UNIVERSITY OF CALIFORNIA
SANTA CRUZ

HYBRID TYPE CHECKING AND TYPE RECONSTRUCTION FOR EXECUTABLE REFINEMENT TYPES

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

MASTERS OF SCIENCE

in

COMPUTER SCIENCE

by

Kenneth L. Knowles

March 2008

The Thesis of Kenneth L. Knowles
is approved:

Professor Cormac Flanagan, Chair

Professor Martín Abadi

Professor Stephen Freund

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Table of Contents

Abstract

Hybrid Type Checking and Type Reconstruction

for Executable Refinement Types

by

Kenneth L. Knowles

Traditional static type systems are very effective for verifying certain basic interface specifications. Dynamically checked contracts support more precise specifications, but these are not checked until run time, resulting in incomplete detection of defects. This thesis explores a system of *executable refinement types* that can express the same specifications as contracts, such as function pre- and postconditions and data structure invariants. Type checking for executable refinement types involves reasoning about implications between arbitrary executable predicates, hence is undecidable. We first introduce *hybrid type checking*, a synthesis of static and dynamic checking that addresses this undecidability by enforcing executable refinement types via static analysis where possible, but also via dynamic checks where necessary.

Owing in part to their greater expressiveness, executable refinements are more verbose than simple types, so type reconstruction is particularly valuable. Yet typeability is also undecidable because it subsumes type checking. Using a generalized notion of type reconstruction, we present the first type reconstruction algorithm for executable refinement types. Our algorithm is a typeability-*preserving* transformation and defers type checking to a subsequent phase, where a strategy such as hybrid type checking

may be employed. The algorithm generates and solves a collection of implication constraints over refinement predicates, inferring maximally precise refinement predicates in a largely syntactic manner that is reminiscent of strongest postcondition calculation. Perhaps surprisingly, our notion of type reconstruction is decidable even though type checking is not.

Acknowledgments

Chapters 1 and 2 of this work are an expository reorganization, with some corrections, of work by Flanagan [2006] which provides crucial background for Chapter 3. Chapter 3 is itself an expanded revision of published work by the present author [Knowles and Flanagan 2007], with notations and definitions modified to be consistent with Chapters 1 and 2.

Chapter 1

Executable Refinement Types

The construction of reliable software is extremely difficult. For large systems, it requires a modular development strategy that, ideally, is based on precise and trusted interface specifications. In practice, however, programmers typically work in the context of a large collection of APIs whose behavior is only informally and imprecisely specified and understood. Practical mechanisms for specifying and verifying precise, behavioral aspects of interfaces are clearly needed.

Static type systems have proven to be effective and practical tools for specifying and verifying basic structural interface specifications, and are widely adopted. However, there are important specifications that cannot generally be expressed with only structural types, such as preconditions, postconditions, adherence to protocols, or potentially arbitrary formal properties of programs.

Many preconditions and postconditions can be expressed as executable *contracts* [Meyer 1988; Findler and Felleisen 2002; Leavens and Cheon 2005; Gomes et al.

1996; Holt and Cordy 1988; Luckham 1990; Parnas 1972; Kölling and Rosenberg 1997]. Dynamic checking can easily support some precise specifications[1], such as:

- Subranges: The function `printDigit` requires an integer in the range [0,9].
- 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 predicate `wellTyped` : `Expr` $\rightarrow$ `Bool`.

In this thesis, we describe a type system with the expressivity of dynamic contracts and the technical apparatus which makes the type system practical. Such precise types, however, make type checking undecidable. Rather than compromise on the design of the type system to maintain decidability, we present practical algorithms for working within an undecidable type sytem.

[1] Notably absent are common properties requiring (potentially) infinite quantification, such as associativity or injectivity

Figure 1.1: Syntax

$v ::=$		*Values:*
	c	constant
	$\lambda x\!:\!S.\, t$	abstraction
$s, t ::=$		*Terms:*
	v	value
	x	variable
	$t\ t$	application
$S, T ::=$		*Types:*
	$x\!:\!S \to T$	dependent function type
	$\{x\!:\!B \mid t\}$	refined basic type
$E ::=$		*Environments:*
	$\emptyset$	empty environment
	$E, x : T$	environment extension

1.1 The Language λ^H

This section introduces a variant of the simply typed λ-calculus extended with executable refinement types. We refer to this language as λ^H.

The syntax of λ^H is summarized in Figure 1.1. Terms include variables, constants, functions, and applications. The λ^H type language includes dependent function types [Cardelli 1988a], for which we use the syntax $x\!:\!S \to T$ of Cayenne [Augustsson 1998] (in preference to the equivalent syntax $\Pi x\!:\!S.\ T$). Here, S is the domain type of the function and the formal parameter x may occur in the range type T. We omit x if it does not occur free in T, yielding the standard function type syntax $S \to T$.

We use B to range over base types, which includes at least `Bool` and `Int`. As

in many languages, these base types are fairly coarse and cannot, for example, denote integer subranges. To overcome this limitation, we introduce *executable refinement types* of the form $\{x{:}B \mid t\}$. Here, the variable x (of base type B) can occur within the boolean term or *predicate* t. Informally, this refinement type denotes the set of constants c of type B that satisfy this predicate, *i.e.*, for which the term $[x \mapsto c]\, t$ evaluates to `true`. Thus, $\{x{:}B \mid t\}$ denotes a subtype of B, and we use a base type B as an abbreviation for the trivial refinement type $\{x{:}B \mid \texttt{true}\}$.

Our refinement types are inspired by prior work on decidable refinement type systems [Mandelbaum et al. 2003; Freeman and Pfenning 1991; Davies and Pfenning 2000; Xi and Pfenning 1999; Xi 2000; Ou et al. 2004]. However, our refinement predicates are arbitrary boolean expressions, so every computable subset of the integers is actually a λ^H type. Not surprisingly, this expressive power causes type checking to become undecidable. More specifically, subtyping between two refinement types $\{x{:}B \mid t_1\}$ and $\{x{:}B \mid t_2\}$ reduces to checking implication between the corresponding predicates, which is clearly undecidable. These decidability difficulties are circumvented by our hybrid type checking algorithm, which we describe in Chapter 2.

The type of each constant is defined by the function $ty : \mathit{Constant} \to \mathit{Type}$ in Figure 1.2. The set *Constant* is implicitly defined as the domain of the mapping.

A *basic constant* is a constant whose type is a refinement type (not a function type). Each basic constant is assigned a singleton type that denotes exactly that constant. For example, the type of an integer n denotes the singleton set $\{n\}$.

A *primitive function* is a constant of function type. For clarity, we use infix

Figure 1.2: Some Types of Constants

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

syntax for applications of some primitive functions (*e.g.*, $+$, $=$, $\Leftrightarrow$). The types for primitive functions are quite precise: The type

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

exactly specifies that this function performs addition. That is, the term $n + m$ has the type $\{z{:}\texttt{Int} \,|\, z = n + m\}$ 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.

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:

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

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

Figure 1.3: Evaluation Rules

Redex Evaluation $\boxed{s \longrightarrow t}$

$$(\lambda x{:}S.\, t)\ s \longrightarrow [x \mapsto s]\, t \qquad [\text{E-}\beta]$$
$$c\ v \longrightarrow \delta(c, v) \qquad [\text{E-PRIM}]$$

Contextual Evaluation $\boxed{s \rightsquigarrow t}$

$$\mathcal{C}[s] \rightsquigarrow \mathcal{C}[t] \quad \text{if } s \longrightarrow t \qquad [\text{E-COMPAT}]$$

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

$$\begin{array}{lll} \mathcal{C} & ::= & \bullet \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \lambda x{:}S.\mathcal{C} \mid \lambda x{:}\mathcal{D}.\, t \\ \mathcal{D} & ::= & x{:}\mathcal{D} \to T \mid x{:}S \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\} \end{array}$$

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

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

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

1.2 Operational Semantics of λ^H

We next describe the run-time behavior of λ^H terms. The relation $s \longrightarrow t$ expresses conversion of redexes; we close this over arbitrary contexts to generate the single-step evaluation relation $s \rightsquigarrow t$. We write $\rightsquigarrow^*$ for the reflexive-transitive closure of $\rightsquigarrow$. As shown in Figure 1.3, the rule [E-β] performs standard β-reduction of function applications. The rule [E-PRIM] evaluates applications of primitive functions according

to their external definitions, given by the partial function

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

For example:

$$\begin{aligned}
\delta(\texttt{not}, \texttt{true}) &= \texttt{false} \\
\delta(+, 3) &= +_3 \\
\delta(+_3, 4) &= 7 \\
\delta(\texttt{not}, 3) &= \textit{undefined} \\
\delta(\texttt{if}_T, \texttt{true}) &= \lambda x{:}T.\, \lambda y{:}T.\, x \\
\delta(\texttt{if}_T, \texttt{false}) &= \lambda x{:}T.\, \lambda y{:}T.\, y \\
\delta(\texttt{fix}_T, t) &= t \;\; (\texttt{fix}_T \; t)
\end{aligned}$$

1.3 The λ^H Type System

We next describe the (undecidable) λ^H type system via the collection of type judgments and rules shown in Figure 1.4. The judgment $E \vdash t : T$ checks that the term t has type T in environment E; the judgment $E \vdash T$ checks that T is a well formed type in environment E; and the judgment $E \vdash S <: T$ checks that S is a subtype of T in environment E. The judgement $E \vdash t_1 \Rightarrow t_2$ is considered external to the system; its axiomatization is discussed in Section 1.4. The rules for typing terms and judging well formedness of environments and types are entirely standard. As usual, 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.

Figure 1.4: Type Rules

Type rules $\boxed{E \vdash t : T}$

$$\frac{(x : T) \in E}{E \vdash x \,:\, T} \text{ [T-Var]} \qquad \frac{}{E \vdash c \,:\, ty(c)} \text{ [T-Const]}$$

$$\frac{E \vdash S \qquad E, x : S \vdash t \,:\, T}{E \vdash (\lambda x{:}S.\, t) \,:\, (x{:}S \to T)} \text{ [T-Fun]}$$

$$\frac{E \vdash t_1 \,:\, (x{:}S \to T) \qquad E \vdash t_2 \,:\, S}{E \vdash t_1 \; t_2 \,:\, [x \mapsto t_2]\, T} \text{ [T-App]}$$

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

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

$$\frac{E \vdash S \qquad E, x : S \vdash T}{E \vdash x{:}S \to T} \text{ [WT-Arrow]} \qquad \frac{E, x : B \vdash t \,:\, \texttt{Bool}}{E \vdash \{x{:}B \mid t\}} \text{ [WT-Base]}$$

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

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

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

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

$$\frac{E, x : B \vdash s \Rightarrow t}{E \vdash \{x{:}B \mid s\} <: \{x{:}B \mid t\}} \text{ [S-Base]}$$

The novel aspects of this system arise from its support of refinement types in subtyping. A type $\{x{:}B \mid t\}$ roughly denotes the set of constants c of type B for which $[x \mapsto c]\, t$ is valid. Subtyping between refinement types reduces to implication between their refinement predicates (the formal details of the implication judgment $E \vdash t_1 \Rightarrow t_2$ is discussed in the next section). As an example, the rule [S-Base] states that the subtyping judgment

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

follows from the validity of the implication:

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

Of course, checking implication between arbitrary predicates is undecidable, which motivates the development of the hybrid type checking algorithm in the following chapter.

