Local Refinement Typing

BENJAMIN COSMAN, UC San Diego
RANJIT JHALA, UC San Diego

We introduce the Fusion algorithm for local refinement type inference, yielding a new SMT-based method for verifying programs with polymorphic data types and higher-order functions. Fusion is *concise* as the programmer need only write signatures for (externally exported) top-level functions and places with cyclic (recursive) dependencies, after which Fusion can *predictably* synthesize the *most precise* refinement types for all intermediate terms (expressible in the decidable refinement logic), thereby checking the program without false alarms. We have implemented Fusion and evaluated it on the benchmarks from the LiquidHaskell suite totalling about 12KLOC. Fusion checks an existing safety benchmark suite using about half as many templates as previously required and nearly 2× faster. In a new set of theorem proving benchmarks Fusion is both 10 – 50× faster and, by synthesizing the most precise types, avoids false alarms to make verification possible.

Additional Key Words and Phrases: Refinement Types, Program Logics, SMT, Verification

ACM Reference format:
Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *Proc. ACM Program. Lang.* 1, 1, Article 26 (September 2017), 31 pages.
https://doi.org/10.1145/3110270

1 INTRODUCTION

Refinement types are a generalization of Floyd-Hoare logics to the higher-order setting. Assertions are generalized to types by constraining basic types so that their inhabitants satisfy predicates drawn from SMT-decidable refinement logics. Assertion checking can then be generalized to a form of subtyping, where subtyping for basic types reduces to checking implications between their refinement predicates [Constable 1986; Rushby et al. 1998]. Thus, thanks to the SMT revolution [Barrett et al. 2016], refinement types have emerged as a modular and programmer-extensible means of expressing and verifying properties of polymorphic, higher order programs in languages like ML [Dunfield 2007; Rondon et al. 2008; Xi and Pfenning 1998], Haskell [Vazou et al. 2014b], Racket [Kent et al. 2016], $F^\sharp$ [Bengtson et al. 2008], and TypeScript [Vekris et al. 2016].

The Problem Unfortunately, the expressiveness and extensibility offered by predicate refinements comes at a price. Existing refinement type systems are either *not concise*, *i.e.* require many type annotations (not just for top-level functions), or *not complete*, *i.e.* programs fail to type check because the checker cannot synthesize suitable types for intermediate terms, or *not terminating*, *i.e.* the checker can diverge while trying to synthesize suitable types for intermediate terms in an iterative counterexample-guided manner. We show that the presence of logical predicates relating *multiple* program variables renders classical approaches like unification or set-constraint based subtyping – even when used in a *bidirectional* [Pierce and Turner 1998] or *local* fashion [Odersky

2475-1421/2017/9-ART26
https://doi.org/10.1145/3110270

et al. 2001] – not up to the task of inference. The problem is especially acute for programs that make ubiquitous use of polymorphic datatypes and higher-order combinators. Consequently, to use refinement types, the programmer must accept one of several distasteful choices: litter her code with type annotations (everywhere), or eschew polymorphic combinators (§ 2).

Fusion: Local Refinement Typing In this paper, we introduce the Fusion algorithm that permits local refinement type inference. Fusion is *concise* as the programmer need only write signatures for (externally exported) top-level functions, and places with cyclic dependencies on refinements, either explicit (*e.g.* recursive functions) or implicit (*e.g.* calls to inductive library functions like `fold`). Fusion then synthesizes the most *precise* refinement types for all intermediate terms that are expressible in the refinement logic, and hence checks the specified signatures. Finally, Fusion ensures that verification remains *decidable* by using the framework of Liquid Typing [Rondon et al. 2008] to synthesize refinements from user-defined templates ("qualifiers") in the presence of cyclic dependencies. Fusion reconciles concision, precision and decidability with three contributions.

(1) **From Programs to NNF Horn Constraints** Our first insight is that we can phrase the problem of refinement type checking as that of checking the satisfiability of (nested) Horn Clause Constraints in Negation Normal Form (NNF). The NNF constraints are *implications* between refinement variables denoting the unknown, to-be-synthesized refinements; a satisfying assignment for these constraints yields refinement types for all intermediate terms, that satisfy the subtyping obligations that must hold for the program to meet its specification. Crucially, our novel NNF constraint formulation retains the *scoping structure* of bindings that makes precise constraint solving practical (§ 4).

(2) **Solving Constraints via Elimination** Our second insight is that we can find a satisfying assignment for the NNF constraint, by systematically computing the *most precise* assignment for each refinement variable, and using it to *eliminate* the variable from the constraints. Unfortunately, we find that a direct application of the elimination algorithm, inspired by the classical notion of "unfolding" from the Logic Programming literature [Burstall and Darlington 1977; Tamaki and Sato 1984], leads to an exponential blowup even for simple programs that arise in practice. We show that such a blowup cannot be avoided *in general* as the problem of refinement type checking is ExpTime-hard. Fortunately, we show how to exploit the scoping structure present in NNF constraints to get compact solutions for real-world programs (§ 5).

(3) **Refinement Typing: Faster and without False Alarms** We have implemented Fusion and evaluated its effectiveness on all the benchmarks from the LiquidHaskell suite [Vazou et al. 2014b]. We show that in *existing* safety benchmarks where qualifiers *were* supplied or extracted from top-level type signatures, Fusion is able to synthesize the types about 2× faster and requires only about half the qualifiers (needed *e.g.* to synthesize types for non-top-level recursive functions). Further, in a *new* set of theorem proving benchmarks which make heavy use of polymorphic higher-order functions and data types, global template-based inference of LiquidHaskell is both prohibitively slow *and* unpredictable, failing with false alarms. In contrast, Fusion is more than 10× faster and, by synthesizing the most precise types, makes verification possible (§ 6).

2 OVERVIEW

We start with a high-level overview illustrating how refinement types can be used to verify programs, the problems that refinements pose for local inference, and our techniques for addressing them. We have picked obviously contrived examples that distill the problem down to its essence: for real-world programs we refer the reader to [Swamy et al. 2011; Vazou et al. 2014a].

```
inc  :: x:Int -> {v:Int | v = x + 1}   -- increment by one
dec  :: x:Int -> {v:Int | v = x - 1}   -- decrement by one
[]   :: List a                        -- list : "nil"
(:)  :: a -> List a -> List a         -- list : "cons"
last :: List a -> a                   -- last element of a list
map  :: (a -> b) -> List a -> List b  -- mapping over a list
(.)  :: (b -> c) -> (a -> b) -> a -> c -- function composition
```

Fig. 1. Type specifications for library functions

Refinement Types We can precisely describe *subsets* of values corresponding to a type by composing basic types with *logical predicates* that are satisfied by values inhabiting the type. For example, `Nat` describes the subset of `Int` comprising (non-negative) natural numbers:

```
type Nat = {v:Int | 0 <= v}
```

Verification Conditions A refinement type checker can use the above signature to check:

```
abs :: Int -> Nat
abs n | 0 <= n    = n
      | otherwise = 0 - n
```

by generating a *verification condition* (VC) [Nelson 1981]:

$$\begin{aligned} \forall n.\, 0 \leq n \Rightarrow \forall \nu.\, \nu = n \quad &\Rightarrow 0 \leq \nu \\ \wedge\, 0 \not\leq n \Rightarrow \forall \nu.\, \nu = 0 - n &\Rightarrow 0 \leq \nu \end{aligned}$$

whose *validity* can be checked by an SMT solver [Barrett et al. 2016; Nelson 1981] to ensure that the program meets the given specification.

2.1 The Problem

For simple, first-order programs like `abs`, VC generation is analogous to the classical Floyd-Hoare approach used *e.g.* by EscJava [Flanagan et al. 2002]. Namely, we have constraints *assuming* the path conditions (and pre-conditions), and *asserting* the post-conditions. Next, we illustrate how VC generation and checking is problematic in the presence of *local variables*, *collections*, and *polymorphic*, *higher order* functions. We do so with a series of small but idiomatic examples comprising "straight-line" code (no explicit looping or recursion), decorated with top-level specifications as required for local type inference. We show how the presence of the above features necessitates the synthesis of intermediate refinement types, which is beyond the scope of existing approaches.

Library Functions Our examples use a library of functions with types as shown in Fig. 1.

Example 1: Local Variables In `ex1` in Figure 2, we illustrate the problem posed by local binders that lack (refinement) type signatures. Here, we need to check that `inc y` is a `Nat` *i.e.* non-negative, assuming the input `x` is. To do so, we must synthesize a sufficiently strong type for the *local* binder `y` which says that its value is an *almost*-`Nat`, *i.e.* that `y` has the type `{y:Int | 0 ≤ y + 1}`.

Example 2: Collections In `ex2` in Figure 2 we show how the idiom of creating intermediate collections of values complicates refinement type checking. Here, we use a `Nat` to create a *list* `ys` whose `last` element is extracted and incremented, yielding a `Nat`. The challenge is to infer that the intermediate collection `ys` is a list of almost-`Nat`s.

Example 3: Higher Order Functions Function `ex3` shows the idiom of composing two functions with the `(.)` operator whose type is in Figure 1. To verify the type specification for `ex3` we need to synthesize suitable instantiations for the type variables `a`, `b` and `c`. Unlike with classical types, unification is insufficient, as while `a` and `c` may be instantiated to `Nat`, we need to infer that `b` must be instantiated with almost-`Nat`.

Example 4: Iteration Finally `ex4` shows how the higher-order `map` and `(.)` are used to idiomatically refactor transformations of collections in a "wholemeal" style [Hinze 2009]. This straight-line function (treating `map` as a library function, abstracted by its type signature) vexes existing refinement type systems [Knowles and Flanagan 2010; Swamy et al. 2011] that cannot infer that `map dec` (resp. `map inc`) transforms a `Nat` list (resp. almost-`Nat` list) into an almost-`Nat` list (resp. `Nat` list).

2.2 Existing Approaches

Existing tools use one of three approaches, each of which is stymied by programs like the above.

1. Existentials First, [Kent et al. 2016; Knowles and Flanagan 2009] shows how to use existentials to hide the types of local binders. In `ex1`, this yields

$$\mathtt{y} \;::\; \exists 0 \leq \mathtt{t}.\ \{\nu{:}\mathtt{Int} \mid \nu = \mathtt{t} - 1\}$$

As `inc y` has the type $\{\nu \mid \nu = \mathtt{y} + 1\}$ we get the VC:

$$\forall \mathtt{y}.\ (\exists \mathtt{t}.\ 0 \leq \mathtt{t} \wedge \mathtt{y} = \mathtt{t} - 1) \implies \forall \nu.\ \nu = \mathtt{y} + 1 \implies 0 \leq \nu$$

which is proved valid by an SMT solver. Sage [Knowles and Flanagan 2010], F* [Swamy et al. 2011] and Racket [Kent et al. 2016] use this method to check `ex1`. However, it is not known how to scale this method beyond local variable hiding, *i.e.* to account for collections, lambdas, higher-order functions *etc.*. Consequently, the above systems *fail to check* `ex2`, `ex3` and `ex4`.

2. Counterexample-Guided Refinement Second, as in Mochi [Unno et al. 2013] we can use *counterexample-guided abstraction-refinement* (CEGAR) to iteratively compute stronger refinements until the property is verified. This approach has two drawbacks. First, it is non-modular in that it requires *closed* programs – *e.g.* the source of library functions like `map` and `(.)` – in order to get counterexamples. Second, more importantly, it is limited to logical fragments like linear arithmetic where Craig Interpolation based "predicate discovery" is predictable, and is notoriously prone to divergence otherwise [Jhala and McMillan 2006]. Consequently, the method has only been applied to small but tricky programs comprising tens of lines, but not scaled to large real-world libraries spanning thousands of lines, and which typically require refinements over uninterpreted function symbols or modular arithmetic [Vazou et al. 2014a].

3. Abstract Interpretation Finally, the framework of Liquid Typing [Rondon et al. 2008] shows how to synthesize suitable refinement types via abstract interpretation. The key idea is to:

(1) ***Represent*** types as *templates* that use κ *variables* to represent unknown refinements.
(2) ***Generate*** subtyping constraints over the templates, that reduce to implications between the κ variables,
(3) ***Solve*** the implications over an *abstract domain*, *e.g.* the lattice of formulas generated by conjunctions of user-supplied atomic predicates, to find an assignment for the κ-variables that satisfies the constraints.

Unfortunately, this approach also has several limitations. First, the user must supply atomic predicates (or more generally, a suitable abstract domain) which should be superfluous in "straight line" code as in `ex1`, `ex2`, `ex3`, and `ex4`. Unfortunately, without the atomic predicate hints, Liquid Typing fails to check any of the above examples. Specifically, unless the user provides the qualifier or template $0 \leq \mathtt{v} + 1$ (which does not appear anywhere in the code or specifications), the system

```
ex1 :: Nat -> Nat
ex1 x =
  let y =
    let t = x
    in
      dec t
  in
    inc y
```

```
ex2 :: Nat -> Nat
ex2 x =
  let ys = let n  = dec x
               p  = inc x
               xs = n : []
           in p : xs
      y  = last ys
  in
    inc y
```

Fig. 2. Examples: (ex1) Local variables, (ex2) Collections

cannot synthesise the almost-Nat types for the various intermediate terms in the above examples. Consequently LiquidHaskell (which implements Liquid Typing) rejects safe (and checkable) programs, leaving the user with the unpleasant task of debugging false alarms. Second, in predicate abstraction, the so-called abstraction operator "α" makes *many* expensive SMT queries, which can slow down verification, or render it impossible when the specifications, and hence predicates, are complicated (§ 6).

