdash e : T \,\&\, C_2}{\Gamma \Vdash (\lambda x{:}S.\, e) : (x{:}S \to T) \,\&\, C_1 \cup C_2}\ [\text{CG-Fun}]$$

$$\frac{\Gamma \Vdash f : T \,\&\, C_1 \qquad \Gamma \Vdash e : S \,\&\, C_2 \qquad \alpha \text{ fresh}}{\Gamma \Vdash f\ e : [x \mapsto e : S] \cdot \alpha \ \ \&\ \ C_1 \cup C_2 \cup \{\Gamma, x{:}S \vdash \alpha,\ \Gamma \vdash T <: (x{:}S \to \alpha)\}}\ [\text{CG-App}]$$

is derived from the typing judgment but, like cast insertion, is made syntax-directed by removing the subsumption rule. The use of subtyping is replaced by the emission of subtyping constraints which, when satisfied, ensure that the term is well typed.

[CG-Var]: A variable x is assigned the type to which it is bound in the environment Γ with no constraints.

[CG-Const]: A primitive k is assigned a type by ty with no constraints.

[CG-Fun]: For a λ-abstraction $\lambda x\!:\!S.\, e$ we first generate constraint sets C_1 and C_2 from its type annotation S and body e, respectively, and assign it the usual type under the union of their constraints, $C_1 \cup C_2$.

[CG-App]: Most of the interest lies in this rule for function applications $f\ e$. Given that f has type T, it may be that T is a type variable, so we cannot extract its domain type. Instead we constrain the lower bound of its domain to be S, the type of e, and its return type to be a fresh unknown type variable α. Thus the return type of the application include a delayed substitution $[x \mapsto e] \cdot \alpha$. Since any solution for α must be well-formed in the environment $\Gamma, x : S$, this type-wellformedness constraint is emitted as well.

5.2.2 Constraint Generation for $\lambda_{\mathcal{H}}^{\mathbf{Recon}}$ types

The constraint generation judgment

$$\Gamma \Vdash T \ \&\ C$$

Figure 5.4: Constraint Generation Rules for $\lambda_{\mathcal{H}}^{\text{Recon}}$ types

Well-formed Type Constraint Generation $\boxed{\Gamma \Vdash T \;\&\; C}$

$$\frac{\Gamma \Vdash S \;\&\; C_1 \qquad \Gamma, x : S \Vdash T \;\&\; C_2}{\Gamma \Vdash x{:}S \to T \;\;\&\;\; C_1 \cup C_2} \text{ [WTC-Arrow]}$$

$$\frac{\Gamma, x : B \Vdash e \;:\; \texttt{Bool} \;\&\; C}{\Gamma \Vdash \{x{:}B \mid e\} \;\&\; C} \text{ [WTC-Base]} \qquad \frac{}{\Gamma \Vdash \alpha \;\&\; \{\Gamma \vdash \alpha\}} \text{ [WTC-Var]}$$

emits in C any constraints which must be satisfied in order for T to be a well formed type in environment Γ.

[WTC-Arrow] and [WTC-Base]: Well-formedness of types is subject to those constraints that come from checking the terms which they contain.

[WTC-Var]: Type variables are well-formed subject to a new type well-formedness constraint defining the variables that may appear free in any solution. Furthermore, only type variables without a delayed substitution are allowed in the source program. This constraint could probably be relaxed but this reflects the reality that delayed substitutions are a formality not present in the statement

of the type reconstruction problem itself.

The constraint set C establishes a firm interface between this phase and the remaining phases; remaining phases need not inspect the original program at all, but merely satisfy the constraints: When applied to a typeable $\lambda_{\mathcal{H}}^{\text{Recon}}$ program, the constraint generation rules emit a satisfiable constraint set. Conversely, if the constraint set derived from a program is satisfiable, then that program is typeable.

Lemma 5.4 (Constraint Generation is Sound and Complete)

1. *For any environment* Γ *and term* e*:*

$$\exists \pi, T.\ \pi\Gamma \vdash \pi e : \pi T \qquad \Longleftrightarrow \qquad \exists \pi', S, C. \begin{cases} \Gamma \Vdash e : S \ \&\ C \\ \pi' \textit{ satisfies } C \end{cases}$$

2. *For any environment* Γ *and type* T*,*

$$\exists \pi.\ \pi\Gamma \vdash \pi T \qquad \Longleftrightarrow \qquad \exists \pi', C. \begin{cases} \Gamma \Vdash T \ \&\ C \\ \pi' \textit{ satisfies } C \end{cases}$$

Proof: We examine the details of the proof for terms only.

($\Leftarrow$) Soundness of constraint generation is simple: let $\pi = \pi'$ and let $T = S$.

($\Rightarrow$) To demonstrate completeness of constraint generation – that every typeable term generates a satisfiable constraint set – we will prove a stronger statement in order to strengthen our induction hypothesis to deal with new type variables

arising during constraint generation:

$$\exists \pi, T.\ \pi\Gamma \vdash \pi e : \pi T \quad \Longrightarrow \quad \exists \pi', S, C. \begin{cases} \Gamma \Vdash e : S \ \&\ C \\ \pi\pi' \text{ satisfies } C \\ \pi\Gamma \vdash \pi\pi' S <: \pi T \\ \pi' \text{ and } \pi \text{ have disjoint domains} \end{cases}$$

Proceeding by (mutual) induction, each case requires calculations with substitutions to produce the required π'. All the interest lies in a single case:

Case [T-App]: We have a derivation (to minimize subscripts, all metavariables used here are fresh – the "e" and "T" below are not that of the theorem but new for this proof case)

$$\frac{\dfrac{\vdots}{\pi\Gamma \vdash \pi f : \pi(x{:}T \to U)} \qquad \dfrac{\vdots}{\pi\Gamma \vdash \pi e : \pi T}}{\pi\Gamma \vdash \pi f\ \pi e : [x \mapsto \pi e : \pi T]\,(\pi U)}$$

By induction we have corresponding subderivations that we can put together with [CG-App]

$$\frac{\dfrac{\vdots}{\Gamma \Vdash f : S_f \ \&\ C_f} \qquad \dfrac{\vdots}{\Gamma \Vdash e : S_e \ \&\ C_e} \qquad \alpha \text{ fresh}}{\Gamma \Vdash f\ e : [x \mapsto e : S_e] \cdot \alpha \ \&\ C_f \cup C_e \cup \{\Gamma, x{:}S_e \vdash \alpha,\ \Gamma \vdash S_f <: x{:}S_e \to \alpha\}}$$

Our strengthened hypothesis provides us with π_e and π_f with disjoint domains that combine with π to satisfy C_e and C_f, respectively, along

with the facts

1. $\pi\Gamma \vdash \pi\pi_e S_e <: \pi T$

2. $\pi\Gamma \vdash \pi\pi_f S_f <: \pi(x{:}T \to U)$

We need to account for the fresh α and the natural choice is the corresponding type from the premise. Namely, we use this solution:

$$\pi' = [\alpha \mapsto \pi U] \circ \pi_f \pi_e$$

Since the domains of all substitutions are disjoint, this also satisfies C_e and C_f immediately, and the remaining constraint simplifies to

$$\pi\pi'\Gamma \vdash \pi'\pi S_f <: \pi\pi'(x{:}S_e \to \pi T)$$

and we can combine 1 and 2 along with the fact that subtyping is a preorder (Lemma 3.1 on page 38) to calculate that this judgment holds. Then to satisfy the strengthened induction hypothesis we also need

$$\pi\Gamma \vdash \pi'\pi([x \mapsto e : S_e] \cdot \alpha) <: [x \mapsto \pi e : \pi T]\, \pi U$$

which simplifies to

$$\pi\Gamma \vdash [x \mapsto \pi\pi' e : \pi\pi' S_e]\, \pi U <: [x \mapsto \pi e : \pi T]\, \pi U$$

in which both sides are syntactically identical since $dom(\pi')$ contains no type variables appearing in e and the type annotation on the substitution is irrelevant. □

Consider the following $\lambda_{\mathcal{H}}^{\text{Recon}}$ term e (the expression `let` $x : T = d$ `in` e is syntactic sugar for $(\lambda x{:}T.\, e)\ d$).

$$\texttt{let } id : (x{:}\alpha_1 \to \alpha_2) = \lambda x{:}\alpha_3.\, x \texttt{ in}$$

$$\texttt{let } w : \{n{:}\texttt{Int} \mid n = 0\} = 0 \texttt{ in}$$

$$\texttt{let } y : \{n{:}\texttt{Int} \mid n > w\} = 3 \texttt{ in}$$

$$id\ (id\ y)$$

Eliding some generated type variables and type well-formedness constraints for clarity, the corresponding constraint generation judgement is

$$\varnothing \Vdash e \ : \ [x \mapsto (id\ y) : \alpha_1] \cdot \alpha_2 \ \&\ C$$

where C contains the following constraints, in which $T_{id} \equiv (x : \alpha_1 \to \alpha_2)$ and the declaration of some variables (w and y in this case) is left implicit at their leftmost occurence.

$$\begin{array}{rcl}
\varnothing \vdash & x{:}\alpha_3 \to \alpha_3 & <: \ x{:}\alpha_1 \to \alpha_2 \\
id : T_{id} \vdash & \{n{:}\texttt{Int} \mid n = 0\} & <: \ \{n{:}\texttt{Int} \mid n = 0\} \\
id : T_{id},\ w = 0 \vdash & \{n{:}\texttt{Int} \mid n = 3\} & <: \ \{n{:}\texttt{Int} \mid n > w\} \\
id : T_{id},\ w = 0, y > w \vdash & \{n{:}\texttt{Int} \mid n > w\} & <: \ \alpha_1 \\
id : T_{id},\ w = 0, y > w \vdash & [x \mapsto y : \alpha_1] \cdot \alpha_2 & <: \ \alpha_1 \\
\varnothing \vdash & x{:}\alpha_1 \to \alpha_2 & \\
\varnothing \vdash & \alpha_3 &
\end{array}$$

5.3 Shape Reconstruction

The second phase of reconstruction is to infer a type's basic shape, deferring resolution of refinement predicates. To defer reconstruction of refinements, we introduce

placeholders $\psi \in \mathit{Placeholder}$ to represent unknown refinement predicates (in the same way that type variables represent unknown types). The net effect of shape reconstruction is to reduce subtyping contraints to implication constraints and type well-formdness constraints to refinement well-typedness constraints. Figure 5.5 shows the full syntax for the resulting language after shape reconstruction. During reconstruction, there may be terms from the union of the two languages, but when it terminates successfully all type variables are eliminated and all constraints are reduced.

- An implication constraint $\Gamma \vdash p \Rightarrow q$ indicates that for the program to be typeable, the corresponding implication must hold.

- A refinement well-typedness constraint $\Gamma \vdash p : \texttt{Bool}$ constrains the variables that may appear free in p.

Like type variables, each placeholder has an associated delayed substitution.

$$p \quad ::= \quad \cdots \mid \theta \cdot \psi$$

A placeholder replacement is a function $\rho : \mathit{Placeholder} \to \mathit{Term}$ and is lifted compatibly to all syntactic structures. As with type replacements, applying placeholder replacement allows any delayed substitutions also to be applied.

$$\begin{aligned} [x \mapsto e : T](\theta \cdot \psi) &= ([x \mapsto e : T], \theta) \cdot \psi \\ \rho(\theta \cdot \psi) &= \rho(\theta)(\rho(\psi)) \end{aligned}$$

For a placeholder replacement ρ, the definition of constraint satisfaction is analogous to Definition 5.3 for type replacements.

Figure 5.5: Syntax for $\lambda_{\mathcal{H}}^{\text{Recon}}$ shape reconstruction

$p, q ::=$	*Predicates:*
$\quad \theta \cdot \psi$	***placeholder with substitution***
$\quad e$	term
$e, d, f, g ::=$	*Terms:*
$\quad k$	constant
$\quad x$	variable
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad f\ e$	application
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	constant
$\quad \lambda x\!:\!S.\, e$	abstraction
$R, S, T, U ::=$	*Types:*
$\quad x\!:\!S \to T$	dependent function type
$\quad \{x\!:\!B \mid p\}$	refined basic type
$\theta ::=$	*Delayed substitution*
$\quad [\,]$	empty substitution
$\quad [x \mapsto e : T] \circ \theta$	substitution extension
$\Gamma \vdash p \Rightarrow q$	*Implication constraint*
$\Gamma \vdash p$: `Bool`	*Refinement well-typedness constraint*

Definition 5.5 *A placeholder replacement* ρ satisfies *a constraint set* P *if and only if* $\rho(P)$ *contains only valid implications and typing judgments.*

The shape reconstruction algorithm, detailed in Figure 5.6 takes as input a subtyping constraint set C and processes the constraints in C nondeterministically according to the rules in Figure 5.6. When the conditions on the left-hand side of a rule are satisfied, the updates described on the right-hand side are performed. The set P of implication constraints, each of the form $\Gamma \vdash p \Rightarrow q$, and the type replacement π are outputs of the algorithm. Each rule in Figure 5.6 resembles a step of traditional type reconstruction; this algorithm is presented as formal evidence that this step of reconstruction can, in fact, be separated from any reasoning about refinement predicates.

1. When a type variable α must have the shape of a function type, it is replaced by $x : \alpha_1 \to \alpha_2$, where α_1 and α_2 are fresh type variables. The standard function *occurs* checks that α has a finite solution, since $\lambda_{\mathcal{H}}^{\text{Recon}}$ does not have recursive types. Occurences of α which appear in refinement predicates or in the range of a delayed substitution are ignored – these occurences do not require a solution involving recursive types.

$$
\begin{aligned}
occurs(\alpha, \{x{:}B \mid e\}) &\stackrel{\text{def}}{=} false \\
occurs(\alpha, \theta \cdot \alpha') &\stackrel{\text{def}}{=} false \quad (\alpha \neq \alpha') \\
occurs(\alpha, \theta \cdot \alpha) &\stackrel{\text{def}}{=} true \\
occurs(\alpha, x{:}S \to T) &\stackrel{\text{def}}{=} occurs(\alpha, S) \vee occurs(\alpha, T)
\end{aligned}
$$

Figure 5.6: Shape Reconstruction Algorithm

Input: C
Output: π, P
Initially: $P = \emptyset$ and $\pi = [\,]$
match some constraint in C, removing it, until quiescent:

1. $\Gamma \vdash \theta \cdot \alpha <: x{:}T_1 \to T_2$ or $\Gamma \vdash x{:}T_1 \to T_2 <: \theta \cdot \alpha$ $\Longrightarrow$ if $occurs(\alpha, x{:}T_1 \to T_2)$ then $fail$ otherwise for fresh α_1, α_2 $\pi := [\alpha \mapsto (x{:}\alpha_1 \to \alpha_2)] \circ \pi$ $C := \pi(C)$ $P := \pi(P)$

2. $\Gamma \vdash \theta \cdot \alpha <: \{x{:}B \mid e\}$ or $\Gamma \vdash \{x{:}B \mid e\} <: \theta \cdot \alpha$ $\Longrightarrow$ for fresh ψ $\pi := [\alpha \mapsto \{x{:}B \mid \psi\}] \circ \pi$ $C := \pi(C)$ $P := \pi(P)$

3. $\Gamma \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)$ $\Longrightarrow$ $C := C \cup \left\{ \begin{array}{l} \Gamma \vdash T_1 <: S_1 \\ \Gamma, x : T_1 \vdash S_2 <: T_2 \end{array} \right\}$

4. $\Gamma \vdash \{x{:}B \mid p\} <: \{x{:}B \mid q\}$ $\Longrightarrow$ $P := P \cup \{\Gamma \vdash p \Rightarrow q\}$

5. $\Gamma \vdash (x{:}S \to T)$ $\Longrightarrow$ $C := C \cup \left\{ \begin{array}{l} \Gamma \vdash S \\ \Gamma, x : S \vdash T \end{array} \right\}$

6. $\Gamma \vdash \{x{:}B \mid p\}$ $\Longrightarrow$ $P := P \cup \{\Gamma, x : B \vdash p : \texttt{Bool}\}$

7. $\Gamma \vdash \{x{:}B \mid p\} <: x{:}S \to T$ or $\Gamma \vdash x{:}S \to T <: \{x{:}B \mid p\}$ or $\Gamma \vdash \{x{:}B \mid p\} <: \{x{:}B' \mid q\}$ $(B \neq B')$ $\Longrightarrow$ *fail*

2. When a type variable must be a refinement of a base type B, the type variable is replaced by $\{x{:}B \mid \psi\}$ where ψ is a fresh placeholder.

3. A subtyping constraint between two function types induces additional constraints between the domains and codomains of the function types.

4. A subtyping constraint between two refined base types is simplified to an implication constraint between their refinements.

5. A type well-formedness constraint on a function type is decomposed into type well-formedness constraints on both of its components.

6. A type well-formedness constraint on a refined base type is replaced by the equivalent refinement well-typedness constraint.

7. Whenever two types have incompatible shapes, the algorithm fails immediately.

The algorithm terminates once no more progress can be made. At this stage, any type variables remaining in $\pi(C)$ are not constrained to be subtypes of any concrete type but may be subtypes of each other. We set these type variables equal to an arbitrary concrete type to eliminate them (the resulting subtyping judgements are trivial by reflexivity).

For a set of subtyping constraints C, one of the following occurs:

1. Shape reconstruction fails, in which case C is unsatisfiable, or

2. Shape reconstruction succeeds, yielding π and P. Then P is satisfiable if and only if C is satisfiable. Furthermore, if ρ satisfies P then $\rho \circ \pi$ satisfies C.

More formally,

Lemma 5.6 (Shape Reconstruction is Sound and Complete)

For any set of subtyping constraints C,

$$\exists \pi.\ \pi \textit{ satisfies } C \iff \exists \pi', P, \rho. \begin{cases} \pi', P \textit{ result from shape reconstruction} \\ \rho \textit{ satisfies } P \end{cases}$$

PROOF: Each step of the algorithm maintains the invariant that C and P are equi-satisfiable with their prior state, so consider only the case where shape reconstruction fails. The algorithm can only fail when two types are utterly incompatible, and by the invariant this constraint originates from a constraint in the original set of subtyping constraints. □

Returning to our example, shape reconstruction returns the type replacement

$$\pi = [\ \alpha_1 \mapsto \{n\!:\!\texttt{Int} \mid \psi_1\},\ \alpha_2 \mapsto \{n\!:\!\texttt{Int} \mid \psi_2\},\ \alpha_3 \mapsto \{n\!:\!\texttt{Int} \mid \psi_3\}\ ]$$

and the following implication constraint set P, where now $T_{id} = x : \{n\!:\!\texttt{Int} \mid \psi_2\} \to \{n\!:\!\texttt{Int} \mid \psi_3\}$, noting that the type variables have been elaborated into refinement types

containing placeholders.

$$
\begin{array}{rll}
n : \texttt{Int} & \vdash & \psi_1 \Rightarrow \psi_3 \\
x : \{n{:}\texttt{Int} \mid \psi_1\},\ n : \texttt{Int} & \vdash & \psi_3 \Rightarrow \psi_2 \\
id : T_{id},\ n : \texttt{Int} & \vdash & (n = 0) \Rightarrow (n = 0) \\
id : T_{id},\ w = 0,\ n : \texttt{Int} & \vdash & (n = 3) \Rightarrow (n > w) \\
id : T_{id}, w = 0,\ y > w,\ n : \texttt{Int} & \vdash & (n > w) \Rightarrow \psi_1 \\
id : T_{id}, w = 0,\ y > w,\ n : \texttt{Int} & \vdash & [x \mapsto y : \{n{:}\texttt{Int} \mid \psi_1\}] \cdot \psi_2 \Rightarrow \psi_1 \\
n : \texttt{Int} & \vdash & \psi_1 \ : \ \texttt{Bool} \\
n : \texttt{Int}, x : \{n{:}\texttt{Int} \mid \psi_1\} & \vdash & \psi_2 \ : \ \texttt{Bool} \\
n : \texttt{Int} & \vdash & \psi_3 \ : \ \texttt{Bool}
\end{array}
$$

5.4 Strongest Refinements

The final phase of type reconstruction solves the residual implication constraint set P by finding a placeholder replacement that preserves satisfiability.

Our approach is based on the intuition – temporarily ignoring the complication of dependent types – that implications are essentially dataflow paths that carry the specifications of data sources (constants and function postconditions) to the requirements of data sinks (function preconditions), with placeholders functioning as intermediate nodes in the dataflow graph. Thus, if a placeholder ψ appears on the right-hand side of two implication constraints $\Gamma \vdash p \Rightarrow \psi$ and $\Gamma \vdash q \Rightarrow \psi$, then our replacement for ψ is simply the disjunction $p \vee q$ (the strongest consequence) of these two lower bounds. Our algorithm repeatedly applies this transformation until no placeholders remain, but

several difficulties arise that are not present in the simpler first-order setting of SP:

1. p or q may contain variables that cannot appear in a solution for ψ
2. ψ may have a delayed substitution
3. ψ may appear in p or q

To help resolve these issues, we extend the language of terms with parallel boolean connectives and existential quantification and will utilize the fixed point operator assumed to be included in the primitives of the language.

$$e \quad ::= \quad \cdots \quad | \quad e \vee e \quad | \quad e \wedge e \quad | \quad \exists x : T.\ e$$

Figure 5.7 summarizes the syntax of the final language in which the output of type reconstruction lies, and Figure 5.8 presents the operational semantics of the parallel boolean connectives.

- The parallel disjunction $e_1 \vee e_2$ (respectively conjunction $e_1 \wedge e_2$) evaluates e_1 and e_2 nondeterministically, reducing to `true` (resp. `false`) if either of them reduces to `true` (resp. `false`).

- The existential term $\exists x : T.\ e$ binds x in e, and evaluates by nondeterministically replacing x with a closed term of type T. The evaluation rules are summarized in Figure 5.8. Note that neither parallel boolean connective (nor, obviously, existential quantification) can be encoded via β or δ reduction.[3]

[3]In synthetic topology [Escardó 2004], where open sets correspond to semidecidable predicates very

Figure 5.7: Syntax for $\lambda_{\mathcal{H}}^{\text{Recon}}$ with strongest refinements

$e, d, f, g, q, p ::=$	*Terms:*
k	constant
$\lambda x\!:\!S.\, e$	abstraction
x	variable
$f\ e$	application
$\exists x : T.\ p$	***existential quantification***
$e \vee e$	***parallel disjunction***
$e \wedge e$	***parallel conjunction***
$v ::=$	*Values: ($\subset$ Terms)*
k	constant
$\lambda x\!:\!S.\, e$	abstraction
$R, S, T, U ::=$	*Types:*
$x\!:\!S \to T$	dependent function type
$\{x\!:\!B \mid p\}$	refined basic type

Figure 5.8: Additional Redex Reduction for Parallel Logical Connectives

$$\texttt{true} \vee e \ \curvearrowright\ \texttt{true} \quad \text{[E-Or-L]} \qquad \texttt{false} \wedge e \ \curvearrowright\ \texttt{false} \quad \text{[E-And-L]}$$

$$e \vee \texttt{true} \ \curvearrowright\ \texttt{true} \quad \text{[E-Or-R]} \qquad e \wedge \texttt{false} \ \curvearrowright\ \texttt{false} \quad \text{[E-And-R]}$$

$$\texttt{false} \vee \texttt{false} \ \curvearrowright\ \texttt{false} \quad \text{[E-Or-F]} \qquad \texttt{true} \wedge \texttt{true} \ \curvearrowright\ \texttt{true} \quad \text{[E-And-T]}$$

$$\exists x : T.\ p \ \curvearrowright\ [x \mapsto e : T]\, p \quad \text{if } \varnothing \vdash e\ :\ T \quad \text{[E-Exists]}$$

$$\mathcal{C} \ ::= \ \cdots \mid e \vee \mathcal{C} \mid \mathcal{C} \vee e \mid \mathcal{C} \wedge e \mid e \wedge \mathcal{C} \mid \exists x : \mathcal{D}.\ p \mid \exists x : T.\ \mathcal{C}$$

5.4.1 Free Variable Elimination

The immediate lower bounds for a placeholder may contain free variables that may not appear free in any solution. In our example program, the type variable α_1 appeared in the empty environment and $\pi(\alpha_1) = \{n{:}\texttt{Int} \mid \psi_1\}$, yielding a refinement well-typedness constraint $n{:}\texttt{Int} \vdash \psi_1 \ : \ \texttt{Bool}$. The only variable that can appear in a solution for ψ_1 is therefore n. But consider the following lower bound for ψ_1:

$$id : T_{id},\ w = 0,\ y > w,\ n : \texttt{Int} \vdash (n > w) \Rightarrow \psi_1$$

Since id, w, and y cannot appear in a solution for ψ_1, we need a way to identify and remove them from the lower bound without changing its logical interpretation.

In general, each placeholder ψ introduced by shape reconstruction has an associated environment Γ_ψ in which it must have type `Bool`, recorded in the refinement well-typedness constraints. This gives us a reasonable definition for the free variables of a placeholder (with its associated delayed subtitution):

$$fv(\theta \cdot \psi) = (dom(\Gamma_\psi) \setminus dom(\theta)) \cup fv(rng(\theta))$$

We then rewrite each implication constraint $\Gamma, y : T \vdash p \Rightarrow q$ where $y \notin fv(q)$ into the constraint $\Gamma \vdash (\exists y : T.\ p) \Rightarrow q$. This corresponds to the following tautology[4]

much like executable refinement types, the "parallel or" connective is required in order for the usual defining property of a topology to apply. Synthetic topology uses a single-valued boolean, denoting falsity by nontermination. Because we denote falsity by termination at `false` we also require "parallel and". Our inputs are essentially open sets, though our syntactic approach obscures this to the point where formally establishing the connection must be left for future research, and our output must also be an open set, which we construct by union and intersection. Further, any *overt* data type – dual to *compact* – is equipped with a semidecidable existential quantifier, which would become relevant if existential quantification were ever to be supported in runtime checks.

[4] A "De Morgan law" for quantification.

$(\forall x.\, p \Rightarrow q) \Leftrightarrow ((\exists x.p) \Rightarrow q)$ when $x \notin fv(q)$; the executable existential is carefully defined so that the tautology holds.

Thus we rewrite the constraint

$$id : T_{id},\ w = 0,\ y > w,\ n : \texttt{Int} \vdash (n > w) \Rightarrow \psi_1$$

to the new constraint

$$n : \texttt{Int} \vdash (\exists id : T_{id}.\ \exists w : w = 0.\ \exists y : y > w.\ n > w) \Rightarrow \psi_1$$

This transformation is semantics-preserving:

Lemma 5.7 (Correctness of Free Variable Elimination) *For* $y \notin fv(q)$, $\Gamma, y : T \vdash p \Rightarrow q$ *if and only if* $\Gamma \vdash (\exists y : T.\ p) \Rightarrow q$

Proof: Consider each direction of entailment separately.

($\Rightarrow$) Suppose $\Gamma, y : T \vdash p \Rightarrow q$ and $y \notin fv(q)$ and consider any σ such that $\vdash \sigma : \Gamma$ and $\sigma(\exists y : T.\ p) \rightsquigarrow^* \texttt{true}$. Then there is some e such that $\varnothing \vdash e : T$ and $\sigma(\exists y : T.\ p) \rightsquigarrow \sigma([x \mapsto e : T]p) \rightsquigarrow^* \texttt{true}$.

Let $\sigma' = \sigma \circ [y \mapsto e]$ so that $\vdash \sigma' : \Gamma, y : T$. Then by assumption, $\sigma(p) \rightsquigarrow^* \texttt{true}$ implies $\sigma(q) \rightsquigarrow^* \texttt{true}$. Since $y \notin fv(q)$, $\sigma'(q) = \sigma(q)$, and we have proved $\Gamma \vdash \exists y : T.\ p \Rightarrow q$.

($\Leftarrow$) Conversely, suppose $\Gamma \vdash \exists y : T.\ p \Rightarrow q$ where $y \notin fv(q)$ and consider σ such that $\vdash \sigma : \Gamma, y : T$ and $\sigma(p) \rightsquigarrow^* \texttt{true}$.

