Type Reconstruction for General Refinement Types

Kenneth Knowles Cormac Flanagan

University of California, Santa Cruz

Abstract. *General refinement types* allow types to be refined by predicates written in a general-purpose programming language, and can express function pre- and postconditions and data structure invariants. In this setting, with expressive and possibly verbose types, type reconstruction is particularly valuable, yet typeability is undecidable because it subsumes type checking. Using a generalized notion of type reconstruction, we present the first type reconstruction algorithm for a type system with base types refined by abitrary program terms. Our algorithm is a typeability-*preserving* transformation and defers type checking to a subsequent phase. 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.

1 Introduction

A *refinement type*, such as $\{x{:}\texttt{Int} \mid x \geq 0\}$, describes the set of terms of type `Int` satisfying the *refinement predicate* $x \geq 0$. Refinement types [12] significantly extend the expressive power of traditional type systems and, when combined with dependent function types, can document expressive function pre- and postconditions, as well as data structure invariants.

In the language λ^H [9], refinement predicates are unrestricted boolean expressions, and so, for example, any computable set of integers can be described by a λ^H type. Type checking requires proving implications between refinement predicates, such as that the postcondition of one function implies the precondition of another. Since the language of refinement predicates is λ^H itself, implication is undecidable, and hence so is type checking.

Hybrid type checking [9] circumvents this decidability limitation by passing each implication to a theorem prover that tries to prove or refute the implication, but also may give up and return "maybe," resulting in an inserted run-time check. 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 [14].

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 [2]. 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 [9], 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.

The following section reviews the syntax, semantics, and type system of λ^H. Section 3 formalizes and discusses the type reconstruction problem, which we reduce to satisfiability of a set of subtyping constraints in Section 4. Sections 5 and 6 then explain how we solve these constraints via shape reconstruction followed by predicate inference. Section 7 states and proves correctness of the reconstruction algorithm. The remaining sections are dedicated to related work and concluding remarks.

Figure 1: Syntax and Semantics

$$
\begin{array}{rcl}
s,t \in Term & ::= & x \\
& | & c \\
& | & \lambda x{:}S.\, t \\
& | & (t\ t) \\
& | & \langle S \triangleright T \rangle\, t
\end{array}
\qquad
\begin{array}{rcl}
S,T \in Type & ::= & x{:}S \to T \ \mid\ \{x{:}B \mid t\} \\
B & ::= & \texttt{Bool} \ \mid\ \texttt{Int} \ \mid\ \cdots \\
E & ::= & \varnothing \ \mid\ E, x:T
\end{array}
$$

Evaluation $\boxed{s \longrightarrow t}$

$$
\begin{array}{rl}
(\lambda x{:}S.\, t)\ s \longrightarrow t[x := s] & \text{[E-}\beta\text{]} \\
c\ t \longrightarrow [\![c]\!](t) & \text{[E-Prim]} \\
\langle (x{:}S_1 \to S_2) \triangleright (x{:}T_1 \to T_2) \rangle\, t \longrightarrow \lambda x{:}T_1.\, \langle S_2 \triangleright T_2 \rangle\, (t\ (\langle T_1 \triangleright S_1 \rangle\, x)) & \text{[E-Cast-F]} \\
\langle \{x{:}B \mid s\} \triangleright \{x{:}B \mid t\} \rangle\ c \longrightarrow c \quad \text{if } t[x := c] \longrightarrow^* \texttt{true} & \text{[E-Cast-C]} \\
\mathcal{E}[s] \longrightarrow \mathcal{E}[t] \quad \text{if } s \longrightarrow t & \text{[E-Compat]}
\end{array}
$$

Contexts $\boxed{\mathcal{E}}$

$$\mathcal{E} \quad ::= \quad \bullet \mid \bullet\ t \mid t\ \bullet \mid \langle S \triangleright T \rangle\ \bullet$$