2.3 Our Solution: Refinement Fusion

Next, we describe our Fusion algorithm for *local refinement typing* and show how it allows us to automatically, predictably and efficiently check refinement types in the presence of variable hiding, collections, lambdas and polymorphic, higher order functions. The key insight is two-fold. First, we present a novel reduction from type inference to the checking the satisfaction of a system of NNF Horn Constraints (over refinement variables) that preserve the scoping structure of bindings in the original source program. Second, we show how to check satisfaction of the NNF constraints by exploiting the scoping structure to compute the strongest possible solutions for the refinement variables. The above steps yield a method that is *concise*, *i.e.* requires no intermediate signatures, and *complete*, *i.e.* is guaranteed to check a program whenever suitable signatures exist, instead of failing with false alarms when suitable qualifiers are not provided. Finally, the method *terminates*, as the acyclic variables can be eliminated (by replacing them with their most precise solution), after which any remaining cyclic variables can be solved via abstract interpretation [Rondon et al. 2008].

Example 1: Local Variables. First, let us see how Fusion synthesizes types for local binders that lack (refinement) type signatures.

1. Templates Fusion starts by generating templates for terms whose type must be synthesized. From the input type of ex1 (Fig 2), the fact that t is bound to x, and the output type of dec we have:

$$\begin{aligned} \texttt{x} &:: \{\nu{:}\texttt{Int} \mid 0 \leq \nu\} \\ \texttt{t} &:: \{\nu{:}\texttt{Int} \mid \nu = \texttt{x}\} \\ \texttt{dec t} &:: \{\nu{:}\texttt{Int} \mid \nu = \texttt{t} - 1\} \end{aligned}$$

The term dec t is bound to y but as t goes out of scope we assign y a template

$$\texttt{y} :: \{\nu{:}\texttt{Int} \mid \kappa(\nu)\}$$

with a fresh κ denoting the (unknown) refinement for the expression. That is, y is a `Int` value ν that satisfies $\kappa(\nu)$. We will constrain κ to be a well-scoped super-type of its body `dec t`. Finally, the output of `ex1` gets assigned

$$\texttt{inc y} :: \{\nu\texttt{:Int} \mid \nu = \texttt{y} + 1\}$$

2. Constraints Next, Fusion generates constraints between the various refinements. At each application, the parameter must be a subtype of the input type; dually, in each function definition, the body must be a subtype of the output type, and at each let-bind the body must be a subtype of the whole let-in expression. For base types, subtyping is exactly implication between the refinement formulas [Constable 1986; Rushby et al. 1998]. For `ex1` we get two constraints. The first relates the body of the let-in with its template; the second the body of `ex1` with the specified output:

$$\forall \texttt{x}.\ 0 \leq \texttt{x} \implies \forall \nu.\ \nu = \texttt{x} - 1 \implies \kappa(\nu) \tag{1}$$

$$\wedge\ \forall \texttt{y}.\ \kappa(\texttt{y}) \implies \forall \nu.\ \nu = \texttt{y} + 1 \implies 0 \leq \nu \tag{2}$$

Fusion ensures that the constraints preserve the *scoping structure* of bindings. Each subtyping (*i.e.* implication) happens under a context where the program variables in scope are bound and required to satisfy the refinements from their previously computed templates. Further, shared binders are explicit in the NNF constraint. For example, in the NNF constraint above, the shared binder x in the source program is also shared across the two implications 1 and 2. This sharing crucially allows Fusion to compute the most precise solution.

3. Solution To *solve* the constraints we need to find a *well-formed assignment* mapping each κ-variable to a predicate over variables in scope wherever the κ-variable appears, such that the formula obtained by replacing the κ with its assignment is *valid*. If no such interpretation exists, then the constraints are *unsatisfiable* and the program is ill-typed.

Fusion computes an interpretation called the *strongest refinements* using a method inspired by the classical notion of "unfolding" from the Logic Programming literature [Burstall and Darlington 1977; Tamaki and Sato 1984]. In essence, if we have implications of the form: $P_i \implies \kappa(\nu)$ then we can assign κ to the disjunction $\kappa(\overline{x}) \doteq \vee_i P_i$ after taking care to existentially quantify variables. In our example, κ is a unary refinement predicate, and so we use (1) to obtain the assignment

$$\kappa(z) \doteq \exists x.\ 0 \leq x \wedge (\exists \nu.\ \nu = x - 1 \wedge \nu = z) \tag{3}$$

which simplifies to $0 \leq z + 1$. The computed assignment above is simply the (existentially quantified) *hypotheses* arising in the conjunction where $\kappa(\cdot)$ is the goal. The "simplification" done above and in the sequel is purely for *exposition*: to *compute* it we would have to resort to quantifier elimination procedures which are prohibitively expensive in practice, and hence avoided by Fusion.

It is easy to check that the above assignment renders the constraint valid (post-substitution). In particular, the solution (3) precisely captures the almost-`Nat` property which allows an SMT solver to prove (2) valid. While the substituted VC contains existential quantifiers, the SMT solver can handle the resulting validity query easily as the quantifiers appear *negatively*, *i.e.* in the antecedents of the implication. Consequently, they can be removed by a simple variable renaming ("skolemization") as the VC $(\exists x.P(x)) \implies Q$ is equivalent to the VC $P(z) \implies Q$, where z is a fresh variable name.

Thus, for local variables, Fusion synthesizes the same intermediate types as the existentials-based method of [Kent et al. 2016; Knowles and Flanagan 2009], and, unlike Liquid Types, is able to verify `ex1` without requiring *any* predicate templates. However, as we see next, Fusion generalizes the existentials based approach to handle collections, lambdas, and polymorphic, higher order functions.

Example 2: Collections. Next, let us see how Fusion precisely synthesizes types for intermediate collections without requiring user-defined qualifiers or annotations.

1. Templates From the output types of `inc` and `dec` we get

$$\begin{aligned} \mathtt{n} &:: \{\nu{:}\mathtt{Int} \mid \nu = \mathtt{x} - 1\} \\ \mathtt{p} &:: \{\nu{:}\mathtt{Int} \mid \nu = \mathtt{x} + 1\} \end{aligned}$$

The lists `xs` and `ys` are created by invoking the nil and cons constructors (Fig 1.) In the two cases, we respectively instantiate the polymorphic type variable `a` (in the constructors' signatures from Figure 1) with the (unknown) templates over fresh κ variables to get

$$\begin{aligned} \mathtt{xs} &:: \mathtt{List}\ \{\nu{:}\mathtt{Int} \mid \kappa_x(\nu)\} \\ \mathtt{ys} &:: \mathtt{List}\ \{\nu{:}\mathtt{Int} \mid \kappa_y(\nu)\} \end{aligned}$$

as the output of `last` has the same type its input list, we get:

$$\begin{aligned} \mathtt{y} &:: \{\nu{:}\mathtt{Int} \mid \kappa_y(\nu)\} \\ \mathtt{inc\ y} &:: \{\nu{:}\mathtt{Int} \mid \nu = \mathtt{y} + 1\} \end{aligned}$$

2. Constraints We get four implication constraints from `ex2`.

$$\begin{aligned} &\forall \mathtt{x}.\ 0 \leq \mathtt{x} \Rightarrow \\ &\quad \forall \mathtt{n}.\ \mathtt{n} = \mathtt{x} - 1 \Rightarrow \\ &\quad\quad \forall \mathtt{p}.\ \mathtt{p} = \mathtt{x} + 1 \Rightarrow \\ &\quad\quad\quad \wedge\ \forall \nu.\ \nu = \mathtt{n} \Rightarrow \kappa_x(\nu) && (4) \\ &\quad\quad\quad \wedge\ \forall \nu.\ \nu = \mathtt{p} \Rightarrow \kappa_y(\nu) && (5) \\ &\quad\quad\quad \wedge\ \forall \nu.\ \kappa_x(\nu) \Rightarrow \kappa_y(\nu) && (6) \\ &\wedge\ \forall \mathtt{y}.\ \kappa_y(\mathtt{y}) \Rightarrow \\ &\quad\quad \forall \nu.\ \nu = \mathtt{y} + 1 \Rightarrow 0 \leq \nu && (7) \end{aligned}$$

As `n` is passed into "cons" to get `xs`, we get that $\{\nu \mid \nu = \mathtt{n}\}$ must be a subtype of $\{\nu \mid \kappa_x(\nu)\}$, yielding (4). Similarly, as `p` is "cons"-ed to `xs` to get `ys`, we get that $\{\nu \mid \nu = \mathtt{p}\}$ and $\{\nu \mid \kappa_x(\nu)\}$ must be subtypes of $\{\nu \mid \kappa_y(\nu)\}$ yielding (5) and (6) respectively. Finally, the type of the return value `y+1` must be a subtype of `Nat` yielding (7). Each implication appears under a context in which the variables in scope are bound to satisfy their template's refinement.

3. Solution Again, we compute the strongest refinements as the (existentially quantified) disjunctions of the hypotheses under which each κ appears. Thus, implication (4) yields:

$$\begin{aligned} \kappa_x(z) \doteq\ &\exists \mathtt{x}.\ 0 \leq \mathtt{x} \wedge \\ &\exists \mathtt{n}.\ \mathtt{n} = \mathtt{x} - 1 \wedge \\ &\exists \mathtt{p}.\ \mathtt{p} = \mathtt{x} + 1 \wedge \\ &\exists \nu.\ \nu = n \ \wedge\ \nu = z \qquad \ldots \text{from (4)} \end{aligned}$$

```
ex3 :: Nat -> Nat
ex3 = let fn = \a -> dec a
          fp = \b -> inc b
      in  fp . fn
```

```
ex4 :: List Nat -> List Nat
ex4 = let fn = \a -> dec a
          fp = \b -> inc b
      in map fp . map fn
```

Fig. 3. Examples: (ex3) Higher-Order Composition, (ex4) Higher-Order Iteration.

which is essentially $0 \leq z + 1$, or almost-Nat, and the disjunction of (5, 6) yields

$$\begin{aligned}\kappa_y(z) \doteq \exists \mathsf{x}.\ & 0 \leq \mathsf{x} \wedge \\ & \exists \mathsf{n}.\ \mathsf{n} = \mathsf{x} - 1 \wedge \\ & \exists \mathsf{p}.\ \mathsf{p} = \mathsf{x} + 1 \wedge \\ & \quad \vee\ \exists \nu.\ \nu = \mathsf{p} \ \wedge\ \nu = z \ldots \text{from (5)} \\ & \quad \vee\ \exists \nu.\ \nu = \mathsf{n} \ \wedge\ \nu = z \ldots \text{from (6)}\end{aligned}$$

which is also $0 \leq z + 1$. Substituting the strongest refinements into (7) yields a valid formula, verifying ex2.

Example 3: Composition. Next, we describe how Fusion synthesizes precise types for inner lambda-terms like those bound to fn and fp, and simultaneously determines how the polymorphic type variables for the (.) combinator can be instantiated in order to verify ex3

1. Templates Fusion scales up to polymorphic higher-order operators like "compose" (.) by using the same approach as for collections: create κ-variables for the unknown instantiated refinements, and then find the strongest solution. In ex3, this process works by respectively instantiating the a, b and c in the signature for (.) (Fig 1) with fresh templates $\{\nu \mid \kappa_a(\nu)\}$, $\{\nu \mid \kappa_b(\nu)\}$, and $\{\nu \mid \kappa_c(\nu)\}$ respectively. Consequently, the arguments and output to . at this instance get the templates:

$$\mathtt{fn} :: \{a{:}\mathtt{Int} \mid \kappa_a(a)\} \rightarrow \{\nu{:}\mathtt{Int} \mid \kappa_b(\nu)\} \tag{8}$$

$$\mathtt{fp} :: \{b{:}\mathtt{Int} \mid \kappa_b(b)\} \rightarrow \{\nu{:}\mathtt{Int} \mid \kappa_c(\nu)\} \tag{9}$$

$$\mathtt{fp}\ .\ \mathtt{fn} :: \{a{:}\mathtt{Int} \mid \kappa_a(a)\} \rightarrow \{\nu{:}\mathtt{Int} \mid \kappa_c(\nu)\}$$

2. Constraints Next, the bodies of fn and fp must be subtypes of the above templates. By decomposing function subtyping into input- and output- subtyping, we get the following implications. We omit the trivial constraints on the input types; the above correspond to checking the output types in an environment assuming the stronger (super-) input type:

$$\forall a.\ \kappa_a(a) \ \Rightarrow\ \forall \nu.\ \nu = a - 1 \Rightarrow\ \kappa_b(\nu) \tag{10}$$

$$\wedge\quad \forall b.\ \kappa_b(b) \ \Rightarrow\ \forall \nu.\ \nu = b + 1 \Rightarrow\ \kappa_c(\nu) \tag{11}$$

Finally, fp . fn must be a subtype of the (function) type ascribed to ex3 yielding input and output constraints:

$$\wedge\quad \forall \nu.\ 0 \leq \nu \ \Rightarrow\ \kappa_a(\nu) \tag{12}$$

$$\wedge\quad \forall \nu.\ \kappa_c(\nu) \ \Rightarrow\ 0 \leq \nu \tag{13}$$

3. Solution Finally Fusion computes the strongest refinements by assigning κ_a to its single hypothesis:

$$\kappa_a(z) \ \doteq\ \exists \nu.\ 0 \leq \nu \wedge \nu = z$$

which simplifies to $0 \leq z$, which plugged into (10) gives:

$$\kappa_b(z) \doteq \exists a.\ 0 \leq a \wedge \exists v.\ v = a - 1 \wedge v = z$$

which simplifies to $0 \leq z + 1$, which in (11) yields

$$\kappa_c(z) \doteq \exists b.\ 0 \leq b + 1 \wedge \exists v.\ v = b + 1 \wedge v = z$$

which is just $0 \leq z$, rendering implication (13) valid.