Since $y \notin dom(\Gamma)$ we can ignore it, so $\vdash \sigma : \Gamma$, and the nondeterministic existential can replace y with $\sigma(y)$, so $\sigma(\exists y : T.\ p) \rightsquigarrow^* \sigma(p) \rightsquigarrow^* \texttt{true}$. By assumption, this implies that $\sigma(q) \rightsquigarrow^* \texttt{true}$, and we have proved $\Gamma, y : T \vdash p \Rightarrow q$. $\square$

Repeatedly applying this transformation, we rewrite each implication constraint until the domain of the environment (and hence the free variables of the left-hand side) is a subset of the free variables of the right-hand side.

5.4.2 Delayed Substitution Elimination

At this stage, all constraints are of the form

$$\Gamma \vdash p \Rightarrow \theta \cdot \psi$$

where $dom(\Gamma) \subseteq fv(\theta \cdot \psi)$ The next issue is the presence of delayed substitutions in constraints. To eliminate the delayed substitution θ we add the variables in its domain to the environment and encode its semantic content as an additional predicate. Formally,

$$\begin{array}{rcl@{\qquad}rcl} env([\,]) & = & \varnothing & pred(\ [\,]\ ) & = & \texttt{true} \\ env([x \mapsto e : T], \theta) & = & x : T, env(\theta) & pred(\ [x \mapsto e : T], \theta\ ) & = & (x = e) \wedge pred(\theta) \end{array}$$

The environment $env(\theta)$ binds all the variables in $dom(\theta)$ while the term $pred(\theta)$ represents the semantic content of θ.

We then transform the constraint

$$\Gamma \vdash p \Rightarrow \theta \cdot \psi$$

into the equivalent constraint

$$\Gamma, env(\theta) \vdash pred(\theta) \wedge p \Rightarrow \psi.$$

But we can rewrite the constraint even more cleanly: Γ must be some prefix of Γ_ψ since by the previous transformation $dom(\Gamma) \subseteq fv(\theta \cdot \psi) \subseteq dom(\Gamma_\psi)$. Any $x \in dom(\theta)$ such that $x \notin dom(\Gamma_\psi)$ can be dropped from θ and we see that $\Gamma, env(\theta)$ is then exactly Γ_ψ. So our constraint is

$$\Gamma_\psi \vdash pred(\theta) \wedge p \Rightarrow \psi$$

To prove this transformation correct, we use the following well formedness judgement $\Gamma \vdash_{\text{wf}} \theta$ which distinguishes those delayed substitutions that may actually occur in context Γ.

$$\frac{}{\Gamma \vdash_{\text{wf}} [\,]}\ [\text{WF-EMPTY}] \qquad \frac{\Gamma \vdash e : T \qquad \Gamma, x : T \vdash_{\text{wf}} \theta'}{\Gamma \vdash_{\text{wf}} [x \mapsto e : T] \circ \theta'}\ [\text{WF-EXT}]$$

Lemma 5.8 (Correctness of Substitution Elimination) *Suppose ρ is a placeholder replacement such that $\rho(\Gamma) \vdash_{\text{wf}} \rho(\theta)$. Then ρ satisfies $\Gamma \vdash p \Rightarrow \theta \cdot \psi$ if and only if ρ satisfies $\Gamma, env(\theta) \vdash pred(\theta) \wedge p \Rightarrow \psi$.*

PROOF: Assume throughout that $\rho\Gamma \vdash_{\text{wf}} \rho\theta$

($\Rightarrow$) Suppose ρ satisfies $\Gamma \vdash p \Rightarrow \theta \cdot \psi$, i.e. $\rho\Gamma \vdash \rho p \Rightarrow \rho\theta(\rho\psi)$ and consider any σ s.t. $\vdash \sigma : \rho\Gamma, env(\rho\theta)$ and $\sigma(\rho(pred(\theta) \wedge p)) \rightsquigarrow^* \texttt{true}$. Then by definition of the conjunction we know that $\sigma(\rho p) \rightsquigarrow^* \texttt{true}$, hence $\sigma(\rho(\theta \cdot \psi)) \rightsquigarrow^* \texttt{true}$ by assumption, which is $\sigma(\rho\theta(\rho\psi)) \rightsquigarrow^* \texttt{true}$.

Thus since σ is a closing substitution for $\rho(\Gamma, env(\theta))$, we know for any $x \in dom(\theta)$ that $\sigma(x) = (\rho\theta)(x)$ hence $\sigma(\rho\psi) = \sigma(\rho\theta(\rho\psi))$, which we already know evaluates to `true`. Hence ρ satisfies $\Gamma, env(\theta) \vdash pred(\theta) \wedge p \Rightarrow \psi$.

($\Leftarrow$) Suppose ρ satisfies $\Gamma, env(\theta) \vdash pred(\theta) \wedge p \Rightarrow \psi$ i.e. $\rho\Gamma, env(\rho\theta) \vdash \rho(pred(\theta) \wedge p) \Rightarrow \rho\psi$ and consider any σ s.t. $\vdash \sigma : \rho\Gamma$ and $\sigma(\rho p) \rightsquigarrow^*$ `true`. Let $\sigma' = \sigma \circ (\rho\theta)$; note that $\vdash \sigma' : \rho\Gamma, env(\rho\theta)$.

Obviously, $\sigma'(\rho p) \rightsquigarrow^*$ `true` since $dom(\theta) \cap fv(p) = \emptyset$. Furthermore, by intensional equality, we see that $\sigma'(pred(\theta)) \rightsquigarrow^*$ `true`. So the conjunction in our assumption evaluates to `true`, and we infer that $\sigma'(\rho\psi) \rightsquigarrow^*$ `true`.

And $\sigma'(\rho\psi) = \sigma(\rho\theta(\rho\psi))$ which is exactly what we need to conclude that ρ satisfies $\Gamma \vdash p \Rightarrow \theta \cdot \psi$ $\square$

5.4.3 Placeholder Solution

After the previous transformations, all lower bounds of a placeholder ψ appear in constraints of the form

$$\Gamma_\psi \vdash p_i \Rightarrow \psi$$

for $i \in \{1..n\}$, assuming ψ has n lower bounds. We want to set ψ equal to the parallel disjunction $p_1 \vee p_2 \vee \cdots \vee p_n$ of all its lower bounds (the disjunction must be parallel because some subterms may be nonterminating). However, ψ may appear in some p_i due to recursion or self-composition of a function. In this case we use a least fixed point operator, conveniently already available in our language, to find a solution to the

Figure 5.9: Shorthands for $\lambda^{\text{Recon}}_{\mathcal{H}}$

$$\overline{T} \to \texttt{Bool} \quad \stackrel{\text{def}}{=} \quad T_1 \to T_2 \to \cdots \to T_k \to \texttt{Bool}$$

$$\lambda\overline{x{:}T}.\, e \quad \stackrel{\text{def}}{=} \quad \lambda x_1{:}T_1.\, \lambda x_2{:}T_2.\, \cdots \lambda x_k{:}T_k.\, e$$

$$f\ \overline{x} \quad \stackrel{\text{def}}{=} \quad f\ x_1\ x_2\ \cdots\ x_k$$

equation $\psi = p_1 \vee \cdots \vee p_n$.

More formally, suppose $\Gamma_\psi = x_1 : T_1, \cdots, x_k : T_k$. Then ψ is a predicate over $x_1 \cdots x_k$ and we can interpret it as a function $\mathcal{F}_\gamma : T_1 \to \cdots \to T_k \to \texttt{Bool}$ and give a recursive definition for the function. Using the shorthands of Figure 5.9 the function $\mathcal{F}_\psi$ can be defined as the following least fixed point computation:

$$\mathcal{F}_\psi \ = \ \texttt{fix}_{\overline{T}\to\texttt{Bool}}\ (\lambda f{:}\overline{T} \to \texttt{Bool}.\, \lambda\overline{x{:}T}.\, [\psi \mapsto f\ \overline{x}](p_1 \vee \cdots \vee p_n))$$

Our solution for ψ is $SR(\psi) = \mathcal{F}_\psi\ \overline{x}$. This is the *strongest refinement* (SR) that is implied by all lower bounds of ψ. This fixed-point calculation will terminate immediately in the event that ψ does not appear in any of its lower bounds. When the program is recursive, the presence of a fixed-point calculation is expected. Applying existing and new techniques to this carefully isolated subproblem remains an open and fruitful area for future research, but has already borne fruit: Subsequent to this work, Rondon et al. [2008] applied predicate abstraction to calculate enough of these fixed points that the program annotations to ensure fully static type-checking were reduced from 31% of the text to less than 1%.

Lemma 5.9 (SR is the Strongest Refinement)

If ρ satisfies P, then ρ satisfies $\Gamma_\psi \vdash SR(\psi) \Rightarrow \psi$.

PROOF: Consider any ρ satisfying P and σ such that $\vdash \sigma : \rho\Gamma_\psi$ and $\sigma\rho(SR(\psi)) \rightsquigarrow^* \texttt{true}$ minimal k. Since the possible nondeterministic evaluations of $\sigma\rho SR(\psi)$ correspond to simultaneously evaluating all lower bounds for ψ simultaneously, one of them must eventually evaluate to `true`. Then by the assumption that ρ satisfies P, we know that $\sigma\rho\psi \rightsquigarrow^* \texttt{true}$. □

The result of equisatisfiability follows from the fact that we have chosen the strongest possible solution for ψ.

Lemma 5.10 (Correctness of Strongest Refinement Calculation)

P is satisfiable if and only if $P[\psi := SR(\psi)]$ is satisfiable.

PROOF:

($\Leftarrow$) For any $\rho : PlaceHolders \rightarrow Terms$ that satisfies $P[\psi := SR(\psi)]$, we have $\rho \circ [\psi := SR(\psi)]$ that satisfies P.

($\Rightarrow$) Consider any $\rho : PlaceHolders \rightarrow Terms$ that satisfies P. By Lemma 5.9 if $\rho(\psi) \Rightarrow p$ occurs in P, then $SR(\psi) \Rightarrow \rho(\psi) \Rightarrow p$; covariant occurences of ψ in environments are analogous. If $p \Rightarrow \rho(\psi)$ occurs in P, then $p \Rightarrow SR(\psi)$ by construction of $SR(\psi)$; contravariant occurences of types in environments do not affect satisfiability. □

In our example, the only lower bound of ψ_3 is ψ_1 and the only lower bound of ψ_2 is ψ_3, so let us set $\psi_3 := \psi_1$ and $\psi_2 := \psi_3$ in order to discuss the more interesting solution for ψ_1. The resulting unsatisfied constraints (simplified for clarity) are:

$$n : \texttt{Int} \ \vdash \ \exists w : \{n{:}\texttt{Int} \mid n = 0\}.\ (n > w) \ \Rightarrow \ \psi_1$$

$$n : \texttt{Int} \ \vdash \quad \exists w : w = 0.\ \exists y : y > w.\ [x \mapsto y] \cdot \psi_1 \Rightarrow \psi_1$$

The exact text of $SR(\psi_1)$ is too large to print here, but it is equivalent to $\exists w \ : \ w = 0.\ n > w$ and thus equivalent to $(n \ > \ 0)$. The resulting explicitly typed program (simplified accordingly) is:

$\texttt{let}\ id : (x{:}\{n{:}\texttt{Int} \mid n > 0\} \to \{n{:}\texttt{Int} \mid n > 0\}) = \lambda x{:}\{n{:}\texttt{Int} \mid n > 0\}.\, x\ \texttt{in}$

$\texttt{let}\ w : \{n{:}\texttt{Int} \mid n = 0\} = 0\ \texttt{in}$

$\texttt{let}\ y : \{n{:}\texttt{Int} \mid n > w\} = 3\ \texttt{in}$

$id\ (id\ y)$

5.5 Type Reconstruction is Typability-Preserving

The output of our algorithm is the composition of the type replacement returned by shape reconstruction and the placeholder replacement returned by the satisfiability routine. Application of this composed replacement is a typeability-preserving transformation. Moreover, for any typeable program, the algorithm succeeds in producing such a replacement.

Theorem 5.11 *For any $\lambda_{\mathcal{H}}^{Recon}$ program e, one of the following occurs:*

1. *Type reconstruction fails, in which case e is untypeable, or*

2. *Type reconstruction returns a type replacement π such that e is typeable if and only if $\pi(e)$ is well typed.*

PROOF:

1. Only shape reconstruction can fail. If it does, then by Lemma 5.6 the subtyping constraints are unsatisfiable. Then by Lemma 5.4, e is not typeable.
2. Type reconstruction solved constraints that were faithful, by Lemma 5.4. Thus by Lemma 5.6 we have π and by Lemma 5.10 we have ρ such that $\rho\pi e$ is typeable (well typed) if and only if e is typeable. □

5.6 Related Work

Constraint-based type reconstruction for systems with subtyping is a tremendously broad topic that we cannot fully review here. The problem is studied in some generality by Mitchell [1984], Fuh and Mishra [1988], Lincoln and Mitchell [1992], Aiken and Wimmers [1993], and Hoang and Mitchell [1995]. Type inference systems parameterized by a subtyping constraint system are developed by Pottier [1996] and Odersky et al. [1999]. This work is complementary to generalized systems in that it focuses on the solution of our particular instantiation of subtyping constraints; we also do not investigate parametric polymorphism, which is included in the mentioned frameworks. Set-based analysis presents many similar ideas, and this chapter also draws inspiration from the works of Heintze [1992], Cousot and Cousot [1995], Fahndrich and Aiken [1996], and Flanagan and Felleisen [1997].

Many of the refinement type systems discussed in Section 1.1.2 perform some manner of local type reconstruction [Pierce and Turner 1998; Dunfield and Pfenning 2004], but require type annotations to guide the type checking process, and none address the general problem of the satisfaction of a set of subtyping constraints between refinement types. Dependent ML Xi and Pfenning [1999] solves systems of linear inequalities to infer a restricted class of type indices.

Liquid types [Rondon et al. 2008], discussed at length in Section 8.2.2 take the solution presented here a step further by using abstract interpretation to solve for closed forms of fixed points such as those produced by our algorithm. The particular abstract interpretation used is *predicate abstraction* where a finite set of predicate templates are instantiated with the variables of the program, so common predicates for a given domain can be outlined and then the system performs full type reconstruction. If a solution cannot be found by the predicate abstraction process, then the program is rejected. A great benefit of Liquid Types is that the predicates inferred can be very readable as they come from a small set of templates and do not include fixed points or existential quantification. Our deliberate separation of the *generation* of the fixed point constraints from the *solving* of them for a closed form leaves other avenues open. For example, it is not generally necessary to find a closed form solution except for nice error messages: our reconstructed predicates are never used as preconditions, but only to satisfy other preconditions, so strategic unrolling and approximation may be sufficient to perform (possibly hybrid) type checking of the resultant program.

Rastogi et al. [2012] developes a type inference system for the much more com-

plex context of Actionscript and evaluate the performance asymptotically and through an implementation with benchmarks time trials.

Chapter 6

Compositional and Decidable Checking

Having established that executable refinement types can be effectively reconstructed from a program with no type annotations and subsequently enforced using hybrid type checking, we return to the question of when – if ever – we can predict that the type checking question *for a particular program* will be decidable. Note that this is distinct from, though related to, the design of a decidable restriction of the type system.

Unfortunately, the essential difference between dependent types and simple structural type systems prevents a satisfactory answer. Simple type systems such as that of $STLC$ perform *compositional reasoning* in that the type of a term depends only on the types of its subterms, not on their semantics. Dependent types offer more expressive abstractions, but violate those abstractions and base their reasoning directly upon the semantics of terms.[1] Pragmatically, noncompositionality makes the decidability of static checking unpredictable.

[1]This is, in fact, the origin of the term "dependent type": the types *depend* on terms.

This chapter shows how compositional reasoning may be restored using standard type-theoretic techniques, namely a form of existential types and subtyping. Despite its compositional nature, the type system is *exact*, in that the type of a term can completely capture its semantics, thus demonstrating that precision and compositionality are compatible. We then address predictability of static checking for executable refinement types by giving a type-checking algorithm for an important class of programs with refinement predicates drawn from a decidable theory. Our algorithm relies crucially on the fact that the type of a term depends only the types of its subterms (which fall into the decidable theory) and not their semantics (which will not, in general).

Chapter Outline

This chapter is organized as follows:

- Section 6.1 clarifies the non-compositionality of dependent type systems and its consequences for executable refinement types.
- Section 6.2 Describes $\lambda_{\mathcal{H}}^{\text{Comp}}$, a variant of $\lambda_{\mathcal{H}}$ augmented to support compositional reasoning.
- Section 6.3 provides an operational semantics to $\lambda_{\mathcal{H}}^{\text{Comp}}$.
- Section 6.5 presents the $\lambda_{\mathcal{H}}^{\text{Comp}}$ type system and proves its compositionality.
- Section 6.7 proves that the $\lambda_{\mathcal{H}}^{\text{Comp}}$ type system is *exact*: Whenever primitives have suitably precise types, every term may be assigned a type that precisely describes its semantics.

- Section 6.8 proves type soundness for $\lambda_{\mathcal{H}}^{\text{Comp}}$, relying crucially on exactness.

- Section 6.6 extends the definition of extensional equivalence for $\lambda_{\mathcal{H}}$ to $\lambda_{\mathcal{H}}^{\text{Comp}}$

- Section 6.9 provides a type checking algorithm for the important special case of programs whose refinement predicates fall in a decidable theory.

- Section 6.10 demonstrates the verification in $\lambda_{\mathcal{H}}^{\text{Comp}}$ of the classic example of binary search trees.

- Section 6.11 discusses work with related goals and surveys other applications of similar type-theoretic techniques.

6.1 Compositional Reasoning

During program development, compositional reasoning separates a program into cognitively manageable pieces. During analysis and verification, compositional reasoning limits the amount of information that must be stored and processed. Before we embark upon a formal definition, consider the following illustration of how structural type systems reason compositionally. In a well-typed program such as

$$\texttt{let}\ x : T = e\ \texttt{in}\ d$$

the type T provides an abstract specification of e that is concise and yet sufficiently precise to verify that e and d interact properly. This verification process is compositional in that the verification of d cannot rely on any properties of e other than those exposed via its type T; the analysis of the superterm relies only on the results of analysis of the

subterm(s). Consequently, replacing e with any equivalently-typed term does not affect the typeability of the overall program. Formally, we can define compositionality of any type system via its type rules.[2]

Definition 6.1 (Compositional Reasoning) *A type rule* [A] *performs* compositional reasoning *if for any derivation* $\Gamma \vdash e : S$ *used in the premise, as in*

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash e : S} \qquad \textit{possibly other premises}}{\Gamma' \vdash \mathcal{C}[e] : T}\ [\mathrm{A}]$$

any derivation of $\Gamma \vdash d : S$ *may be used in place, as in*

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash d : S} \qquad \textit{possibly other premises}}{\Gamma' \vdash \mathcal{C}[d] : T}\ [\mathrm{A}]$$

Such a rule is said to be compositional. A type system containing only compositional rules is also called compositional.

Seen through the lens of the Curry-Howard-Lambek correspondence [Curry 1934; Curry and Feys 1958; Howard 1980; Lambek and Scott 1986] this corresponds to a sort of proof irrelevance: The particular proof of a proposition cannot affect how that property is invoked in proofs of other propositions.

[2]This assumes that a type rule does not inspect the derivation of its antecedents, but only their content. This is, in fact, compositionality (proof irrelevance) of the metatheory.

6.1.1 Dependent Types Perform Non-compositional Reasoning

Executable refinement types provide a natural means to abstract a term's behavior to a greater or lesser degree (or, in the extreme, not abstract its behavior at all). For example, the constant 1 may be given an infinite number of types, including `Int`, `Pos` and the exact type $\{y{:}\texttt{Int} \mid y = 1\}$. These types are related via subtyping in the expected manner:

$$\{y{:}\texttt{Int} \mid y = 1\} \quad <: \quad \texttt{Pos} \quad <: \quad \texttt{Int}$$

As demonstrated throughout this dissertation, executable refinement types cooperate in a clean and expressive manner with dependent function types. For example, the function `add1` may also be given many types, including the simple type $\texttt{Int} \to \texttt{Int}$, the an exact type $x : \texttt{Int} \to \{y{:}\texttt{Int} \mid y = x + 1\}$, or the intermediate specification $x : \texttt{Int} \to \{y{:}\texttt{Int} \mid y > x\}$ describing only that `add1` is increasing. Again, subtyping appropriately relates these various specifications for `add1`:

$$\begin{aligned} & x{:}\texttt{Int} \to \{y{:}\texttt{Int} \mid y = x + 1\} \\ <: \quad & x{:}\texttt{Int} \to \{y{:}\texttt{Int} \mid y > x\} \\ <: \quad & \texttt{Int} \to \texttt{Int} \end{aligned}$$

At first, it appears that a programmer has precise control, via type annotations, over the specifications that must be verified during type-checking. Unfortunately, this is not the case at all! Standard dependent type systems, such as those of Cayenne [Augustsson 1998], Epigram [McBride and McKinna 2004], Agda [Norell 2009] and Coq [The Coq development team 2012], when given a program $\mathcal{C}[e]$, reason about the

interaction between the term e and its context $\mathcal{C}$ using both:

- compositional reasoning based on the *type* of e; and

- non-compositional reasoning based on the *syntax* of e (hence its behavior).

The non-compositional nature of dependent type systems originates from the following standard type rule for dependent function application. (For simplicity, we elide the typing environment).

$$\frac{f \ : \ (x{:}S \rightarrow T) \qquad e \ : \ S}{f \ e \ : \ [x \mapsto e]\, T}$$

The inferred type $[x \mapsto e]\, T$ (denoting T with x replaced by e) of $f\ e$ includes the term e itself, and not just its type S.

To highlight the consequences of the non-compositional nature of this rule, suppose that e is an arbitrary term of type `Pos`, and that f is the exactly-typed identity function on integers:

$$\begin{aligned} e \quad &: \quad \texttt{Pos} \\ f \quad &: \quad (x{:}\texttt{Int} \rightarrow \{y{:}\texttt{Int} \mid y = x\}) \end{aligned}$$

Using compositional reasoning, we should only be able to infer that f returns its argument, which is of type `Pos`, and so $f\ e$ has type `Pos`.

Under the above type rule, however, the application $f\ e$ has the more precise (indeed, exact) type stating that $f\ e$ returns exactly the value of e.

$$f\ e \quad : \quad \{y{:}\texttt{Int} \mid y = e\}$$

Thus, the abstraction `Pos` of e has now been circumvented. To explicitly see the counterexample to compositionality, consider the corresponding derivation

$$\frac{e : \texttt{Pos} \qquad f : (x{:}\texttt{Int} \to \{y{:}\texttt{Int} \mid y = x\})}{f\ e : \{y{:}\texttt{Int} \mid y = e\}}$$

and consider replacing the derivation of $e : \texttt{Pos}$ with another for $d : \texttt{Pos}$, yielding the erroneous derivation

$$\frac{d : \texttt{Pos} \qquad f : (x{:}\texttt{Int} \to \{y{:}\texttt{Int} \mid y = x\})}{f\ e : \{y{:}\texttt{Int} \mid y = e\}}\ \textsc{Invalid}$$

Non-compositional reasoning causes multiple difficulties. First, an arbitrary program term e has leaked into a refinement predicate. Since e could be a large term, any type-checking procedure now needs to deal with extremely large types, probably much larger than any of the types present in the source program. Since the types in source programs are abstract specifications carefully chosen by the programmer, it is not clear that these new, larger, types generated during type checking are at all the right abstractions for performing lightweight verification.

Second, even if the refinement predicates in source programs are carefully chosen from a decidable theory, the fact that program terms leak into refinement predicates during type checking means that static type checking still requires deciding implications between arbitrary program terms, which is of course undecidable.

6.1.2 Compositional Reasoning for Executable Refinement Types

Given the universal and precise abstractions provided by dependent and refinement types, and the benefits of compositional reasoning, this chapter explores an alternative strategy for static checking of executable refinement types with dependent function types. The key idea underlying our approach is illustrated by the following rule for function application:

$$\frac{f : (x{:}T \to U) \qquad e : S \qquad S <: T}{f\ e : (\exists x{:}S.\, U)}$$

The application $f\ e$ must return a value of type U, where x is bound appropriately, but what is the appropriate binding for x?

Compositionality dictates that all we are permitted to know about e (and hence about x) is that it has type S. Thus, the inferred type of the application $f\ e$ is the existential type

$$\exists x{:}S.\, U$$

Roughly speaking, this type denotes the union of all types of the form $[x \mapsto d]\, U$, where d ranges over terms of type S.[3]

For the application $f\ e$ considered earlier, this rule infers the existential type

$$f\ e \quad : \quad (\exists x : \texttt{Pos}.\, \{y{:}\texttt{Int} \mid y = x\})$$

[3]Existential types are also written as dependent pairs $\Sigma x : S.U$, with two equally valid informal readings. The first is that the type is the (disjoint) union of $[x \mapsto d]\, U$ for each d of type S. The second is as the type of pairs of a term d of type S along with a corresponding type $[x \mapsto d]\, U$. The latter is more prevalent in constructive logic, as it emphasizes the reification of the existential witness.

that is (informally) equivalent to the type `Pos`, and indeed is a subtype of `Pos`.

$$(\exists x : \texttt{Pos}.\, \{y : \texttt{Int} \mid y = x\}) \quad <: \quad \texttt{Pos}$$

Thus, the type rule achieves the desired goal of compositional reasoning: $f\ e$ has type `Pos` simply because the argument expression has type `Pos`. All we can know about the argument is its type. Furthermore, note that the expression e itself no longer leaks into the refinement predicate, which provides benefits in terms of smaller inferred types and facilitates decidable type checking.

6.1.3 Expressiveness and Exactness

Given that this type system is strictly limited to compositional reasoning, and cannot, for example, include the traditional rule for dependent function application, a key question that arises is to what degree the requirement for compositional reasoning limits the expressiveness of the type system. Certainly, the refinement type language itself is sufficiently expressive to describe exact types, such as the type

$$\{x : \texttt{Int} \mid x = 1\}$$

for the constant 1. More interestingly, we show that, if every constant in the language is assigned an exact type, then for any well-typed program e, our type system will infer an exact type that exactly captures the run-time semantics of e. This result holds even if e itself includes coarse type specifications such as `Int`. This curious result occurs because all annotations in a program are preconditions to functions, which will be made as precise as allowed by the type of the argument to the function. Thus it is the types of

opaque constants that drives the precision of type checking. This is discussed at length in Section 6.7.

Assigning exact types to primitives is straightforward, via *selfification*. For example, the fixed point primitive $\texttt{fix}_T$ is typically given the inexact type $(T \to T) \to T$. For the case where $T = x{:}S \to B$, an exact type for $\texttt{fix}_T$ is:

$$f{:}(T \to T) \to x{:}S \to \{y{:}B \mid y = \texttt{fix}_T\ f\ x\}$$

In practice, of course, primitives are typically assigned somewhat coarser types. Nevertheless, this completeness result indicates that the precision of the type system is entirely parameterized by the types of these primitives; *the type system itself neither requires nor performs any additional abstraction.*

Put differently, the precision of any compositional analysis is naturally driven by the precision of the abstractions that it composes, and careful choice of abstractions is crucial for achieving precise, scalable analyses. The type system is entirely configurable in this regard; it does not perform any abstraction itself, and instead simply propagates the abstractions inherent in the types of constants and recursive functions, with the result that the precision of the analysis is largely under the control of the programmer.

6.1.4 Decidable Type Checking

A key benefit of compositional type checking is that program terms never leak into refinement predicates, so the language of refinement predicates can be cleanly separated from that of program terms. In particular, if refinement predicates in source

programs are drawn from the decidable theory of linear inequalities, then all proof obligations generated during type checking also fall within a decidable theory, and so type checking is decidable. We use this result to develop a decidable type checking algorithm for an important and easily-identifiable subset of our target language.