2 A Review of λ^H

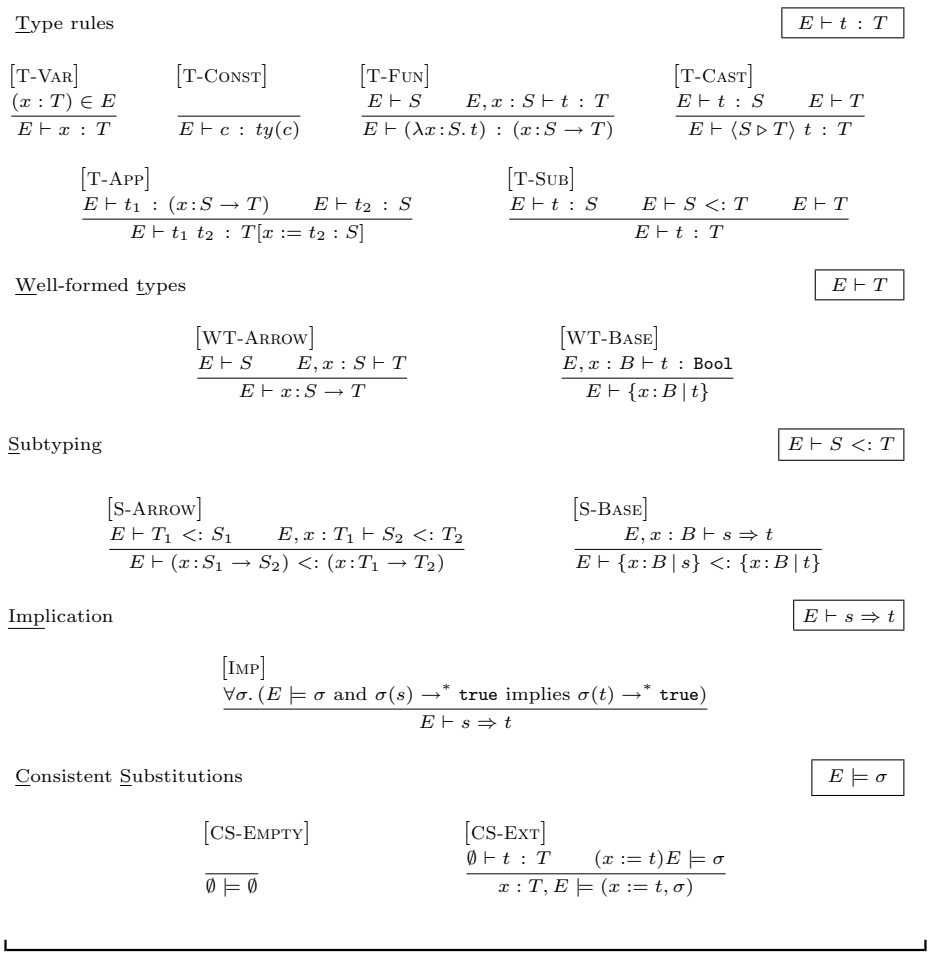
The language λ^H [9] is an extension of the lambda-calculus with dependent function types and refined base types; see Figure 1 for the complete syntax and operational semantics. In the dependent function type $x{:}S \to T$, the argument x is bound in the return type T. (This notation is preferred to the equivalent $\Pi x : S.T$). In a refined base type $\{x{:}B \mid t\}$, B is a base type such as `Int` or `Bool`, and t is a boolean predicate over x. Informally, $\{x{:}B \mid t\}$ is the subset of B for which the predicate t holds

Types have an operational interpretation via run-time casts: The term $\langle S \triangleright T \rangle\ t$ attempts to cast t from type S to type T. A cast to a refined base type is checked by evaluating the refinement predicate, while a cast to a function type is split into a delayed cast on the function's input and another on the function's output.

We assume some countable alphabet of constants c, each with an associated semantic function $[\![c]\!]$ that is applied when c is in the function position of an application. These constants include, for each type T, a fixpoint operator $\texttt{fix}_T$ that computes the least fixed point of a function $t : T \to T$, enabling unrestricted recursion:

$$[\![\texttt{fix}_T]\!](t) \quad = \quad t\ (\texttt{fix}_T\ t)$$

Figure 2: Type Rules



The typing rules for λ^H are reproduced in Figure 2. Each constant c is assigned a type $ty(c)$ by rule [T-Const]; the axioms on constants, detailed in [9], ensure that $ty(c)$ and $[\![c]\!]$ uphold type soundness. The type of a variable is extracted from the environment by rule [T-Var] and functions are assigned dependent function types by rule [T-Fun]. For an application $t_1\ t_2$, the rule [T-App] checks that t_1 has a dependent function type $x{:}S \to T$ and that t_2 has type S. The application is then assigned the type $T[x := t_2 : S]$, which is T with the concrete argument t_2 substituted for the argument variable x. The substitution is annotated with the argument type S to aid type reconstruction.