Example 4: Iteration. Finally, Fusion uses types to scale up to higher-order iterators over collections. As in `ex3` we generate fresh templates for `fn` (8) and `fp` (9). When applied to `map` the above return as output the templates:

$$\texttt{map fn} :: \texttt{List}\ \{a \mid \kappa_a(a)\} \rightarrow \texttt{List}\ \{v \mid \kappa_b(v)\}$$
$$\texttt{map fp} :: \texttt{List}\ \{b \mid \kappa_b(b)\} \rightarrow \texttt{List}\ \{v \mid \kappa_c(v)\}$$

which are the templates from `ex3` *lifted* to lists, yielding

$$\texttt{map fp . map fn} :: \texttt{List}\ \{a \mid \kappa_a(a)\} \rightarrow \texttt{List}\ \{v \mid \kappa_c(v)\}$$

Consequently, the subtyping constraints for `ex4` are the same as for `ex3` but lifted to lists. Covariant list subtyping reduces those into the exact same set of implications as `ex3`. Thus, Fusion computes the same strongest refinements as in `ex3` and hence, verifies `ex4`.

2.4 Avoiding Exponential Blowup

The reader may be concerned that substituting the strongest solution may cause the formulas to expand, leading to an exponential blowup. Recall that, in our examples, we "simplified" the formulas before substituting to keep the exposition short. In general this is not possible as it requires expensive quantifier elimination. We show that the concern is justified in theory, and that a direct substitution strategy leads to a blowup even in practice. Fortunately, we show that a simple optimization that uses the scoping structure of bindings that we have carefully preserved in the NNF constraints allows us to avoid blowups in practice.

Local Refinement Typing is ExpTime-Hard The bad news is that local refinement typing in general, and thus, constraint solving in particular are both ExpTime-hard. This result follows from the result that reachability (*i.e.* safety) of non-recursive first order boolean programs is ExpTime-complete [Godefroid and Yannakakis 2013]. We can encode the reachability problem directly as checking a given refinement type signature over programs using just boolean valued data, and so local refinement typing is ExpTime-hard. Thus, deciding constraint satisfaction is also ExpTime-hard, which means that we cannot avoid exponential blowups in general. Propositional validity queries are in Np so ExpTime-hardness means we must make an exponential number of queries, or a polynomial number of exponential size queries.

Let-chains cause exponential blowup Indeed, the direct "unfolding" based approach [Burstall and Darlington 1977; Tamaki and Sato 1984] that we have seen so far blows up due to a simple and ubiquitous pattern: sequences of let-binders. Consider the constraint generated by the program:

```
exp :: Nat -> Nat
exp x0 = let x1 = id x0
             ⋮
             xn = id xn-1
         in xn
```

The i^{th} invocation of `id :: a -> a` creates a fresh κ_i, which is then used as the template for $\mathtt{x}_i$. Consequently, we get the NNF constraint:

$$\begin{aligned}
\forall \mathtt{x}_0.\ 0 \le \mathtt{x}_0 \ &\Rightarrow \forall \nu.\ \nu = \mathtt{x}_0 \ \Rightarrow\ \kappa_1(\nu) \\
\wedge\ \forall \mathtt{x}_1.\ \kappa_1(\mathtt{x}_1) \ &\Rightarrow \forall \nu.\ \nu = \mathtt{x}_1 \ \Rightarrow\ \kappa_2(\nu) \\
&\ \ \vdots \\
\wedge\ \forall \mathtt{x}_{n-1}.\ \kappa_{n-1}(\mathtt{x}_{n-1}) &\Rightarrow \forall \nu.\ \nu = \mathtt{x}_{n-1} \ \Rightarrow\ \kappa_n(\nu) \\
\wedge\ \forall \mathtt{x}_n.\ \kappa_n(\mathtt{x}_n) \ &\Rightarrow \forall \nu.\ \nu = \mathtt{x}_n \ \Rightarrow\ 0 \le \nu
\end{aligned} \tag{14}$$

Since each κ_i has in its hypotheses, all of $\kappa_1 \ldots \kappa_{i-1}$, the strongest solution for κ_i ends up with 2^i copies of $0 \le x_0$! While this example is contrived, it represents an idiomatic pattern: long sequences of let-binders (*e.g.* introduced by ANF-conversion) with polymorphic instantiation. Due to the ubiquity of the pattern, a direct unfolding based computation of the strongest refinement fails for all but the simplest programs.

Sharing makes Solutions Compact This story has a happy ending. Fortunately, our constraints preserve the scoping structure of binders. Thus, the exponentially duplicated $0 \le x_0$ is "in scope" everywhere each κ_i appears. Consequently, we can *omit* it entirely from the strongest solutions, collapsing each to the compact solution (15), $\kappa_i(z) \doteq z = x_{i-1}$. In § 4 we formalize our algorithm for generating such nested constraints, and in § 5 we show how to use the above insight to derive an optimized solving algorithm, which yields compact, shared solutions and hence, dramatically improves refinement type checking in practice § 6.

The Importance of Preserving Scope Note that the sharing observation and optimization seem obvious at this juncture, precisely because of our novel NNF formulation which carefully preserves the scoping structure of binders. Previous approaches for constraint based refinement synthesis [Hashimoto and Unno 2015; Knowles and Flanagan 2007; Polikarpova et al. 2016; Rondon et al. 2008] yield constraints that discard the scoping structure, yielding the flat (*i.e.* non-nested) constraints

$$\begin{aligned}
&\forall \mathtt{x}_{0,0}.\ 0 \le \mathtt{x}_{0,0} \Rightarrow \forall \nu.\ \nu = \mathtt{x}_{0,0} \ \Rightarrow\ \kappa_1(\nu) \\
\wedge\ &\forall \mathtt{x}_{1,0}.\ 0 \le \mathtt{x}_{1,0} \Rightarrow \forall \mathtt{x}_{1,1}.\ \kappa_1(\mathtt{x}_{1,1}) \Rightarrow \forall \nu.\ \nu = \mathtt{x}_{1,1} \ \Rightarrow\ \kappa_2(\nu) \\
&\ \ \vdots \\
\wedge\ &\forall \mathtt{x}_{n-1,0}.\ 0 \le \mathtt{x}_{n-1,0} \Rightarrow \ldots \Rightarrow \forall \mathtt{x}_{n-1,n-1}.\ \kappa_{n-1}(\mathtt{x}_{n-1,n-1}) \Rightarrow \forall \nu.\ \nu = \mathtt{x}_{n-1,n-1} \ \Rightarrow\ \kappa_n(\nu) \\
\wedge\ &\forall \mathtt{x}_{n,0}.\ 0 \le \mathtt{x}_{n,0} \Rightarrow \ldots \Rightarrow \forall \mathtt{x}_{n,n-1}.\ \kappa_{n-1}(\mathtt{x}_{n,n-1}) \Rightarrow \forall \mathtt{x}_{n,n}.\ \kappa_n(\mathtt{x}_{n,n}) \ \Rightarrow \forall \nu.\ \nu = \mathtt{x}_{n,n} \ \Rightarrow\ 0 \le \nu
\end{aligned}$$

in which the sharing is not explicit as the shared variables are alpha-renamed. Absent sharing, unfolding yields a solution for each κ_i that is exponential in i without any syntactic duplication of a common conjunct, rendering the computation of precise solutions infeasible.

2.5 Relatively Complete Local Refinement Typing

By *fusing* together the constraints appearing as hypotheses for each refinement variable κ, our approach synthesizes the most precise refinements expressible in the decidable refinement logic (Lemma 5.6), and hence, makes local refinement typing *relatively complete*. That is, (assuming no recursion) *if there exist* type ascriptions for intermediate binders and terms that allow a program to typecheck, then Fusion is guaranteed to find them. The above examples *do* in fact typecheck with existing systems but only if the user annotated local variables with their signatures. For example, [Swamy et al. 2011] *can* check `ex2` (Figure 2) *if* the programmer annotates the type of the local `xs` as an almost `Nat`. Unfortunately, such superfluous annotations impose significant overhead in code size as well as effort: the user must painstakingly debug false alarms to find where information was lost due to incompleteness. In contrast, our completeness guarantee means that once the user

has annotated top-level and recursive binders, Fusion *will not* return false alarms due to missing templates, *i.e.* will typecheck the program if and only if it can be, which crucially makes verification concise, precise and decidable, enabling the results in § 6.

Precise Refinements vs. Pre- and Post-Conditions As discussed in § 7, our notion of *most precise refinements* can be viewed as the analog of the method of strongest postconditions from Floyd-Hoare style program logics. F* introduces the notion of Dijkstra Monads [Swamy et al. 2013] as a way to generalize the dual notion of weakest preconditions (WP) to the higher-order setting. Hence, F* can use the notion of Dijkstra Monads to verify the following variant of `ex3` that uses Floyd-Hoare style specification instead of refinement types:

```
let inc x = x + 1
let dec x = x - 1
let compose f g x = f (g x)

let ex3 : i:int -> Pure int
  (requires (0 <= i)) (ensures (fun j -> 0 <= j))
  = compose inc dec
```

However, this approach has a key limitation relative to Fusion in that it requires an implementation or logically equivalent *strong* specification of the `compose` operator `(.)`, instead of the weaker polymorphic type signature required by Fusion. The WP machinery crucially relies upon the strong specification to makes the chaining of functions *explicit*. When the chaining is obscured, *e.g.* by the use of `map` in `ex4`, the WP method is insufficient, and so F* fails to verify `ex4`. In contrast, Fusion uses only the *implicit* subtyping dependencies between refinement types. Hence, the same compositional, type-directed machinery that checks `ex3` carries over to successfully verify `ex4`.

3 PROGRAMS

We start by formalizing the syntax and operational semantics of a core source language λ^R [Knowles and Flanagan 2010; Vazou et al. 2014b].

3.1 Syntax

Terms Figure 4 summarizes the syntax of the terms of λ^R. The *values* of λ^R comprise primitive constants `c` and functions $\lambda x\!:\!\tau.e$. The *terms* of λ^R include values and additionally, variables x, y, z ..., let-binders `let` $x = e$ `in` e, applications $e\ x$ (in ANF, to simplify the application rule [Rondon et al. 2008]). Polymorphism is accounted for via type abstraction $\Lambda\alpha.e$ and instantiation $[e]\ \tau$. We assume that the source program has been annotated with *unrefined* types at λ-abstractions and type abstraction and instantiations, either by the programmer or via classical (unrefined) type inference.

Types The types of λ^R include unrefined types, written τ, and refined types, written t. The unrefined types include basic types like `Int`, `Bool`, type variables α, functions $x\!:\!\tau \rightarrow \tau'$ where the input is bound to the name x, and type schemes $\forall\alpha.\tau$. We refine basic types with *refinement predicates* r to get refined types. The formulas r are drawn from a decidable refinement logic, *e.g.* QF_UFLIA: the quantifier free theory of linear arithmetic and uninterpreted functions.

Notation We write $\{v \mid r\}$ when the base b is clear from the context. We write b to abbreviate $\{v\!:\!b \mid \mathit{true}\}$, and write $t \rightarrow t'$ to abbreviate $x\!:\!t \rightarrow t'$ if x does not appear in t'.

Typing Constants We assume that each constant `c` is equipped with a type prim(c) that characterizes its semantics [Knowles and Flanagan 2010; Vazou et al. 2014b]. For example, literals are assigned their corresponding singleton type and operators have types representing their pre- and

Values	v	::=	c	constants
		|	$\lambda x{:}\tau.e$	function
Terms	e	::=	v	values
		|	$x, y, z, \ldots$	variables
		|	$\texttt{let}\ x = e\ \texttt{in}\ e'$	let-bind
		|	$e\ x$	application
		|	$\Lambda\alpha.e$	type-abstraction
		|	$[e]\ \tau$	type instantiation
Refinements	r	::=	... varies ...	
Basic Types	b	::=	$\texttt{Int} \mid \texttt{Bool} \mid \ldots$	
Types	t	::=	α	variable
		|	$\{x{:}b \mid r\}$	base
		|	$x{:}t \rightarrow t$	function
		|	$\forall\alpha.t$	scheme
Unrefined Types	τ	::=	α	variable
		|	b	base
		|	$x{:}\tau \rightarrow \tau$	function
		|	$\forall\alpha.\tau$	scheme

Fig. 4. Syntax of λ^R

post-conditions:

$$
\begin{aligned}
\texttt{prim(7)} &\doteq \{v{:}\texttt{Int} \mid v = 7\} \\
\texttt{prim(+)} &\doteq x{:}\texttt{Int} \rightarrow y{:}\texttt{Int} \rightarrow \{v \mid v = x + y\} \\
\texttt{prim(/)} &\doteq \texttt{Int} \rightarrow \{v{:}\texttt{Int} \mid v \neq 0\} \rightarrow \texttt{Int} \\
\texttt{prim(assert)} &\doteq \{v{:}\texttt{Bool} \mid v\} \rightarrow \texttt{Unit}
\end{aligned}
$$

3.2 Semantics

λ^R has a standard small-step, contextual *call-by-value* semantics; we write $e \hookrightarrow e'$ to denote that term e steps to e'. We write $e \hookrightarrow^j e'$ if there exists $e_1, \ldots, e_j$ such that $e \equiv e_1$, $e' \equiv e_j$, and for all $1 \leq i < j$, we have $e_i \hookrightarrow e_{i+1}$. We write $e \hookrightarrow^* e'$ if for some (finite) j we have $e \hookrightarrow^j e'$.