6.2 The Language $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

We present our ideas via the language $\lambda_{\mathcal{H}}^{\text{Comp}}$, a variant of $\lambda_{\mathcal{H}}$ augmented with existential types to support compositionality and also new case discrimination constructs (also called "pattern matching") to enable more interesting and challenging examples. Figure 6.1 presents the syntax of $\lambda_{\mathcal{H}}^{\text{Comp}}$, highlighting forms that are not present in $\lambda_{\mathcal{H}}$.

As before, primitives are presumed to include basic operations of the language, such as boolean and arithmetic operations ($+$, $\times$, `not`, $\Leftrightarrow$, etc.), and a fixed point operator $\texttt{fix}_T$ for each type T. However, in $\lambda_{\mathcal{H}}^{\text{Comp}}$ primitives are separated into syntactic classes of primitive functions k and data constructors c. Constructors include the boolean constants (`true`, `false`), integer constants (1, 2, 6, -17, etc), and any constructors for algebraic data types such as the list constructors `nil` and `cons`.

A constructor is eliminated by using the case discrimination expression

$$\texttt{case}\ e\ \texttt{of}\ \overline{c \Mapsto f}$$

where $\overline{c \Mapsto f}$ is a sequence of cases to whose constructor c to which e is compared, and the appropriate one chosen. For simplicity, we assume all case discrimination is disjoint and complete, introducing nonterminating alternatives as necessary. Thus the ordering

Figure 6.1: Syntax of $\lambda_{\mathcal{H}}^{\text{Comp}}$

$d, e, f, g, p, q ::=$	*Expressions:*
$\quad x$	variable
$\quad k$	primitive function
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad f\ e$	application
$\quad c$	***constructor***
$\quad \texttt{case}\ e\ \texttt{of}\ \overline{c \Mapsto f}$	***case discrimination***
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	primitive function
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad c\ \overline{e}$	***constructor application***
$S, T, U ::=$	*Types:*
$\quad \{x\!:\!B \mid p\}$	refinement type
$\quad x\!:\!S \to T$	dependent function type
$\quad \exists x\!:\!S.\, T$	***existential type***

$$
\begin{array}{rcl}
c\ \overline{e} & \stackrel{\text{def}}{=} & c\ e_1\ \cdots\ e_n \\
\overline{x\!:\!S} \to T & \stackrel{\text{def}}{=} & x_1\!:\!S_1 \to \cdots \to x_n\!:\!S_n \to T \\
\exists \overline{x\!:\!S}.\, T & \stackrel{\text{def}}{=} & \exists x_1\!:\!S_1.\ \cdots \exists x_n\!:\!S_n.\, T \\
\overline{c \Mapsto f} & \stackrel{\text{def}}{=} & c_1 \Mapsto f_1, \cdots, c_n \Mapsto f_n
\end{array}
$$

of the cases is irrelevant and they can be freely reordered.

The $\lambda_{\mathcal{H}}^{\text{Comp}}$ type language includes existential types of the form

$$\exists x : S.\, T$$

which classify terms of type T where x represents some unknown term of type S. Since $\lambda_{\mathcal{H}}^{\text{Comp}}$ lacks polymorphism, in addition to `Bool` and `Int` we also assume a base type of lists of integers `IntList` to develop our examples; adding other monomorphic data types is straightforward.

6.3 Operational Semantics of $\lambda_{\mathcal{H}}^{\text{Comp}}$

Figure 6.2 presents the operational semantics of $\lambda_{\mathcal{H}}^{\text{Comp}}$. The notable enhancement is case discrimination, but the complete set of redex reduction rules, contextual evaluation rules, and evaluation contexts is presented for ease of reference.

[E-CASE]: The case discrimination expression $\texttt{case}\ c\ \overline{e}\ \texttt{of}\ \overline{c \Mapsto f}$ reduces by selecting the appropriate case $c \Mapsto f$ and applying f to all the arguments of c. After all of $\overline{e}$, the full term $c\ \overline{e}$ is also passed to f; this is a formality (an important one) that enables path sensitivity, discussed later in Section 6.5.

6.4 Denotation of $\lambda_{\mathcal{H}}^{\text{Comp}}$ types to sets of $STLC$ terms

The definition of implication for $\lambda_{\mathcal{H}}^{\text{Comp}}$ is unchanged from that of $\lambda_{\mathcal{H}}$ (Figure 3.9 on page 39) but the quantification over closing substitutions depends on a denotational

Figure 6.2: Redex Evaluation for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Redex Evaluation $\boxed{e_1 \curvearrowright e_2}$

$$
\begin{array}{rcll}
(\lambda x{:}T.\, e_1)\ e_2 & \curvearrowright & [x \mapsto e_2]\, e_1 & \text{[E-}\beta\text{]} \\
k\ e & \curvearrowright & \delta(k, e) & \text{[E-}\delta\text{]} \\
\texttt{case}\ c\ \overline{e}\ \texttt{of}\ c \Rightarrow f, \cdots & \curvearrowright & f\ \overline{e}\ (c\ \overline{e}) & \text{[E-Case]}
\end{array}
$$

Contextual Evaluation $\boxed{d \rightsquigarrow e}$

$$
\begin{array}{rcll}
\mathcal{C}[d] & \rightsquigarrow & \mathcal{C}[e] \quad \text{if } d \curvearrowright e & \text{[E-Compat]} \\
\mathcal{D}[d] & \rightsquigarrow & \mathcal{D}[e] \quad \text{if } d \curvearrowright e & \text{[E-TyCompat]}
\end{array}
$$

Evaluation Contexts $\boxed{\mathcal{C},\ \mathcal{D}}$

$$
\begin{array}{rcl}
\mathcal{C} & ::= & \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C} \mid \lambda x{:}\mathcal{D}.\, e \mid \texttt{case}\ \mathcal{C}\ \texttt{of}\ \overline{c \Rightarrow f} \mid \texttt{case}\ e\ \texttt{of}\ c \Rightarrow \mathcal{C}, \cdots \\
\mathcal{D} & ::= & x{:}\mathcal{D} \to T \mid x{:}S \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\} \mid \exists x{:}\mathcal{D}.\, T \mid \exists x{:}S.\, \mathcal{D}
\end{array}
$$

Figure 6.3: Shape of $\lambda_{\mathcal{H}}^{\text{Comp}}$ types and terms

$$
\begin{array}{rcl}
\lfloor \{x{:}B \mid p\} \rfloor & \stackrel{\text{def}}{=} & B \\
\lfloor x{:}S \to T \rfloor & \stackrel{\text{def}}{=} & \lfloor S \rfloor \to \lfloor T \rfloor \\
\lfloor \exists x{:}S.\, T \rfloor & \stackrel{\text{def}}{=} & \lfloor T \rfloor
\end{array}
$$

interpretation of all $\lambda_{\mathcal{H}}^{\text{Comp}}$ types as sets of terms. The denotational semantics of Section 3.3 extends naturally to the existential types of $\lambda_{\mathcal{H}}^{\text{Comp}}$, mirroring the denotation of function types.

The extended definition of the shape of a $\lambda_{\mathcal{H}}^{\text{Comp}}$ types is given in Figure 6.3. The shape of an existential type $\exists x{:}S.\, T$ is simply the shape of the underlying type T, since bound variables are irrelevant in $STLC$. The shape of a term is still simply the same term where all types have been replaced by their shape.

Figure 6.4 shows the interpretation of $\lambda_{\mathcal{H}}^{\text{Comp}}$ types as sets of terms. The interpretations of refined basic types and function types are unchanged from $\lambda_{\mathcal{H}}$. An existential types $\exists x{:}S.\, T$ is interpreted as the set of $STLC$ terms e of simple type $\lfloor T \rfloor$ for which there exists some existential witness d in $[\![S]\!]$ such that $e \in [\![[x \mapsto e]\, T]\!]$.

6.5 The $\lambda_{\mathcal{H}}^{\text{Comp}}$ Type System

The judgments of the $\lambda_{\mathcal{H}}^{\text{Comp}}$ type system are those of $\lambda_{\mathcal{H}}$ (Section 3.4 on page 32) but their definitions are considerably more involved due to the enriched type and term languages.

Figure 6.4: Denotation from $\lambda_{\mathcal{H}}^{\text{Comp}}$ types to sets STLC terms

$$
\begin{array}{rcl}
[\![\{x{:}B \mid p\}]\!] & \stackrel{\text{def}}{=} & \{e \mid \vdash_{\text{STLC}} \lfloor e \rfloor : B \ \wedge \ (e \rightsquigarrow^* k \text{ implies } [x \mapsto k]\, p \rightsquigarrow^* \texttt{true})\} \\
[\![x{:}S \to T]\!] & \stackrel{\text{def}}{=} & \{f \mid \vdash_{\text{STLC}} \lfloor f \rfloor : \lfloor S \rfloor \to \lfloor T \rfloor \ \wedge \ \forall e \in [\![S]\!],\ f\ e \in [\![[x \mapsto e]\, T]\!]\} \\
[\![\exists x{:}S.\, T]\!] & \stackrel{\text{def}}{=} & \{e \mid \vdash_{\text{STLC}} \lfloor e \rfloor : \ \wedge \ \exists d \in [\![S]\!],\ e \in [\![[x \mapsto d]\, T]\!]\}
\end{array}
$$

- The typing judgment $\Gamma \vdash e : T$ states that term e may be assigned type T under the assumptions in environment Γ. (Figure 6.5)

- The type well-formedness judgment $\Gamma \vdash T$ ensures that T is a well-formed type under the environment Γ. (Figure 6.7)

- The subtyping judgment $\Gamma \vdash S <: T$ retains the standard reading: Any term of type S may also be assigned type T. (Figure 6.8)

6.5.1 Typing for $\lambda_{\mathcal{H}}^{\text{Comp}}$

The typing judgment

$$\Gamma \vdash e : T$$

for $\lambda_{\mathcal{H}}^{\text{Comp}}$ is enhanced from that of $\lambda_{\mathcal{H}}$ to support existentially quantified variables occurring in types, and case discrimination. These two constructs interact in important ways to provide path-sensitivity, without which precise types for constructors do not provide information during case discrimination.

Figure 6.5: Type Rules for $\lambda^{\text{Comp}}_{\mathcal{H}}$

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{}{\Gamma \vdash c : ty(c)}\ [\text{T-Const}] \qquad \frac{}{\Gamma \vdash k : ty(k)}\ [\text{T-Prim}] \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : \mathbf{self}(T, x)}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash S \qquad \Gamma, x : S \vdash e : T}{\Gamma \vdash (\lambda x{:}S.\, e) : (x{:}S \to T)}\ [\text{T-Fun}] \qquad \frac{\Gamma \vdash e : S \qquad \Gamma \vdash f : S \to T}{\Gamma \vdash f\ e : T}\ [\text{T-App}]$$

$$\Gamma \vdash e : \exists \overline{z{:}T}.\, \{y{:}B \mid q\}$$

For each $c \Rightarrow f$:

$$\frac{ty(c) = \overline{x{:}S} \to \{y{:}B \mid p\} \qquad \Gamma \vdash f : (\overline{x{:}S} \to y{:}\exists \overline{z{:}T}.\, \{B \mid p \wedge q\} \to U)}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \overline{c \Rightarrow f} : U}\ [\text{T-Case}]$$

[T-Const] and [T-Prim]: Constructors c and primitive functions f are assigned types according to ty.

[T-Var]: Variables are assigned types yielded by **self**, defined in Figure 6.6, a key feature discussed in Section 6.5.5.

[T-Fun]: This standard rule for λ-abstractions is unchanged.

[T-App]: Typing of function a application $(f\ e)$ is interesting, since it appears not to support dependent function types at all. We discuss this in detail in Section 6.5.4.

[T-Case]: This rule for a case discrimination expression `case` e `of` $\overline{c \Rightarrow f}$ is rather complex, because it uses only the information contained in the type of c to provide a measure of path-sensitivity, as the type of `if` did in $\lambda_{\mathcal{H}}$.

Each constructor c must necessarily return some refinement $\{x{:}B \mid p\}$ of the same base type B (for example, `nil` and `cons` both return refinements of `IntList`). The type of the discriminand e must also be some refinement of B, but may have existentially quantified variables $\overline{z:T}$, thus the type of e must have the form $\exists\overline{z{:}T}.\,\{y{:}B \mid q\}$.

For each branch $c \Rightarrow f$, where the particular constructor c has a type of the form $\overline{x{:}S} \to \{y{:}B \mid p\}$, if this branch is chosen, then we have "learned" that e was constructed with constructor c, so the predicate p holds for e. So we conjoin p to the type of e, yielding $\exists\overline{z{:}T}.\,\{y{:}B \mid p \wedge q\}$. The handling function f must then take parameters $\overline{x:S}$ corresponding to the arguments to the constructor

as well as the constructed term itself. Though $\overline{x}$ may not occur in U, their occurrences in $p \wedge q$ capture relationships between the variables so that the body of f may assume anything already known about e as well as anything learned by the fact that the constructor used was c. In this way, the precision of the path-sensitivity of [T-Case] is determined by the types of constructors.

For example, `cons` may be assigned a range of types, including:

$$\begin{aligned}
&\texttt{cons} \;:\; \texttt{Int} \to \texttt{IntList} \to \texttt{IntList} \\
&\texttt{cons} \;:\; \texttt{Int} \to x{:}\texttt{IntList} \to \\
&\qquad\qquad \{y{:}\texttt{IntList} \mid \texttt{length}(y) = \texttt{length}(x) + 1\} \\
&\texttt{cons} \;:\; n{:}\texttt{Int} \to x{:}\texttt{IntList} \to \\
&\qquad\qquad \{y{:}\texttt{IntList} \mid y = \texttt{cons}\ n\ x\}
\end{aligned}$$

In particular, if the constructor c has the precise return type $\{y{:}B \mid y = c\ \overline{x}\}$, then the new binding implies $[y \mapsto c\ \overline{x}]\, e'$. However, if c has a more coarse type, then the naïve approach of adding an assumption such as $e = c\ \overline{x}$ is reasoning noncompositionally using the exact semantics of c, ignoring its type.

6.5.2 Type well-formedness for $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

The type well-formedness judgment

$$\Gamma \vdash T$$

for $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$ is that of $\lambda_{\mathcal{H}}$ with the addition of well-formedness for existential types. The rule [WT-Exists] for an existential type $\exists x{:}S.\, T$ ensures that T is well formed in the

Figure 6.6: Definition of **self**

$$
\begin{array}{rcl}
\mathbf{self}(\{x{:}B \mid p\}, e) & = & \{x{:}B \mid p \wedge (x =_B e)\} \\
\mathbf{self}(x{:}S \to T, e) & = & x{:}S \to \mathbf{self}(T, e\ x) \\
\mathbf{self}(\exists x{:}S.\, T, e) & = & \exists x{:}S.\, \mathbf{self}(T, e)
\end{array}
$$

extended environment $\Gamma, x : S$.

6.5.3 Subtyping for $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

Subtyping between existential types (Figure 6.8) is derived from the logical interpretation of subtyping as "implication" between propositions.[4]

[S-Arrow] and [S-Base]: Subtyping between function types and base types is standard.

[S-Bind]: This rule to introduce an existential on the left corresponds to the tautology of first-order logic:

$$(\forall x \in A.\, P(x) \Rightarrow Q) \Longrightarrow (\exists x \in A.\, P(x)) \Rightarrow Q \text{ where } x \notin fv(C)$$

[S-Witness]: To introduce an existential on the right requires a witness term e of type T for the variable x. If S is a subtype of $[x \mapsto e]\, U$, then we can disregard the identity of the witness and retain only its specification to conclude that T is a subtype of $\exists x{:}T.\, U$.

[4] A similar inspiration may underlie Dreyer et al. [2003] where subtyping rules like our own are briefly mentioned, but ultimately not used, as their work had different goals.

Figure 6.7: Type well-formedness for for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Well-formed Types $\boxed{\vdash T}$

$$\frac{\Gamma \vdash S \qquad \Gamma, x : S \vdash T}{\Gamma \vdash x{:}S \to T}\ [\text{WT-Arrow}] \qquad \frac{\Gamma, x : B \vdash p : \texttt{Bool}}{\Gamma \vdash \{x{:}B \mid p\}}\ [\text{WT-Base}]$$

$$\frac{\Gamma \vdash S \qquad \Gamma, x{:}S \vdash T}{\Gamma \vdash \exists x{:}S.\, T}\ [\text{WT-Exists}]$$

Corresponding to our intuition, a combination of [S-Witness] and [S-Bind] shows that the following reassuringly covariant rule is admissible:[5]

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma, x{:}S_1 \vdash S_2 <: T_2}{\Gamma \vdash \exists x{:}S_1.\, S_2 <: \exists x{:}T_1.\, T_2}$$

6.5.4 Non-dependent Function Application

As mentioned above, the typing rule for a function application $(f\ e)$ is extremely simple, and does not refer to dependent types at all. Instead, the combination of existential types and subtyping provides enough power in other areas of the type sys-

[5] The full derivation is not too complex, eliding some the environment prefix and some weakening:

$$\dfrac{\dfrac{\dfrac{x : S_1 \vdash x\ :\ S_1 \qquad \vdash S_1 <: T_1}{x : S_1 \vdash x\ :\ T_1}\ [\text{T-Sub}] \qquad x : S_1 \vdash S_2 <: T_2}{x : S_1 \vdash S_2 <: \exists x{:}T_1.\, T_2}\ [\text{T-Witness}]}{\vdash \exists x{:}S_1.\, S_2 <: \exists x{:}T_1.\, T_2}\ [\text{T-Bind}]$$

Figure 6.8: Subtyping for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Subtyping $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, x : T_1 \vdash S_2 <: T_2}{\Gamma \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \text{ [S-Arrow]}$$

$$\frac{\Gamma, x : B \vdash p \Rightarrow q}{\Gamma \vdash \{x{:}B \mid p\} <: \{x{:}B \mid q\}} \text{ [S-Base]}$$

$$\frac{\Gamma \vdash e \,:\, T \qquad \Gamma \vdash S \;<:\; [x \mapsto e]\, U}{\Gamma \vdash S \;<:\; \exists x{:}T.\, U} \text{ [S-Witness]}$$

$$\frac{\Gamma, x{:}S \vdash T \;<:\; U \qquad x \notin FV(U)}{\Gamma \vdash \exists x : S.\, T \;<:\; U} \text{ [S-Bind]}$$

tem that this straightforward rule is sufficiently expressive. Suppose f has a dependent function type $x:T \to U$ and the argument e has type S where $S <: T$. Then, we use subtyping to specialize the function type as follows:

$$\begin{array}{lll} (x:T \to U) & <: & (x:S \to U) \\ & <: & (x:S \to \exists x:S.\, U) \\ & \equiv & (S \to \exists x:S.\, U) \quad \text{since } x \notin fv(S) \end{array}$$

The new function type $S \to \exists x:S.\, U$ is only applicable to terms of type S. More interestingly, its existential return type now internalizes the assumption that x will be of type S, so we have a non-dependent function type that precisely characterizes the behavior of the function f on arguments of type S, and which does not refer to the exact semantics of the argument e.[6]

Based on the above discussion, the following rule (mentioned in the introduction) is admissible:

$$\frac{\Gamma \vdash f \,:\, x:T \to U \qquad \Gamma \vdash e \,:\, S \qquad \Gamma \vdash S <: T}{\Gamma \vdash f\ e \,:\, (\exists x:S.\, U)}$$

[6] Non-dependent type rules for function application in a dependent type system are also used by Harper and Lillibridge [1994] and Dreyer et al. [2003] in ML module systems where computational effects make the standard substitution-based rule unsound. In the latter, a specialization of the technique we present allows the power of dependent function application when the argument is effect-free.

6.5.5 Self Types

To motivate our non-standard rule for variable references, consider the expression $x - x$, where x has type `Int` and subtraction has an exact type:

$$- \quad : \quad (w{:}\texttt{Int} \to y{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = w - y\})$$

Under the standard rule, whereby a reference to x simply has type `Int`, the type of $x - x$ is given by the judgment

$$x{:}\texttt{Int} \vdash (x - x) \; : \; (\exists w{:}\texttt{Int}.\, \exists y{:}\texttt{Int}.\, \{z{:}\texttt{Int} \mid z = w - y\})$$

which could be any integer. This is needlessly coarse, since the fact that $x - x = 0$ does not depend on the particular value of x. Informally, it would appear that even via compositional reasoning we should be able to give an exact type $\{z{:}\texttt{Int} \mid z = 0\}$. What happened? The type system has lost a key notion of identity, that the two variable references in $x - x$ refer to the same value.

To maintain the necessary information, *selfification* is used to assign to the variable reference x the type $\mathbf{self}(Int, x)$, or $\{y{:}\texttt{Int} \mid y = x\}$, which captures the identity of x, and yields the conclusion:

$$\begin{aligned} x{:}\texttt{Int} \vdash (x - x) \; : \quad & \exists w{:}\{\texttt{Int} \mid w = x\}. \\ & \exists y{:}\{\texttt{Int} \mid y = x\}. \\ & \{z{:}\texttt{Int} \mid z = w - y\} \end{aligned}$$

Since $(w = x) \wedge (y = x) \wedge (z = w - y)$ implies $z = 0$, this is a subtype of $\{z{:}\texttt{Int} \mid z = 0\}$, as intended. In addition to this example, **self** types are also crucial for achieving path-

sensitivity in case discrimination, which requires a notion of identity for the matched expression.

Selfification interacts in delicate ways with our goal of compositional reasoning. For example, Ou et al. [2004], in their declarative system, give a general selfification rule

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash e : \mathbf{self}(T, e)}\ [\text{T-Self}]$$

But this rule is non-compositional as they cannot replace the subderivation $\Gamma \vdash e : T$ with another $\Gamma \vdash e' : T$. In their algorithmic presentation, they give self types only to constants and variables. A key point not highlighted, however, is that giving self types to variables *is* compositional: Since there are no subderivations, compositionality is vacuous. This is not simply a syntactic trick, but the syntactic manifestation of the essence of compositional reasoning: We may know nothing about a variable but its type *and* we may remember from which variable we "fetch" a value. This insight is instrumental to reconciling compositional reasoning with the precision of selfification. The restriction of selfification to variables enables us to prove the following key compositionality theorem for our system:

Theorem 6.2 (Compositionality) *The $\lambda_{\mathcal{H}}^{Comp}$ type system is compositional.*

Proof: This is directly observable by inspection of the typing rules. □

6.6 Extensional equivalence for $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

Figure 6.9 presents the full definition of extensional equivalence for $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$. It is that of $\lambda_{\mathcal{H}}$ from Figure 3.10 on page 49 augmented with a definition for equivalence at existential type: Two terms e_1 and e_2 of existential type $\exists x\!:\!S.\,T$ are equivalent if there exist equivalent witness terms d_1 and d_2 of type S under which e_1 and e_2 are equivalent at type $[x \mapsto d_1]\,T \curlywedge [x \mapsto d_2]\,T$.

As for $\lambda_{\mathcal{H}}$, extensional equivalence in $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$ implies contextual equivalence; the required proofs are straightforward extensions of those in Section 3.6.

6.7 The $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$ Type System is Exact

The requirement for compositional reasoning restricts the kinds of inference rules the $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$ type system can include. For example, we have seen that it forbids both the standard rule for dependent function application and the selfification of arbitrary terms. The type system described above carefully adds the most important capabilities while maintaining compositional reasoning. We now address the important question of how much precision or expressive power our type system has lost because of its restriction to compositional type rules.

In fact, under moderate assumptions, our type system is *exact*: it can assign to each program term a type that completely captures the semantics of that term. Informally, an exact type is any type that captures a terms semantics and an exact type system is one that can capture *any* term's semantics. The intent of the function **self** is

Figure 6.9: Extensional Equivalence Under Reduction for $\lambda_{\mathcal{H}}^{\text{Comp}}$

$\boxed{\Gamma \vdash e_1 \sim e_2 : T}$ (defined by induction on $\lfloor T \rfloor$)

$$\Gamma \vdash e_1 \sim e_2 : \{x{:}B \mid p\} \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \Gamma \vdash e_1 \in \{x{:}B \mid p\}$$
$$\Gamma \vdash e_2 \in \{x{:}B \mid p\}$$
$$\forall \sigma \text{ such that } \vdash \sigma : \Gamma,\ (\sigma(e_1) \rightsquigarrow^* k) \Leftrightarrow (\sigma(e_2) \rightsquigarrow^* k)$$

$$\Gamma \vdash f_1 \sim f_2 : (x{:}S \to T) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \forall e_1, e_2 \text{ such that } \Gamma \vdash e_1 \sim e_2 : S,$$
$$\Gamma \vdash f_1\ e_1 \sim f_2\ e_2 : [x \mapsto e_1]\,T \curlywedge [x \mapsto e_2]\,T$$

$$\Gamma \vdash e_1 \sim e_2 : (\exists x{:}S.\, T) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \exists d_1, d_2 \text{ such that } \Gamma \vdash d_1 \sim d_2 : S,$$
$$\Gamma \vdash e_1 \sim e_2 : [x \mapsto d_1]\,T \curlywedge [x \mapsto d_2]\,T$$

$\boxed{S \curlywedge T}$

$$\{x{:}B \mid p\} \curlywedge \{x{:}B \mid q\} \quad \stackrel{\text{def}}{=} \quad \{x{:}B \mid p \wedge q\}$$
$$(x{:}S_1 \to S_2) \curlywedge (x{:}T_1 \to T_2) \quad \stackrel{\text{def}}{=} \quad x{:}(S_1 \curlywedge T_1) \to (S_2 \curlywedge T_2)$$
$$(\exists x{:}S_1.\, S_2) \curlywedge (\exists x{:}T_1.\, T_2) \quad \stackrel{\text{def}}{=} \quad \exists x{:}(S_1 \curlywedge T_1).\, (S_2 \curlywedge T_2)$$

---

to assign exact types (to variables, in this case) so that quantifications over self types are equivalent to quantifications over a single value. By defining exactness clearly we can prove this is the case.

Definition 6.3 (Exactness) *If* $\Gamma \vdash e : T$ *then the type* T *is* exact *for term* e *in environment* Γ *if for any other term* d *the judgment* $\Gamma \vdash d : T$ *implies* $d \equiv e$. *A type system is exact when it can assign an exact type to every well-typed term.*

Lemma 6.4 (Exactness of self type denotation) *If* $d \in [\![\mathbf{self}(T, e)]\!]$ *then* $d \equiv e$.

PROOF: This is equivalent to showing $\emptyset \vdash d \sim e : \mathbf{self}(T, e)$, which is proved by induction on $\lfloor T \rfloor$. □

Since self types have the desired denotation, the following chain of corollaries establishes that all of our judgments interact with **self** types as expected.

Corollary 6.5

1. *If* $\Gamma \vdash d : \mathbf{self}(T, e)$ *then* $d \equiv e$.
2. *If* $\vdash \sigma : \Gamma, x : \mathbf{self}(T, e), \Gamma'$ *then* $\sigma(x) \equiv \sigma(e)$.
3. $\Gamma, x : \mathbf{self}(T, e), \Gamma' \vdash p \Rightarrow [x \mapsto e]\, p$
4. $\Gamma, x : \mathbf{self}(T, e), \Gamma' \vdash T <: [x \mapsto e]\, T$
5. $\Gamma, \Gamma' \vdash \exists x{:}\mathbf{self}(S, e).\, T <: [x \mapsto e]\, T$

Establishing exactness for $\lambda_{\mathcal{H}}^{\text{Comp}}$ requires that each primitive function k and constructor c) be given an exact type. For the duration of this section, we assume that ty has this property:

Requirement 6.6 (Exact Types for Primitives) *In the exact variant of $\lambda_{\mathcal{H}}^{Comp}$ each primitive f and constructor c is given an exact type by ty.*

The exact information provided by types of constants and the self types of variables is propagated losslessly by the $\lambda_{\mathcal{H}}^{\text{Comp}}$ type system.