1.4 Type Soundness

We prove soundness via the usual "progress" and "preservation" lemmas. Throughout this section, environments are assumed to be well formed, to elide many uninteresting antecedents.

As usual, a term is in *normal form* if it does not reduce to any subsequent term, and a *value* v is either a λ-abstraction or a constant. We assume that the function *ty* maps each constant to an appropriate type, in the following sense:

Assumption 1 (Types of Constants) *For each $c \in Constant$:*

1. *The type of c is well formed,* i.e. $\emptyset \vdash ty(c)$.
2. *If c is a primitive function then it cannot "get stuck" and its operational behavior is compatible with its type,* i.e. *if $\emptyset \vdash c\ v : T$ then $[\![c]\!](v)$ is defined and $\emptyset \vdash [\![c]\!](v) : T$*
3. *If c is a basic constant then it is the unique constant of its type,* i.e. *if $ty(c) = \{x{:}B \mid t\}$ then $[x \mapsto c]\, t \rightsquigarrow^* \texttt{true}$ and $\forall c' \neq c.\ [x \mapsto c']\, t \not\rightsquigarrow^* \texttt{true}$.*

To remain flexible with regard to the logic used for implication, we leave the judgement abstract, and assume only those properties necessary for proving soundness of the type system.

Assumption 2 (Axioms of implication)

1. *(Weakening) If $E, G \vdash p \Rightarrow q$ then $E, x : S, G \vdash p \Rightarrow q$.*
2. *(Transitivity) If $E \vdash p \Rightarrow q$ and $E \vdash q \Rightarrow r$ then $E \vdash p \Rightarrow r$.*
3. *(Reflexivity) $E \vdash p \Rightarrow p$.*
4. *(Faithfulness) $E \vdash p \Rightarrow \texttt{true}$.*
5. *(Consistency) $\emptyset \nvdash \texttt{true} \Rightarrow \texttt{false}$.*
6. *(Evaluation) If $p \leftrightsquigarrow^* q$ then $E \vdash p \Rightarrow q$.*
7. *(Substitution) If $E, x : S, F \vdash p \Rightarrow q$ and $E \vdash t : S$ then $E, \theta F \vdash \theta p \Rightarrow \theta q$ where $\theta = [x \mapsto t]$.*

8. *(Hypothesis) If* $ty(c) = \{x{:}B \mid p\}$*, a singleton type by Assumption 1, and* $E \vdash \texttt{true} \Rightarrow [x \mapsto c]\, p$ *then* $E, x : B \vdash x = c \Rightarrow p$.

9. *(Narrowing) If* $E, x : T, F \vdash p \Rightarrow q$ *and* $E \vdash S <: T$ *then* $E, x : S, F \vdash p \Rightarrow q$.

The type system satisfies the standard type preservation (subject reduction) property, supported by the usual lemmas along with the fact that types remain equivalent when terms within them are reduced.

Lemma 3 (Weakening)

Let $E = E_1, E_2$ *and* $E' = E_1, x : U, E_2$.

1. *If* $E \vdash T$ *and* $E \vdash S$ *and* $E \vdash S <: T$ *then* $E' \vdash S <: T$.

2. *If* $E \vdash t : T$ *then* $E' \vdash t : T$.

3. *If* $E \vdash T$ *then* $E' \vdash T$.

PROOF:

(1) By straightforward induction on the derviation of $E \vdash S <: T$, using Weakening of implication in the case for rule [S-BASE]

(2) and (3) By straightforward mutual induction on the derivations of $E \vdash t : T$ and $E \vdash T$, using part (1) in the case of rule [T-SUB] □

Lemma 4 (Substitution) *Suppose*

$$E_1 \vdash s : S \qquad \theta = [x \mapsto s] \qquad E = E_1, x : S, E_2 \qquad E' = E_1, \theta E_2$$

Then

1. *If* $E \vdash T_1 <: T_2$ *then* $E' \vdash \theta T_1 <: \theta T_2$.

2. *If* $E \vdash T$ *then* $E' \vdash \theta T$.

3. *If* $E \vdash t : T$ *then* $E' \vdash \theta t : \theta T$.

PROOF:

(1) By straightforward induction on the derivation $E \vdash T_1 <: T_2$, using the Substitution axiom of implication in the case of [S-BASE].

(2) and (3) By straightforward mutual induction on the derivations $E \vdash t : T$ and $E \vdash T$. The case for [T-SUB] uses part (1), and in the case for [T-VAR] if $t = x$ then $S = T$ and $\theta t = s$ and we use weakening to extend $E_1 \vdash s : S$ to $E' \vdash s : S$. □

Lemma 5 (Narrowing of Subtyping)

If $E \vdash S <: T$ *and* $E, x : T, F \vdash Q <: R$ *then* $E, x : S, F \vdash Q <: R$.

PROOF: By straightforward induction over the derivation of $E, x : T, F \vdash Q <: R$ using the Narrowing axiom of implication. □

Lemma 6 (Subtyping is a Preorder)

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

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

PROOF:

1. By induction on the derivation of $E \vdash T$, using the Reflexivity axiom of implication.

2. By induction on the derivation of $E \vdash S <: T$ (followed by inversion on $E \vdash T <: U$) using Lemma 5 (Narrowing of Subtyping) for the bindings in function types, and the Transitivity axiom of implication. □

Lemma 7 (Type Equivalence under Evaluation) *If $E \vdash S$ and $S \rightsquigarrow T$ then $E \vdash S <: T$ and $E \vdash T <: S$.*

PROOF: By induction on the derivation of S. In the base case of [WT-BASE] we invoke the Evaluation axiom of theorem proving. □

Theorem 8 (Preservation) *If $\vdash E$ and $E \vdash s : T$ and $s \rightsquigarrow t$ then $E \vdash t : T$*

PROOF: By induction on the typing derivation $E \vdash s : T$, with case analysis on the final rule applied.

Case [T-VAR], [T-CONST]: No evaluation rule applies.

Case [T-FUN], [T-SUB]: Immediate from the inductive hypothesis and Lemma 7.

Case [T-APP]: Consider the possible evaluation rules:

Case [E-PRIM]: Preservation holds by assumption 1 on constants.

Case [E-β]: Apply Lemma 4 (Substitution)

Case [E-COMPAT]: Apply the inductive hypothesis and in Lemma 7 (Type equivalence under evaluation).

□

The type system also satisfies the progress property, completing the syntactic proof of type soundness. We require the usual lemma about canonical forms for types.

Lemma 9 (Canonical Forms) *If* $\emptyset \vdash v : (x{:}T_1 \to T_2)$ *then either*

1. $v = \lambda x{:}S.\, s$ *and* $\emptyset \vdash T_1 <: S$ *and* $x : S \vdash s : T_2$, *or*

2. v *is a constant and* $ty(c)$ *is a subtype of* $x{:}T_1 \to T_2$.

PROOF: By induction on $\emptyset \vdash v : x{:}T_1 \to T_2$.

Case [T-VAR]: Does not apply to values

Case [T-CONST]: Then we are in case 2.

Case [T-FUN]: Then $v = \lambda x{:}T_1.\, s$ and $x : S \vdash s : T_2$ so by reflexivity of subtyping we meet case 1.

Case [T-SUB]: The lemma follows by induction and transitivity of subtyping. □

Theorem 10 (Progress) *If* $\emptyset \vdash s : T$ *then either* s *is a value, or there exists some* s' *such that* $s \rightsquigarrow s'$.

PROOF: By induction on the derivation of $\vdash s : T$, considering the final rule applied.

Case [T-VAR], [T-CONST], [T-FUN]: Then t is already a value or not closed.

Case [T-SUB]: Immediate from the inductive hypothesis.

Case [T-APP]: Then $t = t_1\ t_2$. If t_1 or t_2 are not values, then by induction they can evaluate hence t can as well. If both are values, then there are two possible forms for t_1 (given by Canonical forms):

1. t_1 is a constant of appropriate type – then its meaning function is defined by assumption.

2. t_1 is a lambda term, so β-reduction applies. $\square$

1.5 Related Work

Research on advanced type systems has influenced our choice of how to express program invariants. In particular, Freeman and Pfenning [1991] extended ML with another form of refinement types. They do not support executable refinement predicates, since their system provides both decidable type checking and type inference. Xi and Pfenning have explored the practical application of dependent types in an extension of ML called Dependent ML [Xi and Pfenning 1999; Xi 2000]. Decidability of type checking is preserved by appropriately restricting which terms can appear in types. Despite these restrictions, a number of interesting examples can be expressed in Dependent ML.

Ou, Tan, Mandelbaum, and Walker developed a type system similar to ours, adding an interface between untyped code and code with refinement and dependent types [Ou et al. 2004]. Though their system also includes mutable references, their refinement predicates must be side-effect free, and may not call recursive functions.

Chapter 2

Hybrid Type Checking

Though ongoing research on more powerful type systems, such as those discussed in Section 1.5, attempts to overcome some of the restrictions discussed in Chapter 1, yet these systems are designed to be *statically type safe*, so the specification language is intentionally restricted to ensure that specifications can always be checked statically. An opposing approach is to abandon static checking entirely in favor of dynamic checking. But this has significant disadvantages: First, it consumes cycles that could otherwise perform useful computation. More seriously, dynamic checking provides limited coverage – specifications are only checked on data values and code paths of actual executions. Thus, dynamic checking often results in incomplete and late (possibly post-deployment) detection of defects.

Thus, the twin goals of *complete checking* and *expressive specifications* appear to be incompatible in practice.[1] Static type checking focuses on complete checking of

[1] Complete checking of expressive specifications could be achieved by requiring that each program be accompanied by a proof (perhaps expressed as type annotations) that the program satisfies its

restricted specifications. Dynamic checking focuses on incomplete checking of expressive specifications. Neither approach in isolation provides an entirely satisfactory solution for enforcing precise interface specifications.

In this chapter, we describe an approach for validating precise interface specifications using a synthesis of static and dynamic techniques. By checking correctness properties and detecting defects statically (whenever possible) and dynamically (only when necessary), this approach of *hybrid type checking* provides a potential solution to the limitations of purely static and purely dynamic approaches.

We illustrate the key idea of hybrid type checking by considering the type rule for function application:

$$\frac{E \vdash t_1 \,:\, T \to T' \qquad E \vdash t_2 \,:\, S \qquad E \vdash S <: T}{E \vdash (t_1\ t_2) \,:\, T'}$$

The antecedent $E \vdash S <: T$ checks compatibility of the actual and formal parameter types. If the type checker can prove this subtyping relation, then this application is well typed. 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, once we consider expressive type languages that are not statically decidable, the type checker may encounter situations where its algorithms can neither prove nor refute the subtype judgment $E \vdash S <: T$ (particularly within the time bounds

specification, but manually or interactively writing such proofs appears too heavyweight for widespread use.

imposed by interactive compilation). A fundamental question in the development of expressive type systems is how to deal with such situations where the compiler cannot statically classify the program as either ill typed or well typed:

- *Statically rejecting* such programs would cause the compiler to reject some programs that, on deeper analysis, could be shown to be well typed. This approach seems too brittle for use in practice since it would be 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, which is undesirable.