Typing of λ^H is based on subtyping, which utilizes an implication judgement between refinement predicates, rendering subtyping undecidable. The implication judgement $E \vdash s \Rightarrow t$ is defined by rule [Imp], which reads: term s implies

term t in environment E if, for any substitution σ on the variables bound in E that is consistent with the types of those variables in E, $\sigma(p) \longrightarrow^* \texttt{true}$ implies $\sigma(q) \longrightarrow^* \texttt{true}$. For example, $x : \texttt{Int} \vdash (x > 1) \Rightarrow (x > 0)$, because for any integer i chosen to substitute for x, whenever $(i > 1)$ evaluates to `true`, so does $(i > 0)$.

3 Type Reconstruction

For the type reconstruction problem, 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.

3.1 Delayed Substitutions

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).

$$\begin{array}{rcl} T & ::= & \cdots \mid \theta \cdot \alpha \\ \theta & ::= & [\,] \mid [x := t : T], \theta \end{array}$$

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

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

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

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

4 Constraint Generation

The constraint generation judgement $E \vdash t : T \;\&\; C$ is defined in Figure 3 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.

Figure 3: Constraint Generation Rules

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

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

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

$$\frac{E \vdash t : S' \;\&\; C_1 \qquad E \vdash T \;\&\; C_2}{E \vdash \langle S \triangleright T \rangle\ t : T \;\&\; C_1 \cup C_2 \cup \{E \vdash S' <: S\}}\;[\text{CG-Cast}]$$

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

$$\frac{E \vdash S \;\&\; C_1 \qquad E, x:S \vdash T \;\&\; C_2}{E \vdash x{:}S \to T \;\&\; C_1 \cup C_2}\;[\text{WTC-Arrow}] \qquad \frac{E, x:B \vdash t : \texttt{Bool} \;\&\; C}{E \vdash \{x{:}B \mid t\} \;\&\; C}\;[\text{WTC-Base}] \qquad \frac{}{E \vdash \theta \cdot \alpha \;\&\; \emptyset}\;[\text{WTC-Var}]$$

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.

Lemma 1. *For any environment E and term t:*

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

Proof outline: Each direction proceeds by induction on the respective derivation. (All complete proofs are included in the extended technical report [20].) $\square$

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

$$\begin{array}{l} \texttt{let}\ id : (x{:}\alpha_1 \to \alpha_2) = \lambda x{:}\alpha_3.\, x\ \texttt{in} \\ \texttt{let}\ w : \{n{:}\texttt{Int} \mid n = 0\} = 0\ \texttt{in} \\ \texttt{let}\ y : \{n{:}\texttt{Int} \mid n > w\} = 3\ \texttt{in} \\ id\ (id\ y) \end{array}$$

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

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

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

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

5 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 $\gamma \in \mathit{Placeholder}$ 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 $\rho : \mathit{Placeholder} \to \mathit{Term}$ and is lifted compatibly to all syntactic structures. As with type replacements, applying placeholder replacement allows any delayed substitutions also to be applied.

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

The shape reconstruction algorithm, detailed in figure 4 takes as input a subtyping constraint set C and processes the constraints in C nondeterministically according to the rules in Figure 4. 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 4 resembles a step of traditional type reconstruction. When a type variable α must have the shape of a function type, it is replaced by $x : \alpha_1 \to \alpha_2$, 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 – these occurences do not require a solution involving recursive types.