Theorem 6.7 (The $\lambda_{\mathcal{H}}^{\text{Comp}}$ type system is exact) *Supposing Requirement 6.6 holds, if $\Gamma \vdash e : T$ then also $\Gamma \vdash e : T'$ where T' is an exact type for e and Γ.*

PROOF: This is proved by demonstrating that $\Gamma \vdash e : \mathbf{self}(T, e)$ using Corollary 6.5, by induction on the typing derivation $\Gamma \vdash e : T$. □

Theorem 6.7 seems very strong: for an arbitrary term e of type T (where e's subterms may include coarse types), we can assign e the type $\mathbf{self}(T, e)$ that *exactly* captures the semantics of e, all via compositional reasoning. In exploring the conflict upon which this paper is premised, we have developed a compositional type system as powerful as one based on substitution, but without the violation of abstraction.[7] Rather, our type system insists that whatever precision is desired must be explicitly expressed via types.

[7]We do not claim to have improved upon dependent types in their general use as a foundation for mathematics, but only for the particular use of verifying a program specified by executable refinement types.

As an illustration of Theorem 6.7, consider the type of the fixed point primitive $\texttt{fix}_T$ used to express recursion, which is typically given the inexact type $(T \to T) \to T$. We can make this type exact via selfification. For example, if $T = x{:}U \to B$, then an exact type for $\texttt{fix}_T$ is $\mathbf{self}(\ (T \to T) \to T,\ \texttt{fix}_T)$ which is

$$f{:}(T \to T) \to x{:}U \to \{y{:}B \mid y = \texttt{fix}_T\ f\ x\}$$

For a more interesting example, consider Euclid's algorithm for computing greatest common divisors, where we use `if` expressions to abbreviate case discrimination on type `Bool`:

$$\begin{array}{rrcl} & \texttt{gcd} & = & \texttt{fix}_T\ (\lambda g{:}T.\, \lambda a{:}\texttt{Int}.\, \lambda b{:}\texttt{Int}.\, e) \\ \text{where} & T & = & \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\ \text{and} & e & = & \texttt{if}\ b = 0\ \texttt{then}\ a \\ & & & \texttt{else if}\ a > b\ \texttt{then}\ g\ (a - b)\ b \\ & & & \texttt{else}\ g\ a\ (b - a) \end{array}$$

If $\texttt{fix}_T$ has the exact type $\mathbf{self}(\ (T \to T) \to T,\ \texttt{fix}_T)$, then the type systems infers the following exact type for `gcd`:

$$\exists h{:}(g{:}T \to a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid r = e\}).$$

$$a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid r = \texttt{fix}_T\ h\ a\ b\}$$

Note that this type includes within refinement predicates both fixed point computations and also the body e of `gcd`. Thus, the type system is essentially translating the entire computation into the predicate language. While this illustrates the expressiveness of the type system, reasoning about fixed point computations is notoriously difficult for automated theorem provers.

However, primitive functions in this calculus represent terms whose semantics are unavailable; the predicates that occur in inferred types are determined by the specifications of primitives. The primitives of $\lambda_{\mathcal{H}}^{\text{Comp}}$ model built-in constructs of a language implementation but also names from external modules[8], which will typically be assigned approximate types. The import of the theorem is not that all abstractions can be broken even without dependent types, but that *the type system neither requires nor performs any abstraction itself.*

Returning to the example, a more practical approach is to use the simple type $(T \to T) \to T$ for $\texttt{fix}_T$, and for the programmer to provide an appropriately-precise specification T for the recursive function `gcd`, corresponding to the imperative idea of a loop invariant.[9] The structural specification $T = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ is rather coarse; a more precise (but not exact) specification is:

$$T \quad = \quad a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid a \texttt{ mod } r = 0 \ \wedge b \texttt{ mod } r = 0\}$$

The type system can then verify this specification for `gcd`, with simple proof obligations such as "if $b \texttt{ mod } r = 0$ and $(a - b) \texttt{ mod } r = 0$ then $a \texttt{ mod } r = 0$."

6.8 Type Soundness for $\lambda_{\mathcal{H}}^{\text{Comp}}$

This chapter proves type soundness for $\lambda_{\mathcal{H}}^{\text{Comp}}$ not by the usual means, but by proving type soundness for the exact variant – where all primitives have exact types –

[8] In a well-designed language, built-in primitives and imported symbols are indistinguishable

[9] The wide variety of methods for inferring loop invariants may be applied to avoid this annotation burden, as mentioned in Chapter 5. The most closely related progress on this variant of the problem is discussed in Chapter 8.

which ensures that any more approximate variant is also safe.

Given exactness, an existentially quantified variable with a singleton type is equivalent to having simply performed a substitution (it is essentially an explicit substitution [Abadi et al. 1990]). This intuition is formalized in Corollary 6.5 and the following complementary lemma:

Lemma 6.8 *For any expression e and type T, if $\Gamma \vdash e : S$ and $\Gamma, x{:}S \vdash T$ then $\Gamma \vdash [x \mapsto e]\,T <: \exists x{:}\mathbf{self}(S, e).\,T$*

Proof: By induction on the size of T, utilizing Theorem 6.7 to provide the exact witness e via [S-Witness]. □

From exactness, the standard substitution, preservation, and progress lemmas apply unchanged. The proofs follow the same structure as those for $\lambda_{\mathcal{H}}$, so we omit the details and simply state type safety. If a term is well-typed in any variant of our system, then it is certainly well-typed in the exact variant. Hence, a well-typed term cannot "get stuck" even in those imprecise variants where the substitution lemma cannot be proved.

Theorem 6.9 *If $\emptyset \vdash e : T$, then e either reduces to a value or is nonterminating.*

Proof: Map the derivation of $\Gamma \vdash e : T$ into the exact variant by induction over the derivation, then apply progress and preservation. □

There may be another syntactic approach to type soundness that applies in these systems, but it is informative to consider such systems as coarsenings of the exact variant.

6.9 A Type-checking Algorithm for $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

We now investigate how to provide decidable compositional type checking for an interesting subset of $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$ programs. Type checking for $\lambda_{\mathcal{H}}^{\mathrm{Comp}}$ is undecidable, despite the recovery of compositional reasoning, because implication remains undecidable. However, arbitrary terms are no longer injected into types. So by restricting refinement predicates to a decidable theory we can now provide a decidable, compositional, dependent type system. Note that this decidability result crucially relies on compositional reasoning: in a traditional dependent type system, even when all annotations fall within a decidable theory, substitutions made during type checking may result in proof obligations outside of that theory.

Figure 6.10 defines the type language for which we provide a type-checking algorithm. We leave the exact language of atomic decidable predicates l abstract, so the approach is applicable to any decidable predicate language. During type-checking, atomic predicates will be combined with conjunction and case-splitting on variables, so those are also included in the predicate sublanguage of terms, denoted by metavariables p or q, which are now no longer used for arbitrary terms.

We distinguish two sublanguages of types. The first, ranged over by metavariable Q and R, does not include existential quantifiers and may appear in source programs. Augmented types ranged over by E and F generated during type checking may contain existential quantification, but only in positive positions. Thus $Q \subseteq E \subseteq T$. Only augmented types E are bound in the environment during type checking, so we use

Figure 6.10: Syntax for Algorithmic Typing of $\lambda_{\mathcal{H}}^{\text{Comp}}$

$p ::=$	*Predicates:*
$\quad l$	atomic predicate
$\quad p \wedge p$	conjunction
$\quad \texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \Mapsto p}$	case discrimination
$d, e, f, g, p, q ::=$	*Terms:*
$\quad x$	variable
$\quad k$	primitive function
$\quad \lambda x\!:\!R.\, e$	abstraction
$\quad f\ e$	application
$\quad c$	constructor
$\quad \texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \Mapsto e}$	case discrimination
$v ::=$	*Values: ($\subset$ Terms)*
$\quad k$	primitive function
$\quad \lambda x\!:\!S.\, e$	abstraction
$\quad c\ \overline{e}$	constructor application
$Q, R ::=$	*Restricted Types:*
$\quad \{x\!:\!B \mid p\}$	refinement type
$\quad x\!:\!Q \to R$	restricted function type
$E, F ::=$	*Types with Covariant Existentials:*
$\quad \{x\!:\!B \mid p\}$	refinement type
$\quad x\!:\!R \to E$	augmented function type
$\quad \exists x\!:\!E.\, F$	augmented existential type
$\Delta ::=$	*Algorithmic Typing Environments:*
$\quad \emptyset$	empty environment
$\quad \Delta, x : E$	environment extension

Figure 6.11: Shorthand for pushing case expressions inside types

$$\texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto e} \quad \stackrel{\text{def}}{=} \quad \texttt{case}\ w\ \texttt{of}\ \overline{c \mapsto \lambda\overline{x:R}.\,\lambda y{:}E.\,e}$$

$$\text{where} \quad ty(c) = \overline{x:R} \to \{y{:}B \mid p\}$$

$$\text{and} \quad E \text{ has the form } \exists\overline{z:F}.\,\{y{:}B \mid p \wedge q\}$$

$$(\overline{z:F} \text{ and } q \text{ will be clear from context})$$

$$\texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto \exists\overline{z_c{:}E_c}.\,F_c} \stackrel{\text{def}}{=} \exists\overline{x_c{:}E_c}.\,\texttt{case}\ w\ \texttt{of}\ \overline{c \mapsto F_c}$$

where $\overline{x_c : E_c}$ contains all of the bindings for each c, (possibly empty)

$$\texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto \{z{:}B \mid q\}} = \{y{:}B \mid \texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto q}\}$$

$$\text{where } ty(c) = \overline{x_c{:}R_c} \to \{z{:}B \mid p\}$$

a new metavariable Δ to range over these E-environments.

The type-checking algorithm is presented as a set of syntax-directed inference rules. The judgments are analogous to those of the type system.

- The algorithmic typing judgment $\Delta \Vdash e : E$ reads that e is assigned type E, which may contain covariant existentials.

- The algorithmic type well-formedness judgment $\Delta \Vdash R$ is only invoked for programmer-provided restricted types R, and ensures that R is well-formed in environment Γ.

- The algorithmic subtyping judgment $\Gamma \Vdash E <: R$ checks that the type E is a subtype of the restricted type R. Though E may contain covariant existentials, they are not problematic for algorithmic checking. The upper bound R will never contain existentials.

6.9.1 Algorithmic typing for $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

Our algorithmic type checking judgment

$$\Delta \Vdash e : E$$

is shown in Figure 6.12. It is essentially derived from the typing judgment $\Gamma \vdash e : T$. As usual for syntax-directed type checking, we remove the subsumption rule and instead inline subtyping where needed in function applications to ensure the argument is of the appropriate type, and in case discrimination to merge the types of all the branches.

[A-Const] and [A-Prim]: Constructors and primitive functions are given types by ty.

[A-VAR]: Variables are still given types via **self**. Adding uninterpreted function symbols to the decidable predicate language retains decidability. Other approximations may be considered for efficiency; because of our nonstandard approach to soundness, any approximation is sound.

[A-APP]: For an application $f\ e$, this rule introspects the type of f to discover a functional type within existential quantifiers, of the form $\exists\overline{z:E_z}.\, x:R \to E_1$ (performing the work of as many applications of [S-BIND] as necessary). Then the type E_2 of the argument e is checked for compatibility with R. The result of the application is assigned type $\exists\overline{z:E_z}.\, \exists x:E_2.\, E_1$

[A-CASE] In typing a case discrimination expression, we lift the `case` syntax to types to concisely express the resulting type. Intuitively, $\texttt{case}\ e\ \texttt{of}\ \overline{c\ \overline{x} \mapsto E}$ is equivalent to whichever case succeeds. Figure 6.12 includes the exact definition. To avoid arbitrary expressions appearing in refinement predicates, we require that the inspected subexpression in a `case` construct be a variable. This requirement can be satisfied by rewriting the more general form $\texttt{case}\ e\ \texttt{of}\ \overline{c\ \overline{x} \mapsto f}$ into the semantically equivalent $(\lambda y:S.\, \texttt{case}\ y\ \texttt{of}\ \overline{c\ \overline{x} \mapsto f})\ e$. This rewriting step, which is a burden imposed on the input program, introduces a type R for the variable y, and so ensures that any expression used in case discrimination will have a specification that falls in the restricted type language.[10]

[10] A similar restriction is present in Rondon et al. [2008], where only variables may be inserted into predicate templates, so introducing bindings of intermediate results to variables is crucial to achieving precision while maintaining decidability.

Figure 6.12: Algorithmic Typing for $\lambda^{\text{Comp}}_{\mathcal{H}}$

Algorithmic Typing $\boxed{\Delta \Vdash e \,:\, E}$

$$\frac{}{\Delta \Vdash c \,:\, ty(c)}\ [\text{A-Const}] \qquad \frac{}{\Delta \Vdash k \,:\, ty(k)}\ [\text{A-Prim}]$$

$$\frac{(x : A) \in \Delta}{\Delta \Vdash x \,:\, \mathbf{self}(E, x)}\ [\text{A-Var-Base}] \qquad \frac{\Delta \Vdash R \qquad \Delta, x{:}R \Vdash e \,:\, E}{\Delta \Vdash (\lambda x{:}R.\, e) \,:\, (x{:}R \to E)}\ [\text{A-Fun}]$$

$$\frac{\Delta \Vdash e_1 \,:\, \exists \overline{z{:}E_z}.\, x{:}R \to E_1 \qquad \Delta \Vdash e_2 \,:\, E_2 \qquad \Delta, \overline{z{:}E_z} \Vdash E_2 \mathrel{<:} R}{\Delta \Vdash e_1\ e_2 \,:\, \exists \overline{z{:}E_z}.\, \exists x{:}E_2.\, E_1}\ [\text{A-App}]$$

$$\Delta \Vdash w \,:\, \exists \overline{z{:}E}.\, \{y{:}B \mid q\}$$

For each $c\ \overline{x} \mapsto d$:

$$\frac{ty(c) = \overline{x{:}Q} \to \{y{:}B \mid p\} \qquad \Delta, \overline{x : Q}, y : \exists \overline{z{:}E}.\, \{B \mid q \wedge p\} \Vdash d \,:\, F}{\Delta \Vdash (\texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto d}) \,:\, (\texttt{case}\ w\ \texttt{of}\ \overline{c\ \overline{x} \mapsto F})}\ [\text{A-Case}]$$

Figure 6.13: Algorithmic Type Well-formedness for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Algorithmic Well-formed Types $\boxed{\Delta \Vdash R}$

$$\frac{\Delta \Vdash R_1 \qquad \Delta, x{:}R_1 \Vdash R_2}{\Delta \Vdash x{:}R_1 \to R_2}\ [\text{AT-Arrow}] \qquad \frac{\Delta, x : B \Vdash p : \texttt{Bool}}{\Delta \Vdash \{x{:}B \mid p\}}\ [\text{AT-Base}]$$

6.9.2 Algorithmic type-wellformedness for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Figure 6.14 presents the rules defining the algorithmic type well-formedness judgment

$$\Gamma \Vdash R$$

which checks if programmer-supplied types R are well-formed. It is easy to check well-formedness of types with existential quantification, but for illustrative purposes the rules are deliberately limited to those cases that actually may occur during type checking.

6.9.3 Algorithmic subtyping for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Like typing, algorithmic type checking relies on subtyping, but interestingly only for the asymmetric form

$$\Delta \Vdash E <: R$$

Figure 6.14: Algorithmic Subtyping for $\lambda_{\mathcal{H}}^{\text{Comp}}$

Algorithmic Subtyping $\boxed{\Delta \Vdash E <: R}$

$$\frac{\Delta \Vdash R_2 <: R_1 \qquad \Delta, x : R_2 \Vdash E_1 <: R_3}{\Delta \Vdash (x{:}R_1 \to E_1) <: (x{:}R_2 \to R_3)} \text{ [AS-Arrow]}$$

$$\frac{\Delta, x : B \vdash q \Rightarrow p}{\Delta \Vdash \{x{:}B \mid q\} <: \{x{:}B \mid p\}} \text{ [AS-Base]} \qquad \frac{\Delta, x{:}E_1 \Vdash E_2 <: R_3}{\Delta \Vdash \exists x{:}E_1.\, E_2 <: R_3} \text{ [AS-Bind]}$$

which checks subtyping between an inferred type E, with covariant existential quantifications, and a programmer-specified, restricted type R.

[AS-Bind]: Since existential types never appear on the right hand side, we (fortunately!) never need to "guess" witnesses for existentials. Existentials on the left are unproblematic, because the existential quantification in a negative context transforms into a universal quantification in a positive context, resulting in just another binding in the typing environment. We could add the covariant rule from the end of Section 6.1 to allow some existential quantification on the right side of subtyping, but our syntactic conditions make that unnecessary.

Algorithmic typeability is not closed under evaluation; instead soundness is guaranteed by the correspondence between algorithmic typing and the full type system.

Theorem 6.10 (Soundness of Algorithmic Typing)

If $\Delta \Vdash e : E$ *then* $\Delta \vdash e : E$

PROOF: Proceed by induction on the derivation of $\Delta \Vdash e : E$. $\square$

While the algorithm is not complete with respect to the full type system, it is complete for the class of sublanguages we have described.

Theorem 6.11 (Relative Completeness)

If $\Delta \vdash e : T$ *and* e *is annotated with restricted types, and case discrimination inspects only variables, then there exists a type* E *such that* $\Delta \Vdash e : E$ *and* $\Delta \vdash E <: T$

PROOF: By induction on the derivation of $\Delta \vdash e : T$. $\square$

These algorithmic rules characterize a class of programs for which type checking is decidable: If all program annotations are restricted types whose refinements are expressions in some decidable theory (where this means that queries of the form $\Delta \Vdash q \Rightarrow p$ are decidable, such as linear inequalities with conjunction and case splits), then all implication queries also fall in this theory, and are decidable. Interestingly, because existentials only appear as assumptions in proof obligations, the theory need only support a restricted form of existential quantification that is "productive" in that each existential corresponds to positive existence of some term, and there is never any existential proof obligation.

If a program includes some predicates that do not fall in the predetermined language, this algorithm may still succeed in verifying the program if the theorem prover used is able to dispatch the generated proof obligations. However, in this case a rejection by this algorithm does not indicate that the program is ill-typed. Hybrid type checking may be used as a fallback for such programs. The following informal architecture yields an algorithm always verifies a well-defined subset of programs, but never rejects a well-typed program:

1. Given a program e, does the algorithm yield $\emptyset \Vdash e : E$?
2. If so, then accept the program.
3. If not, and the program's annotations fall in the language of restricted types, then reject the program.
4. Otherwise, perform hybrid type checking on the program.

6.10 Example: Binary Search Trees in $\lambda_{\mathcal{H}}^{\mathbf{Comp}}$

As an illustration of our decidable type system, we now revisit the example of binary search trees. We assume an additional base type `BST` to represents binary search trees, with auxiliary functions:

$$\texttt{lower} \;:\; \texttt{BST} \to \texttt{Int}$$

$$\texttt{upper} \;:\; \texttt{BST} \to \texttt{Int}$$

that return the lower and upper bounds of integers in a BST, and return `maxInt` and `minInt`, respectively, on empty BSTs. We also assume some additional constructors and primitives:

$$\begin{aligned}
\texttt{minInt} &: \texttt{Int} \\
\texttt{maxInt} &: \texttt{Int} \\
\texttt{min} &: x{:}\texttt{Int} \to y{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z \leq x \wedge z \leq y\} \\
\texttt{max} &: x{:}\texttt{Int} \to y{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z \geq x \wedge z \geq y\}
\end{aligned}$$

The type of a binary search tree with integers in the range $[lo, hi)$ is defined by the refinement type:

$$\texttt{BST}_{lo,hi} \;=\; \{x{:}\texttt{BST} \mid \; lo \leq \texttt{lower}(x) \wedge \texttt{upper}(x) < hi \;\}$$

Binary search tree are created using the constructors `empty` and `node`, which are assigned the following precise types:

$$\begin{aligned}
\texttt{empty} &: lo{:}\texttt{Int} \to hi{:}\texttt{Int} \to \texttt{BST}_{lo,hi} \\
\texttt{node} &: lo{:}\texttt{Int} \to hi{:}\texttt{Int} \to \\
&\quad\; v{:}\{v{:}\texttt{Int} \mid lo \leq v < hi\} \to \\
&\quad\; x{:}\texttt{BST}_{lo,v} \to y{:}\texttt{BST}_{v,hi} \to \texttt{BST}_{lo,hi}
\end{aligned}$$

Here, these constructors take additional "index" arguments lo and hi, and so we are using an index-type-like implementation of binary search trees. Using these constructors,

we can define the `insert` operation on `BST`s:

$$
\begin{array}{l}
\texttt{fix}_T(\lambda f{:}T. \\
\quad \lambda lo{:}\texttt{Int}.\, \lambda hi{:}\texttt{Int}. \\
\quad \lambda v{:}\{y{:}\texttt{Int} \mid lo \le y < hi\}. \\
\quad \lambda x{:}\texttt{BST}_{lo,hi}. \\
\quad\quad \texttt{case}\ x\ \texttt{of} \\
\quad\quad\quad (\texttt{empty}\ lo\ hi) \ \triangleright (\texttt{node}\ lo\ hi\ v\ (\texttt{empty}\ lo\ v)\ (\texttt{empty}\ v\ hi)) \\
\quad\quad\quad (\texttt{node}\ lo\ hi\ n\ l\ r) \triangleright \\
\quad\quad\quad\quad (\texttt{case}\ v < n\ \texttt{of} \\
\quad\quad\quad\quad\quad \texttt{true} \triangleright (\texttt{node}\ lo\ hi\ n\ (\texttt{insert}\ lo\ n\ x\ l)\ r) \\
\quad\quad\quad\quad\quad \texttt{false} \triangleright\ (\texttt{node}\ lo\ hi\ n\ l\ (\texttt{insert}\ n\ hi\ x\ r))
\end{array}
$$

where `insert` has type

$$
\begin{array}{rcl}
T & = & lo{:}\texttt{Int} \to hi{:}\texttt{Int} \to \\
 & & v{:}\{v{:}\texttt{Int} \mid lo \le v < hi\} \to \\
 & & \texttt{BST}_{lo,hi} \to \texttt{BST}_{lo,hi}
\end{array}
$$

All the proof obligations generated for this program are formulae over linear integer inequalities, and hence decidable.

In this example, the variables *lo* and *hi* are somewhat awkward, and provide an indirect specification of the behavior of `insert`. We can express this program more naturally by removing these parameters instead expressing the key data invariants directly in terms of the underlying data structure. In this formulation, the `BST` constructors

have the more natural types:

$$\begin{array}{rcl} \texttt{empty} & : & \texttt{BST}_{\texttt{maxInt},\texttt{minInt}} \\ \\ \texttt{node} & : & v{:}\texttt{Int} \rightarrow \\ & & x{:}\{x{:}\texttt{BST} \mid \texttt{upper}(x) < v\} \rightarrow \\ & & y{:}\{y{:}\texttt{BST} \mid v \leq \texttt{lower}(y)\} \rightarrow \\ & & \texttt{BST}_{\texttt{lower}(x),\texttt{upper}(y)} \end{array}$$

The revised `insert` implementation is identical to the one shown above but elides index variables:

$$\begin{array}{l} \texttt{fix}_{T'}(\lambda f{:}T'.\ \ \lambda v{:}\texttt{Int}.\ \ \lambda x{:}\texttt{BST}. \\ \quad \texttt{case}\ x\ \texttt{of} \\ \quad\quad \texttt{empty} \triangleright (\texttt{node}\ v\ \texttt{empty}\ \texttt{empty}) \\ \quad\quad (\texttt{node}\ n\ l\ r) \triangleright \\ \quad\quad\quad (\texttt{case}\ v < n\ \texttt{of} \\ \quad\quad\quad\quad \texttt{true} \triangleright (\texttt{node}\ n\ (\texttt{insert}\ x\ l)\ r) \\ \quad\quad\quad\quad \texttt{false} \triangleright (\texttt{node}\ n\ l\ (\texttt{insert}\ x\ r)) \end{array}$$

and has the following type T':

$$\begin{array}{rcl} T' & = & v{:}\texttt{Int} \rightarrow x{:}\texttt{BST} \rightarrow \\ & & \texttt{BST}_{\texttt{min}(\texttt{lower}(x),v),\texttt{max}(\texttt{upper}(x),v)} \end{array}$$

6.11 Related Work

First, we present a high-level comparison of refinement types with indexed types, an alternative approach towards similar goals. We next survey other applications

of existential quantification in type systems. Finally, we briefly outline the development of refinement and refinement types which this work directly builds upon.

6.11.1 Indexed Types

Indexed types are discussed in Section 1.1.2 in general, so here we focus on their relationship to the concept of compositionality. Formally, the type systems of Dependent ML [Xi and Pfenning 1999], ATS [Cui et al. 2005], and Ωmega [Sheard 2005], distinguish compile-time from run-time data, and allow types to depend only on compile-time data. This approach makes compositionality a formally vacuous proposition, and indeed can sometimes be encoded in existing polymorphic type systems without use of dependent types [Zenger 1997; McBride 2002].

As an illustrative example, consider the family of types $\texttt{IntList}_n$, where n indicates the length of lists inhabiting the type. The types of list constructors and the `append` function are as follows. We use the type $\mathbb{N}$ to distinguish the compile-time type of natural numbers.

$$\begin{aligned}
\texttt{nil} &: \texttt{IntList}_0 \\
\texttt{cons} &: n{:}\mathbb{N} \to \texttt{Int} \to \texttt{IntList}_n \to \texttt{IntList}_{n+1} \\
\texttt{append} &: m{:}\mathbb{N} \to n{:}\mathbb{N} \to \\
&\qquad \texttt{IntList}_m \to \texttt{IntList}_n \to \texttt{IntList}_{m+n}
\end{aligned}$$

The connection between run-time invariants and compile-time data is based essentially on injecting an abstraction of run-time data into compile-time data; for example lists indexed by their length use a compile-time copy of the natural numbers. The expres-

sions that are reflected into indices can be carefully controlled to ensure that type compatibility remains decidable.

We can naturally express the indexed type $\texttt{IntList}_n$ as the refinement type $\{x{:}\texttt{IntList} \mid \texttt{length}(x) = n\}$, and the above types and all proof obligations remain equivalent. In this way, by reifying the abstraction function from a value to its associated index, arbitrary indexed types can be embedded into executable refinement types. The abstraction function (`length` in this case) may be treated as an uninterpreted symbol – its definition cannot be necessary in proof obligation since it is not even available to indexed types.

The above type for `append` is somewhat awkward, though, as it requires the additional index parameters n and m.[11] In contrast, executable refinement types allow a more natural expression of the same specification for `append`, that eliminates these index parameters:

$$x{:}\texttt{IntList} \to y{:}\texttt{IntList} \to$$
$$\{z{:}\texttt{IntList} \mid \texttt{length}(z) = \texttt{length}(x) + \texttt{length}(y)\}$$

More critical than the aesthetic issue of index parameters, the index of a data structure, decided by its implementor, determines which properties may be reasoned about, inhibiting reuse and composition of data structures. For example, if one is interested in verifying the ordering of a list, rather that its length, then a different index is required, specifically the minimum element (at the head of the list) to specify `cons`, and also the maximum element (at the tail) to specify `append`. We also need

[11] Note that these index parameters can be inferred in many cases.

integers indexed by their exact values $\texttt{Int}_i$, where i is drawn from compile-time integers $\mathbb{Z}$ (with $\pm\infty$ for corner cases). We define the type $\texttt{IntList}_{i,j}$ of lists in ascending order with minimum element i and maximum element j using the following constructors. Note that predicate subtypes are used on indices of compile-time type $\mathbb{Z}$ but not on runtime terms.

$$\begin{aligned}
\texttt{nil} &: \texttt{IntList}_{\infty,-\infty} \\
\texttt{cons} &: i{:}\mathbb{Z} \to j{:}\{j{:}\mathbb{Z} \mid j \geq i\} \to k{:}\mathbb{Z} \to \\
&\qquad \texttt{Int}_i \to \texttt{IntList}_{j,k} \to \texttt{IntList}_{i,\texttt{max}(i,k)} \\
\texttt{append} &: i{:}\mathbb{Z} \to j{:}\mathbb{Z} \to k{:}\{k{:}\mathbb{Z} \mid k \geq j\} \to l{:}\mathbb{Z} \to \\
&\qquad \texttt{IntList}_{i,j} \to \texttt{IntList}_{k,l} \to \texttt{IntList}_{\texttt{min}(i,k),l}
\end{aligned}$$

In general, since the index of a data type determines the properties that may be reasoned about, more complex properties require embedding of more data as indices. In the limit, giving the precise type $x{:}\,\texttt{IntList} \to \{y{:}\texttt{IntList} \mid y = x\}$ to the identity function on lists (or any type) requires embedding the entire type of lists (or any type) into the index language, which reduces the utility of the syntactic distinction of compile-time data.