Hence, we argue that the most satisfactory approach is for the compiler to accept such programs on a provisional basis, but to insert sufficient dynamic checks to ensure that specification violations never occur at run time. Of course, checking that $E \vdash S <: T$ at run time is still a difficult problem and would violate the principle of *phase distinction* [Cardelli 1988b]. Instead, our hybrid type checking approach transforms the above application into the code

$$t_1\ (\langle T \lhd S \rangle\ t_2)$$

where the additional *typecast* or *coercion* $\langle T \lhd S \rangle\ t_2$ dynamically checks that the value produced by t_2 is in the domain type T. Note that hybrid type checking supports precise types, and T could in fact specify a detailed precondition of the function, for

<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>

Figure 2.1: Hybrid type checking on various programs.

example, that it only accepts prime numbers. In this case, the run-time cast would involve performing a primality check.

The behavior of hybrid type checking on various kinds of programs is illustrated in Figure 2.1. Although every program can be classified as either ill typed or well typed, for expressive type systems it is not always possible to make this classification statically. However, the compiler can 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 *subtle* programs, dynamic type casts are inserted to check any unverified correctness properties at run time. 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 subtle manner that cannot easily be detected at compile time, the inserted casts may fail. As static analysis technology improves, we expect that the category of subtle programs in Figure 2.1 will shrink, as more ill typed programs are rejected and more well typed programs are fully verified at compile time.

Hybrid type checking provides several desirable characteristics:

1. It supports precise interface specifications, which are essential for modular development of reliable software.

2. As many defects as is 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.

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 never occur; they are always detected, either statically or dynamically.

Our proposed specifications extend traditional static types, and so we view hybrid type checking as an extension of traditional static type checking. In particular, hybrid type checking supports precise specifications while preserving a key benefit of static type systems; namely, the ability to detect simple errors at compile time. Moreover, as we shall see, for any decidable static type checker S, it is possible to develop a hybrid type checker H that performs somewhat better than S in the following sense:

1. H dynamically detects errors that would be missed by S, since H supports more precise specifications than S and can detect violations of these specifications dynamically.

2. H statically detects all errors that would be detected by S, provided the specifications have the same interpretation, since H can statically perform the same reasoning as S.

3. H actually detects errors *statically* that S does not, since H supports more precise specifications, and could reasonably detect some violations of these precise specifications statically.

The last property is perhaps the most surprising; Section 2.5 contains a proof that clarifies this argument. Note that if S interprets specifications differently, there may be faulty programs which are accepted by H but rejected by S, but the justification for rejecting the program would not be valid in H.

Hybrid type checking may facilitiate the evolution and adoption of advanced static analyses, by allowing software engineers to experiment with sophisticated specification strategies that cannot (yet) be verified statically. Such experiments can then motivate and direct static analysis research. In particular, if a hybrid type checker fails to decide (*i.e.*, verify or refute) a subtyping query, it could send that query back to the compiler writer. Similarly, if a hybrid-typed program fails an inserted cast $\langle T \rangle\ v$, the value v is a witness that refutes an undecided subtyping query, 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.

Indeed, just as different compilers for the same language may yield object code of different quality, we might imagine a variety of hybrid type checkers with different

trade-offs between static and dynamic checks (and between static and dynamic error messages). 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 1990; 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` [Abadi et al. 1989; S. Thatte 1990], 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 to be applicable to a variety of programming languages and to various specification languages. We illustrate the key ideas of hybrid type checking for a fairly expressive dependent type system that is statically undecidable. Specifically, we work with an extension of λ^H developed in the previous chapter.

2.1 Additional Syntax, Semantics, and Typing

The additional language constructs and their semantics (static and dynamic) are shown in Figure 2.2. The cast $\langle T \triangleleft S \rangle\ t$ dynamically checks that the result of t is

Figure 2.2: Additional Syntax, Semantics, and Typing

$$
\begin{array}{lr}
v ::= \cdots & \textit{Values:} \\
\quad \langle T \triangleleft S \rangle & \text{type cast} \\
\\
s, t ::= \cdots & \textit{Terms:} \\
\quad \langle T, t, c \rangle & \text{check in progress}
\end{array}
$$

Additional Redex Evaluation $\boxed{s \longrightarrow t}$

$$\langle x{:}T_1 \to T_2 \triangleleft x{:}S_1 \to S_2 \rangle\ v \longrightarrow \lambda x{:}T_1.\, (\langle T_2 \triangleleft S_2 \rangle \circ v \circ \langle S_1 \triangleleft T_1 \rangle)\ x \qquad \text{[E-Cast-F]}$$

$$\langle \{x{:}B \mid t\} \triangleleft \{x{:}B \mid s\} \rangle\ c \longrightarrow \langle \{x{:}B \mid t\}, [x \mapsto c]\, t, c \rangle \qquad \text{[E-Cast-Begin]}$$

$$\langle \{x{:}B \mid s\}, \mathtt{true}, c \rangle \longrightarrow c \qquad \text{[E-Cast-End]}$$

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

$$\mathcal{C} ::= \cdots \mid \langle T, \mathcal{C}, c \rangle \mid \langle \mathcal{D}, t, c \rangle \mid \langle T \triangleleft \mathcal{D} \rangle \mid \langle \mathcal{D} \triangleleft S \rangle$$

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

$$\frac{E \vdash S \qquad E \vdash T}{E \vdash \langle T \triangleleft S \rangle\ :\ S \to T}\ \text{[T-Cast]}$$

$$\frac{E \vdash \{x{:}B \mid t\} \qquad E \vdash c : B \qquad E \vdash s : \mathtt{Bool} \qquad E \vdash s \Rightarrow [x \mapsto c]\, t}{E \vdash \langle \{x{:}B \mid t\}, s, c \rangle\ :\ \{x{:}B \mid t\}}\ \text{[T-Checking]}$$

of type T (in a manner similar to coercions [S. Thatte 1990], contracts [Findler 2002; Findler and Felleisen 2002], and to type casts in languages such as Java [Gosling et al. 2005]).

The operational semantics of casts to function types is somewhat involved. As described by the rule [E-Cast-F], casting a function t of type $x : S_1 \to S_2$ to the type $x : T_1 \to T_2$ yields a new function

$$\lambda x : T_1.\, (\langle T_2 \lhd S_2 \rangle \circ t \circ \langle S_1 \lhd T_1 \rangle)\ x$$

where $\circ$ is the usual associative composition operator (in the absence of polymorphism, we actually need a family of composition operators, as for `fix` and `if`). This function is of the desired type $x : T_1 \to T_2$; it takes an argument x of type T_1, casts it to a value of type S_1, which is passed to the original function t, and the result of that application is then cast to the desired result type T_2. 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 of higher order contracts [Findler and Felleisen 2002].[2]

The rules [E-Cast-Begin] and [E-Cast-End] deal with casting a basic constant c to a base refinement type $\{x : B \mid t\}$. Via rule [E-Cast-Begin] a cast application $\langle \{x : B \mid t\} \lhd \{x : B \mid s\} \rangle\ c$ evaluates to a cast-in-progress $\langle \{x : B \mid t\}, [x \mapsto c]\, t, c \rangle$. The instantiated predicate $[x \mapsto c]\, t$ then evaluates via the closure rule [E-Ctx], either diverging or terminating in a value v. If v is `true` then the cast-in-progress evaluates to c, as it

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

has been dynamically verified that c has type $\{x{:}B \mid t\}$, otherwise the cast-in-progress is "stuck."

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 1988b], in that there is no type checking of actual program syntax at run time.

To conclude this section, before treating the hybrid type checking algorithm, we note that the Preservation theorem holds as stated, with the implication $E \vdash s \Rightarrow [x \mapsto c]\, t$ ensuring that when a cast ends, it has certainly checked the necessary property.

On the other hand, the Progress theorem holds only modulo failed casts – a term may be "justifiably" stuck because a cast has failed.

Definition 11 (Failed Casts) *A* failed cast *is a term of the form* $\langle\{x{:}B \mid s\}, t, c\rangle$ *where* t *cannot evaluate further, yet* $t \neq \texttt{true}$.

Theorem 12 (Progress) *If* $\emptyset \vdash s : T$ *then either* s *is a value,* s *contains a failed cast, or there is some* s' *such that* $s \rightsquigarrow s'$.

Proof: By induction on the derivation of $\emptyset \vdash s : T$ as before, except in the case for [T-Checking] where s may be a failed cast. In combination with the Evaluation axiom of theorem proving and specification of constants, the premise $E \vdash s \Rightarrow [x \mapsto c]\, t$ ensures that casts on basic constants can only get stuck when the constant does not have the desired type. In the case for [T-App] we know that the function position cannot be a failed cast because they occur only at base types, not function types. □

2.2 Hybrid Type Checking for λ^H

We now describe how to perform hybrid type checking for the language λ^H. We believe this general approach extends to other languages with similarly expressive type systems.

Hybrid type checking relies on an algorithm for conservatively approximating implication between predicates. We assume that for any conjectured implication $E \vdash s \Rightarrow t$, this algorithm returns one of three possible results, which we denote as follows:

- The judgment $E \vdash^{\surd}_{alg} s \Rightarrow t$ means the algorithm finds a proof that $E \vdash s \Rightarrow t$.
- The judgment $E \vdash^{\times}_{alg} s \Rightarrow t$ means the algorithm finds a proof that $E \nvdash s \Rightarrow t$.
- The judgment $E \vdash^{?}_{alg} s \Rightarrow t$ means the algorithm terminates due to a timeout without either discovering a proof of either $E \vdash s \Rightarrow t$ or $E \nvdash s \Rightarrow t$.

We lift this 3-valued algorithmic implication judgment $E \vdash^{a}_{alg} s \Rightarrow t$ (where $a \in \{\surd, \times, ?\}$) to a 3-valued algorithmic subtyping judgment:

$$E \vdash^{a}_{alg} S <: T$$

as shown in Figure 2.3. The subtyping judgment between base refinement types reduces to a corresponding implication judgment, via the rule [SA-Base]. Subtyping between function types reduces to subtyping between corresponding contravariant domain and covariant range types, via the rule [SA-Arrow]. This rule uses the following conjunction

Figure 2.3: Cast insertion Rules

Cast insertion on terms $\boxed{E \vdash s \hookrightarrow t : T}$

$$\frac{(x : T) \in E}{E \vdash x \hookrightarrow x : T}\ [\text{C-Var}] \qquad \frac{}{E \vdash c \hookrightarrow c : ty(c)}\ [\text{C-Const}]$$

$$\frac{E \vdash S_1 \hookrightarrow T_1 \qquad E, x : T_1 \vdash s \hookrightarrow t : T_2}{E \vdash (\lambda x{:}S_1.\, s) \hookrightarrow (\lambda x{:}T_1.\, t) : (x{:}T_1 \to T_2)}\ [\text{C-Fun}]$$

$$\frac{E \vdash s_1 \hookrightarrow t_1 : (x{:}T_1 \to T_2) \qquad E \vdash s_2 \hookrightarrow t_2 \downarrow T_1}{E \vdash s_1\ s_2 \hookrightarrow t_1\ t_2 : [x \mapsto t_2]\, T_2}\ [\text{C-App}]$$

Cast insertion and checking $\boxed{E \vdash s \hookrightarrow t \downarrow T}$

$$\frac{E \vdash s \hookrightarrow t : S \qquad E \vdash^{\checkmark}_{alg} S <: T}{E \vdash s \hookrightarrow t \downarrow T}\ [\text{CC-Ok}]$$

$$\frac{E \vdash s \hookrightarrow t : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash s \hookrightarrow \langle T \triangleleft S \rangle\ t \downarrow T}\ [\text{CC-Chk}]$$