Constants We assume that when values are applied to a primitive constant, the expression is reduced to the output of the primitive constant operation in a single step. For example, consider +, the primitive `Int` addition operator. We assume that $[\![+]\!](n)$ equals $+_n$ where for all m, $[\![+_n]\!](m)$ equals the integer sum of n and m.

Safety We say a term e *is stuck* if there does not exist any e' such that $e \hookrightarrow e'$. We say that a term e *is safe*, if whenever $e \hookrightarrow^* e'$ then either e' is a value or e' is *not* stuck. We write safe(e) to denote that e is safe. Informally, we assume that the primitive operations (*e.g.* division) get stuck on values that do not satisfy their input refinements (*e.g.* when the divisor is `0`). Thus, e is safe when every (primitive) operation is invoked with values on which it is defined (*e.g.* there are no divisions by `0`.)

Constraint Syntax c

$$
\begin{array}{rrcl}
\textbf{\textit{Predicates}} & p & ::= & r \\
 & & \mid & \kappa(\overline{y}) \\
 & & \mid & p_1 \wedge p_2 \\
\textbf{\textit{Constraints}} & c & ::= & p \\
 & & \mid & c_1 \wedge c_2 \\
 & & \mid & \forall x{:}b.\ p \Rightarrow c \\
\textbf{\textit{Environments}} & \Gamma & ::= & \emptyset \mid \Gamma; x{:}t \\
\textbf{\textit{Assignment}} & \sigma & ::= & K \rightarrow R \\
\textbf{\textit{Assumption}} & G & ::= & \sigma \mid G, \{x{:}b \mid p\}
\end{array}
$$

Well-formedness $\Gamma \vdash c$

$$
\frac{\Gamma \vdash r : \texttt{Bool}}{\Gamma \vdash r}\ \text{W-Ref}
\qquad
\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 \wedge c_2}\ \text{W-And}
\qquad
\frac{\Gamma; x{:}b \vdash c \quad \Gamma; x{:}b \vdash r}{\Gamma \vdash \forall x{:}b.\ r \Rightarrow c}\ \text{W-Imp}
$$

Satisfaction $G \models c$

$$
\frac{\emptyset \vdash \sigma(c) \quad \mathsf{SmtValid}(\sigma(c))}{\sigma \models c}\ \text{Sat-Base}
\qquad
\frac{G \models \forall x{:}b.\ p \Rightarrow c}{G, \{x{:}b \mid p\} \models c}\ \text{Sat-Ext}
\qquad
\frac{G \models c \text{ for each } c \in cs}{G \models cs}\ \text{Sat-Many}
$$

Fig. 5. Constraints: Syntax and Semantics

4 CONSTRAINTS

Local refinement typing proceeds in two steps. First, we use λ^R terms to generate a system of constraints (§ 4.1) whose satisfiability (§ 4.2) implies that the term is safe (§ 4.4). Second, we solve the constraints to find a satisfying assignment (§ 5).

4.1 Syntax

Refinement Variables Figure 5 describes the syntax of constraints. A *refinement variable* represents an unknown n-ary relation, *i.e.* is an unknown refinement over the free variables $z_1, \ldots, z_n$ (abbreviated to $\overline{z}$). We refer to $\overline{z{:}\tau}$ as the *parameters* of a variable κ, written $\mathsf{params}(\kappa)$, and say that n is the *arity* of κ. We write K for the set of all refinement variables.

Predicates A *predicate* is a refinement r from a decidable logic, or a *refinement variable* application $\kappa(x_1, \ldots, x_n)$, or a conjunction of two sub-predicates. We assume each κ is applied to the same number of variables as its arity.

Constraints A constraint is a "tree" where each "leaf" is a *goal* and each "internal" node either (1) universally *quantifies* some variable x of a basic type b, subjecting it to satisfy a *hypothesis* p, *or* (2) *conjoins* two sub-constraints.

NNF Horn Clauses Our constraints are Horn Clauses in Negation Normal Form [Bjørner et al. 2015]. NNF constraints are a generalization of the "flat" Constraint Horn Clause (CHC) formulation used in Liquid Typing [Rondon et al. 2008]. Intuitively, each root-to-leaf path in the tree is a CHC that requires that the leaf's goal be implied by the internal nodes' hypotheses. The nesting structure of NNF permits the scope-based optimization that makes Fusion practical (§ 5.1).

4.2 Semantics

We describe the semantics of constraints by formalizing the notion of constraint satisfaction. Intuitively, a constraint is satisfiable if there is an interpretation or *assignment* of the κ-variables to concrete refinements r such that the resulting logical formula is *well-formed* and *valid*.

Well-formedness An *environment* Γ is a set of type bindings $x : \tau$. We write $\Gamma \vdash c$ to denote that c is *well-formed* under environment Γ. Figure 5 summarizes the rules for establishing well-formedness. We write $\Gamma \vdash r : \texttt{Bool}$ if r can be typed as a `Bool` using the standard (non-refined) rules, and use it to check individual refinements (W-Ref). A conjunction of sub-constraints is well-formed if each conjunct is well-formed (W-And). Finally, each implication is well formed if the hypothesis is well-formed and the consequent is well-formed under the extended environment (W-Imp). (Note that a constraint containing a κ-application is *not* well-formed.)

Assignments An *assignment* σ is a map from the set of refinement variables K to the set of refinements R. An assignment is *partial* if its range is predicates (containing κ-variables, not just refinements). The function $\sigma(\cdot)$ substitutes κ-variables in predicates p, constraints c, and sets of constraints C with their σ-assignments:

$$
\begin{array}{lcll}
\sigma(\kappa(\overline{y})) & \doteq & \sigma(\kappa)\,[\overline{y}/\overline{x}] & \text{where } \overline{x} = \mathsf{params}(\kappa) \\
\sigma(r) & \doteq & r & \\
\sigma(p \wedge p') & \doteq & \sigma(p) \wedge \sigma(p') & \\
\sigma(\forall x{:}b.\, p \Rightarrow c) & \doteq & \forall x{:}b.\, \sigma(p) \Rightarrow \sigma(c) & \\
\sigma(c_1 \wedge c_2) & \doteq & \sigma(c_1) \wedge \sigma(c_2) & \\
\sigma(C) & \doteq & \{\sigma(c) \mid c \in C\} &
\end{array}
$$

Satisfaction A *verification condition* (VC) is a constraint c that has no κ applications; *i.e.* $\mathsf{kvars}(c) = \emptyset$. We say that an assignment σ *satisfies* constraint c, written $\sigma \models c$, if $\sigma(c)$ is a VC that is: (a) *well-formed*, and (b) *logically valid* [Nelson 1981]. Figure 5 formalizes the notion of satisfaction and lifts it to sets of constraints. We say that c *is satisfiable* if *some* assignment σ satisfies c. The following lemmas follow from the definition of satisfaction and validity:

Lemma 4.1 (**Weaken**). *If* $G \models p$ *then* $G \models p \vee p'$.

Lemma 4.2 (**Strengthen**). *If* $\sigma \models \forall x{:}b.\, p \Rightarrow c$ *and* $\sigma \models \forall x{:}b.\, p' \Rightarrow p$ *then* $\sigma \models \forall x{:}b.\, p' \Rightarrow c$.

Composing Assignments For two assignments σ and σ' we write $\sigma{\cdot}\sigma'$ to denote the assignment:

$$(\sigma{\cdot}\sigma')(\kappa) \doteq \sigma(\sigma'(\kappa))$$

The following Lemmas, proved by induction on the structure of c, characterize the relationship between assignment composition and satisfaction.

Lemma 4.3 (**Composition-Preserves-Sat**). *If* $\sigma \models c$ *then* $\sigma' {\cdot} \sigma \models c$.

Lemma 4.4 (**Composition-Validity**). $\sigma{\cdot}\sigma' \models c$ *if and only if* $\sigma \models \sigma'(c)$.

4.3 Dependencies and Decomposition

Next, we describe various properties of constraints that we will exploit to determine satisfiability.

Flat Constraints A *flat* or non-nested constraint c is of the form:

$$\forall x_1{:}b_1.\, p_1 \Rightarrow \ldots \Rightarrow \forall x_n{:}b_n.\, p_n \Rightarrow p$$

By induction on the structure of c we can show that the procedure $\mathsf{flat}(c)$, shown in Figure 12, returns a set of flat constraints that are satisfiable exactly when c is satisfiable.

Lemma 4.5 (**Flattening**). $\sigma \models c$ *if and only if* $\sigma \models \mathsf{flat}(c)$.

Heads and Bodies Let $c \equiv \forall x_1 : b_1.\ p_1 \Rightarrow \ldots \Rightarrow \forall x_n : b_n.\ p_n \Rightarrow p$ be a flat constraint. We write $\mathsf{head}(c)$ for the predicate p that is the right-most "consequent" of c. We write $\mathsf{body}(c)$ for the set of predicates $p_1, \ldots, p_n$ that are the "antecedents" of c.

Dependencies Let $\mathsf{kvars}(c)$ denote the the set of κ-variables in a constraint c. We can define the following sets of *dependencies*:

$$\begin{aligned}
\mathsf{deps}(\sigma) &\doteq \{(\kappa, \kappa') \mid \kappa \in \sigma(\kappa')\} && \text{(of a } \textit{solution } \sigma\text{)} \\
\mathsf{deps}(c) &\doteq \{(\kappa, \kappa') \mid \kappa \in \mathsf{body}(c), \kappa' \in \mathsf{head}(c)\} && \text{(of a } \textit{flat constraint } c\text{)} \\
\mathsf{deps}(c) &\doteq \cup_{c' \in \mathsf{flat}(c)} \mathsf{deps}(c') && \text{(of an } \textit{NNF constraint } c\text{)} \\
\mathsf{deps}(\hat{K},\ c) &\doteq \mathsf{deps}(c) \setminus (K \times \hat{K} \cup \hat{K} \times K) && \text{(of a constraint } \textit{excluding } \hat{K}\text{)}
\end{aligned}$$

We write $\mathsf{deps}^*(c)$ (resp. $\mathsf{deps}^*(\hat{K},\ c)$) for the reflexive and transitive closure of $\mathsf{deps}(c)$ (resp. $\mathsf{deps}(\hat{K},\ c)$). Intuitively, the dependencies of a constraint comprise the set of pairs (κ, κ') where κ appears in a body (hypothesis) for a clause where κ' is in the head (goal).

Cuts and Cycles A set of variables $\hat{K}$ *cuts* c if the relation $\mathsf{deps}^*(\hat{K},\ c)$ is *acyclic*. A set of variables K' is *acyclic* in c if $K - K'$ cuts c. Intuitively, a set K' is acyclic in c if the closure of dependencies *restricted to* K' (*i.e.* excluding variables not in K') has no cycles. A single variable κ is *acyclic* in c if $\{\kappa\}$ is acyclic in c. Intuitively, a single variable κ is acyclic in c if there is no clause in c where κ appears in both the head and the body. A constraint is *acyclic* if the set of *all* K is acyclic in c, *i.e.* if $\mathsf{deps}^*(c)$ is acyclic. In the sequel, for clarity of exposition we assume the invariant that whenever relevant, κ is acyclic in c. (Of course, our algorithm maintains this property, as described in § 5.5.)

Definitions and Uses The *definitions* (resp. *uses*) of κ in c, written $c{\downarrow}\kappa$ (resp. $c{\uparrow}\kappa$), are the subset of $\mathsf{flat}(c)$ such that the head contains (resp. does not contain) κ:

$$\begin{aligned}
c{\downarrow}\kappa &\doteq \{c' \mid c' \in \mathsf{flat}(c),\ \kappa \in \mathsf{head}(c')\} && \text{(definitions)} \\
c{\uparrow}\kappa &\doteq \{c' \mid c' \in \mathsf{flat}(c),\ \kappa \notin \mathsf{head}(c')\} && \text{(uses)}
\end{aligned}$$

By induction on the structure of c we can check that:

Lemma 4.6 (**Flatten-Definitions**). $\forall x.\ p \Rightarrow c{\downarrow}\kappa \equiv (\forall x.\ p \Rightarrow c){\downarrow}\kappa$

Since the definitions and uses of a κ partition $\mathsf{flat}(c)$, we can show that σ satisfies c if it satisfies the definitions and uses of κ in c:

Lemma 4.7 (**Partition**). *If κ is acyclic in c then $\sigma \models c$ if and only if $\sigma \models c{\downarrow}\kappa$ and $\sigma \models c{\uparrow}\kappa$.*

4.4 Generation

Next, we show how to map a term e into a constraint c whose satisfiability implies the safety of e.

Shapes (*shape*) Procedure shape, elided for brevity, takes as input a refined type t and returns as output the non-refined version obtained by erasing the refinements from t.