In contrast, executable refinement types can naturally express a wide variety of refinements over the same `IntList` type, such as

$$\{x{:}\texttt{IntList} \mid \texttt{minElem}(x) = i \wedge \texttt{maxElem}(x) = j\}$$

or the most natural $\{x{:}\texttt{IntList} \mid \texttt{isSorted}(x)\}$. The proof obligations for the latter may be more problematic, and the current work is progress towards characterizing those situations where they are not difficult.

Reasoning about how the constructors of a type relate to the global predicate upon the type is an open question. Kawaguchi et al. [2009] defines a class of "measures" which are global predicates calculated by catamorphism over the defining equations of a recursive datatype, and automatically generates the appropriate refinements for the constructors. Atkey et al. [2011] generalizes this approach to a category-theoretic analysis of when a type defined as the initial algebra of a functor is actually a refinement of a type defined by another functor.

6.11.2 Existential Types

Formal existential quantification in type-theory has found a variety of uses, including closure conversion [Morrisett et al. 1999; Grossman 2002], module systems [Mitchell and Plotkin 1988; Harper and Lillibridge 1994; Dreyer et al. 2003], and semantics of object orientation [Bruce et al. 1999]. Most similar to the present work are ML module systems, which also combine existentials with subtyping and a different (formal) form of singleton types.

The standard substitution-based typing rule for function application is unsound in the presence of effects, so dependencies have to be "forgotten" [Harper and Lillibridge 1994]. Dreyer et al. [2003] ameliorate this restriction with a sophisticated effect system that restores the power of dependent application in the event that the argument to a function is effect-free.

Also similar is the need to express sharing constraints. Just as we give self types to variables, Stone and Harper [2000] and subsequently Dreyer et al. [2003] assign

singleton kinds (a special syntactic form) to modules in order to preserve information when applying a dependently-kinded functor. In contrast to module languages, we examine the consequences and form of existentials and selfification in the context of executable refinements with case discrimination and automatic theorem proving. Our singletons are expressed in the existing type language rather than adding a special form, and in our setting a type of a variable x may constrain other bound variables so we maintain this information while adding the identity information to the known type of x.

Our different approaches and goals led us to emphasize compositional reasoning as a key aspect of our system, which we present in a more minimal calculus than the large and pragmatism-oriented ML module type systems. Compositionality is obviously important in their work as well, since it affects separate (re-)compilation.

Our existential types may be considered a limited expression of Σ types for dependent pairs in powerful type theories such as those underlying Coq [The Coq development team 2012], Hoare Type Theory [Nanevski et al. 2006], and Epigram [McBride and McKinna 2004]. Dependent ML, in fact, makes use of types such as $\Sigma x{:}T_1.T_2$ with explicit introduction and elimination forms (i.e. without subtyping) to allow functions operating over arguments of unknown index.

With respect to Σ types, the work presented in this chapter can be interpreted as a way of providing automation for a simple and common case where the full power of these type theories is not necessary.

6.11.3 Liquid Types

Liquid Types [Rondon et al. 2008] present a similar system, with predicates drawn from a carefully designed language of predicate templates. To avoid inserting arbitrary terms into predicates, only variables are used to instantiate these templates, and variables are required to have types within the selected predicate language. For programs in *a-normal form*, where every subexpression is bound to a variable via a `let` expression, every subexpression is given a type within the language of predicate templates. This provides some of the benefits of compositionality, as no predicate will need to be proved that falls outside of the selected predicate language. Instead, as with our approximate variant, the approximation occurs because a precise type for each expression may not be expressible within the decidable predicate language.

Chapter 7

Sage: An Implementation of Executable Refinement Types

This chapter presents Sage, an exploratory implementation of executable refinement types and hybrid type checking. In creating a new language with experimental features, we hope not for mass adoption, but to ascertain the technical feasibility of the eventual incorporation of our ideas into mainstream languages. Of interest are question such as:

- How long does type checking take?
- How effective are state-of-the-art theorem provers at dispatching the obligations that arise?
- Can realistic programs be easily written and specified?

To place these questions in the introductory context of this dissertation, consider again this variety of specifications for a function that inverts a matrix, reproduced here verbatim for reference:

1. The argument can be any (dynamically-typed) value.
2. The argument must be an array of arrays of numbers.
3. The argument must be a *matrix*, that is, a rectangular (non-ragged) array of arrays of numbers.
4. The argument must be a square matrix.
5. The argument must be a square matrix with nonzero determinant.
6. The argument must be a square matrix satisfying a custom invertibility test procedure `isInvertible`.

Sage supports *all* of these specifications in a unified type system.

Chapter Outline

- Section 7.1 gives a brief overview of the key features of Sage.
- Section 7.2 illustrates the Sage language through a series of examples written in Sage's concrete syntax.
- Section 7.3 presents the abstract syntax for Sage Core, a minimal calculus for discussing Sage's operational semantics and type system.

- Section 7.4 defines a small-step operational semantics for Sage Core.

- Section 7.5 presents the type system of Sage Core.

- Section 7.6 presents a hybrid type checking / cast insertion algorithm for the language.

- Section 7.7 and Section 7.8 describe our implementation and experimental results.

7.1 Overview of Sage features

On a technical level, the Sage type system can be viewed as a synthesis of three concepts: the type `Dynamic` of terms that "may be" any value expressible in the language, executable refinement types, and first-class types. These features add expressive power in three orthogonal directions, yet they all cooperate neatly within Sage's hybrid static/dynamic checking framework.

For this section and the next, Sage concrete syntax will be used. Sage's concrete syntactic style is borrowed from OCaml and Coq so that the transliterations of mathematical symbols into ASCII are standard and familiar. Here are some brief examples:

- The dependent function type $x{:}\texttt{Int} \to \texttt{Int}$ is written `x:Int -> Int`.

- The function $\lambda x{:}S.\, x + 1$ is written `fn (x:S) => x + 1`.

- Top-level definitions are introduced with `let` and terminated with a semicolon. For example, this defines the function `add1`.

```
let add1 = fn (x:Int) => x + 1;
```

The function arguments may also be placed to the left of the equals sign. The definition of `add1` is equivalently written as

```
let add1 (x:Int) = x + 1;
```

Any further details of the concrete syntax should be, hopefully, intuitive, but the full syntactic sugar provided by the SAGE implementation is technically uninteresting.

7.1.1 Type `Dynamic`

SAGE supports dynamically-typed programming with the type `Dynamic` [Thatte 1990; Henglein 1994]. `Dynamic` is a supertype of all types; any value can be upcast to type `Dynamic`. A value of type `Dynamic` can be implicitly downcast (via a run-time check) to a more precise type. Downcasts are implicitly inserted when necessary, such as when applying the primitive addition operator +, which requires arguments of type `Int`, to arguments of type `Dynamic`. Thus, declaring variables to have type `Dynamic` (which is the default if type annotations are omitted) yields the usual programming style and runtime characteristics of dynamically-typed programming languages such as Scheme, Python, Perl, Ruby, etc.

These dynamically-typed programs can later be annotated with traditional type specifications like `Int` and `Bool`. The programmer need not fully annotate the program with types in order to reap some benefit. Types enable SAGE to check more

properties statically, but it is still able to fall back to dynamic checking whenever the type `Dynamic` is encountered.

7.1.2 Executable Refinement Types

SAGE supports high precision specifications via executable refinement types. For example, the following code snippet defines the type of integers in the range from `lo` (inclusive) to `hi` (exclusive):

```
{ x:Int | lo <= x && x < hi }
```

SAGE extends prior chapters by allowing refinement of any type, not only base types. This results in additional challenges to decidability. For example, whether an arbitrary function may be assigned the type

```
{ f : Int -> Int | f(0) == f(3) }
```

is undecidable. It may even be difficult to translate proof obligations to the input language of an off-the-shelf theorem prover. Nonetheless, hybrid type checking techniques still apply to this language of types, with difficult cases resulting in simply more run-time checks.

7.1.3 First-Class Types

Finally, SAGE elevates types to be first-class values, following the Type Theory of Martin-Löf [1975] and Pure Type Systems of Barendregt [1991][1]. To the usual functions from *terms* to *terms* that exist in all languages, first-class types add:

[1] Type Theory and Pure Type System have inspired other languages, including Cayenne, Agda, and Epigram.

- Functions from *terms* to *types*. The following function `Range` takes two integers and returns the type of integers within that range:

```
let Range (lo:Int) (hi:Int) : * = { x:Int | lo <= x && x < hi };
```

  Here, `*` is the type of types and indicates that `Range` returns a type.

- Functions from *types* to *types*. The following function `endofunction` accepts a type `T` as a parameter and returns the type of of endofunctions over `T`:

```
let endofunction (T:*) : * = T -> T;
```

- Functions from *types* to *terms*. The following function `id` accepts a type `T` as its first parameter and returns the identity function on `T`:

```
let id (T:*) (x:T) : T = x;
```

The traditional limitation of both first-class types (in the presence of nonterminating programs) and executable refinement types is type checking is not statically decidable. SAGE applies hybrid type checking to both problems. The same mechanism supports dynamic typing. In practice, most errors are detected statically, while only more complicated violations escape static checking and are detected at run time.

7.2 Examples of Sage programs

This section examines some code snippets and then several longer example programs to illustrate the features of SAGE and give a sense of the language. The

following examples have fairly complete specifications to highlight the specification features, rather than leveraging the type `Dynamic`.

7.2.1 Binary Search Trees

We will first demonstrate SAGE programming with the oft-studied example of binary search trees, whose SAGE implementation is shown in Figure 7.1. For completeness we include the entirety of the code here, with line numbers for reference, beginning with the definition of `Range` from Section 7.1.3.

```
1: let Range (lo:Int) (hi:Int) : * =
2:   {x:Int | lo <= x && x < hi};
```

A binary search tree (`BST lo hi`) is an ordered tree containing integers in the range $[\texttt{lo}, \texttt{hi})$. A tree may either be `Empty`, or a `Node` containing an integer $\texttt{v} \in [\texttt{lo}, \texttt{hi})$ and two subtrees containing integers in the ranges $[\texttt{lo}, \texttt{v})$ and $[\texttt{v}, \texttt{hi})$, respectively.

```
4: datatype BST (lo:Int) (hi:Int) =
5:     Empty
6:   | Node of (v:Range lo hi) * (BST lo v) * (BST v hi);
```

This definition is desugared into the core language of Section 7.3. The exact encoding is not important, but the types of the constructors are, using SAGE's concrete syntax for types,

```
Empty : (lo:Int) -> (hi:Int) -> BST lo hi
Node  : (lo:Int) -> (hi:Int) ->
        (v:Range lo hi) -> (BST lo v) -> (BST v hi) ->
        BST lo hi
```

Thus, the type of binary search trees explicates the requirement that these trees must be ordered. This is a blend of the style of indexed types and direct style: The `BST` itself is indexed, but the values on the nodes are constrained by direct refinement.

The function `search` takes as arguments two integers `lo` and `hi`, a binary search tree of type (`BST lo hi`), and an integer `x` in the range contained in the tree. It returns `true` if `x` is contained within the tree.

```
 8: let rec search (lo:Int) (hi:Int) (t:BST lo hi)
 9:                (x:Range lo hi) : Bool =
10:   case t of
11:       Empty _ _ -> false
12:     | Node v l r ->
13:         if x = v then true
14:         else if x < v
15:              then search lo v l x
16:              else search v hi r x;
```

Note the use of dependent function types, where the types of the parameters `t` and `x` to `search` depend on the values of the parameters `lo` and `hi`.

The function `insert` takes similar arguments and extends the given tree with the integer `x`. It is a useful demonstration of the construction of the tree.

```
18: let rec insert (lo:Int) (hi:Int) (t:BST lo hi)
19:                (x:Range lo hi) : (BST lo hi) =
20:   case t of
21:       Empty -> Node lo hi x (Empty lo x) (Empty x hi)
22:     | Node v l r ->
23:         if x < v
24:         then Node lo hi v (insert lo v l x) r
25:         else Node lo hi v l (insert v hi r x);
```

The SAGE system uses an automatic theorem prover to statically verify that the specified ordering invariants on binary search trees are satisfied by these two functions.

No run-time checks are required for this example.

The precise type specifications enable SAGE to detect various common programming errors. For example, suppose we inadvertently used the wrong conditional test:

```
23:       if x <= v
```

For this (incorrect and ill-typed) program, SAGE will report that the specification for `insert` is violated by the first recursive call:

```
line 25:  x does not have type (Range lo v)
```

Similarly, if one of the arguments to the constructor `Node` is incorrect, *e.g.*:

```
26:       else Node lo hi v r (insert v hi r x);
```

SAGE will report the type error:

```
line 26:  r does not have type (BST lo v)
```

Using this `BST` implementation, constructing trees with specific constraints is straightforward and verifiable. For example, the following code constructs a tree containing only positive numbers:

```
let PosBST : * = BST 1 MAXINT;

let nil :  PosBST = Empty 1 MAXINT;
let add (t:PosBST) (x:Range 1 MAXINT) : PosBST =
        insert 1 MAXINT t x;
let find (t:PosBST) (x:Range 1 MAXINT) : Bool =
        search 1 MAXINT t x;

let t :  PosBST = add (add (add nil 1) 3) 5;
```

Figure 7.1: Binary Search Trees in Sage

```
 1: let Range (lo:Int) (hi:Int) : * =
 2:   {x:Int | lo <= x && x < hi};
 3:
 4: datatype BST (lo:Int) (hi:Int) =
 5:     Empty
 6:   | Node of (v:Range lo hi) * (BST lo v) * (BST v hi);
 7:
 8: let rec search (lo:Int) (hi:Int) (t:BST lo hi)
 9:                (x:Range lo hi) : Bool =
10:   case t of
11:       Empty -> false
12:     | Node v l r ->
13:         if x = v then true
14:         else if x < v
15:              then search lo v l x
16:              else search v hi r x;
17:
18: let rec insert (lo:Int) (hi:Int) (t:BST lo hi)
19:                (x:Range lo hi) : (BST lo hi) =
20:   case t of
21:       Empty -> Node lo hi x (Empty lo x) (Empty x hi)
22:     | Node v l r ->
23:         if x < v
24:         then Node lo hi v (insert lo v l x) r
25:         else Node lo hi v l (insert v hi r x);
```

Figure 7.2: Regular Expressions in Sage

```
datatype Regexp =
  Alpha
| AlphaNum
| Kleene of Regexp
| Concat of Regexp * Regexp
| Or of Regexp * Regexp
| Empty;

let match (r:Regexp) (s:String) :  Bool = ...

let Credential = {s:String | match (Kleene AlphaNum) s};
```

Note that this fully-typed BST implementation inter-operates with dynamically-typed client code:

```
let t :  Dynamic = (add nil 1) in find t 5;
```

7.2.2 Regular Expressions

Regular expressions are a common means of validating string data, and serve as a good example of a specification beyond the common constraints using linear arithmetic. Figure 7.2 declares the `Regexp` data type and the function `match`, which determines if a string matches a regular expression. To avoid irrelevant complexity, `Regexp` only expresses regular expressions over the two character classes of alphabetic characters (`Alpha`) and alphanumeric characters (`AlphaNum`). In addition to these primitive character classes, we include the usual the Kleene closure, concatenation, and choice operators. As an example, the regular expression "`[a-zA-Z0-9]*`" would be represented

in our datatype as `(Kleene AlphaNum)`.[2]

The code then uses `match` to define the type `Credential`, which refines the type `String` to allow only alphanumeric strings. We use the type `Credential` to enforce an important, security-related interface specification for the following function `authenticate`. This function performs authentication by querying a SQL database (where '^' denotes string concatenation):

```
let authenticate (user:Credential) (pass:Credential) :  Bool =
   let query :  String =
     ("SELECT count(*) FROM client WHERE name =" ^
     user ^ " and pwd=" ^ pass) in
   executeSQLquery(query) > 0;
```

This code is prone to SQL injection attacks if given specially-crafted non-alphanumeric strings. For example, calling

```
authenticate  "admin --"  ""
```

breaks the authentication mechanism because "`--`" starts a comment in SQL and consequently "comments out" the password part of the query. To prohibit this vulnerability, the type:

```
authenticate :  Credential → Credential → Bool
```

specifies that `authenticate` should be applied only to alphanumeric strings.

Next, consider the following user-interface code:

```
let username :  String = readString() in
let password :  String = readString() in
authenticate username password;
```

[2]Assuming ASCII, since a more robust character set is not important for this demonstration.

This code is ill-typed, since it passes arbitrary user input of type `String` to `authenticate`. However, proving that this code is ill-typed is quite difficult, since it depends on complex reasoning showing that the user-defined function `match` is not a tautology, *i.e.* not all `String`s are `Credential`s.

In fact, the current implementation of SAGE cannot statically verify or refute this code. Instead, it inserts the following casts at the call site to enforce the specification for `authenticate` dynamically:

```
authenticate (⟨Credential⟩username) (⟨Credential⟩password);
```

At run time, these casts check that `username` and `password` are alphanumeric strings satisfying the predicate `match (Kleene AlphaNum)`. If the `username` "`admin --`" is ever entered, the cast `(⟨Credential⟩username)` will fail and halt program execution.

As a bonus, a dynamic cast failure actually strengthens SAGE's ability to detect type errors statically. In particular, the string "`admin --`" is a witness proving that not all `String`s are `Credential`s, *i.e.*, $\not\forall$ `String` <: `Credential`. Rather than discarding this information, and potentially observing the same error on later runs or in different programs, such refuted subtype relationships are stored in a database. If the above code is later revisited, the SAGE type checker will discover upon consulting this database that `String` is not a subtype of `Credential`, and it will statically reject the call to `authenticate` as ill-typed. Additionally, the database stores a list of other programs previously type-checked under the assumption that `String` may be a subtype of `Credential`. These programs may also fail at run time and SAGE will also report

that they should be revisited using the more-informed type checker. This database is discussed further after the formal developments, in Section 7.7.2.

7.2.3 Printf

Our final example is the classic `printf` function. The number and type of the expected arguments to `printf` depends in subtle ways on the format string (the first argument). In SAGE, we can assign to `printf` the precise type:

```
printf :  (format:String) -> (Printf_Args format)
```

where the user-defined function

```
Printf_Args :  String -> *
```

returns the `printf` argument types for the given format string.[3] For example,

```
(Printf_Args "%d%d")
```

evaluates to the type

```
Int -> Int -> Unit
```

Thus, the SAGE language is sufficiently expressive to need no special support for accommodating `printf` and catching errors in `printf` clients statically. In contrast, other language implementations require special rules in the compiler or run time to ensure the type safety of calls to `printf`. For example, Scheme [Sperber et al. 2007] and GHC [Marlow 2010] leave all type checking of arguments to the run time. OCaml [Leroy

[3] The implementation of `Printf_Args` is more complex than warrants textual inclusion.

et al. 2013], on the other hand, performs static checking, but it requires the format string to be constant.

SAGE can statically check many uses of `printf` with non-constant format strings, as illustrated by the following example:

```
let repeat (s:String) (n:Int) : String =
  if (n = 0) then "" else (s \verb@^@ (repeat s (n-1)));

  // checked statically:
printf (repeat "\%d" 2) 1 2;
```

The SAGE type checker infers that `printf (repeat "%d" 2)` has type `Printf_Args (repeat "%d" 2)`, which evaluates (at cast insertion time) to `Int` $\rightarrow$ `Int` $\rightarrow$ `Unit`, and hence this call is well-typed. Conversely, the type checker would statically reject the following ill-typed call:

```
// compile-time error:
printf (repeat "%d" 2) 1 false;
```

For efficiency, and to avoid non-termination, the type checker performs only a bounded number of evaluation steps before resorting to dynamic checking. Thus, the following call requires a run-time check:

```
// run-time error:
printf (repeat "%d" 200) 1 2 ...  199 false;
```

As expected, the inserted dynamic cast catches the error.

The current SAGE implementation is not yet able to statically verify that the implementation of `printf` matches its specification,

```
(format:String) -> (Printf_Args format)
```

As a result, the type checker inserts a single dynamic type cast into the `printf` implementation. This example illustrates the flexibility of hybrid checking — the `printf` specification is enforced dynamically on the `printf` implementation, but also enforced (primarily) statically on client code. We revisit this example in Section 7.6.4 to illustrate Sage's cast insertion algorithm.

7.3 The Sage Core Language

Sage programs are desugared into a small core language shown in Figure 7.3. We use the same metavariable conventions from the rest of this dissertation with few differences that will be clear from context. Most importantly, the syntactic sort of types and terms is merged, so the metavariables used for terms (e, f, p, etc) and those used for types (S, T, etc) now refer to the same syntactic sort. The choice of which metavariable to use in a particular situation is only to provide some intuition.

First-class types also allow us to express all type constructors other than dependent function types as primitives, similar to the treatment of conditionals (via `if`) and recursion (via `fix`) in $\lambda_{\mathcal{H}}$. This is the fruition of Type Theory: The core constructs need not change, but form a meta-theoretical framework; we need only discuss the semantics of the primitives we include in the framework. The following primitives are crucial to the definition of Sage:

- The primitive `Refine` is used to construct executable refinement types. Suppose $f : T \to \texttt{Bool}$ is some arbitrary predicate over type T. Then the type `Refine` T f

Figure 7.3: Syntax of SAGE Core

$d, e, f, g, p, q, S, T, U ::=$	*Terms:*
x	variable
k	primitive
$\texttt{let}\ x{:}S = e\ \texttt{in}\ e$	binding
$\lambda x{:}S.\, e$	abstraction
$f\ e$	application
$x{:}S \to T$	function type
$u, v, V, W ::=$	*Values*
$\lambda x{:}S.\, e$	abstraction
$x{:}S \to T$	function type
k	primitive
$k\ v_1\ \ldots\ v_n$	primitive, $0 < n < \mathit{arity}(k)$
$\texttt{Refine}\ W\ v$	refinement
$\texttt{fix}\ W\ v$	recursive type

denotes the executable refinement type classifying all values of type T that satisfy the predicate f. Following our notation established in previous chapters, we use the shorthand $\{x{:}T \mid p\}$ to abbreviate `Refine` T $(\lambda x{:}T.\, p)$.

- The primitive `fix` enables the definition of recursive functions and recursive types. For example, the type of integer lists is defined via the fixed point calculation:

$$\texttt{fix}\ \texttt{*}\ (\lambda L{:}\texttt{*}.\, \texttt{Sum Unit (Pair Int}\ L))$$

which (roughly speaking) returns a type `L` satisfying the equation:

$$L \quad = \quad \texttt{Sum Unit (Pair Int}\ L)$$

(Here, `Sum` and `Pair` are the usual type constructors for sums and pairs, respectively.)

The subtyping algorithm treats recursion in types differently than the operational semantics treats recursion in terms, as most recursive types are infinite and result in infinite reduction sequences.

- The primitive `Dynamic` [Thatte 1990; Abadi et al. 1991] can be thought of as the most general type. Every value may be assigned type `Dynamic`, and casts can be used to convert values from type `Dynamic` to other types (and of course such downcasts may fail if applied to inappropriate values).

- The primitive `cast` performs dynamic coercions between types. It takes as arguments a type T and a value (of type `Dynamic`), and it attempts to cast that value

Figure 7.4: SAGE Shorthands

$$
\begin{array}{rcl}
\langle T \rangle & = & \texttt{cast}\ T \\
\{x{:}T \mid p\} & = & \texttt{Refine}\ T\ (\lambda x{:}T.\, p) \\
\langle \{x{:}T \mid p\}, q, v \rangle & = & \texttt{Checking}\ T\ (\lambda x{:}T.\, p)\ q\ v \\
\texttt{if}_T\ p\ \texttt{then}\ d\ \texttt{else}\ e & = & \texttt{if}\ T\ p\ (\lambda x{:}\{\texttt{Unit} \mid p\}.\, d)\ (\lambda x{:}\{\texttt{Unit} \mid \texttt{not}\ p\}.\, e)
\end{array}
$$

to type T. We use the shorthand $\langle T \rangle\ e$ to abbreviate $\texttt{cast}\ T\ e$. Thus, for example, the expression

$$\langle \{x{:}\texttt{Int} \mid x \geq 0\} \rangle\ y$$

casts the integer y to the refinement type of natural numbers, and fails if y is negative.

- The primitive `Checking` is used to construct casts-in-progress while a refinement predicate is being checked. Following the notation established in previous chapters, we will write $\langle \{x{:}T \mid p\}, q, v \rangle$ to abbreviate $\texttt{Checking}\ T\ (\lambda x{:}T.\, p)\ v\ q$.

Shorthands for selected primitives are summarized in Figure 7.4.

7.4 Operational Semantics of Sage Core

Figure 7.5 and Figure 7.6 describe the meaning of SAGE programs with a small-step operational semantics. Because much of the interest lies in the meaning of

primitives, we do not embed all of this meaning in a δ function, but write out in full the meaning of those primitives whose semantics bear discussion. We do, however, elide standard semantics such as the basics of boolean and arithmetic connectives. The semantics the primitive `if` are presented as an example of a primitive with standard semantics. The semantics of `cast` are presented in full.

Evaluation is performed inside call-by-value evaluation contexts $\mathcal{E}$, to model the actual SAGE execution strategy. Arbitrary contexts are still written $\mathcal{C}$.

[E-APP] and [E-LET] Both λ-abstractions and `let`-bindings evaluate according to β-reduction, replacing their bound variable with the actual value in their body.

[E-FIX]: The operation `fix` is used to define recursive functions and types, which are considered values, and hence `fix` U v is also a value. However, when this construct `fix` U v appears in a strict position (*i.e.*, in a function position or in a cast), the rule [E-FIX] performs one step of unrolling to yield v (`fix` U v).

[E-IF1] and [E-IF2]: These rules encode the standard definition for a Church-encoded conditional. Each of the branches of the conditional is a constant function taking a `unit` parameter and returning the value of that branch.

Figure 7.6 displays all redex evaluation rules pertaining to casts in SAGE. The expanded type language and first-class types result in many more possibilities than in $\lambda_{\mathcal{H}}$.

[E-CAST-DYN] Casts to type `Dynamic` always succeed.