Cast insertion on types $\boxed{E \vdash S \hookrightarrow T}$

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

$$\frac{E, x : B \vdash s \hookrightarrow t : \{y{:}\texttt{Bool} \mid t'\}}{E \vdash \{x{:}B \mid s\} \hookrightarrow \{x{:}B \mid t\}}\ [\text{C-Base}]$$

Subtyping Algorithm $\boxed{E \vdash^{a}_{alg} S <: T}$

$$\frac{E \vdash^{b}_{alg} T_1 <: S_1 \qquad E, x : T_1 \vdash^{c}_{alg} S_2 <: T_2 \qquad a = b \otimes c}{E \vdash^{a}_{alg} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)}\ [\text{SA-Arrow}]$$

$$\frac{E, x : B \vdash^{a}_{alg} s \Rightarrow t \qquad a \in \{\checkmark, \times, ?\}}{E \vdash^{a}_{alg} \{x{:}B \mid s\} <: \{x{:}B \mid t\}}\ [\text{SA-Base}]$$

operation $\otimes$ between three-valued results:

$\otimes$	$\surd$	?	$\times$
$\surd$	$\surd$	?	$\times$
?	?	?	$\times$
$\times$	$\times$	$\times$	$\times$

If the appropriate subtyping relation holds between the domain and range components (*i.e.*, $b = c = \surd$), then the subtyping relation holds between the function types (*i.e.*, $a = \surd$). If the appropriate subtyping relation does not hold between either the domain or range components (*i.e.*, $b = \times$ or $c = \times$), then the subtyping relation does not hold between the function types (*i.e.*, $a = \times$). Otherwise, in the uncertain case, subtyping *may* hold between the function types (*i.e.*, $a = ?$). Thus, like the implication algorithm, the subtyping algorithm need not return a definite answer in all cases.

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. We characterize this process of simultaneous type checking and cast insertion via the *cast insertion judgment*:

$$E \vdash s \hookrightarrow t : T$$

Here, the environment E provides bindings for free variables, s is the original source program, t is a modified version of the original program with additional casts, and T is the inferred type for t. Since types contain terms, we extend this cast insertion process to types via the judgment $E \vdash S \hookrightarrow T$. Some of the cast insertion rules rely on the

auxiliary *cast insertion and checking* judgment:

$$E \vdash s \hookrightarrow t \downarrow T$$

This judgment takes as input an environment E, a source term s, and a desired result type T, and checks that s is converted by cast insertion to a term of this result type.

The rules defining these judgments are shown in Figure 2.3 are mostly straightforward. The rules [C-VAR] and [C-CONST] say that variable references and constants do not require additional casts. The rule [C-FUN] inserts casts into an abstraction $\lambda x{:}S_1.\, s$ by first inserting casts into the type S_1 to yield T_1 and then processing s to yield a term t of type T_2; the resulting abstraction $\lambda x{:}T_1.\, t$ has type $x{:}T_1 \to T_2$. Rule [C-CAST] similarly recurses on the part of a cast. The rule [C-APP] for an application $s_1\ s_2$ processes s_1 to a term t_1 of type $x{:}T_1 \to T_2$ then invokes the cast insert and checking judgement to convert s_2 into a term of the appropriate argument type T_1.

The two rules defining the cast insertion and checking judgment $E \vdash s \hookrightarrow u \downarrow T$ demonstrate the key idea of hybrid type checking. Both rules start by processing s to a term t of some type S. The crucial question is then whether this type S is a subtype of the expected type T:

- If the subtyping algorithm succeeds in proving that S is a subtype of T (*i.e.*, $E \vdash^{\checkmark}_{alg} S <: T$), then t is clearly of the desired type T, and so the rule [CC-OK] returns t.

- If the subtyping algorithm can show that S is not a subtype of T (*i.e.*, $E \vdash^{\times}_{alg} S <: T$), then the program is rejected since no rule is applicable.

- Otherwise, in the uncertain case where $E \vdash^{?}_{alg} S <: T$, the rule [CC-Chk] inserts the type cast $\langle T \triangleleft S \rangle$ to dynamically ensure that values returned by t are actually of the desired type T.

These rules for cast insertion and checking illustrate the key benefit of hybrid type checking: specific static analysis problem instances (such as $E \vdash S <: T$) that are undecidable or computationally intractable can be avoided in a convenient manner simply by inserting appropriate dynamic checks. Of course, we should not abuse this facility, and so ideally the subtyping algorithm should yield a precise answer in most cases. However, the critical contribution of hybrid type checking is that it avoids the very strict requirement of demanding a precise answer for *all* subtyping questions.

Cast insertion on types is straightforward. The rule [C-Arrow] inserts casts in the domain and codomain of a function type $x{:}S \to T$ and reassembles the components. The rule [C-Base] inserts casts into the refinement t of a base type $\{x{:}B \mid t\}$, producing t' (whose type should be a subtype of `Bool`), and then yielding the base refinement type $\{x{:}B \mid t'\}$.

Since checking well formedness of a type is actually a cast insertion process which returns a well formed type (possibly with added casts), we only perform cast insertion on types where necessary, at λ-abstractions and casts, when we encounter (possibly ill formed) types in the source program. In particular, the cast insertion rules do not explicitly check that the environment is well formed, since that would involve repeatedly processing all types in that environment. Instead, the rules assume that the environment is well formed.

2.3 An Example

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 λ^H 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:

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

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

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

We now define the function `serializeMatrix` as:

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

The elided term ... 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`:

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

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:

$$n : \texttt{Int},\ m : \texttt{Int},\ a : \texttt{Array}\ \vdash (\texttt{asize}\ a = e) \Rightarrow (\texttt{asize}\ a = (n \times m))$$

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 (*e.g.*, Detlefs et al. [2005]; Blei et al. [2000]) 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 in the presence of an extremely 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 no casts are yet necessary. Note that a theorem prover which is not complete for arbitrary multiplications 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 due to space constraints we have elided the source type of the cast):

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

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

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

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:

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

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 any of these cases, since the type T uses nonlinear 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 clearly an improvement over not detecting these errors at all, leading to subsequent crashes or incorrect results.

2.4 Correctness of Cast Insertion

Since hybrid type checking relies on necessarily incomplete algorithms for subtyping and implication, we next investigate what correctness properties are guaranteed by this cast insertion process.

We assume the 3-valued algorithm for checking implication between boolean terms is sound in the following sense:

Assumption 13 (Soundness of $E \vdash^{a}_{alg} s \Rightarrow t$) *Suppose $\vdash E$.*

1. *If $E \vdash^{\checkmark}_{alg} s \Rightarrow t$ then $E \vdash s \Rightarrow t$.*

2. *If $E \vdash^{\times}_{alg} s \Rightarrow t$ then $E \nvdash s \Rightarrow t$.*

Note that this algorithm does not need to be complete (indeed, an extremely naive algorithm could simply return $E \vdash^{?}_{alg} s \Rightarrow t$ in all cases). A consequence of the soundness of the implication algorithm is that the algorithmic subtyping judgment $E \vdash_{alg} S <: T$ is also sound.

Lemma 14 (Soundness of $E \vdash^{a}_{alg} S <: T$) *Suppose $\vdash E$.*

1. *If $E \vdash^{\checkmark}_{alg} S <: T$ then $E \vdash S <: T$.*

2. *If $E \vdash^{\times}_{alg} S <: T$ then $E \nvdash S <: T$.*

PROOF: By induction on derivations using Assumption 13. □

Becasue algorithmic subtyping is sound, the hybrid cast insertion algorithm generates only well typed programs:

Theorem 15 (Compilation Type Soundness) *Suppose* $\vdash E$.

1. *If* $E \vdash t \hookrightarrow t' : T$ *then* $E \vdash t' : T$.

2. *If* $E \vdash t \hookrightarrow t' \downarrow T$ *and* $E \vdash T$ *then* $E \vdash t' : T$.

3. *If* $E \vdash T \hookrightarrow T'$ *then* $E \vdash T'$.

PROOF: By induction on cast insertion derivations. □

Since the generated code is well typed, standard type-directed cast insertion and optimization techniques [Tarditi et al. 1996; Morrisett et al. 1999] are applicable. Furthermore, the generated code includes all the type specifications present in the original program, and so by the Preservation Theorem these specifications will never be violated at run time. Any attempt to violate a specification is detected via a combination of static checking (where possible) and dynamic checking (only when necessary).

If closed term t has type S which is a subtype of T, then it is almost self-evident that t is equivalent to $\langle T \triangleleft S \rangle\ t$, hence casts inserted during compilation are redundant. Rather than prove this directly from the evaluation relation, we present a straightforward argument based on the usual notion of extensionality for functions.

Lemma 16 (Behavioral Correctness of Cast Insertion) *If* $\vdash t : S$ *and* $\vdash S <: T$ *then* t *is observationally equivalent to* $\langle T \triangleleft S \rangle\ t$

PROOF: By induction on the height of T in terms of the function space arrow – which equals the height of S.

Case T is a refined base type $\{y{:}B \mid s\}$. Then t either diverges, or evaluates to a basic constant c of type B, in which case $[y \mapsto c]\, s \rightsquigarrow^* \texttt{true}$ by assumption and the cast succeeds, evaluating to c. Confluence of evaluation then assures that t, c, and $\langle T \triangleleft S \rangle\ t$ are observationally equivalent.

Case T is a function type $y{:}T_1 \to T_2$. Then $S = y{:}S_1 \to S_2$ by inversion of subtyping, and we invoke extensionality. Consider any s of type T_1. Since T_1 is a subtype of S_1, also s has type S_1 and by induction is equivalent to $\langle S_1 \triangleleft T_1 \rangle\ s$. Now $t\ s$ has type $[y \mapsto s]T_2$ so by induction and combined with the above, we have

$$t\ s = \langle [y \mapsto s]\, T_1 \triangleleft [y \mapsto s]\, S_2 \rangle\ t\ (\langle S_1 \triangleleft T_1 \rangle\ s)$$

which is an unfolding of the result of the function cast so t is observationally equivalent to $\langle y{:}T_1 \to T_2 \triangleleft y{:}S_1 \to S_2 \rangle\ t$ $\square$

Since for a well typed term cast insertion inserts only such redundant casts, it does not change the behavior of well typed programs. This is key to understanding a form of completeness for hybrid type checking.

Theorem 17 (Completeness of Cast Insertion)

1. *If* $E \vdash s : S$ *then* $\exists t, T$ *such that* $E \vdash s \hookrightarrow t : T$.

2. *If* $E \vdash s : S$ *and* $E \vdash S <: T$ *then* $\exists t$ *such that* $E \vdash s \hookrightarrow t \downarrow T$.

3. *If* $E \vdash S$ *then* $\exists T$ *such that* $E \vdash S \hookrightarrow T$.

PROOF: By mutual induction on the antecedent derivations.

1. *Case* [T-VAR], [T-CONST] : Immediate.

   Case [T-SUB] : Immediate from the subderivation.

   Case [T-APP], [T-FUN] : By induction, we can insert casts into each subterm yielding well typed terms which we reassemble. Lemmas 7 and 16 ensure that the types involved remain equivalent under cast insertion, so all the same subtyping judgements hold.

2. By part one, there exist t and U such that $E \vdash s \hookrightarrow t : U$. By Lemma 14 we know that $E \vdash t : S$, so a sound algorithmic subtyping relation cannot reject t, but only insert casts, yielding t' such that $E \vdash t \hookrightarrow t' \downarrow T$.