Templates (*fresh*) Procedure $\mathsf{fresh}(\Gamma,\ \tau)$ (Figure 6) takes as input an environment and a non-refined type τ, and returns a template whose shape is τ. Recall that each refinement variable denotes an n-ary relation. For example, a refinement variable that appears in the *output* type of a function can refer to the *input* parameter, *i.e.* can be a *binary* relation between the input and the output value. We track the allowed parameters by using the environment Γ. In the base case, fresh generates a fresh κ whose parameters correspond to the binders of Γ. In the function type case, fresh recursively generates templates for the input and output refinements, extending the environment for the output with the input binder.

fresh	:	$(\Gamma \times T) \to \hat{T}$
fresh($[y_1 : \tau_1, \ldots], b$)	$\doteq$	$\{v : b \mid \kappa(y_1, \ldots, v)\}$
where		
κ	=	fresh K variable
s.t. params(κ)	=	$[z_1 : \tau_1, \ldots, z : b]$
fresh($\Gamma,\ x : \tau \to \tau'$)	$\doteq$	$x : t \to t'$
where		
t	=	fresh($\Gamma,\ \tau$)
t'	=	fresh($\Gamma; x : \tau,\ \tau'$)
fresh($\Gamma,\ \alpha$)	$\doteq$	α
fresh($\Gamma,\ \forall \alpha.\tau$)	$\doteq$	$\forall \alpha.$fresh($\Gamma,\ \tau$)

Fig. 6. Fresh Templates: $\hat{T}$ denotes *templates*, *i.e.* types with κ-variables representing unknown refinements.

sub	:	$(T \times T) \to C$
sub($\{x : b \mid p\},\ \{y : b \mid q\}$)	$\doteq$	$\forall x : b.\ p \Rightarrow q[x/y]$
sub($x : s \to s',\ y : t \to t'$)	$\doteq$	$c \wedge ((y :: t) \Rightarrow c')$
where		
c	=	sub($t,\ s$)
c'	=	sub($s'[y/x],\ t'$)
sub($\alpha,\ \alpha$)	$\doteq$	*true*
sub($\forall \alpha.t,\ \forall \alpha.t'$)	$\doteq$	sub($t,\ t'$)

Fig. 7. Subtyping Constraints

Subtyping (sub***)*** Procedure sub($t,\ t'$) (Figure 7) returns the constraint c that must be satisfied for t to be a subtype of t', which intuitively means, the set of values denoted by t to be subsumed by those denoted by t'. (See [Knowles and Flanagan 2010; Vazou et al. 2014b] for details.) In the base case, the sub-type's predicate must imply the super-type's predicate. In the function case, we conjoin the contra-variant input constraint and the co-variant output constraint. For the latter, we additionally add a hypothesis assuming the stronger input type, using a *generalized implication* that drops binders with non-basic types as they are not supported by the first-order refinement logic:

$$\begin{array}{lcl} (x :: \{y : b \mid p\}) \Rightarrow c & \doteq & \forall x : b.\ p[x/y] \Rightarrow c \\ (x :: t) \Rightarrow c & \doteq & c \end{array}$$

Generation (cgen***)*** Finally, procedure cgen($\Gamma,\ e$) (Figure 8) takes as input an environment Γ and term e and returns as output a pair (c, t) where e typechecks if c holds, and has the (refinement) template t. The procedure computes the template and constraint by a syntax-directed traversal of the input term. The key idea is two-fold. First: generate a fresh template for each position where the (refinement) types cannot be directly synthesized – namely, function inputs, let-binders, and

polymorphic instantiation sites. Second: generate subtyping constraints at each position where values flow from one position into another.

- ***Constants and Variables*** yield the trivial constraint *true*, and templates corresponding to their primitive types, or environment bindings respectively. (The helper single strengthens the refinement to state the value equals the given variable, dubbed "selfification" by [Ou et al. 2004], enabling path-sensitivity [Knowles and Flanagan 2010; Rondon et al. 2008].)
- ***Let-binders*** (`let` x `=` e_1 `in` e_2) yield the conjunction of three sub-constraints. First, we get a constraint c_1 and template t_1 from e_1. Second, we get a constraint c_2 and template t_2 from e_2 checked under the environment extended with t_1 for x. Third, to hide x which may appear in t_2, we generate a fresh template $\hat{t}$ with the shape of t_2 and use sub to constrain t_2 to be a subtype of $\hat{t}$. The well-formedness requirement ensures $\hat{t}$, which is returned as the expression's template, is assigned a well-scoped refinement type.
- ***Functions*** ($\lambda x\!:\!\tau.e$) yield a template where the input is assigned a fresh template of shape τ, and the output is the template for e.
- ***Applications*** ($e\ y$) yield the output template with the formal x substituted for the actual y. (ANF ensures the substitution does not introduce arbitrary expressions into the refinements.) Furthermore, the input value y is constrained to be a subtype of the function's input type.
- ***Type-Instantiation*** ($e[\tau]$) is handled by generating a fresh template whose shape is that of the polymorphic instance τ, and substituting that in place of the type variable α in the type (scheme) obtained for e.

Soundness of Constraint Generation We can prove by induction on the structure of e that the NNF constraints generated by cgen capture the refinement type checking obligations of [Knowles and Flanagan 2010; Vazou et al. 2014b]. Let $\text{cgen}(e)$ denote the constraint returned by $\text{cgen}(\emptyset,\ e)$. We prove that if $\text{cgen}(e)$ is satisfiable, then $\emptyset \vdash e : t$ for some refinement type t. This combined with the soundness of refinement typing (Theorem 1, [Vazou et al. 2014b]) yields:

Theorem 4.8. *If* $\text{cgen}(e)$ *is satisfiable then* e *is safe.*

5 ALGORITHM

Next, we describe our algorithm for solving the constraints generated by cgen, *i.e.* for for finding a satisfying assignment – and hence inferring suitable refinement types. Our algorithm is an instance of the classical "unfolding" method for logic programs [Burstall and Darlington 1977; Pettorossi and Proietti 1994; Tamaki and Sato 1984]. We show how to apply unfolding to the NNF constraints derived from refinement typing, in a *scoped* fashion that avoids the blowup caused by long let-chains. First, we describe a procedure for computing the *scope* of a κ variable (§ 5.1). Second, we show how to use the scope to compute the *strongest solution* for a single κ variable (§ 5.2). Third, we describe how we can *eliminate* a κ by replacing it with its strongest solution (§ 5.3). Fourth, we show how the above procedure can be repeated when the constraints are acyclic (§ 5.4). Finally, we describe how to use our method when the constraints have cycles (§ 5.5).

5.1 The Scope of a Variable

Procedure $\text{scope}(\kappa, c)$ (Fig. 9) returns a sub-constraint of c of the form: $\forall(\overline{x_i\!:\!p_i}) \Rightarrow c'$ such that (1) κ does not occur in any p_i, and (2) *all* occurrences of κ in c occur in c'. As occurrences of κ in c occur under $\overline{x_i\!:\!p_i}$, we can omit these binders from the solution of κ, thereby avoiding the blowup from unfolding let-chains described in § 2.4. Restricting unfolding to $\text{scope}(\kappa, c)$ does not affect satisfiability. The omitted hypotheses are redundant as they are already present at all *uses* of κ. This intuition is captured by the following lemmas that follow by induction on c.

cgen	:	$(\Gamma \times E) \to (C \times T)$
cgen(Γ, c)	$\doteq$	$(\mathit{true},\ \mathsf{prim}(c))$
cgen(Γ, x)	$\doteq$	$(\mathit{true},\ \mathsf{single}(\Gamma,\ x))$
cgen(Γ, `let` $x = e_1$ `in` e_2)	$\doteq$	$(c \wedge \mathsf{sub}(t_2,\ \hat{t}),\ \hat{t})$
where		
c	$=$	$c_1 \wedge ((x :: t_1) \Rightarrow c_2)$
$\hat{t}$	$=$	$\mathsf{fresh}(\emptyset,\ \mathsf{shape}(t_2))$
(c_1, t_1)	$=$	$\mathsf{cgen}(\Gamma,\ e_1)$
(c_2, t_2)	$=$	$\mathsf{cgen}(\Gamma; x{:}t_1,\ e_2)$
cgen(Γ, $\lambda x{:}\tau.e$)	$\doteq$	$(c, x{:}\hat{t} \to t)$
where		
(c, t)	$=$	$\mathsf{cgen}(\Gamma; x{:}\hat{t},\ e)$
$\hat{t}$	$=$	$\mathsf{fresh}(\emptyset,\ \tau)$
cgen(Γ, $e\ y$)	$\doteq$	$(c \wedge c_y, t'\,[y/x])$
where		
c_y	$=$	$\mathsf{sub}(\mathsf{single}(\Gamma,\ y),\ t)$
$(c, x{:}t \to t')$	$=$	$\mathsf{cgen}(\Gamma,\ e)$
cgen(Γ, $\Lambda\alpha.e$)	$\doteq$	$(c, \forall\alpha.t)$
where		
(c, t)	$=$	$\mathsf{cgen}(\Gamma; \alpha,\ e)$
cgen(Γ, $e[\tau]$)	$\doteq$	$(c, t\,[\hat{t}/\alpha])$
where		
$(c, \forall\alpha.t)$	$=$	$\mathsf{cgen}(\Gamma,\ e)$
$\hat{t}$	$=$	$\mathsf{fresh}(\emptyset,\ \tau)$

single	:	$\Gamma \times X \to T$
single(Γ, x)		
$\mid\ t = \{x{:}b \mid p\}$	$\doteq$	$\{v{:}b \mid p\,[v/x] \wedge v = x\}$
$\mid$ otherwise	$\doteq$	t
where t	$=$	$\Gamma(x)$

Fig. 8. Constraint Generation

Lemma 5.1 (**Scoped-Definitions**). $c{\downarrow}\kappa \equiv \mathsf{scope}(\kappa, c){\downarrow}\kappa$

Lemma 5.2 (**Scoped-Uses**). $c{\uparrow}\kappa \equiv \mathsf{scope}(\kappa, c){\uparrow}\kappa \cup C$ *for some flat constraints* C *such that* $\kappa \notin C$.

Lemma 5.3 (**Scoped-Satisfaction**). *If* $\sigma \models c$ *then* $\sigma \models \mathsf{scope}(\kappa, c)$.

5.2 The Strongest Solution

Procedure sol1(κ, c) (Fig. 10) returns a predicate that is guaranteed to satisfy all the clauses where κ appears as the head. The procedure recursively traverses c to compute the returned predicate as the *disjunction* of those bodies in c where κ appears as the head. When the head does not contain κ, the output predicate is the empty disjunction, *false*.

scope	:	$(K \times C) \rightarrow C$
$\text{scope}(\kappa, c_1 \wedge c_2)$		
$\quad \mid \kappa \in c_1, \kappa \notin c_2$	$\doteq$	$\text{scope}(\kappa, c_1)$
$\quad \mid \kappa \notin c_1, \kappa \in c_2$	$\doteq$	$\text{scope}(\kappa, c_2)$
$\text{scope}(\kappa, \forall x\!:\!b.\ p \Rightarrow c')$		
$\quad \mid \kappa \notin p$	$\doteq$	$\forall x\!:\!b.\ p \Rightarrow \text{scope}(\kappa, c')$
$\text{scope}(\kappa, c)$	$\doteq$	c

Fig. 9. The scope of a κ in constraint c.

sol1	:	$(K \times C) \rightarrow P$
$\text{sol1}(\kappa, c_1 \wedge c_2)$	$\doteq$	$\text{sol1}(\kappa, c_1) \vee \text{sol1}(\kappa, c_2)$
$\text{sol1}(\kappa, \forall x\!:\!b.\ p \Rightarrow c)$	$\doteq$	$\exists x\!:\!b.\ p \wedge \text{sol1}(\kappa, c)$
$\text{sol1}(\kappa, \kappa(\overline{y}))$	$\doteq$	$\bigwedge_i x_i = y_i \quad \text{where } \overline{x} = \text{params}(\kappa)$
$\text{sol1}(\kappa, p)$	$\doteq$	false

Fig. 10. The Strongest Solution for a κ-Variable.

The Strongest Solution The *strongest* (resp. *scoped*) solution for κ in c, written σ^c_κ (resp. $\hat{\sigma}^c_\kappa$) is:

$$\sigma^c_\kappa \doteq [\kappa \mapsto \lambda \overline{x}.\text{sol1}(\kappa, c)]$$
$$\hat{\sigma}^c_\kappa \doteq [\kappa \mapsto \lambda \overline{x}.\text{sol1}(\kappa, c')] \text{ where } \text{scope}(\kappa, c) \equiv \forall(\overline{x_i\!:\!p_i}) \Rightarrow c' \text{s.t. } \kappa \notin p_i$$

The name "strongest solution" is justified by the following theorems. First, we prove that σ^c_κ satisfies the definitions of κ in c.

Lemma 5.4 (**Strongest-Sat-Definitions**). $\sigma^c_\kappa \models c{\downarrow}\kappa$

As an example, consider the constraint: $c_5 \doteq \forall a.\ p\ a \Rightarrow \forall b.\ q\ b \Rightarrow \kappa(a) \wedge \kappa(b)$. Below, we show each sub-constraint c_i, and the corresponding solution returned by $\text{sol1}(\kappa, c_i)$:

c_1	$\doteq$	$\kappa(a)$	$\text{sol1}(\kappa, c_1)$	$\doteq$	$x = a$
c_2	$\doteq$	$\kappa(b)$	$\text{sol1}(\kappa, c_2)$	$\doteq$	$x = b$
c_3	$\doteq$	$c_1 \wedge c_2$	$\text{sol1}(\kappa, c_3)$	$\doteq$	$x = a \vee x = b$
c_4	$\doteq$	$\forall b.\ q\ b \Rightarrow c_4$	$\text{sol1}(\kappa, c_4)$	$\doteq$	$\exists b.\ q\ b \wedge (x = a \vee x = b)$
c_5	$\doteq$	$\forall a.\ p\ a \Rightarrow c_4$	$\text{sol1}(\kappa, c_5)$	$\doteq$	$\exists a.\ p\ a \wedge (\exists b.\ q\ b \wedge (x = a \vee x = b))$

Note that in essence, $\sigma^{c_5}_\kappa \doteq [\kappa \mapsto \lambda x.\text{sol1}(\kappa, c_5)]$ maps κ to the disjunction of the two hypotheses (bodies) under which κ appears, thereby satisfying both implications, and hence satisfying $c_5{\downarrow}\kappa$.