Figure 7.5: Operational Semantics for SAGE Core

Redex Evaluation $\boxed{d \curvearrowright e}$

$$(\lambda x{:}S.\, e)\ v \curvearrowright [x \mapsto v]\, e \qquad \text{[E-App]}$$

$$\texttt{let}\ x{:}S = v\ \texttt{in}\ e \curvearrowright [x \mapsto v]\, e \qquad \text{[E-Let]}$$

$$\mathcal{S}[\texttt{fix}\ U\ v] \curvearrowright \mathcal{S}[v\ (\texttt{fix}\ U\ v)] \qquad \text{[E-Fix]}$$

$$\texttt{if}_T\ \texttt{true}\ v_1\ v_2 \curvearrowright v_1\ \texttt{unit} \qquad \text{[E-If1]}$$

$$\texttt{if}_T\ \texttt{false}\ v_1\ v_2 \curvearrowright v_2\ \texttt{unit} \qquad \text{[E-If2]}$$

Evaluation $\boxed{d \rightsquigarrow e}$

$$\mathcal{E}[d] \rightsquigarrow \mathcal{E}[e] \quad \text{if } d \curvearrowright e \qquad \text{[E-Compat]}$$

Call-by-value Contexts $\boxed{\mathcal{C}, \mathcal{D}}$

$$\mathcal{E} ::= \bullet \mid \mathcal{E}\ e \mid v\ \mathcal{E}\ \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C}$$

Unfolding Contexts $\boxed{\mathcal{S}}$

$$\mathcal{S} ::= \bullet\ v \mid \langle \bullet \rangle\ v$$

Arbitrary Contexts $\boxed{\mathcal{C}, \mathcal{D}}$

$$\mathcal{C} ::= \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid x{:}\mathcal{C} \to T \mid x{:}S \to \mathcal{C} \mid \lambda x{:}\mathcal{C}.\, e \mid \lambda x{:}S.\, \mathcal{C}$$
$$\mid \texttt{let}\ x{:}S = \mathcal{C}\ \texttt{in}\ e \mid \texttt{let}\ x{:}\mathcal{C} = e\ \texttt{in}\ e \mid \texttt{let}\ x{:}S = e\ \texttt{in}\ \mathcal{C}$$

Figure 7.6: Operational Semantics for SAGE Casts

Redex Evaluation $\boxed{d \curvearrowright e}$

$$\langle \texttt{Dynamic} \rangle\ v \ \curvearrowright\ v \qquad \text{[E-CAST-DYN]}$$

$$\langle k \rangle\ v \ \curvearrowright\ v \qquad \text{[E-CAST-BASIC]}$$

if for $k \in \{\texttt{Int}, \texttt{Bool}, \texttt{Unit}\}$ and appropriate v

$$\langle x{:}S \to T \rangle\ v \ \curvearrowright\ \lambda x{:}S.\, \langle T \rangle\ (v\ (\langle D \rangle\ x)) \qquad \text{[E-CAST-FN]}$$

where $D = \mathit{domain}(v)$

$$\langle \texttt{Refine}\ T\ f \rangle\ v \ \curvearrowright\ \texttt{Checking}\ T\ f\ v\ (f\ (\langle T \rangle\ v)) \qquad \text{[E-CAST-BEGIN]}$$

$$\texttt{Checking}\ T\ f\ v\ \texttt{true} \ \curvearrowright\ \langle T \rangle\ v \qquad \text{[E-CAST-END]}$$

$$\langle * \rangle\ v \ \curvearrowright\ v \qquad \text{[E-CAST-TYPE]}$$

if v is of one of the forms $\texttt{Int}, \texttt{Bool}, \texttt{Unit}, \texttt{Dynamic}, *, x{:}S \to T, \texttt{Refine}\ T\ f$

$$\begin{aligned} \mathit{domain} &: \mathit{Value} \to \mathit{Term} \\ \mathit{domain}(\lambda x{:}T.\, e) &= T \\ \mathit{domain}(\texttt{fix}\ (x{:}T \to T')\ v) &= T \\ \mathit{domain}(k\ v_1 \ldots v_{i-1}) &= \text{type of } i^{th} \text{ argument to } k \end{aligned}$$

[E-Cast-Basic] Casts to primitive types such as `Int`, `Bool`, and `Unit` succeed when applied to appropriate values.

[E-Cast-Fn] Casts to function types are more involved. The casts of Sage only retain their target type (an experimental design decision) so we require the following partial function *domain* returns the domain of a function value. The rule casts a function v to type $x\!:\!S \to T$ by creating a new function:

$$\lambda x\!:\!S.\, \langle T \rangle\ v\ (\langle D \rangle\ x)$$

where $D = \mathit{domain}(v)$ is the domain type of the function v. Just as in $\lambda_{\mathcal{H}}$, this new function takes a value x of type S, casts it to D, applies the given function v, and casts the result to the desired result type T.

[E-Cast-Begin] and [E-Cast-End]: These two rules are analogous to those for $\lambda_{\mathcal{H}}$. To cast a term v to a refined type `Refine` $T\ f$, first v is cast to type T and the predicate $f\ (\langle T \rangle\, v)$ is evaluated to determine its truth or falsity. If this check succeeds, then the result of the cast is $\langle T \rangle\, v$.[4]

[E-Cast-Type]: Casts to type `*` succeed only for special values of type `*`. With first-class types, the programmer may write nonsensical function "types" such as $3 \to 6$, since this is syntactically valid. Suppose this type occurs in a cast $\langle 3 \to 6 \rangle$. Though this term is ill-typed, the hybrid type checker may not be able to determine that this is so (for example if 3 and 6 were the result of

[4]There is clearly room for optimized evaluation strategies that duplicate less work when casting to a refinement type, but the current presentation has been chosen for clarity.

a long-running computation) and may output the cast $\langle\langle * \rangle\, 3 \to \langle * \rangle\, 6\rangle$. Later evaluation via [E-Cast-Fn] will expand the cast on the domain type to $\langle\langle * \rangle\, 3\rangle$ which is not a value, and will result in a failed cast.

7.5 The Sage Core Type System

The Sage type system is defined via the following judgments:

- The typing judgment $\Gamma \vdash e : T$ denotes that expression e has type T under the assumptions in environment Γ.
- The subtyping judgment $\Gamma \vdash S <: T$ denotes that type S is a subtype of T under the assumptions in environment Γ.
- The validity judgment $\Gamma \models p$ denotes that p is valid in environment Γ. This replaces the use of $\Gamma \vdash p \Rightarrow q$ in prior chapters.

All of the above judgments utilitize an environment Γ that binds variables to types and, in some cases, to values. We assume that variables are bound at most once in an environment and, as usual, we apply implicit α-renaming of bound variables to maintain this assumption and to ensure that substitutions are capture-avoiding. In addition, we assume that the typing environment is well-formed according to the judgment $\vdash \Gamma$.

7.5.1 Typing for Sage Core

The main typing judgment

$$\Gamma \vdash e : T$$

assigns type T to term e in the environment Γ, and is defined by the rules in Figure 7.7. Note that the type well-formedness judgment $\Gamma \vdash T$ from prior chapters is merged with this judgment: Since types are terms, they are validated by being assigned the type `*` via the typing judgment $\Gamma \vdash T : \texttt{*}$. Thus, `Int`, `Bool`, and `Unit` all have type `*`. Also, `*` itself has type `*`.[5]

[T-Prim]: The auxiliary function ty returns the type of the primitive k, as defined in Figure 7.8. In Sage these types can be quite rich; note the heavy use of first-class types.

Many types are precise as in $\lambda_{\mathcal{H}}$; an integer n has the precise type $\{m{:}\texttt{Int} \mid m = n\}$ denoting the singleton set $\{n\}$.

In particular, note the type of `if`. The "then" parameter to `if` is a thunk of type $(\{d{:}\texttt{Unit} \mid p\} \to X)$. That thunk can be invoked only if the domain $\{d{:}\texttt{Unit} \mid p\}$ is inhabited, *i.e.*, only if the test expression p evaluates to `true`. Thus the type of `if` precisely specifies its behavior; Sage has a *path-sensitive* type system.

[5] The relationship `*` : `*` results in a paradox known as Girard's Paradox [Girard 1972; Coquand 1986; Hurkens 1995] which allows one to construct a non-terminating term that is a proof of $\bot$ (falsity, or the empty type). When type theory is proposed as a foundation for mathematics, the solution is to introduce a stratification of `*` into $\texttt{*}_n$ indexed by a natural number, and insist that $\texttt{*}_n : \texttt{*}_m$ only if $n < m$. However this complexity is not needed since the paradox (not the only one in Sage) does not affect the use of a language for programming.

Figure 7.7: Type Rules for SAGE Core

$\Gamma ::=$ *Environments:*

$\emptyset$	empty environment
$\Gamma, x : T$	environment extension
$\Gamma, x = v : T$	environment term extension

Well-formed environment $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ [\text{WE-EMPTY}] \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash T \ : \ *}{\vdash \Gamma, x : T}\ [\text{WE-EXT1}]$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T \ : \ * \qquad \Gamma \vdash e \ : \ T}{\vdash \Gamma, x : T = e}\ [\text{WE-EXT2}]$$

Type rules $\boxed{\Gamma \vdash e \ : \ T}$

$$\frac{}{\Gamma \vdash k \ : \ ty(k)}\ [\text{T-CONST}] \qquad \frac{(x : T) \in \Gamma \quad \text{or} \quad (x = v : T) \in \Gamma}{\Gamma \vdash x \ : \ \{y{:}T \mid \ y = x\}}\ [\text{T-VAR}]$$

$$\frac{\Gamma \vdash S \ : \ * \qquad \Gamma, x : S \vdash e \ : \ T}{\Gamma \vdash (\lambda x{:}S.\, e) \ : \ (x{:}S \to T)}\ [\text{T-FUN}] \qquad \frac{\Gamma \vdash S \ : \ * \qquad \Gamma, x : S \vdash T \ : \ *}{\Gamma \vdash (x{:}S \to T) \ : \ *}\ [\text{T-ARROW}]$$

$$\frac{\Gamma \vdash f \ : \ (x{:}S \to T) \qquad \Gamma \vdash e \ : \ S}{\Gamma \vdash f \ e \ : \ [x \mapsto e]\, T}\ [\text{T-APP}]$$

$$\frac{\Gamma \vdash v \ : \ S \qquad \Gamma, (x = v : S) \vdash e \ : \ T}{\Gamma \vdash \texttt{let}\ x{:}S = v\ \texttt{in}\ e \ : \ [x \mapsto v]\, T}\ [\text{T-LET}] \qquad \frac{\Gamma \vdash e \ : \ S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e \ : \ T}\ [\text{T-SUB}]$$

Figure 7.8: Example types of SAGE constants

$$
\begin{array}{rcl}
\texttt{*} & : & \texttt{*} \\
\texttt{Unit} & : & \texttt{*} \\
\texttt{Bool} & : & \texttt{*} \\
\texttt{Int} & : & \texttt{*} \\
\texttt{Dynamic} & : & \texttt{*} \\
\texttt{Refine} & : & X{:}\texttt{*} \to (X \to \texttt{Bool}) \to \texttt{*} \\
 & & \\
\texttt{unit} & : & \texttt{Unit} \\
\texttt{true} & : & \{b{:}\texttt{Bool} \mid b\} \\
\texttt{false} & : & \{b{:}\texttt{Bool} \mid \texttt{not}\ b\} \\
\texttt{not} & : & b{:}\texttt{Bool} \to \{b'{:}\texttt{Bool} \mid b' = \texttt{not}\ b\} \\
n & : & \{m{:}\texttt{Int} \mid m = n\} \\
+ & : & n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
= & : & x{:}\texttt{Dynamic} \to y{:}\texttt{Dynamic} \to \{b{:}\texttt{Bool} \mid b = (x = y)\} \\
 & & \\
\texttt{if} & : & X{:}\texttt{*} \to p{:}\texttt{Bool} \to \\
 & & (\{d{:}\texttt{Unit} \mid p\} \to X) \to \\
 & & (\{d{:}\texttt{Unit} \mid \texttt{not}\ p\} \to X) \to \\
 & & X \\
 & & \\
\texttt{fix} & : & X{:}\texttt{*} \to (X \to X) \to X \\
\texttt{cast} & : & X{:}\texttt{*} \to \texttt{Dynamic} \to X \\
\texttt{Checking} & : & X{:}\texttt{*} \to f{:}(X \to \texttt{Bool}) \to x{:}X \to \{p{:}\texttt{Bool} \mid p \Rightarrow f\ x\} \to X
\end{array}
$$

[T-VAR]: For a variable x we extract the type T of x from the environment, and assign to x the singleton refinement type[6] $\{y{:}T \mid y = x\}$.

[T-FUN]: For a function $\lambda x{:}S.\, e$ we infer the type T of e in an extended environment and return the dependent function type $x{:}S \to T$, where x may occur free in T.

[T-ARROW]: The type $x{:}S \to T$ is itself a term, which is assigned type $*$, provided that both S and T have type $*$ in appropriate environments.

[T-APP]: This rule is completely standard for function application in dependent types.[7] For an application $(f\ e)$ we first check that f has a function type $x{:}S \to T$ and that e is in the domain of f. The result type is T with all occurrences of the formal parameter x replaced by the actual parameter e.

[T-LET]: For the term `let` $x{:}S = v$ `in` e we first check that the type of the bound value v is S. Then e is typed in an environment that contains both the type *and* the value of x.[8] The precise bindings from `let` expressions are used in the subtype judgment, as described below.

[T-SUB]: Subtyping is allowed at any point in a typing derivation via this general subsumptoin rule.

[6] The necessity of this "selfification" was discovered prior to the work of Chapter 6 and was influenced more by Ou et al. [2004] but seems prescient in retrospect.

[7] The compositionality-restoring techniques of Chapter 6 were developed after SAGE and have not been attempted in this more challenging context.

[8] In the surface syntax of SAGE a non-value may still be bound to an expression as in `let x:S = d in e` but this is desugared into $(\lambda x{:}S.\, e)\ d$.

7.5.2 Subtyping for Sage Core

The subtype judgment

$$\Gamma \vdash S <: T$$

states that S is a subtype of T in the environment Γ, and it is defined as the greatest solution (to permit infinite types) to the collection of subtype rules in Figure 7.9.

[S-Refl]: Every type is a subtype of itself, axiomatically.

[S-Dyn]: Every type is a subtype of the type `Dynamic`, which functions essentially as a "top" type in the static type system, though its algorithmic implementation during hybrid type checking differs.

[S-Fun]: Subtyping between function types is entirely standard.

[S-Var]: A variable may be hygienically replaced with the value to which it is bound during subtyping.

[S-Eval-L] and [S-Eval-R]: The subtype relation is closed under evaluation of terms in arbitrary positions.

[S-Ref-L]: If S is a subtype of T, then any refinement of S is also a subtype of T.

[S-Ref-R]: When S is a subtype of T we invoke the validity judgment $\Gamma \models f\ x$, discussed below, to determine if $f\ x$ is valid for all values x of type S. If so, then S is a subtype of `Refine` $T\ f$.

Figure 7.9: Subtyping Rules for Sage Core

Subtype rules $\boxed{\Gamma \vdash S <: T}$

$$\frac{}{\Gamma \vdash T <: T}\ \text{[S-Refl]} \qquad \frac{}{\Gamma \vdash T <: \texttt{Dynamic}}\ \text{[S-Dyn]}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, x : T_1 \vdash S_2 <: T_2}{\Gamma \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)}\ \text{[S-Fun]}$$

$$\frac{\Gamma, [x \mapsto v]\,\Gamma' \vdash [x \mapsto v]\,S <: [x \mapsto v]\,T}{\Gamma, x = v : U, \Gamma' \vdash S <: T}\ \text{[S-Var]} \qquad \frac{d \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}[d] <: T}{\Gamma \vdash \mathcal{C}[e] <: T}\ \text{[S-Eval-L]}$$

$$\frac{d \rightsquigarrow e \qquad \Gamma \vdash S <: \mathcal{C}[d]}{\Gamma \vdash S <: \mathcal{C}[e]}\ \text{[S-Eval-R]} \qquad \frac{\Gamma \vdash S <: T}{\Gamma \vdash (\texttt{Refine}\ S\ f) <: T}\ \text{[S-Ref-L]}$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma, x : S \models f\ x}{\Gamma \vdash S <: (\texttt{Refine}\ T\ f)}\ \text{[S-Ref-R]}$$

7.5.3 Validity for Sage Core

Our type system is parameterized with respect to the validity judgment

$$\Gamma \models p$$

which defines the validity of term p in an environment Γ. Chapter 3 demonstrates one way to construct such a judgment using denotational semantics. Since this chapter is concerned with the implementation scenario where validity checking will be implemented by an off-the-shelf theorem prover, it is more appropriate and useful to discuss the axioms to which an implementation must adhere. In the following, all environments and terms are assumed to be well-formed.

Requirement 7.1 (Validity Axioms) *A validity judgment for* Sage *must obey the following axioms.*

1. *Faithfulness:* $\emptyset \models \texttt{true}$ *and* $\emptyset \not\models \texttt{false}$.[9]
2. *Hypothesis: If* $(x : \{y{:}S \mid p\}) \in \Gamma$ *then* $\Gamma \models [y \mapsto x]\, p$.
3. *Weakening: If* $\Gamma, \Gamma'' \models p$ *then* $\Gamma, \Gamma', \Gamma'' \models p$.
4. *Substitution: If* $\Gamma, (x : S), \Gamma' \models p$ *and* $\Gamma \vdash e : S$ *then* $\Gamma, [x \mapsto e]\, \Gamma' \models [x \mapsto e]\, p$.
5. *Definition:* $\Gamma, (x = v : S), \Gamma' \models p$ *if and only if* $\Gamma, [x \mapsto v]\, \Gamma' \models [x \mapsto v]\, p$.
6. *Preservation: If* $e \rightsquigarrow^* e'$, *then* $\Gamma \models \mathcal{C}[e]$ *if and only if* $\Gamma \models \mathcal{C}[e']$.

[9] Early presentations of this material erroneously included an environment in this axiom. We extend thanks to Michael Greenberg for pointing this out.

7. *Narrowing: If* $\Gamma, (x : T), \Gamma' \models p$ *and* $\Gamma \vdash S <: T$ *then* $\Gamma, (x : S), \Gamma' \models p$.

In addition to the undecidability induced by theorem proving, the subtype judgment may require an unbounded amount of evaluation during type checking. These decidability limitations motivate the development of the hybrid type checking techniques of the following section.

The SAGE type system guarantees type soundness through the standard lemmas progress (well-typed programs can only get stuck due to failed casts) and preservation (evaluation of a term preserves its type). The proofs are lengthy but standard; for detail consult Gronski et al. [2006].

7.6 Hybrid Type Checking for Sage

Hybrid type checking for SAGE, as for $\lambda_{\mathcal{H}}$, relies on the cooperation between a subtyping algorithm and a cast insertion algorithm, as well as an algorithmic theorem proving judgment. The key judgments are:

- The cast insertion judgment $\Gamma \Vdash d \hookrightarrow e : T$ elaborates the source term d, in environment Γ, to a safe term e adding casts where necessary and synthesizing T.
- The cast insertion and checking judgment $\Gamma \Vdash d \hookrightarrow e \downarrow T$ takes as an input the desired type T and ensures that e has type T by adding necessary casts as determined by the conclusions from the subtyping algorithm.
- The algorithmic subtyping judgment $\Gamma \Vdash^a S <: T$ checks for a subtyping relationship between S and T and has a three-valued result a, as usual for HTC.

- The algorithmic validity judgment, also called "theorem proving", $\Gamma \models^{a}_{alg} p$ attempts to prove or refute p, and has a three-valued result a as well.

- The database of counterexamples $\Gamma \vdash^{\times}_{db} S <: T$ refutes subtyping judgments for which a witness has been gathered via a runtime failure.

The cast insertion rules guarantee that the resulting program is well-typed, and thus it can only go wrong due to failed casts. In addition, this property permits type-directed optimizations on compiled code.

7.6.1 Cast Insertion and Checking

The only judgment that directly inserts casts,

$$\Gamma \Vdash d \hookrightarrow e \downarrow T$$

is defined identically for SAGE and for $\lambda_{\mathcal{H}}$.

[CC-OK]: When checking a whether term d of type S may be considered to have type T, if the S can be proven a subtype of T then the term is unchanged.

[CC-CHK]: If, however, the subtype relationship can be neither proven nor refuted, then the check is deferred until run time by inserting the cast $\langle T \rangle$.

If subtyping is refuted, the lack of an inference rule forces a static rejection of the program.

Figure 7.10: Cast Insertion and Checking for Sage Core

Cast Insertion and Checking rules $\boxed{\Gamma \Vdash d \hookrightarrow e \downarrow T}$

$$\frac{\Gamma \Vdash d \hookrightarrow e : S \qquad \Gamma \Vdash^{\checkmark} S <: T}{\Gamma \Vdash d \hookrightarrow e \downarrow T} \text{ [CC-Ok]}$$

$$\frac{\Gamma \Vdash d \hookrightarrow e : S \qquad \Gamma \Vdash^{?} S <: T}{\Gamma \Vdash d \hookrightarrow (\langle T \rangle\, e) \downarrow T} \text{ [CC-Chk]}$$

7.6.2 Cast Insertion and Synthesis

The cast insertion and checking judgment is applied for a full program by the cast insertion and synthesis judgment

$$\Gamma \Vdash d \hookrightarrow e : T$$

defined by rules in Figure 7.11. Many are similar to the corresponding type rules or analogous to the rules from Chapter 4, but a few have interesting new consequences due to Sage's additional features.

[C-Var]: This rule provides selfification of variables analogously to the type rule [T-Var].

[C-Prim]: Primitives are assumed to be well-typed, so no casts are needed.

[C-Fun]: For a function $(\lambda x : S.\, e)$ the input annotation S is, itself, a term, and is checked against type $*$.

Figure 7.11: Cast Insertion for Sage Core

Cast Insertion rules $\boxed{\Gamma \Vdash d \hookrightarrow e : T}$

$$\frac{(x : T) \in \Gamma \quad \text{or} \quad (x = e : T) \in \Gamma}{\Gamma \Vdash x \hookrightarrow x : \{y{:}T \mid y = x\}}\ [\text{C-Var}] \qquad \frac{}{\Gamma \Vdash k \hookrightarrow k : ty(k)}\ [\text{C-Prim}]$$

$$\frac{\Gamma \Vdash S \hookrightarrow S' \downarrow \star \qquad \Gamma, x : S' \Vdash e \hookrightarrow e' : T}{\Gamma \Vdash (\lambda x{:}S.\, e) \hookrightarrow (\lambda x{:}S'.\, e') : (x{:}S' \to T)}\ [\text{C-Fun}]$$

$$\frac{\Gamma \Vdash S \hookrightarrow S' \downarrow \star \qquad \Gamma, x : S' \Vdash T \hookrightarrow T' \downarrow \star}{\Gamma \Vdash (x{:}S \to T) \hookrightarrow (x{:}S' \to T') : \star}\ [\text{C-Arrow}]$$

$$\frac{\Gamma \Vdash f \hookrightarrow f' : U \qquad \mathit{unrefine}(U) = x{:}S \to T \qquad \Gamma \Vdash e \hookrightarrow e' \downarrow S}{\Gamma \Vdash f\ e \hookrightarrow f'\ e' : [x \mapsto e']\, T}\ [\text{C-App1}]$$

$$\frac{\Gamma \Vdash f \hookrightarrow f' \downarrow (\texttt{Dynamic} \to \texttt{Dynamic}) \qquad \Gamma \Vdash e \hookrightarrow e' \downarrow \texttt{Dynamic}}{\Gamma \Vdash f\ e \hookrightarrow f'\ e' : \texttt{Dynamic}}\ [\text{C-App2}]$$

$$\frac{\Gamma \Vdash S \hookrightarrow S' \downarrow \star \qquad \Gamma \Vdash v \hookrightarrow v' \downarrow S' \qquad \Gamma, (x = v' : S') \Vdash e \hookrightarrow e' : T}{\Gamma \Vdash \texttt{let}\ x{:}S = v\ \texttt{in}\ e \hookrightarrow \texttt{let}\ x{:}S' = v'\ \texttt{in}\ e' : [x \mapsto v']\, T'}\ [\text{C-Let}]$$

[C-Arrow]: For a function type $x : S \to T$, the two component types S and T are checked against type $*$.

[C-Let]: The term $\texttt{let}\ x{:}S = v\ \texttt{in}\ e$ is processed by recursively processing v, S and e in appropriate environments.

[C-App1] and [C-App2]: The rules for function application are more interesting. The rule [C-App1] processes an application $f\ e$ by processing the function f to some term f' of some type U. The type U may be a function type embedded inside refinements as in

$$\{x{:}(\texttt{Int} \to \texttt{Int}) \mid\ :\}x(0) > 0$$

In order to extract the domain and codomain of the function, we use *unrefine* to remove any outer refinements of U before checking the type of the argument e against the expected type. Formally, *unrefine* is defined as follows:

$$\begin{aligned}
\mathit{unrefine} : \mathit{Term} &\to \mathit{Term} \\
\mathit{unrefine}(x{:}S \to T) &= x{:}S \to T \\
\mathit{unrefine}(\texttt{Refine}\ T\ f) &= \mathit{unrefine}(T) \\
\mathit{unrefine}(S) &= \mathit{unrefine}(S') \qquad \text{if } S \rightsquigarrow S'
\end{aligned}$$

The last clause permits S to be simplified via evaluation while removing outer refinements. Given the expressiveness of the type system, this evaluation may not converge within a given time bound. Hence, to ensure that our compiler accepts all (arbitrarily complicated) well-typed programs, the rule [C-App2] provides a backup compilation strategy for applications that requires less static

analysis, but performs more dynamic checking. This rule checks that the function expression has the most general function type `Dynamic` $\rightarrow$ `Dynamic`, and correspondingly coerces e to type `Dynamic`, resulting in an application with type `Dynamic`.

7.6.3 Algorithmic Subtyping

For any subtype query $\Gamma \vdash S <: T$, the algorithmic subtyping judgment

$$\Gamma \Vdash^{a} S <: T$$

returns a result $a \in \{\surd, \times, ?\}$ depending on whether the algorithm succeeds in proving ($\surd$) or refuting ($\times$) the subtype query, or whether it cannot decide the query (?).

Our algorithm conservatively approximates the subtype specification in Figure 7.7. However, special care must be taken in the treatment of `Dynamic`. Since we would like values of type `Dynamic` to be implicitly cast to other types, such as `Int`, the subtype algorithm should conclude $\Gamma \Vdash^{?}$ `Dynamic` $<:$ `Int` (forcing a cast from `Dynamic` to `Int`), even though clearly $\Gamma \nvdash$ `Dynamic` $<:$ `Int`.

A naïve subtype algorithm could always return the result "?" and thus trivially satisfy these requirements, but more precise results enable SAGE to verify more properties and to detect more errors statically.

The rules in Figure 7.12 illustrate the multiple sorts of reasoning that our algorithm uses. However, these rules are not syntax directed so they do not define an algorithm without some more description. The algorithm attempts to apply the rules

in the order in which they are presented in the figure, falling into three main categories:

1. The counterexample database is checked first, so that subtyping can be immediately rejected if a counterexample is known.

2. Then the structural rules are tried in order to attempt to decompose the subtyping problem into smaller queries. If types have incompatible shapes, then the algorithm returns "$\times$".

3. Finally, if none of the above apply, the type is evaluated for some finite number of steps to try to reduce it to a form where other rules apply. The form of evaluation is specialized to only perform useful reductions: In particular, we do not expand fixpoints unless they appear in function position.

4. If no rule applies, the algorithm returns "?"

[AS-Db]: Before applying any other rules, the algorithm attempts to refute that $\Gamma \vdash S <: T$ by querying the database of previously refuted subtype relationships. The judgment $\Gamma \vdash^{\times}_{db} S <: T$ indicates that the database includes an entry stating that S is not a subtype of T in an environment Γ', where Γ and Γ' are compatible in the sense that they include the same bindings for the free variables in S and T. This compatibility requirement ensures that we only re-use a refutation in a typing context in which it is meaningful.