3. Straightforward by induction, since valid implications may not be rejected. □

2.5 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 compare to conventional static type checkers.

On the experimental side, the SAGE language implementation demonstrates that hybrid type checking interacts comfortably with a variety of typing constructs, including first-class types, polymorphism, recursive data structures, as well as the type `Dynamic`, and that the number of inserted casts for some example programs is low or none [Gronski et al. 2006].

So let us now examine this question theoretically. Suppose we are given a static type checker that targets a restricted subset of λ^H for which type checking is statically decidable. Specifically, we assume there exists a subset $\mathcal{D}$ of *Term* such that for all $t_1, t_2 \in \mathcal{D}$ and for all environments E (containing only $\mathcal{D}$-terms), the judgment $E \vdash t_1 \Rightarrow t_2$ is decidable. We introduce the language λ^S that is obtained from λ^H by only permitting $\mathcal{D}$-terms in refinement types.

As an extreme, we could take $\mathcal{D} = \{\texttt{true}\}$, in which case the λ^S type language is essentially the simply typed λ-calculus:

$$T ::= B \mid T \to T$$

However, to yield a more general argument, we assume only that $\mathcal{D}$ is a subset of *Term* for which implication is decidable. It then follows that subtyping and type checking for λ^S are also decidable, and we denote this type checking judgment as $E \vdash^{\mathtt{S}} t : T$.

Clearly, the hybrid implication algorithm can give precise answers on (decidable) $\mathcal{D}$-terms, and so we assume that for all $t_1, t_2 \in \mathcal{D}$ and for all environments E, the judgment $E \vdash^a_{alg} t_1 \Rightarrow t_2$ 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) λ^S programs.

Theorem 18 *For all λ^S terms t, λ^S environments E, and λ^S types T, the following three statements are equivalent:*

1. $E \vdash^{\mathtt{S}} t : T$
2. $E \vdash t : T$

3. $E \vdash t \hookrightarrow t : T$

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

Thus, to a λ^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 λ^H programmer. To enable this comparison, we need to map expressive λ^H types into the more restrictive λ^S types, and in particular to map arbitrary boolean terms into $\mathcal{D}$-terms. We assume the computable function

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

performs this mapping. The function *erase* then maps λ^H refinement types to λ^S refinement types by using γ to abstract boolean terms:

$$\mathit{erase}\{x{:}B \mid t\} \;=\; \{x{:}B \mid \gamma(t)\}$$

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

As might be expected, the *erase* function must lose information, with the consequence that for any computable mapping γ there exists some program P such that hybrid type checking of P performs better than static type checking of $\mathit{erase}(P)$. In

other words, because the hybrid type checker supports more precise specifications, it performs better than a traditional static type checker, which necessarily must work with less precise but decidable specifications.

Theorem 19 *For any computable mapping γ either:*

1. *the static type checker rejects the erased version of some well typed λ^H program, or*

2. *the static type checker accepts the erased version of some ill typed λ^H program for which the hybrid type checker would statically detect the error.*

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

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

$$\{t \mid (E \vdash t \Rightarrow \texttt{false})\} \quad \neq \quad \{t \mid (E \vdash \gamma(t) \Rightarrow \gamma(\texttt{false}))\}$$

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:

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

$$E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$$

We construct as a counter-example the program P_1:

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

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

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

holds. Hence, P_1 is well typed, and (by Theorem 17) is accepted by the hybrid type checker. However, from the assumption $E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$ the erased version of the subtyping judgment does not hold:

$$\emptyset \not\vdash \mathit{erase}(\{x{:}\texttt{Int} \,|\, w\}) <: \mathit{erase}(\{x{:}\texttt{Int} \,|\, \texttt{false}\})$$

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

2. Conversely, suppose:

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

$$E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$$

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

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

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

$$P_2 \quad = \quad \lambda x{:}\{x{:}\texttt{Int} \,|\, w\}.\, (x \ \texttt{as}\ \{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n = m)\})$$

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 t ever evaluates to `true`.

$$\frac{[x \mapsto a]\, t \longrightarrow^{b} \texttt{true}}{E \vdash^{\times}_{alg} t \Rightarrow (\texttt{false} \wedge a = b)}$$

This rule enables the implication algorithm to conclude that:

$$E \vdash^{\times}_{alg} w \Rightarrow \texttt{false} \wedge (n = m)$$

Hence, the subtyping algorithm can conclude:

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

Therefore, the hybrid type checker rejects P_2, which by Lemma 17 is therefore ill typed.

$$\forall P, T. \quad \not\vdash P_2 \hookrightarrow P : T$$

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 judgement holds:

$$E \vdash \gamma(\texttt{false}) \Rightarrow \gamma(\texttt{false} \wedge (n = m))$$

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

$$E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false} \wedge (n = m))$$

Hence the subtyping judgement

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

holds and the program $erase(P_2)$ is accepted by the static type checker:

$$\emptyset \vdash erase(P_2) \,:\, \{x{:}\texttt{Int} \,|\, \gamma(w)\} \rightarrow \{x{:}\texttt{Int} \,|\, \gamma(\texttt{false} \wedge (n = m))\}$$

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

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

This program is well typed, from the subtype judgment:

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

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

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

Hence, $erase(P_3)$ is rejected by the static type checker:

$$\forall T. \quad \emptyset \not\vdash^{\mathrm{S}} erase(P_3) \,:\, T \;\square$$

Note that in the second case of the proof, we see that a pessimistic erasure will allow the static checker to reject some faulty programs that the hybrid checker cannot. For example, replacing instances of the halting problem with instances of the bounded-length halting problem, or generally when $\neg t$ implies $\neg\gamma(t)$. The converse does not hold in general, so we can be certain that S rejects good programs as well.

2.6 Related Work

Much prior work has focused on dynamic checking of expressive specifications, or *contracts* [Meyer 1988; Findler and Felleisen 2002; Leavens and Cheon 2005; Gomes et al. 1996; Holt and Cordy 1988; Luckham 1990; Parnas 1972; Kölling and Rosenberg 1997]. An entire design philosophy, *Contract Oriented Design*, has been based on dynamically-checked specifications. Hybrid type checking embraces precise specifications, but extends prior purely dynamic techniques to verify (or detect violations of) expressive specifications statically, wherever possible.

The programming language Eiffel [Meyer 1988] supports a notion of hybrid specifications by providing both statically-checked types as well as dynamically-checked contracts. Having separate (static and dynamic) 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.

Other authors have considered pragmatic combinations of both static and dynamic checking. Abadi et al. [1989] 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). Henglein [1994] characterized the *completion process* of inserting the necessary coercions, and presented a rewriting system for generating minimal completions. S. Thatte [1990] developed a similar system in which the necessary casts are implicit. These systems are intended to support looser type specifications. In contrast, our work

uses similar, automatically-inserted casts to support more precise type specifications. An interesting avenue for further exploration is the combination of both approaches to support a large range of specifications, from `Dynamic` at one end to precise hybrid-checked specifications at the other.

The static checking tool ESC/Java [Flanagan et al. 2002] checks expressive JML specifications [Burdy et al. 2003; Leavens and Cheon 2005] using the Simplify automatic theorem prover [Detlefs et al. 2005]. However, Simplify does not distinguish between failing to prove a theorem and finding a counter-example that refutes the theorem, and so ESC/Java's error messages may be caused either by incorrect programs or by limitations in its theorem prover.

The limitations of purely static and purely dynamic approaches have also motivated other work on hybrid analyses. For example, CCured [Necula et al. 2002] is a sophisticated hybrid analysis for preventing the ubiqutous 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. Specialized hybrid analyses have been proposed for other problems as well, such as data race condition checking [von Praun and Gross 2001; O'Callahan and Choi 2003; Agarwal and Stoller 2004].

Prior work (*e.g.* [Breazu-Tannen et al. 1991]) introduced and studied implicit coercions in type systems. Note that there are no implicit coercions in the λ^H type system itself, but only in the cast insertion algorithm, and so we do not need a coherence theorem for λ^H, but instead reason about the connection between the type system and

cast insertion algorithm.

Chapter 3

Type Reconstruction

We have addressed the problem of type-checking for a language like λ^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, 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 λ^H, type reconstruction is clearly undecidable.

Instead of surrendering to undecidability, we separate type reconstruction from 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. The generalized definition also extends to undecidable type systems, since alternative techniques, such as hybrid type checking, can be applied to the resulting program. In particular, type reconstruction for λ^H is now 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 fixpoint operations in the specification logic.

In the richer setting of λ^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 resulting, explicitly typed program can then be checked by the λ^H compilation algorithm [Flanagan 2006], which reasons about local implications between refinement predicates. If the compilation algorithm cannot prove or refute a particular implication, it dynamically enforces the desired property via a run-time check. These dynamic checks are only ever necessary for user-specified predicates; inferred predicates

(which may include existential quantification and fixpoint operations) are correct by construction.

3.1 Type Reconstruction

For the type reconstruction problem, we consider only the basic language, ignoring casts since they add no interest to type reconstruction.

We extend the type language with type variables $\alpha \in TyVar$. Type reconstruction yields a function $\pi : TyVar \to Type$, here called a *type replacement*. Application of a type replacement is lifted compatibly to all syntactic sorts, and is not capture avoiding.

The three phases of type reconstruction proceed as follows:

1. The input program is processed to yield a set C of subtyping constraints of the form $E \vdash S <: T$ (the same as the subtyping judgement).

2. The shape reconstruction phase then reduces C into a set P of implication constraints, each of the form $E \vdash p \Rightarrow q$ (the same as the implication judgement).

3. The last phase of type reconstruction solves P.

To facilitate our development, we require that the language be closed under substitution. But a substitution cannot immediately be applied to a type variable, so

each type variable α has an associated delayed substitution θ (which may be empty).

$$T \quad ::= \quad \cdots \quad | \quad \theta \cdot \alpha$$

$$\theta \quad ::= \quad [\,] \quad | \quad [x \mapsto t : T], \theta$$

The usual definition of capture-avoiding substitution is extended to type variables, which simply delay that substitution:

$$[x \mapsto s : T]\ (\theta \cdot \alpha) \ = \ ([x \mapsto s : T], \theta) \cdot \alpha$$

When a type replacement is applied to a type variable α with a delayed substitution θ, the substitution $\pi(\theta)$ is immediately applied to $\pi(\alpha)$:

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

Notice that we have added a type annotation to these explicit substitution – this is for later syntactic processing and does not affect the semantics of substitution.

3.2 Constraint Generation

The constraint generation judgement $E \vdash t \ : \ T \ \& \ C$ is defined in Figure 3.1 and reads: term t has type T in environment E, subject to the constraint set C. Each rule is derived from the corresponding type rule, with subsumption distributed throughout the derivation to make the rules syntax-directed.

For a type replacement π, if $\pi(C)$ contains only valid subtyping relationships, then π *satisfies* C. When applied to a typeable λ^H 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.