The Strongest Scoped Solution The addition of hypotheses – the binders $\overline{x_i\!:\!p_i}$ under which κ always occurs – preserves validity. Thus, Lemma 5.4 implies that the strongest scoped solution satisfies the definitions of κ in c.

Theorem 5.5 (**Strongest-Scoped-Sat-Definitions**). $\hat{\sigma}^c_\kappa \models c{\downarrow}\kappa$.

Furthermore, if there exists a solution σ that satisfies c then the composition of σ and the strongest scoped solution also satisfies c.

elim1	:	$(K \times C) \to C$
$\text{elim1}(\kappa, c)$	$\doteq$	$\text{elim}^*(\hat{\sigma}^c_\kappa, c)$
where		
$\hat{\sigma}^c_\kappa$	=	$[\kappa \mapsto \lambda\overline{x}.\text{sol1}(\kappa, c')]$
$\forall(\overline{x_i : p_i}) \Rightarrow c'$	=	$\text{scope}(\kappa, c)$
$\overline{x}$	=	$\text{params}(\kappa)$
elim^*	:	$(\Sigma \times C) \to C$
$\text{elim}^*(\sigma, c_1 \wedge c_2)$	$\doteq$	$\text{elim}^*(\sigma, c_1) \wedge \text{elim}^*(\sigma, c_2)$
$\text{elim}^*(\sigma, \forall x : b.\ p \Rightarrow c)$	$\doteq$	$\forall x : b.\ \sigma(p) \Rightarrow \text{elim}^*(\sigma, c)$
$\text{elim}^*(\sigma, \kappa(\overline{y}))$		
$\mid\ \kappa \in \text{domain}(\sigma)$	$\doteq$	*true*
$\text{elim}^*(\sigma, p)$	$\doteq$	p

Fig. 11. Eliminating Variables.

Theorem 5.6 (**Strongest-Scoped-Sat-Uses**). *If* $\sigma \models c$ *then* $\sigma \cdot \hat{\sigma}^c_\kappa \models c$.

By Theorem 5.5 and Lemma 4.7 it suffices to restrict attention to uses of κ, *i.e.* to prove that if $\sigma \models c \upharpoonright \kappa$ then $\sigma \cdot \hat{\sigma}^c_\kappa \models c \upharpoonright \kappa$. We show this via a key lemma that states that *any* solution σ that satisfies c, must assign κ to a predicate that is implied by, *i.e.* weaker than, $\text{sol1}(\kappa, c)$.

Lemma 5.7 (**Strongest-Solution**). *If* $\sigma, \overline{x_i : p_i} \models c$ *then* $\sigma, \overline{x_i : p_i} \models \sigma^c_\kappa(p) \Rightarrow p$.

We use the above lemma to conclude that the hypotheses in $c \upharpoonright \kappa$ are *strengthened* under $\sigma \cdot \hat{\sigma}^c_\kappa$, and hence, that each corresponding goal remains valid under the composed assignment.

5.3 Eliminating One Variable

Next, we show how to eliminate a single κ variable by replacing it with its strongest solution.

Variable Elimination ***(***$\text{elim}^*(\sigma, c)$***)*** Procedure $\text{elim}^*(\sigma, c)$ (Fig. 11) *eliminates* from c all the κ variables defined in σ by replacing them with $\sigma(\kappa)$. That is, the procedure returns as output a constraint c' where: (1) every *body*-occurrence of a κ *i.e.* where κ appears as a hypothesis, is replaced by $\sigma(\kappa)$, and (2) every *head*-occurrence of a κ *i.e.* where κ appears as a goal, is replaced by *true* if κ is in the domain of σ, and left unchanged otherwise. We prove by induction over c that eliminating a single variable κ yields a constraint over just the *uses* of that variable:

Lemma 5.8 (elim^*). *If* $\text{domain}(\sigma) = \{\kappa\}$ *and* $c' = \text{elim}^*(\sigma, c)$ *then* $\kappa \notin c'$ *and* $\text{flat}(c') = \sigma(c \upharpoonright \kappa)$.

Eliminating One Variable Procedure $\text{elim1}(\kappa, c)$ (Fig. 11) eliminates κ from a constraint c by invoking elim on the strongest (scoped) solution for κ in c. We prove that if κ is acyclic in c then $\text{elim1}(\kappa, c)$ returns an constraint *without* κ that is satisfiable if and only if c is satisfiable.

Theorem 5.9 (**Elim-Preserves-Satisfiability**). $\text{elim1}(\kappa, c)$ *is satisfiable iff* c *is satisfiable.*

Intuitively, if $\sigma \models \text{elim1}(\kappa, c)$ then, via Theorem 5.6 and Lemma 4.3, we have $\sigma \cdot \hat{\sigma}^c_\kappa$ satisfies c. Dually, if $\sigma \models c$ then we show that $\sigma \models \text{elim1}(\kappa, c)$. See § A for details.

5.4 Eliminating Acyclic Variables

When the constraint c is acyclic, we can iteratively eliminate all the variables in K to obtain a VC whose validity determines the satisfiability of c, as each elimination preserves acyclicity:

Lemma 5.10 (**Elim-Acyclic**). *If K' is acyclic in c,* $\mathsf{deps}(\sigma) \subseteq \mathsf{deps}(c)$ *then K' is acyclic in* $\mathsf{elim}^*(\sigma, c)$.

Eliminating Many Variables Procedure $\mathsf{elim}(K, c)$ (Fig. 12) eliminates a set of acyclic K variables by iteratively eliminating each variable via $\mathsf{elim1}$. Using Theorem 5.9, Lemma 5.8 and Lemma 5.10 we show that the elimination of *multiple* acyclic κ-variables also preserves satisfiability.

Corollary 5.11 (**Exact-Satisfaction**). *Let* $c' \doteq \mathsf{elim}(K, c)$.

(1) $\mathsf{kvars}(c') = \emptyset$, *i.e. c' is a VC, and*
(2) c' is satisfiable iff c is satisfiable.

Unavoidable Worst-Case Blowup Note that in the worst case, the elim procedure can cause an exponential blowup in the constraint size. In general some form of blowup is unavoidable as even the Boolean version of the constraint satisfaction problem is complete for exponential time. A *boolean constraint* is an acyclic constraint where each variable is of type `Bool`. The *Boolean Constraint Satisfaction* (BcSat) problem is to decide whether a given boolean constraint is satisfiable.

Theorem 5.12. *BcSat is ExpTime-complete.*

The above is a corollary of [Godefroid and Yannakakis 2013] and the fact that reachability of (non-recursive) boolean programs is equivalent to the satisfaction of (acyclic) boolean constraints via the correspondence of [Jhala et al. 2011]. Thus, as validity checking is in Np, it is not possible to always generate compact (polynomial sized) formulas, assuming ExpTime $\neq$ Np.

Avoiding Let-Chain Blowup Recall the example from § 2.4. Let c be the NNF constraint in (14) and consider the sequence of intermediate constraints produced by eliminating $\kappa_1, \ldots, \kappa_n$ from c:

$$\begin{aligned} c_1 &\doteq c \\ c_{i+1} &\doteq \mathsf{elim1}(\kappa_i, c_i) \quad \text{for} \quad i \in 1, \ldots, n \end{aligned}$$

At each step, the strongest solution for κ_i is computed from $\mathsf{scope}(\kappa, c_i)$, *i.e.* the part of the constraint derived from the subterm where the binder x_i is in scope. Thus, for each $i \in 1, \ldots, n$ we get solutions

$$\mathsf{sol1}(\kappa_i, c_i) \doteq \exists \nu.\ \nu = \mathsf{x}_{i-1} \wedge z = \nu \tag{15}$$

where $\mathsf{params}(\kappa_i) = z$. For brevity of exposition, we simplify the above to

$$\mathsf{sol1}(\kappa_i, c_i) \doteq z = \mathsf{x}_{i-1}$$

and hence, note that $\mathsf{elim}([\kappa_1, \ldots, \kappa_n], c,)$ yields the linear-sized VC

$$c_{n+1} \doteq \forall \mathsf{x}_0.\ 0 \leq \mathsf{x}_0 \Rightarrow \forall \mathsf{x}_1.\ \mathsf{x}_1 = \mathsf{x}_0 \Rightarrow \ldots \Rightarrow \forall \mathsf{x}_n.\ \mathsf{x}_n = \mathsf{x}_{n-1} \Rightarrow \forall \nu.\ \nu = \mathsf{x}_n \Rightarrow 0 \leq \nu$$

that is easily proved valid by the SMT solver.

5.5 Eliminating Cyclic Variables

When the constraint c is cyclic, the satisfaction problem is undecidable via a reduction from the problem of checking the safety of while-programs [Bjørner et al. 2015]. Consequently, we can only compute over-approximate or conservative solutions via abstract interpretation [Jhala et al. 2011]. Our approach is to (1) compute the set of variables that make the constraints cyclic (*i.e.* whose absence would make the constraints acyclic), (2) eliminate all *except* those variables, and (3) use predicate abstraction to solve the resulting constraint, yielding a method that is much faster *and* more precise (§ 6), than "global" Liquid Typing [Rondon et al. 2008].

sat	:	$(C \times 2^R) \rightarrow \{\textsc{True}, \textsc{False}\}$
$\text{sat}(c, \mathbb{Q})$	$\doteq$	$\text{solve}(cs, \mathbb{Q})$
where		
c'	=	$\text{elim}(\hat{K}, c)$ where $\hat{K}$ cuts c
cs	=	$\text{flat}(c')$
flat	:	$C \rightarrow 2^C$
$\text{flat}(true)$	$\doteq$	$\emptyset$
$\text{flat}(p)$	$\doteq$	$\{p\}$
$\text{flat}(c \wedge c')$	$\doteq$	$\text{flat}(c) \cup \text{flat}(c')$
$\text{flat}(\forall x\!:\!b.\ p \Rightarrow c)$	$\doteq$	$\{\forall x\!:\!b.\ p \Rightarrow c' \mid c' \in \text{flat}(c)\}$
elim	:	$(2^K \times C) \rightarrow C$
$\text{elim}([], c)$	$\doteq$	c
$\text{elim}(\kappa : \kappa s, c)$	$\doteq$	$\text{elim}(\kappa s, \text{elim1}(\kappa, c))$

Fig. 12. Checking Constraint Satisfaction

Cut Variables A set of refinement variables $\hat{K}$ *cuts* a constraint c if $K - \hat{K}$ is acyclic in c. We cannot compute the *minimum* set of cut variables as this is the NP-Complete minimum feedback vertex set problem [Karp 1972]. Instead we use a greedy heuristic to compute $\hat{K}$. We compute the strongest connected component (SCC) digraph for $\text{deps}^*(c)$, and iteratively remove κ-variables (and recompute the digraph) until each SCC has a single vertex *i.e.* the graph is acyclic. The reasoning for Lemma 5.11 yields the following corollary:

Corollary 5.13. *If $\hat{K}$ cuts c then* $\text{elim}(K - \hat{K}, c)$ *returns a constraint c' such that* $\text{kvars}(c') \subseteq \hat{K}$ *and c' is satisfiable iff c is satisfiable.*

Exact and Approximate Constraint Satisfaction Procedure $\text{sat}(c, \mathbb{Q})$ (Fig. 12) computes a set of cut variables $\hat{K}$, and then uses elim to compute *exact* solutions for non-cut variables, and finally computes *approximate* solutions for the residual cut variables left over after elimination, using the solve procedure from [Rondon et al. 2008] which computes the satisfaction of (non-nested) constrained Horn Clauses (CHC) (obtained via flat) using predicate abstraction [Flanagan et al. 2001; Graf and Saïdi 1997].

Theorem 5.14 (**Satisfaction**).

(1) If $\text{sat}(c, \mathbb{Q})$ *then c is satisfiable.*
(2) If c is acyclic, then $\text{sat}(c, \mathbb{Q})$ *iff c is satisfiable.*

These results follow from Corollary 5.11, 5.13, the properties of solve (Theorem 2, [Rondon et al. 2008]), and as solve reduces to $\text{SmtValid}(c)$ when $\text{kvars}(c)$ is empty.

6 EVALUATION

We have implemented Fusion within the LiquidHaskell refinement type checker [Vazou et al. 2014b]. It is used by default in the current version [1]. We evaluate the *speed* and *precision* of Fusion on two sets of benchmarks from the LiquidHaskell project totalling more than 12KLOC. Our formalism based on call-by-value evaluation is sound for Haskell as we also simultaneously prove termination for all potentially bottom-inhabited terms, as described in [Vazou et al. 2014b]. Our results show that Fusion yields nearly 2× *speedups* for safety verification benchmarks and, by synthesizing the most precise types more than 10× faster, actually *enables* theorem proving.

Safety Property Benchmarks The first set of benchmarks are drawn from the Haskell standard libraries and detailed in [Vazou et al. 2014a]. Data-Struct comprises applicative data structures like red-black trees, lists (`Data.List`), splay trees (`Data.Set.Splay`), and binary search trees (`Data.Map.Base`); we verify termination and structure specific invariants like ordering and balance. Vector-Algorithms comprises a suite of imperative (*i.e.* monadic) array-based sorting algorithms; we verify termination and the correctness of array accesses. Text and Bytestring are the standard libraries for high-performance unicode text and byte-array processing which are implemented via low-level pointer arithmetic; we verify termination, memory safety and correctness properties specified by the library API.