[AS-Eval-L] and [AS-Eval-R]: These two rules evaluate the terms representing types. The algorithm only applies these two rules a bounded number of times before

Figure 7.12: Algorithmic Subtyping for Sage Core

Algorithmic subtyping rules $\boxed{\Gamma \Vdash^{a} S <: T}$

$$\frac{\Gamma \vdash^{\times}_{db} S <: T}{\Gamma \Vdash^{\times} S <: T}\ [\text{AS-Db}] \qquad \frac{}{\Gamma \Vdash^{\surd} T <: T}\ [\text{AS-Refl}]$$

$$\frac{\Gamma \Vdash^{a} T_1 <: S_1 \qquad \Gamma, x : T_1 \Vdash^{b} S_2 <: T_2}{\Gamma \Vdash^{a \sqcap b} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)}\ [\text{AS-Fun}] \qquad \frac{}{\Gamma \Vdash^{?} \texttt{Dynamic} <: T}\ [\text{AS-Dyn-L}]$$

$$\frac{}{\Gamma \Vdash^{\surd} S <: \texttt{Dynamic}}\ [\text{AS-Dyn-R}] \qquad \frac{\Gamma \Vdash^{a} S <: T \quad a \in \{\surd, ?\}}{\Gamma \Vdash^{a} (\texttt{Refine}\ S\ f) <: T}\ [\text{AS-Ref-L}]$$

$$\frac{\Gamma \Vdash^{a} S <: T \qquad \Gamma, x : S \models^{b}_{alg} f\ x}{\Gamma \Vdash^{a \sqcap b} S <: (\texttt{Refine}\ T\ f)}\ [\text{AS-Ref-R}]$$

$$\frac{\Gamma, [x \mapsto v]\, F \Vdash^{a} [x \mapsto v]\, S <: [x \mapsto v]\, T}{\Gamma, x = v : u, F \Vdash^{a} S <: T}\ [\text{AS-Var}]$$

$$\frac{e \curvearrowright e' \quad \Gamma \Vdash^{a} D[e'] <: T}{\Gamma \Vdash^{a} D[e] <: T}\ [\text{AS-Eval-L}] \qquad \frac{e \curvearrowright e' \quad \Gamma \Vdash^{a} S <: D_2[e']}{\Gamma \Vdash^{a} S <: D_2[e]}\ [\text{AS-Eval-R}]$$

$$D \ ::= \ \bullet \mid N\ D \qquad \text{where } N \text{ is a normal form}$$

timing out and forcing the algorithm to use a different rule or return "?". This prevents non-terminating computation as well as infinite unrolling of recursive types.

[AS-Dyn-L] and [AS-Dyn-R]: These rules ensure that any type can be considered a subtype of `Dynamic` and that converting from `Dynamic` to any type requires an explicit coercion.

[AS-Ref-R]: This rule for checking whether S is a subtype of a specific refinement type relies on a theorem-proving algorithm, $\Gamma \models^{a}_{alg} p$, for checking validity. This algorithm is an approximation of some validity judgment $\Gamma \models p$ satisfying the axioms in Requirement 7.1. As with subtyping, the result $a \in \{\surd, ?, \times\}$ indicates whether or not the theorem prover could prove or refute the validity of p. The algorithmic theorem proving judgment must be conservative with respect to the logic it is approximating, as captured in the following requirement:

Requirement 7.2 (Algorithmic Theorem Proving)

1. *If* $\Gamma \models^{\surd}_{alg} p$ *then* $\Gamma \models p$.
2. *If* $\Gamma \models^{\times}_{alg} p$ *then* $\forall \Gamma', p'$ *obtained from* Γ *and* p *by replacing each occurrence of the type* `Dynamic` *by any type, we have that* $\Gamma' \not\models p'$.

Our current implementation of this theorem-proving algorithm translates the query $\Gamma \models^{a}_{alg} p$ into input for the Simplify theorem prover [Detlefs et al. 2005].

For example, the query

$$x : \{x{:}\texttt{Int} \mid x \geq 0\} \models^{a}_{alg} x + x \geq 0$$

is translated into the Simplify query:

```
(IMPLIES (>= x 0) (>= (+ x x) 0))
```

for which Simplify returns `Valid`. Given the incompleteness of Simplify (and other theorem provers), care must be taken in how the Simplify results are interpreted. For example, for the translated version of the query

$$x : \texttt{Int} \models^{a}_{alg} x * x \geq 0$$

Simplify returns `Invalid`, because it is incomplete for arbitrary multiplication. In this case, the SAGE theorem prover returns the result "?" to indicate that the validity of the query is unknown. We currently assume that the theorem prover is complete for linear integer arithmetic. Simplify has very effective heuristics for integer arithmetic, but does not fully satisfy this specification. More recent SMT provers are explicit about when they mean "maybe" instead of erroneously saying "no".

An effective and mature hybrid type checking implementation will rely on compile-time evaluation of terms, a counterexample database, and a theorem prover that returns three-valued results.

Assuming that $\Gamma \models^{a}_{alg} p$ satisfies Requirement 7.2 and that $\Gamma \vdash^{\times}_{db} S <: T$ only if $\Gamma \nvdash$

$S <: T$ (*i.e.*the database has not been corrupted), it is straight-forward to show that the subtype algorithm $\Gamma \Vdash^a S <: T$ satisfies Lemma 7.3.

Lemma 7.3 (Algorithmic Subtyping)

1. *If* $\Gamma \Vdash^{\surd} S <: T$ *then* $\Gamma \vdash S <: T$.

2. *If* $\Gamma \Vdash^{\times} T_1 <: T_2$ *then* $\forall \Gamma', S_1, S_2$ *that are obtained from* Γ, T_1, T_2 *by replacing the type* `Dynamic` *by any type, we have that* $\Gamma' \nvdash S_1 <: S_2$.

7.6.4 Walkthrough of hybrid type checking in Sage

To illustrate in detail the process of hybrid type checking for Sage, this section walks through an example, revisiting `printf`. Consider the following term

$$e \stackrel{\text{def}}{=} \texttt{printf "\%d" 4}$$

For this term, the rule [C-App1] will first compile the subexpression `(printf "%d")` via the following cast insertion judgment (based on the type of `printf` from Section 7.2.3):

$$\emptyset \Vdash \texttt{(printf "\%d")} \hookrightarrow \texttt{(printf "\%d") : (Printf\_Args "\%d")}$$

The rule [C-App1] then calls the function *unrefine* to evaluate `(Printf_Args "%d")` to the normal form `Int → Unit`. Since 4 has type `Int`, the term e is therefore accepted as is; no casts are needed.

However, the computation for `(Printf_Args "%d")` may not terminate within a preset time limit. In this case, rule [C-App2] applies to e:

$$(\langle \texttt{Dynamic} \to \texttt{Dynamic} \rangle \texttt{(printf "\%d")})\ 4$$

At run time, `(printf "%d")` will evaluate to some function $(\lambda x : \texttt{Int}.\, e')$ that expects an `Int`, yielding the application:

$$(\langle \texttt{Dynamic} \to \texttt{Dynamic} \rangle (\lambda x : \texttt{Int}.\, e'))\ \ 4$$

The rule [E-Cast-Fn] then reduces this term to:

$$(\lambda x : \texttt{Dynamic}.\, \langle \texttt{Dynamic} \rangle ((\lambda x : \texttt{Int}.\, e')\ (\langle \texttt{Int} \rangle x)))\ \ 4$$

where the nested cast $\langle \texttt{Int} \rangle x$ dynamically ensures that the next argument to `printf` must be an integer.

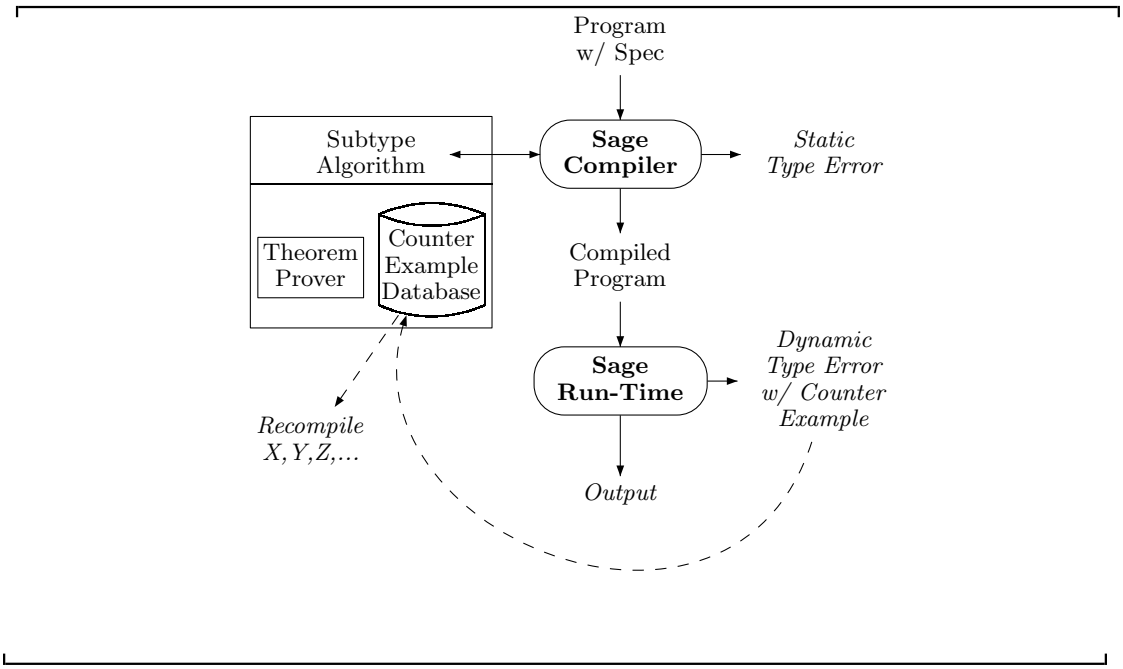
7.7 The Sage Architecture

The overall architecture of the Sage system is shown in Figure 7.13. The subtyping algorithm described formally earlier is where the novelty of the implementation architecture lies, specifically in the interplay between a counterexample database and an automated theorem prover.

7.7.1 Theorem Prover

Testing subtyping between refinement types reduces to testing implication between refinement predicates. In order to reason about these predicates, the subtype algorithm translates each implication between Sage predicates into a validity test of a logical formula, which can be passed to an automated theorem prover. Some Sage terms may not readily translate to the language of the prover, in which case we may

Figure 7.13: SAGE Architecture



either return ? or approximate the formula by one that is stronger (then a $\surd$ answer implies soundness, while $\times$ from the prover must be translated to ?) or one that is weaker (in this case it is $\surd$ that must be translated to ?).

7.7.2 Counter-Example Database

Suppose SAGE inserts the cast $(\langle T \rangle\ e)$ because it cannot prove or refute some subtype test $\Gamma \vdash S <: T$. If that cast fails, the run time inserts an entry into the database asserting that $\Gamma \nvdash S <: T$ (the syntactic representation of which is is $\Gamma \vdash^{\times}_{db} S <: T$) The subtype algorithm consults this database during type checking and will subsequently reject any program that relies on S being a subtype of T. Thus, dynamic type errors actually improve the ability of the SAGE type checker to detect type errors statically.

Moreover, when such a cast fails, SAGE will report a list of known programs that contain a cast that this counterexample refutes, since these programs may also fail at run time. Thus, the counter-example database functions somewhat like a regression test suite, in that it can detect errors in previously type-checked programs.

Function casts must be treated with care to ensure blame is assigned appropriately upon failure [Findler and Felleisen 2002]. In particular, if a cast inserted during the lazy evaluation of a function cast fails, an entry for the original, top-level function cast is inserted into the database in addition to the "smaller" cast on the argument or return value. The "larger" counterexample more directly corresponds to the error in the original source code, while the "smaller" counterexample refutes potentially more

subtyping queries. The details of this process have not been formally modeled.

Currently we prototype via a local database, but in practice this would take the form of a "crash report" sent back to a program's maintainer. Such reports have become ubiquitous subsequent to this research; this confluence of our experiments and social currents has opened up an exciting avenue by which to integrate our techniques into common practice. Over time, we predict that the database will grow to be a valuable repository of common but invalid subtype relationships, leading to further improvements in the checker's precision and less reliance on inserted casts.

7.8 Experimental Results

Our prototype SAGE implementation consists of roughly 5,000 lines of OCaml code, including parsing, desugaring, primitive definitions, operational semantics, and cast insertion. The theorem prover used is Simplify [Detlefs et al. 2005] and the counterexample database is prototyped using a simple local file and serialized OCaml data structures. The complete implementation is available at `http://sage.soe.ucsc.edu/`.

We evaluated the SAGE language and implementation using the benchmarks listed in Figure 7.1. The program `arith.sage` defines and uses a number of mathematical functions, such as `min`, `abs`, and `mod`, where refinement types provide precise specifications. The programs `bst.sage` and `heap.sage` implement and use binary search trees and heaps, and the program `polylist.sage` defines and manipulates polymorphic lists. The types of these data structures ensure that every operation preserves key invariants.

The program `stlc.sage` implements a type checker and evaluator for the simply-typed lambda calculus (STLC), where SAGE types specify that evaluating an STLC-term preserves its STLC-type. We also include the sorting algorithm `mergesort.sage`, as well as the `regexp.sage` and `printf.sage` examples discussed earlier.

Figure 7.1 characterizes the performance of the subtype algorithm on these benchmarks. We consider two configurations of this algorithm, both with and without the theorem prover. For each configuration, the figure shows the number of subtyping judgments proved (denoted by $\surd$), refuted (denoted by $\times$), and left undecided (denoted by ?). The benchmarks are all well-typed, so no subtype queries are refuted. Note that the theorem prover enables SAGE to decide many more subtype queries. In particular, many of the benchmarks include complex refinement types that use integer arithmetic to specify ordering and structure invariants; theorem proving is particularly helpful in verifying these benchmarks.

Our subtyping algorithm verifies a large majority of subtype tests. Only a small number of undecided queries result in casts. For example, in `regexp.sage`, SAGE cannot statically verify subtyping relations involving regular expressions (they are checked dynamically) but it statically verifies all other subtype judgments. Some complicated tests in `stlc.sage` and `printf.sage` must also be checked dynamically.

Despite the use of a theorem prover, type-checking times for these benchmarks is quite manageable. On a 3GHz Pentium 4 Xeon processor running Linux 2.6.14, type-checking required fewer than 10 seconds for each of the benchmarks, except for `polylist.sage` which took approximately 18 seconds. We also measured the number of

Benchmark	Lines of code	Without Prover √	?	×	With Prover √	?	×
`arith.sage`	45	132	13	0	145	0	0
`bst.sage`	62	344	28	0	372	0	0
`heap.sage`	69	322	34	0	356	0	0
`mergesort.sage`	80	437	31	0	468	0	0
`polylist.sage`	397	2338	5	0	2343	0	0
`printf.sage`	228	321	1	0	321	1	0
`regexp.sage`	113	391	2	0	391	2	0
`stlc.sage`	227	677	11	0	677	11	0
Total	1221	4962	125	0	5073	14	0

Table 7.1: Subtyping Algorithm Statistics

evaluation steps required during each subtype test. We found that 83% of the subtype tests required no evaluation, 91% required five or fewer steps, and only a handful of the the tests in our benchmarks required more than 50 evaluation steps. Of course, evaluation steps are not good measurements for run-time performance, as a mature implementation may have a highly optimized method of normalizing types, but this serves as a qualitative indication that the amount of evaluation is extremely small or none.

Chapter 8

Contemporaneous Related Work

Subsequent to and simultaneous with the work presented here, many of the fields with which this dissertation is in conversation have experienced advances from other research. This chapter discusses this related work in four categories, roughly parallel to those from Chapter 2: Dynamic contracts (Section 8.1), static contract checking (Section 8.2), dependent/refinement types (Section 8.3), and gradual types (Section 8.4).

8.1 Dynamic Contracts

The higher-order contracts of Findler and Felleisen [2002] have seen many advances since the research presented in this dissertation was undertaken. Let us consider them in three broad categories: foundations for higher-order contracts and blame, explorations of contracts in typed languages, and static checking of contracts.

8.1.1 Foundations for untyped higher-order contracts and blame

Findler and Felleisen [2002] does not directly address the question of what it means for a program to satisfy a contract. Instead, the meaning of contract satisfaction is left implicit in the semantics of contract checking: If the contract does not blame a piece of code for a violation, then that code obeys the contract. Subsequent examination of the meaning of contract satisfaction and blame is closely related to the dynamic checking of executable refinement types. In turn, the correspondence between refined notions of correctness for dynamic checking and the firmly established definition of type soundness is a primary contribution of variants of $\lambda_{\mathcal{H}}$, especially $\lambda_{\mathcal{H}}^{\text{HTC}}$.

Blume and McAllester [2006] and Findler and Blume [2006] explore denotational models of contracts in an untyped setting, proposing non-tautological, implementation-independent, definitions of contract satisfaction. In the process, they correct some missed contract violations from Findler and Felleisen [2002]. The denotational semantics for executable refinement types of Chapter 3 are inspired by these denotational models, but made simpler by leveraging the simply-typed λ-calculus.

In all of the above research on contracts, a contract is annotated with two blame labels: One for the term it is meant to describe, and one for its context. If the term violates its contract, then its label receives the blame. For example, the term 3 does not satisfy the contract $\{x\!:\!\texttt{Int} \mid x = 4\}$ so it will be blamed if the contract is applied. On the other hand, if the context violates the contract of a term, then the context's label receives blame. For example, if f has the contract $\{x\!:\!\texttt{Int} \mid x > 0\} \to \texttt{Int}$ and the

context attempts to evaluate $f\ 0$, then the context is to blame for the resulting contract violation.

Gronski and Flanagan [2007] question whether both blame labels are necessary, and answer ambivalently. They exhibit a mapping from a language λ_C with dynamic contracts that are independent of the type system to a language like $\lambda_{\mathcal{H}}^{\text{HTC}}$ with type casts that are tightly integrated into the type system. In λ_C, contract checks are of the form $\langle T \rangle^{l,l'}$ – they have *one* contract and *two* blame labels. In their variant of $\lambda_{\mathcal{H}}^{\text{HTC}}$, type casts are of the form $\langle T \lhd S \rangle^{l}$ – they have *two* types per cast but just *one* blame label. A single blame label suffices per cast, but each contract check from λ_C is interpreted as two composed casts carrying each label. In essence, the contract check

$$\langle T \rangle^{l,l'}$$

is translated into the composed type casts

$$\langle T \lhd \lfloor T \rfloor \rangle^{l'} \circ \langle \lfloor T \rfloor \lhd T \rangle^{l}$$

where one cast only ever blames the term to which the contract is applied and one cast only every blames its context. This decomposition corresponds with the reasoning of Hinze et al. [2006] and Wadler and Findler [2009], each of whom disentangle the "positive" and "negative" components of a contract.

Greenberg et al. [2012] follows up Gronski and Flanagan [2007] with an examination of whether a domain contract is rechecked when the input to a function appears in the codomain contract, yielding distinct contract disciplines. When the domain contract is not rechecked, the system is called "lax". A function contract $(\langle x{:}T_1 \to T_2 \rangle\ f)\ e$

in in the lax system reduces to

$$\langle [x \mapsto e]\, T_2 \rangle \quad (f \quad (\langle T_1 \rangle \ e))$$

When function contracts are reechecked, the system is called "picky". In the picky system, the same contract check reduces to

$$\langle [x \mapsto \langle T_1 \rangle \ e]\, T_2 \rangle \quad (f \quad (\langle T_1 \rangle \ e))$$

The lax discipline misses violations of T_1 in the codomain T_2, thus may lead to erroneous crashes; it is incorrect regardless of blame. Dimoulas and Felleisen [2011], Dimoulas et al. [2011] and Dimoulas et al. [2012] further examine the two-label contract disciplines and expose that while the lax discipline misses some violations, the picky system may blame the wrong module. This is resolved by introducing a third blame label for the contract itself, treating it as an independent party mediating the interaction between a term and its context. Notably, the lax/picky distinction is impossible in $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$. Supposing f has original type $x\!:\!S_1 \to S_2$, the analogous term $(\langle x\!:\!T_1 \to T_2 \lhd x\!:\!S_1 \to S_2 \rangle \ f)\ e$ reduces to

$$\langle [x \mapsto e]\, T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle \ e]\, S_2 \rangle \quad (f \quad (\langle S_1 \lhd T_1 \rangle \ e))$$

This enforcement of a function contract is entirely derived from the types; there is no choice of where to recheck the domain contract. The domain contract T_1 is *not* rechecked in T_2, because the type system makes any violation impossible. Conversely, it *must* be rechecked in S_2, otherwise this term would be ill-typed. If S_2 and T_2 are function types, then this does result in checks that are not present in the lax system, but fewer than in the picky system. The type system has yielded an integral insight into exactly which checks are needed, and why.

8.1.2 Higher-order contracts for lazy, typed languages

The foundational work for contracts took place in an untyped setting, but this dissertation is rooted firmly in type theory. The theory of executable refinement types provides one possible foundation for contracts in a typed setting. But alongside this theoretical work, other research on the foundations of typed contracts has been driven by implementations of typed higher-order contracts, generally as a library rather than as a language construct.

Hinze et al. [2006] presents an implementation of contracts as a library for Haskell. Unfortunately, applying a contract to a term can change the semantics of the term by altering its strictness. Chitil et al. [2004], Chitil and Huch [2007a,b], and Chitil [2011] develop a sound semantics for lazy assertions (which do not include blame or higher-order functions) but Degen et al. [2009, 2012] demonstrates some cases in which lazy assertions are not suitable for use in contracts due to incomplete contract monitoring. Chitil [2012] presents an implementation of contracts including blame and "picky" semantics in Haskell and briefly claims that it is easy to incorporate the "indy" monitoring of Dimoulas et al. [2011].

The feasibility of implementing typed contracts as a library is a testament to the power of modern type systems and languages, but executable refinement relate the correctness of the contract system to the correctness of the type system in a way that is not possible when contracts are implemented as a library. Hybrid type checking, for example, would add safety while improving performance, but requires contracts and

types to share their specification language.

Executable refinement types are related to the discussion of laziness only to the extent that laziness is inextricably related to concerns of complete monitoring, but some of the properties promoted in arguing for the correctness of the implementations may be interesting to consider both for the semantics of contract enforcement in $\lambda_{\mathcal{H}}^{\text{HTC}}$ and for the denotational interpretation of executable refinement types.

8.1.3 Parametric polymorphism in contracts

The languages in this dissertation lack parametric polymorphism. While functions in Sage may take a type as a parameter, the property of relational parametricity is certainly not ensured. Parametric polymorphism is common in modern typed programming languages, but it is generally a feature of a static type system. The foundation of *dynamic* enforcement of parametric polymorphism is more recent.

Pierce and Sumii [2000] and Sumii and Pierce [2004] connect the dynamic sealing of Morris [1973a,b] with parametricity using bisimulation, and this technique was refined by Sangiorgi et al. [2007]. In these systems, polymorphic values are essentially encrypted so that it is not possible to violate parametricity.

Guha et al. [2007] present initial approaches to using blame instead of encryption to enforce parametricity, then Matthews and Ahmed [2008] and Ahmed et al. [2009] combine the technique with gradual typing (discussed below in Section 8.4).

Belo et al. [2011] supports a combination of executable refinement types and polymorphic contracts, including refinements of any type. Unfortunately their refine-

ment system is considerably weakened to only include closed refinements, and their type system is noncompositional in a fundamental way, so the result is not directly applicable nor comparable.

Combining the techniques presented in this dissertation and advances in dynamic enforcement of parametric polymorphism is a promising approach both to the theoretical goal of building a unified theory of static and dynamic polymorphism and towards the pragmatic goal of an effective language with both hybrid types and guarantees of parametricity.

8.2 Static Contract Checking

The field of static contract checking is broad and not all directly relevant, so this section reviews only the most closely related work, and some highlights from the rest of the field of "SMT as a platform" [Bierman et al. 2010]. In particular, ESC/Haskell and Liquid Types are in direct conversation with work on hybrid type checking and executable refinement types

8.2.1 ESC/Haskell

Xu [2006] introduces ESC/Haskell, named by analogy with ESC/Java, that statically checks contract annotations in Haskell programs. ESC/Haskell verifies specifications by first inserting the symbol `BAD` where a contract failure would occur, then attempting to eliminate all occurrences of `BAD` with inlining, symbolic evaluations, and counterexample-guided unrolling. If the symbol remains in the program, then the pro-

gram is rejected.

In order to discuss the correctness of ESC/Haskell, Xu et al. [2009] gives a declarative definition of what it means for a term to satisfy a contract, similar to those discussed in Section 8.1 and the denotational model of Section 3.3, and proves that if ESC/Haskell eliminates all occurrences of `BAD` and the term is well-typed (like us, leveraging the underlying type system in a crucial way) then the term satisfies the contract.

Xu [2012] then changes context to O'Caml, a strict language, and makes two significant changes: It leverages an SMT solver to determine reachability of code and allows `BAD` to be left in the program optionally and checked at runtime. In the latter situation no program is rejected statically, unlike HTC.

8.2.2 Liquid Types

Rondon et al. [2008] introduces *logically qualified types*, or "liquid types" for short, which are comparable to executable refinement types in that they are based upon predicates written in the programming language at hand, but given meaning by translation to SMT[1] terms. These predicates are combined with the abstract interpretation technique of *predicate abstraction* to automatically infer refinements including loop invariants, which are verified statically. A programmer omits all refinements from their program but provides a set of template *logical qualifiers* such as $\{x > \star, \star > x, x \geq \star\}$ which are instantiated by replacing $\star$ with arbitrary program variables until constraints

[1] *Satifiability Modulo Theories*, or SMT, is a now-standard technique for automated theorem provers.

are satisfied or the program is proven to be ill-typed.[2] In addition to supporting nearly full type reconstruction for programs whose predicates fall within the provided template language, Rondon et al. [2008] includes an implementation for O'Caml and hence support ML-style polymorphism. Terauchi [2010] demonstrates how to omit the set of template logical qualifiers by using counterexample-guided refinement and an interpolating theorem prover to generate new templates until either a program is found to not be typable (vs the templates simply being too coarse) or a suitably fine-grained refinement is discovered. Rondon et al. [2010] applies liquid types to C, demonstrating the generality of the approach.

Kawaguchi et al. [2009] incorporates a modified form of dependent sum type to use the liquid types technology for verifying recursively-defined data types. They also automatically convert "measures", designated functions defined via catamorphism, into refinements on the constructors of a data type, the initial refinement design of Freeman and Pfenning [1991].

Jhala et al. [2011] explains the underlying technology in another way by reframing type reconstruction steps analogous to those from Knowles and Flanagan [2007] and Rondon et al. [2008] as a translation from a higher-order functional program to a first-order imperative program whose verification implies the verification of the source program. Chugh et al. [2012b] embeds the typing and subtyping relations themselves into the SMT logic, in order to reason about them *within* refinement predicates in a dynamically typed language.

[2] The term "liquid" type plays two roles: both as name for the whole system, but also for types whose refinements fall within the language of logical qualifiers.

Vazou et al. [2013] adds the ability for a function to be polymorphic over refinement predicates, so that even monomorphic functions such as $\texttt{max} : [\texttt{Int}] \to \texttt{Int}$ may be given types such as $[\{x\!:\!\texttt{Int} \mid isPrime\ x\}] \to \{x\!:\!\texttt{Int} \mid isPrime\ x\}$. While one could already write max as a function of type $p\!:\!(\texttt{Int} \to \texttt{Int}) \to \{x\!:\!\texttt{Int} \mid p\ x\} \to \{x\!:\!\texttt{Int} \mid p\ x\}$ with equivalent proof obligations, this is extremely cumbersome and requires anticipation of all places where a refinement may be added.

8.2.3 Other work

Bengtson et al. [2008] augments the F# language with refinement predicates written directly in an SMT-compatible constraint language to statically verify security properties of programs such as encryption protocols and access control implementations.

Bierman et al. [2010] embeds the type relation of a small functional calculus into the logic of SMT and shows that subtyping can then be implemented as implication between predicate interpretations of types.

8.3 Dependent/Refinement Types

This section briefly discusses the continuation of the work on refinement types *other than* those expressed as logical predicates ornamenting structural types. Programming directly in powerful languages based on Type Theory is now also becoming more common, and each of these allows the direct expression of many of the ideas promoted in refinement type systems.

The Stardust language [Dunfield 2007] serves as a good summary of the field, as

well as a novel system, by combining *datasort refinements* [Freeman and Pfenning 1991] and *index refinements* [Xi and Pfenning 1999] into single language that also includes *intersection types* [Davies and Pfenning 2000] and *union types* [Dunfield and Pfenning 2003, 2004].