Figure 3.1: Constraint Generation Rules

Constraint Generation rules $\boxed{E \vdash t : T \;\&\; C}$

$$\frac{(x : T) \in E}{E \vdash x : T \;\&\; \emptyset}\ [\text{CG-Var}] \qquad \frac{}{E \vdash c : ty(c) \;\&\; \emptyset}\ [\text{CG-Const}]$$

$$\frac{E \vdash S \;\&\; C_1 \qquad E, x : S \vdash t : T \;\&\; C_2}{E \vdash (\lambda x{:}S.\, t) : (x{:}S \to T) \;\&\; C_1 \cup C_2}\ [\text{CG-Fun}]$$

$$\frac{E \vdash t_1 : T \;\&\; C_1 \qquad E \vdash t_2 : S \;\&\; C_2 \qquad \alpha \text{ fresh}}{E \vdash t_1\ t_2 : [x \mapsto t_2 : S] \cdot \alpha \;\&\; C_1 \cup C_2 \cup \{E \vdash T <: (x{:}S \to \alpha)\}}\ [\text{CG-App}]$$

Well-formed Type Constraint Generation $\boxed{E \vdash T \;\&\; C}$

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

$$\frac{E, x : B \vdash t : \texttt{Bool} \;\&\; C}{E \vdash \{x{:}B \mid t\} \;\&\; C}\ [\text{WTC-Base}] \qquad \frac{}{E \vdash \theta \cdot \alpha \;\&\; \emptyset}\ [\text{WTC-Var}]$$

Lemma 20 *For any environment E and term t:*

$$\exists \pi, T.\ \pi(E) \vdash \pi(t) : \pi(T) \quad \Longleftrightarrow \quad \exists \pi', S, C. \begin{cases} E \vdash t : S \ \&\ C \\ \pi' \textit{ satisfies } C \end{cases}$$

PROOF: ($\Rightarrow$): Suppose $\pi E \vdash \pi t : \pi T$. then $\exists \pi', S, C. E \vdash t : S \ \&\ C$ and $\pi'\pi$ satisfies C and $\pi E \vdash \pi' \pi S <: \pi T$. Proceed by induction on the derivation of $\pi E \vdash \pi t : \pi T$, with this strengthened hypothesis.

Case [T-APP]: Given: $t = t_1\ t_2$ and $T = [x \mapsto t_2 : T_1]\, T_2$ where

$$\frac{\pi E \vdash \pi t_1 : x{:}\pi T_1 \to \pi T_2 \qquad \pi E \vdash \pi t_2 : \pi T_1}{\pi E \vdash \pi t_1\ \pi t_2 : [x \mapsto \pi t_2 : T_1]\,(\pi T_2)}$$

By induction we have π_1, π_2, S_1, S_2, C_1, C_2 such that

$E \vdash t_1 : S_1 \ \&\ C_1$ $\quad$ $\pi_1\pi$ satisfies C_1 $\quad$ $\pi E \vdash \pi_1 \pi S_1 <: x{:}\pi T_1 \to \pi T_2$

$E \vdash t_2 : S_2 \ \&\ C_2$ $\quad$ $\pi_2\pi$ satisfies C_2 $\quad$ $\pi E \vdash \pi_2 \pi S_2 <: \pi T_1$

Hence by [CG-APP] we derive $E \vdash t_1\ t_2 : [x \mapsto t_2 : S_2]\,\alpha \ \&\ C$ where $C = C_1 \cup C_2 \cup \{E \vdash S_1 <: x{:}S_2 \to \alpha\}, \alpha$ fresh. Note that $dom(\pi_1), dom(\pi_2), \alpha$ are pairwise disjoint.

Let $\pi'(\alpha) = [\alpha \mapsto \pi(T_2)] \circ \pi_1\pi_2$. Then $\pi'\pi(C) = (\pi_1\pi C_1) \cup (\pi_2\pi C_2) \cup \{\pi E \vdash \pi_1\pi S_1 <: x{:}\pi_2\pi S_2 \to \pi T_2\}$ where the indicated subtyping constraint is valid (with the given data, apply [S-ARROW], reflexivity and transivitity of subtyping). Thus $\pi'\pi$ satisfies C.

And finally, $\pi'\pi(\alpha[x \mapsto t_2 : S_2]) = \pi(T_2[x \mapsto t_2 : S_1])$

hence $\pi E \vdash \pi'\pi(\alpha[x \mapsto t_2 : S_2]) <: \pi(T_2[x \mapsto t_2 : T_1])$

The remaining cases and reverse direction are straightforward. □

Consider the following λ^H term t (the expression $\texttt{let}\ x : T = s\ \texttt{in}\ t$ is syntactic sugar for $(\lambda x{:}T.\, t)\ s$).

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

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

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

$$id\ (id\ y)$$

Eliding some generated type variables for clarity, the corresponding constraint generation judgement is

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

where C contains the following constraints, in which $T_{id} \equiv (x : \alpha_1 \to \alpha_2)$ and $T_y \equiv \{n{:}\texttt{Int} \mid n > w\}$:

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

3.3 Shape Reconstruction

The second step of reconstruction is to infer a type's basic shape, ignoring refinement predicates. To defer reconstruction of refinements, we introduce placeholders

$\gamma \in Placeholder$ to represent unknown refinement predicates (in the same way that type variables represent unknown types) Like type variables, each placeholder has an associated delayed substitution.

$$t \quad ::= \quad \cdots \mid \theta \cdot \gamma$$

A placeholder replacement is a function $\rho : Placeholder \rightarrow Term$ and is lifted compatibly to all syntactic structures. As with type replacements, applying placeholder replacement allows any delayed substitutions also to be applied.

$$\begin{aligned} [x \mapsto t : T](\theta \cdot \gamma) &= ([x \mapsto t : T], \theta) \cdot \gamma \\ \rho(\theta \cdot \gamma) &= \rho(\theta)(\rho(\gamma)) \end{aligned}$$

The shape reconstruction algorithm, detailed in Figure 3.2 takes as input a subtyping constraint set C and processes the constraints in C nondeterministically according to the rules in Figure 3.2. 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 $E \vdash p \Rightarrow q$, and the type replacement π are outputs of the algorithm. For a placeholder replacement ρ, if $\rho(P)$ contains only valid implications, then ρ *satisfies* P.

Each rule in Figure 3.2 resembles a step of traditional type reconstruction. When a type variable α must have the shape of a function type, it is replaced by $x{:}\alpha_1 \rightarrow \alpha_2$, where α_1 and α_2 are fresh type variables. The function $occurs$ checks that α has a finite solution, since λ^H does not have recursive types. Occurences of α which appear in refinement predicates or in the range of a delayed substitution are ignored –

Figure 3.2: Shape Reconstruction Algorithm

Input: C

Output: π, P

Initially: $P = \emptyset$ and $\pi = [\,]$

match some constraint in C until quiescent:

$$\begin{array}{lll}
E \vdash \theta \cdot \alpha <: x{:}T_1 \to T_2 & \Longrightarrow & \text{if } \mathit{occurs}(\alpha, x{:}T_1 \to T_2) \text{ then } \mathit{fail} \\
\text{or} \quad E \vdash x{:}T_1 \to T_2 <: \theta \cdot \alpha & & \text{otherwise for fresh } \alpha_1, \alpha_2 \\
& & \pi := [\alpha \mapsto x{:}\alpha_1 \to \alpha_2] \circ \pi \\
& & C := \pi(C) \\
& & P := \pi(P) \\
E \vdash \theta \cdot \alpha <: \{x{:}B \,|\, t\} & \Longrightarrow & \text{for fresh } \gamma \\
\text{or} \quad E \vdash \{x{:}B \,|\, t\} <: \theta \cdot \alpha & & \pi := [\alpha \mapsto \{x{:}B \,|\, \gamma\}] \circ \pi \\
& & C := \pi(C) \\
& & P := \pi(P) \\
E \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2) & \Longrightarrow & C := C \cup \left\{ \begin{array}{l} E \vdash T_1 <: S_1, \\ E, x : T_1 \vdash S_2 <: T_2 \end{array} \right\} \\
E \vdash \{x{:}B \,|\, p\} <: \{x{:}B \,|\, q\} & \Longrightarrow & P := P \cup \{E \vdash p \Rightarrow q\} \\
E \vdash \{x{:}B \,|\, p\} <: x{:}S \to T & \Longrightarrow & \mathit{fail} \\
\text{or} \quad E \vdash x{:}S \to T <: \{x{:}B \,|\, p\} & &
\end{array}$$

these occurences do not require a solution involving recursive types.

$$occurs(\alpha, \{x{:}B \mid t\}) = false$$
$$occurs(\alpha, \theta \cdot \alpha') = false \quad (\alpha \neq \alpha')$$
$$occurs(\alpha, \theta \cdot \alpha) = true$$
$$occurs(\alpha, x{:}S \to T) = occurs(\alpha, S) \vee occurs(\alpha, T)$$

When a type variable must be a refinement of a base type B, the type variable is replaced by $\{x{:}B \mid \gamma\}$ where γ is a fresh placeholder. A subtyping constraint between two function types induces additional constraints between the domains and codomains of the function types. When two refined base types are constrained to be subtypes, a corresponding implication constraint between their refinements is added to P.

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).

Lemma 21 *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.*

PROOF:

Case 1: By inversion of subtyping and lack of recursive types, C is unsatisfiable.

Case 2: Each step of the algorithm maintains the invariant that C is satisfiable if and only if there exists some π' and ρ such that $\rho\pi'\pi C$ contains only valid subtyping relationships.

When shape reconstruction terminates, π' is necessarily empty since all type variables are eliminated. The aforementioned ρ also satisfies P by inversion of [S-Base].

Then for any other ρ' which satisfies P, see that all constraints involving implication that were satisfied by ρ are also satisfied by ρ' thus $\rho'\pi$ satisfies C. $\square$

Returning to our example, shape reconstruction returns the type replacement

$$\pi = [\ \alpha_1 := \{n{:}\texttt{Int} \mid \gamma_1\},\ \alpha_2 := \{n{:}\texttt{Int} \mid \gamma_2\},\ \alpha_3 := \{n{:}\texttt{Int} \mid \gamma_3\}\ ]$$

and the following implication constraint set P, in which $T_{id} = x : \{n{:}\texttt{Int} \mid \gamma_2\} \to \{n{:}\texttt{Int} \mid \gamma_3\}$ and $T_y = \{n{:}\texttt{Int} \mid n > w\}$:

$$\begin{aligned}
n : \texttt{Int} \ &\vdash \quad \gamma_1 \ \Rightarrow\ \gamma_3 \\
x : \{n{:}\texttt{Int} \mid \gamma_1\},\ n : \texttt{Int} \ &\vdash \quad \gamma_3 \ \Rightarrow\ \gamma_2 \\
id : T_{id},\ n : \texttt{Int} \ &\vdash \quad (n = 0) \ \Rightarrow\ (n = 0) \\
id : T_{id},\ w : \{n{:}\texttt{Int} \mid n = 0\},\ n : \texttt{Int} \ &\vdash \quad (n = 3) \ \Rightarrow\ (n > w) \\
id : T_{id}, w : \{n{:}\texttt{Int} \mid n = 0\},\ y : T_y,\ n : \texttt{Int} \ &\vdash \quad (n > w) \ \Rightarrow\ \gamma_1 \\
id : T_{id}, w : \{n{:}\texttt{Int} \mid n = 0\},\ y : T_y,\ n : \texttt{Int} \ &\vdash \quad [x \mapsto y : \{n{:}\texttt{Int} \mid \gamma_1\}] \cdot \gamma_2 \Rightarrow \gamma_1
\end{aligned}$$

3.4 Satisfiability

The final phase of type reconstruction solves the residual implication constraint set P by finding a placeholder replacement that preserves satisfiability. To simplify the proofs, we argue assuming that environments are quantified over well typed terms, making explicit the previously abstract implication relation.