Theorem Proving Benchmarks Recent work shows how refinement typing can convert legacy languages like Haskell into *proof assistants* where ordinary programs can be used to write arbitrary proofs of correctness about the "deep specifications" of other programs, and have the proofs checked via refinement typing [Vazou and Jhala 2016]. The second set of benchmarks corresponds to a set of programs corresponding to such proofs. Arith includes theorems about the *growth* of the fibonacci and ackermann functions; Fold includes theorems about the *universality* of traversals; Monoid, Functor, Applicative and Monad includes proofs of the respective category-theoretic *laws* for the `Maybe`, `List`, and `Id` instances of the respective typeclasses; and finally, SatSolver and Unification are fully verified implementations of the respective algorithms from the Zombie suite which, absent SMT support, requires significantly more local annotations (proof terms) from the user [Casinghino et al. 2014; Sjöberg and Weirich 2015].

Methodology For each benchmark, we compare the performance of LiquidHaskell using the (L) *global* refinement inference from [Rondon et al. 2008], and using our (F) *local* refinement algorithm. We compare the amount of **Time(s)**, in seconds, it took to check each benchmark. Each benchmark represents several Haskell files (modules); we report **Qualifiers**, the average number of qualifiers (predicates) that were required to synthesize the types needed for verification per file. In addition to the variables needed to eliminate cycles, we aggressively mark all refinement variables appearing in templates for "top-level" types as cut-variables to ensure that simple refinements (over qualifiers) are synthesized for such functions. All benchmarks were run on a MacBook Pro with a 2.2 GHz Intel Core i7 processor, using the Z3 SMT solver for checking validity. Table 1 summarizes the results.

Safety Verification Results Two points emerge from the safety verification benchmarks. First, for the larger benchmarks (text and bytestring), for which there already exist suitable qualifiers permitting global inference, the new local Fusion algorithm yields significant speedups – often *halving* the time taken for verification. (Note this includes end-to-end time, including name resolution, plain type checking *etc.*, and so the actual speedup for just refinement checking is even greater.) Second, even for these benchmarks, Fusion reduces (nearly halves) the number of *required* qualifiers. This makes the checker significantly easier to use, as the programmer does not have

[1] https://github.com/ucsd-progsys/liquidhaskell

Table 1. Comparing Liquid inference (L) [Rondon et al. 2008] with Fusion (F). The number of modules (files) per benchmark is listed in parentheses. **Code** is the total number of lines of non-comment and non-whitespace lines of code and **Spec** is the total number of lines of specification (*i.e.* top-level signatures), computed by `sloccount`. **Qualifiers** is the number of qualifiers needed for refinement inference; * means a false-positive, *i.e.* the benchmark could not be verified using the qualifiers provided as the intermediate terms have refinements not expressible using the given qualifiers. **Time(s)** is the total time in seconds needed to verify the benchmark (or to return a false positive, for times with a *).

Benchmark	**Code**	**Spec**	**Qualifiers**		**Time(s)**	
			L	F	L	F
Data-Struct (8)	1818	408	5	4	126	94
Vec-Algos (11)	1252	279	4	4	78	61
Bytestring (11)	4811	726	18	11	233	136
Text (17)	3157	818	9	5	349	231
Arith (2)	270	46	*	0	*63	5
Fold (1)	70	29	0	0	78	1
Monoid (2)	85	16	0	0	3	1
Functor (3)	137	28	0	0	55	3
Applicative (2)	146	36	*	0	*70	2
Monad (3)	180	42	*	0	*35	3
Sat-Solver (1)	98	31	*	0	*48	1
Unification (1)	139	53	*	1	*240	3

reason about which qualifiers to use for intermediate terms. By synthesizing strongest refinements, Fusion removes a key source of unpredictability and hard-to-diagnose false alarms.

Theorem Proving Results The improvement is more stark for the theorem proving benchmarks: most of them can only be checked using local inference. There are several reasons for this. The proofs are made possible by heavy use of polymorphic proof combinators. As the specifications are much more complicated, the combinators' type variables must be (automatically) instantiated with significantly more complex refinements that relate many program variables. Thus, it is very difficult for the user to even *determine* the relevant qualifiers. Even if they could, the qualifiers have many free variable (parameters) which causes an exponential blowup when matching against actual program variables, making global, abstract interpretation based refinement synthesis impossible. In contrast, since the theorem proving benchmarks have almost no cyclic dependencies, Fusion makes short work of automatically synthesizing the relevant refinement instantiations, making complex proofs possible.

Comparison with other Tools We are not aware of any other tool that scales up to these programs. F* [Swamy et al. 2011] requires local annotations as described in § 2, Mochi [Unno et al. 2013] requires no annotations but may diverge, and does not support uninterpreted functions which precludes all of our benchmarks. Similarly, existing Horn Solvers like μZ3 may diverge, while Eldarica [Rümmer et al. 2015], HSF [Grebenshchikov et al. 2012], and Spacer [Komuravelli et al. 2016] do not support uninterpreted functions.

7 RELATED WORK

Floyd-Hoare Logics and Model Checking The process of eliminating refinement variables to get a single verification condition (checked by SMT) is analogous to the VC-generation step used in ESC-style checkers [Flanagan et al. 2002; Nelson 1981]. Indeed we use the term "strongest refinement" or "solution" for the intermediate types to highlight the analogy with the notion of "strongest postconditions" from Floyd-Hoare Logic. While Fusion seeks to minimize the size of the generated VC, unlike in the intra-procedural case [Flanagan and Saxe 2000; Leino 2005], ExpTime-completeness means we cannot get compact VCs (Theorem 5.12). In the presence of loops, direct VC generation is insufficient, and one must compute over-approximations. Fusion's elimination procedure can be viewed as a generalization of the *large block encoding* method [Beyer et al. 2009] where over-approximation is performed not at each instruction, but only at "back edges" in the control-flow graph. Similarly, in program analyses it is common to "inline" the code for a procedure at a call site to improve precision. In [Swamy et al. 2013], the authors show how to generalize the notions of composing weakest preconditions across higher-order functions via the notion of a *Dijkstra Monad*. Fusion's elimination procedure is a way to systematically generalize and lift the Floyd-Hoare notions of strongest-postconditions (dually, weakest-preconditions), large block encodings, and procedure inlining to the typed, higher order setting. However, unlike Dijkstra Monads, Fusion exploits the compositional structure of types to locally synthesize precise refinements (*i.e.* invariants) in the presence of polymorphic collections and higher-order functions, to allow checking examples like `ex2`, `ex3`, and `ex4` (§ 2).

Local Type Inference Fusion performs a *local inference* in that *if* the constraints are acyclic, then Fusion is able to synthesize all intermediate refinement types exactly. However, refinements render the problem (and our solution) quite different than classical local typing [Pierce and Turner 1998], even with subtyping [Odersky et al. 2001]. First, even when all top-level (recursive) functions have signatures, the constraints may get cycles, for example, at instantiation sites for `fold` functions (whose polymorphic type variables must be instantiated with the analogue of a "loop invariant".) Second, our approach is orthogonal to bidirectional type checking. Indeed, they can be (and in our implementation, are) combined to yield a simpler system of constraints, but we still need solK and elim to synthesize the strongest refinements *relating* different program variables.

Refinement Inference There are several other approaches to synthesizing refinements. First, [Knowles and Flanagan 2009] shows how existentials can be used to type let-binders. Second, [Bengtson et al. 2008] shows how a form of bidirectional typing can be used to infer *some* intermediate types. Third, [Rondon et al. 2008] introduces the liquid typing framework for synthesizing refinements via abstract interpretation. Fourth, [Polikarpova et al. 2016] shows that liquid typing can be made bidirectional and presents a new demand driven ("round trip") algorithm for doing the abstract interpretation (and also solving for the weakest solution.) However, none of the above approaches is able to handle the idiomatic examples shown in § 2, or can only do so *if* given a suitable abstract domain (via templates). We can try to infer such templates via abstraction refinement [Jhala et al. 2011; Kobayashi et al. 2011; Unno et al. 2013] but that approach is notoriously unpredictable and prone to diverging, especially in the presence of uninterpreted functions which are ubiquitous in our examples. Finally, [Zhu et al. 2015] shows how machine learning over dynamic traces can be used to learn refinements (in a generalization of the approach pioneered by [Ernst et al. 2001]). However, this needs closed programs (that can be run), which can limit applicability to higher order functions.

Horn Clauses Horn clauses have recently become a popular "intermediate representation" for verification problems [Bjørner et al. 2015], as they can be used to encode the proof rules for classical imperative Floyd-Hoare logics, and concurrent programs [Grebenshchikov et al. 2012] among

others. However, current Horn Clause solvers *e.g.* [Grebenshchikov et al. 2012; Hoder and Bjørner 2012; Rümmer et al. 2015] are based on CEGAR and interpolation and hence, to quote a recent survey [Bjørner et al. 2015]: "mainly tuned for real and linear integer arithmetic and Boolean domains" rendering them unable to check any of our benchmarks which make extensive use of uninterpreted functions. Our work shows how to (1) algorithmically generate NNF clauses from typed, higher-order programs, in a way that preserves scoping, (2) use an optimized form of "unfolding" [Burstall and Darlington 1977; Pettorossi and Proietti 1994; Tamaki and Sato 1984] to synthesize the most precise type and (3) thereby, obtain a method for improving the *speed*, *precision* and *completeness* of refinement type checking.

ACKNOWLEDGMENTS

We thank the anonymous referees, Nadia Polikarpova, Eric Seidel and Niki Vazou for their invaluable feedback on earlier drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant Numbers CCF-1422471, CCF-1223850, and CCF-1218344, and a generous gift from Microsoft Research.

REFERENCES

C. Barrett, P. Fontaine, and C. Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. 2008. Refinement Types for Secure Implementations. In *CSF*.

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *FMCAD*.

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation*.

Rod M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (1977), 44–67.

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *POPL*.

R.L. Constable. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.

J. Dunfield. 2007. Refined typechecking with Stardust. In *PLPV*.

M.D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE TSE* 27(2) (2001), 1–25.

C. Flanagan, R. Joshi, and K. R. M. Leino. 2001. Annotation inference for modular checkers. *Inform. Process. Lett.* (2001).

C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. 2002. Extended static checking for Java. In *PLDI*.

C. Flanagan and J.B. Saxe. 2000. Avoiding exponential explosion: generating compact verification conditions. In *POPL*.

Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *TACAS*.

S. Graf and H. Saïdi. 1997. Construction of abstract state graphs with PVS. In *CAV*. Springer, 72–83.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *PLDI*.

Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. 199–216. https://doi.org/10.1007/978-3-662-48288-9_12

Ralf Hinze. 2009. Functional pearl: la tour d'Hanoï. In *ICFP*.

Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *SAT*.

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *CAV*. https://doi.org/10.1007/978-3-642-22110-1_38

R. Jhala and K.L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *TACAS 06*.

Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations*.

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

K. Knowles and C. Flanagan. 2007. Type Reconstruction for General Refinement Types. In *ESOP*. http://kenn.frap.net/publications/knowles-flanagan.esop.07.type.pdf

K.W. Knowles and C. Flanagan. 2010. Hybrid type checking. *ACM TOPLAS* (2010).

K. W. Knowles and C. Flanagan. 2009. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*.
Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*.
Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4
K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inf. Process. Lett.* 93, 6 (2005).
G. Nelson. 1981. *Techniques for program verification.* Technical Report CSL81-10. Xerox Palo Alto Research Center.
Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL*.
X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*.
Alberto Pettorossi and Maurizio Proietti. 1994. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming* 19 (1994), 261 – 320.
B. C. Pierce and D. N. Turner. 1998. Local Type Inference. In *POPL*.
Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI*.
P. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI*.
Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2015. On recursion-free Horn clauses and Craig interpolation. *Formal Methods in System Design* 47, 1 (2015), 1–25.
J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE TSE* (1998).
Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to Congruence. In *POPL*.
N. Swamy, J. Chen, C. Fournet, P-Y. Strub, K. Bhargavan, and J. Yang. 2011. Secure distributed programming with value-dependent types. In *ICFP*.
N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *PLDI*.
Hisao Tamaki and Taisuke Sato. 1984. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984.* 127–138.
H. Unno, T. Terauchi, and N. Kobayashi. 2013. Relatively complete verification of higher-order functional programs. In *POPL*.
N. Vazou and R. Jhala. 2016. Refinement Reflection (or, how to turn your favorite language into a proof assistant using SMT). *ArXiv e-prints* (Oct. 2016). arXiv:cs.PL/1610.04641
N. Vazou, E. L. Seidel, and R. Jhala. 2014a. LiquidHaskell: Experience with refinement types in the real world. In *Haskell*.
N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones. 2014b. Refinement Types for Haskell. In *ICFP*.
Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *PLDI*.
H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types.. In *PLDI*.
He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning refinement types. In *ICFP*.

A APPENDIX: PROOFS

We include below the proofs of the key theorems.

Proof. (Lemma 5.7) Assume that $p \equiv \kappa(\overline{z})$ where $\overline{z} = \text{params}(\kappa)$ The other cases are trivial as $\sigma^c_\kappa(p) = p$, and $\sigma(p_1) \Rightarrow p_1$ and $\sigma(p_2) \Rightarrow p_2$ implies $\sigma(p_1 \wedge p_2) \Rightarrow p_1 \wedge p_2$. The proof is by induction on the structure of c.

Case: $c \equiv \kappa(\overline{y})$.

$$\begin{aligned}
\text{By definition of sol1}(c,\kappa) \quad & \sigma^c_\kappa(\kappa(\overline{z})) = \overline{z} = \overline{y} && (16)\\
\text{Assume} \quad & \sigma, \overline{x_i : p_i} \models \kappa(\overline{y}) \\
\text{Hence} \quad & \sigma, \overline{x_i : p_i} \models \overline{z} = \overline{y} \Rightarrow \kappa(\overline{z}) \\
\text{By 16} \quad & \sigma, \overline{x_i : p_i} \models \sigma^c_\kappa(\kappa(\overline{z})) \Rightarrow \kappa(\overline{z})
\end{aligned}$$

Case: $c \equiv p'$, *such that* $\kappa \notin p'$.