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

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

Figure 4: Shape Reconstruction Algorithm

Input: C
Output: π, P
Initially: $P = \emptyset$ and $\pi = [\,]$
match some constraint in C until quiescent:

$E \vdash \theta \cdot \alpha <: x{:}T_1 \to T_2$ or $E \vdash x{:}T_1 \to T_2 <: \theta \cdot \alpha$	$\Longrightarrow$	if $occurs(\alpha, x{:}T_1 \to T_2)$ then $fail$ otherwise for fresh α_1, α_2 $\pi := [\alpha := x{:}\alpha_1 \to \alpha_2] \circ \pi$ $C := \pi(C)$ $P := \pi(P)$
$E \vdash \theta \cdot \alpha <: \{x{:}B \mid t\}$ or $E \vdash \{x{:}B \mid t\} <: \theta \cdot \alpha$	$\Longrightarrow$	for fresh γ $\pi := [\alpha := \{x{:}B \mid \gamma\}] \circ \pi$ $C := \pi(C)$ $P := \pi(P)$
$E \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)$	$\Longrightarrow$	$C := C \cup \left\{ \begin{array}{l} E \vdash T_1 <: S_1, \\ E, x : T_1 \vdash S_2 <: T_2 \end{array} \right\}$
$E \vdash \{x{:}B \mid p\} <: \{x{:}B \mid q\}$	$\Longrightarrow$	$P := P \cup \{E \vdash p \Rightarrow q\}$
$E \vdash \{x{:}B \mid p\} <: x{:}S \to T$ or $E \vdash x{:}S \to T <: \{x{:}B \mid p\}$	$\Longrightarrow$	$fail$

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 2. *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 outline: Each step maintains the invariant that C is satisfiable if and only if $\exists \pi', \rho$ such that ρ satisfies P and $\rho \circ \pi' \circ \pi$ satisfies C. □

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 $T_y = \{n{:}\texttt{Int} \mid n > w\}$:

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

6 Satisfiability

The final phase of type reconstruction solves the residual implication constraint set P by finding a placeholder replacement that preserves satisfiability.

Our approach is based on the intuition that implications are essentially data-flow 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 data-flow 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.

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

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 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 5.

6.1 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:

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

Figure 5: Additional Evaluation Rules

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

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

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

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

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 3. *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 outline: The single-step evaluations of the existential term are in one-to-one correspondence with the possible values of $y : T$ in a closing substitution. □

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.

6.2 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 $[\![\theta]\!]$:

$$env([\,]) = \varnothing \qquad [\![\,[\,]\,]\!] = \texttt{true}$$
$$env([x := t : T], \theta) = x : T, env(\theta) \qquad [\![\,[x := t : T], \theta\,]\!] = (x = t) \wedge [\![\theta]\!]$$

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.

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

Lemma 4. *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 outline: The evaluations of the antecedents of each judgement can be mapped into the evaluations of the other. □

6.3 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:

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

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

Proof outline: For any σ such that $\rho(E_\gamma) \models \sigma$, the lemma follows by induction on the length of the reduction sequence of $\sigma(\rho(LB(\gamma))) \longrightarrow^* \texttt{true}$. □

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

Lemma 6. *P is satisfiable if and only if $P[\gamma := LB(\gamma)]$ is satisfiable.*

Proof outline: ($\Rightarrow$): Consider any $\rho : PlaceHolders \rightarrow Terms$ that satisfies P. By Lemma 5 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. □

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:

$$n : \texttt{Int} \vdash \exists w : \{n{:}\texttt{Int} \mid n = 0\}.\ (n > w) \Rightarrow \gamma_1$$
$$n : \texttt{Int} \vdash \exists w : \{n{:}\texttt{Int} \mid n = 0\}.\ \exists y : \{n{:}\texttt{Int} \mid n > w\}.\ [x := y] \cdot \gamma_1 \Rightarrow \gamma_1$$

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

$$\begin{array}{l}\texttt{let } id : (x{:}\{n{:}\texttt{Int} \mid n > 0\} \rightarrow \{n{:}\texttt{Int} \mid n > 0\}) = \lambda x{:}\{n{:}\texttt{Int} \mid n > 0\}.\, x \texttt{ in} \\ \texttt{let } w : \{n{:}\texttt{Int} \mid n = 0\} = 0 \texttt{ in} \\ \texttt{let } y : \{n{:}\texttt{Int} \mid n > w\} = 3 \texttt{ in} \\ id\ (id\ y)\end{array}$$