Our approach is based on the intuition that implications are essentially dataflow paths that carry the specifications of data sources (constants and function post-conditions) to the requirements of data sinks (function pre-conditions), with placeholders functioning as intermediate nodes in the dataflow graph. Thus, if a placeholder γ appears on the right-hand side of two implication constraints $E \vdash p \Rightarrow \gamma$ and $E \vdash q \Rightarrow \gamma$, then our replacement for γ 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:

1. p or q may contain variables that cannot appear in a solution for γ
2. γ may have a delayed substitution
3. γ may appear in p or q

To help resolve these issues, we extend the language with the following terms.

$$s, t \in Term \quad ::= \quad \cdots \quad | \quad t \vee t \quad | \quad t \wedge t \quad | \quad \exists x : T.\ t$$

The parallel disjunction $t_1 \vee t_2$ (respectively conjunction $t_1 \wedge t_2$) evaluates t_1 and t_2 nondeterministically, reducing to `true` (resp. `false`) if either of them reduces

Figure 3.3: Additional Evaluation Rules

$$\texttt{true} \vee t \longrightarrow \texttt{true} \quad \text{[E-Or-L]} \qquad \texttt{false} \wedge t \longrightarrow \texttt{false} \quad \text{[E-And-L]}$$

$$t \vee \texttt{true} \longrightarrow \texttt{true} \quad \text{[E-Or-R]} \qquad t \wedge \texttt{false} \longrightarrow \texttt{false} \quad \text{[E-And-R]}$$

$$\texttt{false} \vee \texttt{false} \longrightarrow \texttt{false} \quad \text{[E-Or-F]} \qquad \texttt{true} \wedge \texttt{true} \longrightarrow \texttt{true} \quad \text{[E-And-T]}$$

$$\exists x : T.\ t \longrightarrow t[x \mapsto s : T] \quad \text{if } \varnothing \vdash s : T \quad \text{[E-Exists]}$$

$$\mathcal{C} ::= \cdots \mid t \vee \bullet \mid \bullet \vee t \mid \bullet \wedge t \mid t \wedge \bullet$$

to `true` (resp. `false`). The existential term $\exists x : T.\ t$ binds x in t, and evaluates by nondeterministically replacing x with a closed term of type T. The evaluation rules are summarized in Figure 3.3. These additions are relatively unsurprising, considering their importance in computability and topology of data types [Escardo 2004].

3.4.1 Free Variable Elimination

In our example program, the type variable α_1 appeared in the empty environment and $\pi(\alpha_1) = \{n{:}\texttt{Int} \mid \gamma_1\}$, so the solution for γ_1 should be a well-formed boolean expression in the environment $n : \texttt{Int}$. The only variable that can appear in a solution for γ_1 is therefore n. But consider the following constraint over γ_1:

$$id : T_{id},\ w : \{n{:}\texttt{Int} \mid n = 0\},\ y : T_y,\ n : \texttt{Int} \vdash (n > w) \Rightarrow \gamma_1$$

Since id, w, and y cannot appear in a solution for γ_1, we rewrite this constraint as

$$n : \texttt{Int} \vdash (\exists id : T_{id}.\ \exists w : \{n{:}\texttt{Int} \mid n = 0\}.\ \exists y : T_y.\ n > w) \Rightarrow \gamma_1$$

In general, each placeholder γ introduced by shape reconstruction has an associated environment E_γ in which it must have type `Bool`. This gives us a reasonable definition for the free variables of a placeholder (with its associated delayed subtitution):

$$fv(\theta \cdot \gamma) = (dom(E_\gamma) \setminus dom(\theta)) \cup fv(rng(\theta))$$

We then rewrite each implication constraint $E, y : T \vdash p \Rightarrow q$ where $y \notin fv(q)$ into the constraint $E \vdash (\exists y : T.\ p) \Rightarrow q$. This transformation is semantics-preserving (Lemma 22)

For the proof of this fact, let us define closing substitutions. We say $E \models \sigma$ if for every x in E, $\vdash \sigma(x) : \sigma(E(x))$. Then a predicate is valid if and only if it evaluates to `true` for all closing substitutions.

Lemma 22 *For $y \notin fv(q)$, $E, y : T \vdash p \Rightarrow q$ if and only if $E \vdash (\exists y : T.\ p) \Rightarrow q$*

PROOF: ($\Rightarrow$) Suppose $E, y : T \vdash p \Rightarrow q$ and $y \notin fv(q)$ and consider any σ such that $E \models \sigma$ and $\sigma(\exists y : T.\ p) \rightsquigarrow^*$ `true`. Then there is some t such that $\varnothing \vdash t : T$ and $\sigma(\exists y : T.\ p) \rightsquigarrow \sigma([x \mapsto t : T]p) \rightsquigarrow^*$ `true`.

Let $\sigma' = (\sigma, y := t)$; clearly $E, y : T \models \sigma'$. Then by assumption, $\sigma(p) \rightsquigarrow^*$ `true` implies $\sigma(q) \rightsquigarrow^*$ `true`. Since $y \notin fv(q)$, $\sigma'(q) = \sigma(q)$, and we have proved $E \vdash \exists y : T.\ p \Rightarrow q$.

($\Leftarrow$) Conversely, suppose $E \vdash \exists y : T.\ p \Rightarrow q$ where $y \notin fv(q)$ and consider σ such that $E, y : T \models \sigma$ and $\sigma(p) \rightsquigarrow^*$ `true`.

Clearly we can ignore $\sigma(y)$, so $E \models \sigma$, and the nondeterministic existential can replace y with $\sigma(y)$, so $\sigma(\exists y : T.\ p) \rightsquigarrow^* \sigma(p) \rightsquigarrow^*$ `true`. By assumption, this implies

that $\sigma(q) \rightsquigarrow^* \texttt{true}$, and we have proved $E, y : T \vdash p \Rightarrow q$. $\square$

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.

3.4.2 Delayed Substitution Elimination

The next issue is the presence of delayed substitutions in constraints of the form $E \vdash p \Rightarrow \theta \cdot \gamma$. To eliminate the delayed substitution θ we first split it into an environment $env(\theta)$ and a term $[\![\theta]\!]$:

$$\begin{array}{rcl@{\qquad}rcl} env([\,]) & = & \varnothing & [\![\,[\,]\,]\!] & = & \texttt{true} \\ env([x \mapsto t : T], \theta) & = & x : T, env(\theta) & [\![\,[x \mapsto t : T], \theta\,]\!] & = & (x = t) \wedge [\![\theta]\!] \end{array}$$

The environment $env(\theta)$ binds all the variables in $dom(\theta)$ while the term $[\![\theta]\!]$ represents the semantic content of θ.

We then transform the constraint $E \vdash p \Rightarrow \theta \cdot \gamma$ into $E, env(\theta) \vdash [\![\theta]\!] \wedge p \Rightarrow \gamma$. But we can rewrite the constraint even more cleanly: E must be some prefix of E_γ since by the previous transformation $dom(E) \subseteq fv(\theta \cdot \gamma) \subseteq dom(E_\gamma)$. Any $x \in dom(\theta)$ such that $x \notin dom(E_\gamma)$ can be dropped from θ and we see that $E, env(\theta)$ is then exactly E_γ. So our constraint is

$$E_\gamma \vdash [\![\theta]\!] \wedge p \Rightarrow \gamma$$

To prove this transformation correct, we use the following well formedness judgement $E \vdash_{\text{wf}} \theta$ which distinguishes those delayed substitutions that may actually

occur in context E.

$$\frac{}{E \vdash_{\text{wf}} [\,]}\ [\text{WF-EMPY}] \qquad \frac{E \vdash t : T \qquad E, x : T \vdash_{\text{wf}} \theta'}{E \vdash_{\text{wf}} [x \mapsto t : T], \theta'}\ [\text{WF-EXT}]$$

Lemma 23 *If $E \models \sigma_E$ and $\sigma_E F \models \sigma_F$, where $dom(\sigma_E) = dom(E)$ and $dom(\sigma_F) = dom(F)$, then $E, F \models \sigma_E, \sigma_F$*

PROOF: Consider a variable $x \in dom(E, F)$. If $x \in dom(E)$ then the lemma is trivial. If $x \in dom(F)$, then $(\sigma_E \sigma_F)(x) = \sigma_F(x)$ which has type $\sigma_F(\sigma_E F)(x)$ by assumption, which is $(\sigma_E, \sigma_F)((E, F)(x))$. □

Lemma 24 *If $E \models \sigma$ and $E \vdash_{\text{wf}} \theta$ then $\sigma(env(\theta)) \models \sigma(\theta)$*

PROOF: Consider any $x \in dom(\theta) = dom(E)$. By assumption, $\vdash \sigma(x) : \sigma(E(x))$, but because $E \vdash_{\text{wf}} \theta$ we know (by a trivial induction) that $\sigma(E(x)) = \sigma(env(\theta)(x))$. □

Lemma 25 *Suppose ρ is a placeholder replacement such that $\rho(E) \vdash_{\text{wf}} \rho(\theta)$. Then ρ satisfies $E \vdash p \Rightarrow \theta \cdot \gamma$ if and only if ρ satisfies $E, env(\theta) \vdash [\![\theta]\!] \wedge p \Rightarrow \gamma$*

PROOF: Assume throughout that $\rho E \vdash_{\text{wf}} \rho\theta$

($\Rightarrow$) Suppose ρ satisfies $E \vdash p \Rightarrow \theta \cdot \gamma$, i.e. $\rho E \vdash \rho p \Rightarrow \rho\theta(\rho\gamma)$ and consider any σ s.t. $\rho E, env(\rho\theta) \models \sigma$ and $\sigma(\rho([\![\theta]\!] \wedge p)) \rightsquigarrow^* \texttt{true}$. Then by definition of the conjunction we know that $\sigma(\rho p) \rightsquigarrow^* \texttt{true}$, hence $\sigma(\rho(\theta \cdot \gamma)) \rightsquigarrow^* \texttt{true}$ by assumption, which is $\sigma(\rho\theta(\rho\gamma)) \rightsquigarrow^* \texttt{true}$.

But since σ is a closing substitution for $\rho(E, env(\theta))$, we know for any $x \in dom(\theta)$ that $\sigma(x) = (\rho\theta)(x)$ hence $\sigma(\rho\gamma) = \sigma(\rho\theta(\rho\gamma))$, which we already know evaluates to `true`. Hence ρ satisfies $E, env(\theta) \vdash [\![\theta]\!] \wedge p \Rightarrow \gamma$.

($\Leftarrow$): Suppose ρ satisfies $E, env(\theta) \vdash [\![\theta]\!] \wedge p \Rightarrow \gamma$ i.e. $\rho E, env(\rho\theta) \vdash \rho([\![\theta]\!] \wedge p) \Rightarrow \rho\gamma$ and consider any σ s.t. $\rho E \models \sigma$ and $\sigma(\rho p) \rightsquigarrow^*$ `true`. Let $\sigma' = \sigma \circ (\rho\theta)$; note that $\rho E, env(\rho\theta) \models \sigma'$ by Lemmas 23 and Lemma 24.