Epigram [McBride and McKinna 2004], Agda [Norell 2009], and Coq [The Coq development team 2012] are each languages based on Type Theory where via propositions-as-types correspondence programmers may explicitly prove correctness of their programs (or any other property they may choose). Proofs are generally quite challenging and manual, but proponents often profess a great appreciation of the process of structuring their programs to be provable. Each of these languages supports and promotes programming with indexed types.

Atkey et al. [2011] and Dagand and McBride [2012] examine the foundations of when a particular recursive function over a recursively-defined type may be expressed as an index to its recursive formulation, offering a foundational look at what Kawaguchi et al. [2009] calls a "measure".

Ωmega [Sheard 2005, 2007] is a proposal that combines indexed types and user-defined kinds with common extensions of Haskell, specifically generalized algebraic data types (GADTs, also a feature of all systems built on Type Theory) and type-level functions, to provide many of the benefits of programming in Type Theory while retaining a familiar Haskell-esque flavor.

Concoqtion [Fogarty et al. 2007] layers Coq proofs atop MetaOCaml to allow manual proofs of precise specifications. Hoare Type Theory and its language YNot

[Nanevski et al. 2006, 2007, 2008, 2011, 2013] also blend Coq proofs into a programming language but specifically add monadic side-effecting computations which may be reasoned about with Hoare-style axiomatic semantics that can be related to facts proven about program values.

The balance between manual proof, automation and assurance is at stake in all such systems. A program, viewed as a document explicating the problem it solves, may be enhanced by explicit proofs, or its meaning may be obscured. One may also ask whether it is necessary for the program to be written in the same language used for carrying out proofs. Could a checker of executable refinement types generate proofs in Coq or another metatheoretical framework? In what manner can manual proof or other forms of automation be integrated into hybrid type checking?

8.4 Gradual Typing

Gradual typing [Siek and Taha 2006] is so named because it addresses the problem of mixing typed and untyped program text while being able to recover the root cause of errors. The formal techniques of gradual typing resemble those of hybrid type checking, while the use of blame is derived directly from research on higher-order contracts. Thus gradual typing is more closely related than it may appear at first.

Unlike refinement types, gradual typing is based not on subtyping but on an idea of type consistency. Two types are consistent if they can be unified by replacing occurrences of `Dynamic` in their syntax with arbitrary other types. For example `Int` $\rightarrow$

`Dynamic` is consistent when `Int` $\to$ `Int` but also with simply `Dynamic`. Unlike subtyping, compatibility cannot be transitive or else all types would be compatible. Gradual typing formalizes an idea that is implicit is SAGE's algorithmic treatment of `Dynamic`.

Siek and Taha [2007] applies gradual typing to an object-oriented language, adding subtyping alongside consistency. Siek and Vachharajani [2008] adds unification-based type inference. Notably, many naïve approaches to adding unification fail to maintain the "gradual" flavor. Rastogi et al. [2012] explores dataflow-based type inference for gradual typing in the context of ActionScript. Sagonas and Luna [2008] applies gradual typing to existing Erlang codebases, while Ren et al. [2013] does the same for Ruby. Wadler and Findler [2009] examines the use of blame in a language blending statically typed code and dynamically typed code, with contracts as an intermediary, and proves that only the dynamically typed portion can ever receive blame. By a careful tracking of blame labels, a new syntactic technique is used to prove soundness of a gradual typing system somewhat simpler than $\lambda_{\mathcal{H}}$. Ahmed et al. [2009] incorporates recent research on polymorphic contracts into a gradually typed language. Ina and Igarashi [2009, 2011] apply this to a more mainstream object-oriented setting where polymorphism is referred to as "generics".

Swamy et al. [2009] attempts a unified theory of type-directed coercion insertion. This would, if successful, apply to both hybrid type checking and gradual typing. Subtyping, where it appears in HTC, is replaced by *coercion generation*. Instead of a subsumption rule, the theory features coercion generation at those same points, in a

reorganization of the same reasoning. Consider this rule as an illustration:

$$\frac{\Gamma \Vdash d \hookrightarrow e : S \qquad \Gamma \Vdash \langle T \lhd S \rangle \hookrightarrow c}{\Gamma \Vdash d \hookrightarrow \langle\langle c \rangle\rangle(e) \downarrow T}$$

where $\Gamma \Vdash \langle T \lhd S \rangle \hookrightarrow c$ states that in order to coerce a term of type S to type T, the coercion c must be applied. In the event that c is the identity, $\langle\langle c \rangle\rangle(e)$ elides its application, resulting in just e.[3] This framework can generate a system equivalent to hybrid type checking via the following two rules for generating coercions

$$\frac{\Gamma \Vdash^{?} S <: T}{\Gamma \Vdash \langle T \lhd S \rangle \hookrightarrow \langle T \lhd S \rangle} \qquad \frac{\Gamma \Vdash^{\surd} S <: T}{\Gamma \Vdash \langle T \lhd S \rangle \hookrightarrow id}$$

where id is a designated symbol and

$$\langle\langle \langle T \lhd S \rangle \rangle\rangle(e) \stackrel{\text{def}}{=} \langle T \lhd S \rangle\ e$$

$$\langle\langle id \rangle\rangle(e) \stackrel{\text{def}}{=} e$$

Gradual typing may be similarly generated by using consistency checks to determine the coercion.

Implementation of gradual typing faces performance challenges from the use of higher-order coercions and also simply having dynamically typed portions. Siek and Wadler [2009] and Herman et al. [2010] address the problem of the proliferation of function contract cast wrappers, which pose a serious problem to adoption of higher-order constracts due to memory consumption. Allende and Fabry [2011] proposes some

[3]The operation $\langle\langle c \rangle\rangle$ is a metafunction carried out on the syntax, not a term of the language.

initial steps towards recovering type-based optimizations for programs using gradual typing.

Bayne et al. [2011] allows a programmer to work while their program is in an ill-typed state by optionally deferring type errors until run time. Their proposal works by replacing ill-typed terms by $\bot$, which may be assigned any type. The Glasgow Haskell Compiler now incorporates this feature. In a hybrid type checking system, this may be viewed as optionally using a theorem prover that converts all "no" answers to "maybe" during development.

Wolff et al. [2011] blends gradual typing with *typestate*, where imperative update may change the type of an imperative reference cell. In the context of executable refinement types, this may present a way to temporarily abandon refinements and then dynamically check that they have been restored.

Politz et al. [2012] takes a complementary approach to gradual typing: Instead of offering a single form of soundness and allowing the program to omit types here and there to control how much they take advantage of the guarantee, their system offers control over what *guarantee* the programmer wants and then enforces types appropriately. This obviously applies to HTC, and especially Sage. Any HTC system incorporating may present itself as different subtyping algorithms that are more or less confident in their rejections of programs.

Chapter 9

Conclusion

9.1 Summary

Types are a proven and widespread language for describing programs rigorously, but most type system express only very coarse specifications. Refinement types increase the expressiveness of types, but remain limited in order to ensure decidable and efficient checking. Dynamic contracts and extended static checking each support more expressive specification languages that structural type system, but suffer from incomplete checking and excessive false positives, respectively.

Executable refinement types enrich traditional type systems with precise specifications as expressive as those of dynamic contracts and extended static checking. This expressiveness renders traditional forms of type checking and type reconstruction undecidable.

This dissertation has introduced and explored executable refinement types

from theoretical and practical perspectives, yielding the following contributions:

- Executable refinement types may be interpreted via a lightweight denotational semantics as sets of terms of the simply-typed λ-calculus. Based on this denotational semantics, it follows that implication and subtyping are proper preorders and that the type system of $\lambda_{\mathcal{H}}$ is sound. In addition to type safety, this dissertation presents a new definition of extensional equivalence for dependently typed calculi and proves that for $\lambda_{\mathcal{H}}$ extensional equivalence implies contextual equivalence.

- Chapter 4 defines hybrid type checking, a technique combining static type checking with dynamic contract checking, and applies it to executable refinement types via the calculus $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$. By employing hybrid type checking, we may ensure that all specification violations are caught either statically or dynamically. Furthermore, for any decidable approximation of the $\lambda_{\mathcal{H}}^{\mathrm{HTC}}$ type system, there is some program which hybrid type checking rejects *statically* that the decidable approximation fails to reject. Using our new proof extensional equivalence, we have quickly proved that whenever a program is well-typed, all inserted run-time checks succeed.

- Type reconstruction as traditionally defined conflates the reconstruction of omitted types with the checking of well-typedness of a program. This conflation obscures useful distinctions between the difficulty of the type reconstruction problem and the type checking problem, and renders the definition useless for undecidable type systems. This dissertation proposes a novel definition of type reconstruction

that applies to undecidable type systems while coincides with the old definition for decidable type systems. Using this new definition, a type reconstruction algorithm for executable refinement types is presented.

- Standard formulations of dependent types rely on non-compositional reasoning, violating the abstraction that types represent. We have shown how to restore compositionality by a new use of existential types. Even with this formulation that is restricted to compositional reasoning, type checking can still captures the complete semantics of a term – the precision of the system is entirely parameterized by the precision of the types of primitives. When all predicates are written in a decidable language, compositional type checking of executable refinement types is decidable.

- Sage is a prototype implementation of hybrid type checking for executable refinement types. Experiments show that type checking time – including use of an SMT solver – is short and that the number of run-time checks inserted for many examples is small. Sage also demonstrates an architecture whereby any run-time failure is fed back into the static portion of the hybrid type checker and becomes a static type error: Every run-time failure can happen at most once.

9.2 Open Avenues

In addition to laying a foundation for future work that has yet to be conceived, this dissertation suggests concrete directions for research related to executable

refinement types.

9.2.1 Parametric Polymorphism

Parametric polymorphism is standard in modern type theories, and combines well with static refinement type systems such as Dependent ML and Liquid Types. But parametricity for dynamic contracts is still advancing.

The addition of parametric polymorphism demands a new development parallel to this dissertation: While types of $\lambda_{\mathcal{H}}$ are easily interpreted as sets of $STLC$ terms, does this approach immediately generalize to System F? How is extensional equivalence affected? What are the consequences for hybrid type checking when the semantics of relationally parametric contracts must be related to the semantics of polymorphic types?

9.2.2 Multiple Representations For Specifications

Executable refinement types are easy for certain purposes, such as translation to an SMT prover, or communicating with other humans. But indexed types enforce many interesting specifications by carefully architecture types for data constructors. Atkey et al. [2011], Dagand and McBride [2012], and Vazou et al. [2013] all investigate how to relate refinements on data constructors to global properties of a data structure. This problem is far from solved, and integral to making executable refinement types more usable. What specifications do we wish to write and which specifications do we hope for an implementation to derive from them? Can this be automated sufficiently that we can always write simply the global property of interest?

More generally, the cited works study equivalences between representations that are best for different purposes: Some representations are more effective for static checking, others more effective for human comprehension. The practice of hybrid type checking introduces a third goal to start to consider: *effective run-time checking*. This means using state-of-the-art data structures, potentially mutable data, and probably much more obscure code. Dynamic contract systems support arbitrary specifications, but connecting this representation to the former two remains incomplete.

9.2.3 Contract Properties

This dissertation has focused on establishing that executable refinement types enjoy the desirable properties expected of a type system. However, contract systems have other correctness properties, such as blame-correctness and complete monitoring. When a counterexample is reported back to an implementation such as Sage, how does correct blame affect *e.g.*the error message to report?

9.2.4 Compositionality

Compositionality of the sort analyzed in Chapter 6 is a property that denotational semantics generally enjoys, and closely parallels homomorphisms between semantics (where a type system is a morphism between the syntactic identity semantics and the semantics that maps each term to its type). Especially considering this in light of abstract interpretation may yield insight into the relationship between type systems and abstract interpretation, further leading to advances in connections between

syntactic and semantic approaches to programming language theory.

9.2.5 Improving hybrid type checking

Our presentation of hybrid type checking is only proven to insert sufficiently many casts to ensure safety. But, in fact, some components of casts are extraneous: In a function subtyping question, if only the domain type needs a cast, the function will nonetheless be fully wrapped. How to define the smallest number of casts, and prove that an HTC system is optimal, is a moderately interesting theoretical question. More realistically, an HTC implementation may simply informally develop techniques for inserting casts that do not perform needless checks.

9.2.6 Side Effects

Side effects in contracts are a huge issue not directly part of this research thread, but are urgent in this context due to the connection of dynamic contracts (where side effects could be used on a "buyer beware" basis) and static semantics (where side effects do not readily translate into simple specifications). JML actually had this issue as well, because you want to use existing libraries. An alternative is to develop *logical* libraries that do not depend on side effects. But these will generally not be efficient to execute, again raising the problem of connecting efficient execution with effective verification.

9.2.7 Error messages, Testing, and Counterexamples

Type error message in modern languages can already be difficult to interpret. When a type incompatibility results from the inability to prove an implication between refinement predicates – which may be very large, especially when they are generated via type reconstruction – the problem is exacerbated. Mathematically, a constructive way to refute a proposition is by providing a counterexample, which would also yield a simple way to communicate an error message. But SMT is nonconstructive by design, so new theorem proving techniques may need to be applied or developed in order to generate readable error messages.

Counterexamples have a larger role to play. In Sage, they strengthen the type checker by refuting more subtyping queries statically. Another good source of counterexamples aside from a counterexample-producing theorem prover is a testing process. Randomized testing tools such as QuickCheck [Claessen and Hughes 2000] and DART [Godefroid et al. 2005] may be effective for enhancing the static capabilities of a hybrid type checking system.

9.2.8 Large-Scale Implementation for Empirical Work

Sage demonstrates that executable refinement types can be incorporated into a realistic programming language, but this has not yet been attempted. How powerful does the counterexample database become when it receives crash reports from all users of a system? How fully are proposed benefits of precise specification realized? For what sorts of programs are executable refinement types most useful?

9.3 Closing Remarks

The theory of executable refinement types is sound and versatile. It combines the strengths of type theory and dynamic contracts, benefiting from advances in each. This exploration of executable refinement types has yielded not only new algorithms to address the standard problems of type systems, but also new definitions of the problems. Through these new definitions have come new approaches to type checking, type reconstruction, compositionality, and system architecture.

However, most of the executable refinement types of this dissertation are based upon the simply-typed λ-calculus; their significance should not be overstated. They lack parametric polymorphism, side-effects, and type classes, to name a few other features we may want in a programming language. They are a foundation upon which to build systems that more closely resemble today's programming practice.

Conversely, programming practice should be influenced to move towards concepts with well-developed foundations. Concepts such as lexical scoping, higher-order functions, structural types, and careful management of side effects all open the door from common practice to the state of the art in programming languages, including executable refinement types.

Just as this dissertation could take for granted the simply-typed λ-calculus as a basis, future work may take for granted that executable refinement types are sound in theory and effective in practice. New theories may build up from this point, and practice may build towards it.

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 31–46, New York, NY, USA, 1990. ACM.

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13 (2):237–268, Apr. 1991.

A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, page 1–13, New York, NY, USA, 2009. ACM.

A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, page 31–41, New York, NY, USA, 1993. ACM.

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, page 163–173, New York, NY, USA, 1994. ACM.

E. Allende and J. Fabry. Application optimization when using gradual typing. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, page 3:1–3:6, New York, NY, USA, 2011. ACM.

R. Atkey, P. Johann, and N. Ghani. When is a type refinement an inductive type? In *Proceedings of the 14th international conference on Foundations of software science and computational structures: part of the joint European conferences on theory and practice of software*, FOSSACS'11/ETAPS'11, page 72–87, Berlin, Heidelberg, 2011. Springer-Verlag.

L. Augustsson. Cayenne - a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, page 239–250, New York, NY, USA, 1998. ACM.

R. J. R. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6): 593–624, Aug. 1988.

H. Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, Apr. 1991.

M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, page 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 521–530, New York, NY, USA, 2011. ACM.

J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In G. Barthe, editor, *Programming Languages and Systems*, number 6602 in Lecture Notes in Computer Science, pages 18–37. Springer Berlin Heidelberg, Jan. 2011.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, page 17–32, Washington, DC, USA, 2008. IEEE Computer Society.

G. M. Bierman, A. D. Gordon, C. Hri\ctcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, page 105–116, New York, NY, USA, 2010. ACM.

M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, July 2006.

V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, Nov. 1999.

L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal of Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

L. Cardelli. Phase distinctions in type theory. Technical report, 1988.

O. Chitil. A semantics for lazy assertions. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, page 141–150, New York, NY, USA, 2011. ACM.

O. Chitil. Practical typed lazy contracts. *SIGPLAN Not.*, 47(9):67–76, Sept. 2012.

O. Chitil and F. Huch. Monadic, prompt lazy assertions in haskell. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS'07, page 38–53, Berlin, Heidelberg, 2007a. Springer-Verlag.

O. Chitil and F. Huch. A pattern logic for prompt lazy assertions in haskell. In *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages*, IFL'06, page 126–144, Berlin, Heidelberg, 2007b. Springer-Verlag.

O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, page 1–19, Berlin, Heidelberg, 2004. Springer-Verlag.

R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of*

the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '12, page 587–606, New York, NY, USA, 2012a. ACM.

R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. *SIGPLAN Not.*, 47(1):231–244, Jan. 2012b.

A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56, June 1940.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. ACM.

T. Coquand. An analysis of girard's paradox. In *In Symposium on Logic in Computer Science*, page 227–236. IEEE Computer Society Press, 1986.

P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, page 170–181, New York, NY, USA, 1995. ACM.

S. Cui, K. Donnelly, and H. Xi. ATS: a language that combines programming with theorem proving. In *Proceedings of the 5th international conference on Frontiers of Combining Systems*, FroCoS'05, page 310–320, Berlin, Heidelberg, 2005. Springer-Verlag.

H. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, Nov. 1934. PMID: 16577644 PMCID: PMC1076489.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 103–114, New York, NY, USA, 2012. ACM.

R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 198–208, New York, NY, USA, 2000. ACM.

M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages. In *Arbeitstagung Programmiersprachen (ATPS'09)*, 2009.

M. Degen, P. Thiemann, and S. Wehr. The interaction of contracts and laziness. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, page 97–106, New York, NY, USA, 2012. ACM.

E. Denney. Refinement types for specification. In *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, PROCOMET '98, page 148–166, London, UK, UK, 1998. Chapman & Hall, Ltd.

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1976.

C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5):16:1–16:29, Nov. 2011.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 215–226, New York, NY, USA, 2011. ACM.

C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, page 214–233, Berlin, Heidelberg, 2012. Springer-Verlag.

D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 236–249, New York, NY, USA, 2003. ACM.

J. Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*, PLPV '07, page 21–32, New York, NY, USA, 2007. ACM.

J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Proceedings of the 6th International conference on Foundations of Software Science and Computation Structures and joint European conference on*

Theory and practice of software, FOSSACS'03/ETAPS'03, page 250–266, Berlin, Heidelberg, 2003. Springer-Verlag.

J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 281–292, New York, NY, USA, 2004. ACM.

M. Escardó. Synthetic topology of data types and classical spaces. In *Electronic Notes in Theoretical Computer Science*, page 2004. Elsevier, 2004.

M. Fagan. *Soft typing: an approach to type checking for dynamically typed languages.* PhD thesis, Rice University, Houston, TX, USA, 1991. UMI Order No. GAX91-36021.

M. Fahndrich and A. Aiken. Making set-constraint program analyses scale. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1996.

R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the 8th international conference on Functional and Logic Programming*, FLOPS'06, page 226–241, Berlin, Heidelberg, 2006. Springer-Verlag.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, page 48–59, New York, NY, USA, 2002. ACM.

C. Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 245–256, New York, NY, USA, 2006. ACM.

C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, page 235–248, New York, NY, USA, 1997. ACM.

C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 23–32, New York, NY, USA, 1996. ACM.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, page 234–245, New York, NY, USA, 2002. ACM.

R. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium in Applied Mathematics: Mathematical Aspects of Computer Science*, pages 19–32, 1967.

S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, page 112–121, New York, NY, USA, 2007. ACM.

T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, page 268–277, New York, NY, USA, 1991. ACM.

A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4): 19:1–19:64, Sept. 2008.

Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, Jan. 1988.

J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. ACM.

M. Greenberg, B. c. Pierce, and S. Weirich. Contracts made manifest. *Journal of Functional Programming*, 22(3):225–274, May 2012.

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *In Eighth Symposium on Trends in Functional Programming*, 2007.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Proceedings of the 7th workshop on Scheme and functional programming*, Portland, OR, 2006. ACM.

D. Grossman. Existential types for imperative languages. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, page 21–35, London, UK, UK, 2002. Springer-Verlag.

A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, page 29–40, New York, NY, USA, 2007. ACM.

R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, page 123–137, New York, NY, USA, 1994. ACM.

S. Hayashi. Logic of refinement types. In *Proceedings of the international workshop on Types for proofs and programs*, TYPES '93, page 108–126, Secaucus, NJ, USA, 1993. Springer-Verlag New York, Inc.

N. C. Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22866.

F. Henglein. Dynamic typing: syntax and proof theory. In *Selected papers of the symposium on Fourth European symposium on programming*, ESOP'92, page 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbolic Compututation*, 23(2):167–189, June 2010.

R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *In*

FLOPS '06: Functional and Logic Programming: 8th International Symposium, page 208–225. Springer-Verlag, 2006.

M. Hoang and J. C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, page 176–185, New York, NY, USA, 1995. ACM.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

R. C. Holt and J. R. Cordy. The turing programming language. *Communications of the ACM*, 31(12):1410–1423, Dec. 1988.

W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479—490, 1980.

A. J. C. Hurkens. A simplification of girard's paradox. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, TLCA '95, page 266–278, London, UK, UK, 1995. Springer-Verlag.

L. Ina and A. Igarashi. Towards gradual typing for generics. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, page 17–29, New York, NY, USA, 2009. ACM.

L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, page 609–624, New York, NY, USA, 2011. ACM.

R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, page 470–485, Berlin, Heidelberg, 2011. Springer-Verlag.

M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, page 304–315, New York, NY, USA, 2009. ACM.

K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European conference on Programming*, ESOP'07, page 505–519, Berlin, Heidelberg, 2007. Springer-Verlag.

K. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, page 27–38, New York, NY, USA, 2009. ACM.

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):6:1–6:34, Feb. 2010.

M. Kölling and J. Rosenberg. Blue - a language for teaching object-oriented programming. *SIGCSE Bulletins*, 28(1):190–194, Mar. 1996.

J. Lambek and P. J. Scott. *Introduction to higher order categorical logic.* Cambridge University Press, New York, NY, USA, 1986.

G. T. Leavens and Y. Cheon. *Design by Contract with JML.* 2006.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.01. Technical report, Sept. 2013.

P. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, page 293–304, New York, NY, USA, 1992. ACM.

D. C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying ADA Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, page 213–225, New York, NY, USA, 2003. ACM.

S. Marlow. Haskell 2010 language report, 2010.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editor, *Studies in Logic and the Foundations of Mathematics*, volume Volume 80 of *Logic Colloquium '73 Proceedings of the Logic Colloquium*, pages 73–118. Elsevier, 1975.

J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, page 16–31, Berlin, Heidelberg, 2008. Springer-Verlag.

C. McBride. Faking it (simulating dependent types in haskell). *Journal of Functional Programming*, 12(5):375–392, July 2002.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, Jan. 2004.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

J. C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, page 175–185, New York, NY, USA, 1984. ACM.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

J. H. Morris. Protection in programming languages. *Commun. ACM*, 16(1):15–21, Jan. 1973a.

J. H. Morris. Types are not sets. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, page 120–124, New York, NY, USA, 1973b. ACM.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *Proceedings of the 16th European conference on Programming*, ESOP'07, page 189–204, Berlin, Heidelberg, 2007. Springer-Verlag.

A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, page 165–179, Washington, DC, USA, 2011. IEEE Computer Society.

A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2):6:1–6:41, July 2013.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, page 62–73, New York, NY, USA, 2006. ACM.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, page 229–240, New York, NY, USA, 2008. ACM.

G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005.

U. Norell. Dependently typed programming in agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, page 1–2, New York, NY, USA, 2009. ACM.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, Jan. 1999.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, number 155 in IFIP International Federation for Information Processing, pages 437–450. Springer US, Jan. 2004.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972a.

D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972b.

B. Pierce and E. Sumii. Relating cryptography and polymorphism. 2000.

B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings of the 25th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98, page 252–265, New York, NY, USA, 1998. ACM.

J. G. Politz, H. Quay-de la Vallee, and S. Krishnamurthi. Progressive types. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, page 55–66, New York, NY, USA, 2012. ACM.

F. Pottier. Simplifying subtyping constraints. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, page 122–133, New York, NY, USA, 1996. ACM.

A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, page 481–494, New York, NY, USA, 2012. ACM.

B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The ruby type checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1565–1572, New York, NY, USA, 2013. ACM.

P. Rondon, A. Bakst, M. Kawaguchi, and R. Jhala. CSolve: verifying c with liquid types. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, page 744–750, Berlin, Heidelberg, 2012. Springer-Verlag.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Proceedings of*

the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, page 131–144, New York, NY, USA, 2010. ACM.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. ACM.

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, Sept. 1998.

K. Sagonas and D. Luna. Gradual typing of erlang programs: a wrangler experience. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, page 73–82, New York, NY, USA, 2008. ACM.

D. Sangiorgi, N. Kobayashi, and E. Sumii. Logical bisimulations and functional languages. In *Proceedings of the 2007 international conference on Fundamentals of software engineering*, FSEN'07, page 364–379, Berlin, Heidelberg, 2007. Springer-Verlag.

D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. 1969.

T. Sheard. Putting curry-howard to work. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, page 74–85, New York, NY, USA, 2005. ACM.

T. Sheard. Type-level computation using narrowing in Ωmega. *Electron. Notes Theor. Comput. Sci.*, 174(7):105–128, June 2007.

J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the 7th workshop on Scheme and functional programmin*, 2006.

J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, page 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, page 7:1–7:12, New York, NY, USA, 2008. ACM.

J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, page 34–46, New York, NY, USA, 2009. ACM.

M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten, R. Kelsey, W. Clinger, J. Rees, R. B. Findler, and J. Matthews. The revised report on the algorithmic language scheme. Technical report, Sept. 2007.

R. Statman. Logical relations and the typed lambda-calculus. pages 85–97, 1985.

C. A. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, page 214–227, New York, NY, USA, 2000. ACM.

D. Stoutamire and S. Omohundro. The sather 1.1 specification. 1996.

E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *SIGPLAN Notices*, 39 (1):161–172, Jan. 2004.

N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, page 329–340, New York, NY, USA, 2009. ACM.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, page 181–192, New York, NY, USA, 1996. ACM.

T. Terauchi. Dependent types from counterexamples. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 119–130, New York, NY, USA, 2010. ACM.

S. Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, page 367–381, New York, NY, USA, 1990. ACM.

The Coq development team. The coq proof assistant reference manual. Technical report, 2012.

N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, page 209–228, Berlin, Heidelberg, 2013. Springer-Verlag.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as*

Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, ESOP '09, page 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, page 459–483, Berlin, Heidelberg, 2011. Springer-Verlag.

A. K. Wright and R. Cartwright. A practical soft type system for scheme. *SIGPLAN Lisp Pointers*, VII(3):250–262, July 1994.

H. Xi. Imperative programming with dependent types. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, LICS '00, page 375–, Washington, DC, USA, 2000. IEEE Computer Society.

H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, page 214–227, New York, NY, USA, 1999. ACM.

D. N. Xu. Extended static checking for haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, page 48–59, New York, NY, USA, 2006. ACM.

D. N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of*

the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, PEPM '12, page 107–116, New York, NY, USA, 2012. ACM.

D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 41–52, New York, NY, USA, 2009. ACM.

C. Zenger. Indexed types. *Theoretical Compututer Science*, 187(1-2):147–165, Nov. 1997.