7 Correctness

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 1. *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 2 the subtyping constraints are unsatisfiable. Then by Lemma 1, t is not typeable.
Case 2: Type reconstruction solved constraints that were faithful, by Lemma 1. Thus by Lemma 2 we have π and by Lemma 6 we have ρ such that $(\rho \circ \pi)(t)$ is typeable (well-typed) if and only if t is typeable. □

8 Related Work

Freeman and Pfenning introduced *datasort refinements*, which express restrictions on the recursive structure of algebraic datatypes [12]. Type reconstruction for the finite set of programmer-specified datasort refinements is decided by abstract interpretation. Hayashi [15] and Denney [5] explored various logics for refinement predicates, while Davies and Pfenning [4], and Mandelbaum *et al* [23] combined refinements with computational effects. All of these systems require type annotations, though many perform some manner of local type inference [27].

Xi and Pfenning [29] 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 [7] 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.

Recently, Ou *et al* [26] developed a system with dependent types and refinement types where a section of code may be dynamically typed in order to reduce the annotation burden. 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 [19], Fuh and Mishra [13], Lincoln and Mitchell [22], Aiken and Wimmers [1], and Hoang and Mitchell [17]. Type inference systems parameterized by a subtyping constraint system are developed by Pottier [28] and Odersky *et al* [25]. This paper is complimentary 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 [16], Cousot and Cousot [3], Fähndrich and Aiken [8], and Flanagan and Felleisen [10].

The precondition/postcondition discipline for imperative programs dates back to the work of Floyd [11], Hoare [18], and Dijkstra [6]. 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 [2]. Nanevski *et al* [24] 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.

9 Conclusions and Future Work

Refinement type systems are a promising method for expressing precise program specifications, but many such specifications are not decidable at compile time. Hybrid type checking offers a practical strategy to enforce undecidable refinement types. This work demonstrates that while typeability for such systems is undecidable, a generalized notion of type reconstruction *is* decidable and resembles a natural application of specification techniques for imperative programs in a declarative context.

The connection with predicate transformations used in the analysis of imperative programs deserves further attention, and one clear avenue of future work is propagating information "backwards" as in a weakest precondition calculation, and combining this information with the information we propagate "forwards", in order to infer the least type for any term. We infer the strongest possible refinement predicates, but in the most precise type for a function, the contravariant domain has the *weakest* possible refinement.

Inferred refinement predicates may be large and unsuitable for use in error messages, much like the verification conditions of axiomatic semantics. Instead of simply presenting the user with a counterexample to the verification condition, ESC/Java illustrates each warning message with a partial trace of the program [21]; it may be possible to present similar traces for untypable programs.

References

1. 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.
2. R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.
3. 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.
4. R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the ACM International Conference on Functional Programming*, pages 198–208, 2000.
5. 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.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
7. J. Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, CMU School of Computer Science, Pittsburgh, Penn., 2002.
8. M. Fähndrich and A. Aiken. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley, 1996.
9. C. Flanagan. Hybrid type checking. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 245 – 256, 2006.

10. 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.
11. Floyd, R. W. Assigning meaning to programs. In *Proceedings of the Symposium in Applied Mathematics: Mathematical Aspects of Computer Science*, pages 19–32, 1967.
12. 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.
13. Fuh, Y. and P. Mishra. Type inference with subtypes. In *Proceedings of the European Symposium on Programming*, pages 155–175, 1988.
14. 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.
15. S. Hayashi. Logic of refinement types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 157–172, 1993.
16. N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.
17. 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.
18. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
19. John C. Mitchell. Coercion and type inference. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 175 – 185, 1983.
20. K. Knowles and C. Flanagan. Type reconstruction for general refinement types. `http://www.soe.ucsc.edu/~cormac/papers/htr-full.pdf`, 2007.
21. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.
22. 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.
23. 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.
24. 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.
25. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
26. 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.
27. 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.
28. F. Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM International Conference on Functional Programming*, pages 122–133, 1996.
29. 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.