Obviously, $\sigma'(\rho p) \rightsquigarrow^*$ `true` since $dom(\theta) \cap fv(p) = \emptyset$. Furthermore, by intensional equality, we see that $\sigma'([\![\theta]\!]) \rightsquigarrow^*$ `true`. So the conjunction in our assumption evaluates to `true`, and we infer that $\sigma'(\rho\gamma) \rightsquigarrow^*$ `true`.

But $\sigma'(\rho\gamma) = \sigma(\rho\theta(\rho\gamma))$ which is exactly what we need to conclude that ρ satisfies $E \vdash p \Rightarrow \theta \cdot \gamma$ $\square$

3.4.3 Placeholder Solution

After the previous transformations, all lower bounds of a placeholder γ appear in constraints of the form

$$E_\gamma \vdash p_i \Rightarrow \gamma$$

for $i \in \{1..n\}$, assuming γ has n lower bounds. We want to set γ 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, γ 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 equation $\gamma = p_1 \vee \cdots \vee p_n$.

More formally, suppose $E_\gamma = x_1 : T_1, \cdots, x_k : T_k$. Then γ is a predicate over $x_1 \cdots x_k$ and we can interpret it as a function $\mathcal{F}_\gamma : T_1 \to \cdots \to T_k \to \texttt{Bool}$. We use the following notation for clarity:

$$\begin{aligned} \bar{T} \to \texttt{Bool} &\equiv T_1 \to T_2 \to \cdots \to T_k \to \texttt{Bool} \\ \lambda \bar{x}{:}\bar{T}.\, t &\equiv \lambda x_1{:}T_1.\, \lambda x_2{:}T_2.\, \cdots \lambda x_k{:}T_k.\, t \\ f\ \bar{x} &\equiv f\ x_1\ x_2\ \cdots\ x_k \end{aligned}$$

The function $\mathcal{F}_\gamma$ can then be defined as the following least fixed point computation:

$$\mathcal{F}_\gamma\ =\ \texttt{fix}_{\bar{T} \to \texttt{Bool}}\ (\lambda f{:}\bar{T} \to \texttt{Bool}.\, \lambda \bar{x}{:}\bar{T}.\, [\gamma \mapsto f\ \bar{x}](p_1 \vee \cdots \vee p_n))$$

Our solution for γ is $LB(\gamma) = \mathcal{F}_\gamma\ \bar{x}$. This is the strongest consequence that is implied by all lower bounds of γ and is in some sense canonical, analogously to the strongest postcondition of a code block.

Lemma 26 *If a placeholder replacement ρ satisfies P, then ρ satisfies $E_\gamma \vdash LB(\gamma) \Rightarrow \gamma$.*

Proof: Consider any ρ satisfying P and σ such that $\rho E_\gamma \models \sigma$ and $\sigma\rho(LB(\gamma)) \rightsquigarrow^*$ `true` minimal k. Since the possible nondeterministic evaluations of $\sigma\rho LB(\gamma)$ correspond to simultaneously evaluating all lower bounds for γ simultaneously, one of them must eventually evaluate to `true`. Then by the assumption that ρ satisfies P, we know that $\sigma\rho\gamma \rightsquigarrow^*$ `true`. □

The result of equisatisfiability follows from the fact that we have chosen the strongest possible solution for γ.

Lemma 27 *P is satisfiable if and only if $P[\gamma := LB(\gamma)]$ is satisfiable.*

PROOF: ($\Rightarrow$): Consider any $\rho : PlaceHolders \rightarrow Terms$ that satisfies P. By Lemma 26 if $\rho(\gamma) \Rightarrow p$ occurs in P, then $LB(\gamma) \Rightarrow \rho(\gamma) \Rightarrow p$; covariant occurences of γ in environments are analogous. If $p \Rightarrow \rho(\gamma)$ occurs in P, then $p \Rightarrow LB(\gamma)$ by construction of $LB(\gamma)$; contravariant occurences of types in environments do not affect satisfiability. $\square$

In our example, the only lower bound of γ_3 is γ_1 and the only lower bound of γ_2 is γ_3, so let us set $\gamma_3 := \gamma_1$ and $\gamma_2 := \gamma_3$ in order to discuss the more interesting solution for γ_1. The resulting unsatisfied constraints (simplified for clarity) are:

$$n : \texttt{Int} \ \vdash \ \exists w : \{n{:}\texttt{Int} \,|\, n = 0\}.\ (n > w) \ \Rightarrow \ \gamma_1$$

$$n : \texttt{Int} \ \vdash \quad \exists w : \{n{:}\texttt{Int} \,|\, n = 0\}.\ \exists y : \{n{:}\texttt{Int} \,|\, n > w\}.\ [x := y] \cdot \gamma_1 \Rightarrow \gamma_1$$

The exact text of $LB(\gamma_1)$ is too large to print here, but it is equivalent to $\exists w : \{n{:}\texttt{Int} \,|\, n = 0\}.\ (n > w)$ and thus equivalent to $(n > 0)$. The resulting explicitly typed program (simplified according to the previous sentence's discussion) is:

$$\texttt{let}\ id : (x{:}\{n{:}\texttt{Int} \,|\, n > 0\} \rightarrow \{n{:}\texttt{Int} \,|\, n > 0\}) = \lambda x{:}\{n{:}\texttt{Int} \,|\, n > 0\}.\, x\ \texttt{in}$$

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

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

$$id\ (id\ y)$$

3.5 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 28 *For any λ^H program t, one of the following occurs:*

1. *Type reconstruction fails, in which case t is untypeable, or*
2. *Type reconstruction returns a type replacement π such that t is typeable if and only if $\pi(t)$ is well typed.*

PROOF:

Case 1: Only shape reconstruction can fail. If it does, then by Lemma 21 the subtyping constraints are unsatisfiable. Then by Lemma 20, t is not typeable.

Case 2: Type reconstruction solved constraints that were faithful, by Lemma 20. Thus by Lemma 21 we have π and by Lemma 27 we have ρ such that $(\rho \circ \pi)(t)$ is typeable (well typed) if and only if t is typeable. □

3.6 Related Work

Freeman and Pfenning [1991] introduced *datasort refinements*, which express restrictions on the recursive structure of algebraic datatypes. Type reconstruction for the finite set of programmer-specified datasort refinements is decided by abstract interpretation. Hayashi [1993] and Denney [1998] explored various logics for refinement predicates, while Davies and Pfenning [2000], and Mandelbaum et al. [2003] combined

refinements with computational effects. All of these systems require type annotations, though many perform some manner of local type inference [Pierce and Turner 1998].

Xi and Pfenning [1999] developed Dependent ML, which uses dependent types along with *index types* to express invariants for complex data structures such as red-black trees. Dependent ML solves systems of linear inequalities to infer a restricted class of type indices. Dunfield [2002] combined index types and datasort refinements in a system with decidable type checking, but the programmer is required to provide sufficient type annotations to guide the type checking process.

In the system of Ou et al. [2004], a section of code may be dynamically typed in order to reduce the annotation burden of refinement types. For the static dependently typed portion of a program, they forbid recursive functions in refinement predicates to ensure decidability of type checking, and perform no type reconstruction.

Constraint-based type reconstruction for systems with subtyping is a tremendously broad topic, and we cannot fully review it here. The problem is studied in some generality by Mitchell [1983], 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 we draw inspiration from the works of Heintze [1992], Cousot and Cousot [1995], Fähndrich and Aiken [1996], and Flanagan

and Felleisen [1997].

The precondition/postcondition discipline for imperative programs dates back to the work of Floyd [1967], C. A. R. Hoare [1969], and Dijkstra [1976]. General refinement types apply similar ideas to functional, higher order, programs. Our transformation of predicates to infer refinements resembles and is inspired by Dijkstra's weakest precondition calculation but is most closely related to the related strongest postcondition defined by Back [1988]. Nanevski et al. [2006] have introduced another relationship between axiomatic semantics and type systems with their Hoare Type Theory, which adds pre- and postconditions to the types of effectful monadic computation.

Bibliography

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 213–227, 1989.

R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In

Proceedings of the ACM Symposium on Principles of Programming Languages, pages 163–173, 1994.

L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM International Conference on Functional Programming*, pages 239–250, 1998. ISBN 1-58113-024-4.

R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25 (6):593–624, 1988. ISSN 0001-5903. doi: http://dx.doi.org/10.1007/BF00291051.

D. Blei, C. Harrelson, R. Jhala, R. Majumdar, G. C. Necula, S. P. Rahul, W. Weimer, and D. Weitz. Vampyre. Information available from `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre/`, 2000.

V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, 1991.

L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications, 2003.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

L. Cardelli. Typechecking dependent types and subtypes. In *Lecture notes in computer science on Foundations of logic and functional programming*, pages 45–57, 1988a.

L. Cardelli. Phase distinctions in type theory. Manuscript, 1988b. URL `citeseer.ist.psu.edu/cardelli88phase.html`.

P. Cousot and R. Cousot. Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the International Conference on Functional Programming and Computer Architecture*, pages 170–181, 1995.

R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the ACM International Conference on Functional Programming*, pages 198–208, 2000.

E. Denney. Refinement types for specification. In *Proceedings of the IFIP International Conference on Programming Concepts and Methods*, volume 125, pages 148–166. Chapman & Hall, 1998. ISBN 0-412-83760-9.

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

J. Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, CMU School of Computer Science, Pittsburgh, Penn., 2002.

M. Escardo. Synthetic topology of data types and classical spaces. *Electronic Notes in Theoretical Computer Science*, 87:21–156, November 2004.

M. Fagan. *Soft Typing.* PhD thesis, Rice University, 1990.

M. Fähndrich and A. Aiken. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley, 1996.

R. B. Findler. *Behavioral Software Contracts*. PhD thesis, Rice University, 2002.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.

C. Flanagan. Hybrid type checking. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 245 – 256, 2006.

C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 235–248, 1997.

C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Finding bugs in the web of program invariants. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

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 Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium in Applied Mathematics: Mathematical Aspects of Computer Science*, pages 19–32, 1967.

T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

Y. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of the European Symposium on Programming*, pages 155–175, 1988.

P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A language manual for Sather 1.1, 1996.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming*, 2007.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 93–104, 2006.

S. Hayashi. Logic of refinement types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 157–172, 1993.

N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

M. Hoang and J. C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 176 – 185, 1995.

R. C. Holt and J. R. Cordy. The Turing programming language. *Communications of the ACM*, 31:1310–1424, 1988.

K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *European Symposium on Programming*, 2007.

M. Kölling and J. Rosenberg. Blue: Language specification, version 0.94, 1997.

G. T. Leavens and Y. Cheon. Design by contract with JML, 2005. avaiable at `http://www.cs.iastate.edu/~leavens/JML/`.

P. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 293 – 304, 1992.

D. Luckham. Programming with specifications. *Texts and Monographs in Computer Science*, 1990.

Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the ACM International Conference on Functional Programming*, pages 213–225, 2003.

B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

J. C. Mitchell. Coercion and type inference. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 175 – 185, 1983.

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, 1999.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the International Conference on Functional Programming*, pages 62–73, 2006.

G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 128–139, 2002.

R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. ISSN 1074-3227.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 252–265, 1998.

F. Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM International Conference on Functional Programming*, pages 122–133, 1996.

S. Thatte. Quasi-static typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 31(5):181–192, 1996. ISSN 0362-1340.

C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001. URL `citeseer.nj.nec.com/boyapati01parameterized.html`.

A. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 250–262, 1994.

H. Xi. Imperative programming with dependent types. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 214–227, 1999.