By definition of validity	$\sigma, \overline{x_i : p_i} \models \mathit{false} \Rightarrow \kappa(\overline{z})$
By the definition of $\mathsf{sol1}(c, \kappa)$	$\sigma^c_\kappa(\kappa(\overline{z})) = \mathit{false}$
Hence,	$\sigma, \overline{x_i : p_i} \models \sigma^c_\kappa(\kappa(\overline{z})) \Rightarrow \kappa(\overline{z})$

Case: $c \equiv c_1 \wedge c_2$.

Let $p_j = \mathsf{sol1}(c_j, \kappa)$ for $j = 1, 2$

Thus, by the definition of $\mathsf{sol1}(c, \kappa)$	$\sigma^c_\kappa(\kappa(\overline{z})) = p_1 \vee p_2$	(17)
Assume, for $j = 1, 2$	$\sigma, \overline{x_i : p_i} \models \wedge_j c_j$	
That is, for $j = 1, 2$	$\sigma, \overline{x_i : p_i} \models c_j$	
By IH, for $j = 1, 2$	$\sigma, \overline{x_i : p_i} \models p_j \Rightarrow \kappa(\overline{z})$	
Hence,	$\sigma, \overline{x_i : p_i} \models (p_1 \vee p_2) \Rightarrow \kappa(\overline{z})$	
Thus, by 17,	$\sigma, \overline{x_i : p_i} \models \sigma^c_\kappa(\kappa(\overline{z})) \Rightarrow \kappa(\overline{z})$	

Case: $c \equiv \forall x : b.\ p \Rightarrow c'$.

Let $p' = \mathsf{sol1}(c', \kappa)$.

Thus, by the definition of $\mathsf{sol1}(c, \kappa)$	$\sigma^{c'}_\kappa(\kappa(\overline{z})) = p'$	(18)
	$\sigma^c_\kappa(\kappa(\overline{z})) = \exists x : b.\ p \wedge p'$	(19)
Assume	$\sigma, \overline{x_i : p_i} \models \forall x : b.\ p \Rightarrow c'$	
By Sat-Ext	$\sigma, \overline{x_i : p_i}, x : p \models c'$	
By IH	$\sigma, \overline{x_i : p_i}, x : p \models \sigma^{c'}_\kappa(\kappa(\overline{z})) \Rightarrow \kappa(\overline{z})$	
By 18	$\sigma, \overline{x_i : p_i}, x : p \models p' \Rightarrow \kappa(\overline{z})$	
By Sat-Ext	$\sigma, \overline{x_i : p_i} \models \forall x : b.\ p \Rightarrow p' \Rightarrow \kappa(\overline{z})$	
That is,	$\sigma, \overline{x_i : p_i} \models \forall x : b.\ (p \wedge p' \Rightarrow \kappa(\overline{z}))$	
As $x \notin \kappa(\overline{z})$	$\sigma, \overline{x_i : p_i} \models (\exists x : b.\ p \wedge p') \Rightarrow \kappa(\overline{z})$	
By 19	$\sigma, \overline{x_i : p_i} \models \sigma^c_\kappa(\kappa(\overline{z})) \Rightarrow \kappa(\overline{z})$	

□

Proof. (Theorem 5.4) The proof is an induction on the structure of c.

Case: $c \equiv \kappa(\overline{y})$.

By definition	$c{\downarrow}\kappa \doteq \kappa(\overline{y})$	
and	$\sigma^c_\kappa(\kappa) \doteq \lambda \overline{x}.\ \wedge_i x_i = y_i$	where $\overline{x} = \mathsf{params}(\kappa)$
Hence,	$\sigma^c_\kappa(c{\downarrow}\kappa) \doteq \wedge_i y_i = y_i$	(20)
As 20 is a tautology	$\sigma^c_\kappa \models c{\downarrow}\kappa$	

Case: $c \equiv p$.

By definition	$c{\downarrow}\kappa \doteq \mathit{true}$	
Hence,	$\sigma^c_\kappa(c{\downarrow}\kappa) \doteq \mathit{true}$	(21)
As 21 is a tautology	$\sigma^c_\kappa \models c{\downarrow}\kappa$	

Case: $c \equiv c_1 \wedge c_2$.

For $i \in \{1,2\}$, let $p_i = \mathrm{sol1}(\kappa, c_i)$, $\sigma_\kappa^{c_i}(\kappa) = \lambda\overline{x}.p_i$.

$$\begin{aligned}
\text{By definition} \quad & c{\downarrow}\kappa \doteq c_1{\downarrow}\kappa \cup c_2{\downarrow}\kappa && (22)\\
\text{By IH} \quad & \sigma_\kappa^{c_i} \models c_i{\downarrow}\kappa && (23)
\end{aligned}$$

Let $c' \in c_i{\downarrow}\kappa$. As κ *only* in head, $c' \equiv \forall(\overline{y:p}) \Rightarrow \kappa(\overline{x})$.

$$\begin{aligned}
\text{By 23} \quad & \overline{y:p} \models \sigma_\kappa^{c_i}(\kappa(\overline{x}))\\
\text{By Lemma 4.1} \quad & \overline{y:p} \models \sigma_\kappa^{c_1}(\kappa(\overline{x})) \vee \sigma_\kappa^{c_2}(\kappa(\overline{x}))\\
\text{That is} \quad & \overline{y:p} \models \sigma_\kappa^{c}(\kappa(\overline{x}))\\
\text{So for any } c' \in c_i{\downarrow}\kappa \quad & \sigma_\kappa^{c} \models c'\\
\text{Hence} \quad & \sigma_\kappa^{c} \models c_i{\downarrow}\kappa\\
\text{and hence, by 22} \quad & \sigma_\kappa^{c} \models c{\downarrow}\kappa
\end{aligned}$$

Case: $c \equiv \forall x{:}b.\ p \Rightarrow c'$.

Let $p' = \mathrm{sol1}(\kappa, c')$, $\sigma' = [\kappa \mapsto \lambda\overline{z}.p']$, and $\sigma_\kappa^c = [\kappa \mapsto \lambda\overline{z}.\exists x{:}b.\ p \wedge p']$,

$$\begin{aligned}
\text{By IH} \quad & \sigma' \models c'{\downarrow}\kappa && (24)
\end{aligned}$$

Consider any arbitrary $c'' \in c'{\downarrow}\kappa$.

$$\begin{aligned}
\text{As } \kappa \in \mathrm{head}(c'') \quad & c'' \doteq \forall x_1.\ p_1 \Rightarrow \ldots \Rightarrow \forall x_n.\ p_n \Rightarrow \kappa(\overline{y}) && (25)\\
\text{By 24} \quad & \sigma' \models \forall x_1.\ p_1 \Rightarrow \ldots \Rightarrow \forall x_n.\ p_n \Rightarrow \kappa(\overline{y})\\
\text{By Sat-Base} \quad & [] \models \forall x_1.\ p_1 \Rightarrow \ldots \Rightarrow \forall x_n.\ p_n \Rightarrow p'\,[\overline{y}/\overline{z}]\\
\text{By Sat-Ext} \quad & \overline{x_i:p_i} \models p'\,[\overline{y}/\overline{z}] && (26)\\
\text{Trivially,} \quad & x{:}p \models p && (27)\\
\text{Hence, by 26, 27} \quad & x{:}p, \overline{x_i:p_i} \models p \wedge p'\,[\overline{y}/\overline{z}]\\
\text{and so by definition of validity} \quad & x{:}p, \overline{x_i:p_i} \models \exists x{:}b.\ p \wedge p'\,[\overline{y}/\overline{z}]\\
\text{by 25} \quad & \sigma_\kappa^c \models \forall x{:}b.\ p \Rightarrow c''
\end{aligned}$$

As the above holds for an arbitrary $c'' \in c'{\downarrow}\kappa$, we get $\sigma_\kappa^c \models c{\downarrow}\kappa$. □

Proof. (Theorem 5.5)

$$\begin{aligned}
\text{Let} \quad & \mathrm{scope}(\kappa, c) \doteq \forall(\overline{x:p}) \Rightarrow c' \text{s.t. } \kappa \notin \overline{p} && (28)\\
\text{By definition} \quad & \hat{\sigma}_\kappa^c = \sigma_\kappa^{c'}\\
\text{By Theorem 5.4} \quad & \hat{\sigma}_\kappa^c \models c'{\downarrow}\kappa\\
\text{By definition of validity} \quad & \hat{\sigma}_\kappa^c \models \forall(\overline{x:p}) \Rightarrow (c'{\downarrow}\kappa)\\
\text{By Lemma 4.6} \quad & \hat{\sigma}_\kappa^c \models (\forall(\overline{x:p}) \Rightarrow c'){\downarrow}\kappa\\
\text{By (28) and Lemma 5.1} \quad & \hat{\sigma}_\kappa^c \models c{\downarrow}\kappa
\end{aligned}$$

□

Proof. (Theorem 5.6) By Theorem 5.5 it suffices to prove that if $\sigma \models c$ then $\sigma \cdot \hat{\sigma}^c_\kappa \models c \uparrow \kappa$.

$$\text{Let} \quad \text{scope}(\kappa, c) \doteq \forall(\overline{y:q}) \Rightarrow c' \text{s.t. } \kappa \notin \overline{q} \text{ and } \hat{\sigma}^c_\kappa = \sigma^{c'}_\kappa \tag{29}$$

$$\text{Assume} \quad \sigma \models c \tag{30}$$

$$\text{By (29), Lemma 5.3} \quad \sigma \models \forall(\overline{y:q}) \Rightarrow c' \tag{31}$$

$$\text{By Lemma 5.2, (29)} \quad c \uparrow \kappa = \forall(\overline{y:q}) \Rightarrow (c' \uparrow \kappa) \cup C \quad \text{where } \kappa \notin C \tag{32}$$

$$\text{Hence, by (35) and (37)} \quad \sigma \cdot \hat{\sigma}^c_\kappa \models c \uparrow \kappa$$

Case: $\sigma \cdot \hat{\sigma}^c_\kappa \models \forall(\overline{y:q}) \Rightarrow (c' \uparrow \kappa)$ (35)

$$\text{By (30,32), Lemma 4.7} \quad \sigma \models \forall(\overline{y:q}) \Rightarrow (c' \uparrow \kappa) \tag{33}$$

$$\text{Let } c'' \in c' \uparrow \kappa \text{ s.t.} \quad c'' \equiv \forall(\overline{x_i:p_i}) \Rightarrow p \text{ and } \kappa \notin p \tag{34}$$

$$\text{By (33)} \quad \sigma \models \forall(\overline{y:q}) \Rightarrow \forall(\overline{x_i:p_i}) \Rightarrow p$$

$$\text{By Sat-Ext} \quad \sigma, \overline{y:q} \models \forall(\overline{x_i:p_i}) \Rightarrow p$$

$$\text{By (31)} \quad \sigma, \overline{y:q} \models c'$$

$$\text{By (29), Lemma 5.7, for } i = 1 \ldots n \quad \sigma, \overline{y:q} \models \sigma^{c'}_\kappa(p_i) \Rightarrow p_i$$

$$\text{As } \hat{\sigma}^c_\kappa = \sigma^{c'}_\kappa \text{ (29)} \quad \sigma, \overline{y:q} \models \hat{\sigma}^c_\kappa(p_i) \Rightarrow p_i$$

$$\text{By repeating Lemma 4.2 and } \kappa \notin p \quad \sigma, \overline{y:q} \models \hat{\sigma}^c_\kappa(\forall(\overline{x_i:p_i}) \Rightarrow p)$$

$$\text{That is} \quad \sigma, \overline{y:q} \models \hat{\sigma}^c_\kappa(c'')$$

$$\text{As by (29) } \kappa \notin \overline{q} \quad \sigma \models \hat{\sigma}^c_\kappa(\forall(\overline{y:q}) \Rightarrow c'')$$

$$\text{Thus, by (34)} \quad \sigma \models \hat{\sigma}^c_\kappa(\forall(\overline{y:q}) \Rightarrow c' \uparrow \kappa)$$

$$\text{And by Lemma 4.4} \quad \sigma \cdot \hat{\sigma}^c_\kappa \models \forall(\overline{y:q}) \Rightarrow c' \uparrow \kappa \tag{35}$$

Case: $\sigma \cdot \hat{\sigma}^c_\kappa \models C$ (37)

$$\text{By (30, 32), Lemma 4.7} \quad \sigma \models C \tag{36}$$

$$\text{As } \kappa \notin C, \text{domain}(\hat{\sigma}^c_\kappa) = \{\kappa\} \quad \hat{\sigma}^c_\kappa(C) = C$$

$$\text{By (36)} \quad \sigma \models \hat{\sigma}^c_\kappa(C)$$

$$\text{By Lemma 4.4} \quad \sigma \cdot \hat{\sigma}^c_\kappa \models C \tag{37}$$

□

Lemma A.1 (**Eliminate-Definitions**). $\text{flat}(\text{elim1}(\kappa, c)) = \hat{\sigma}^c_\kappa(c \uparrow \kappa)$.

Proof. (Lemma A.1) The above follows from the observations that: (a) $\text{domain}(\hat{\sigma}^c_\kappa) = \{\kappa\}$, (b) $\text{elim1}(\kappa, c) = \text{elim}^*(\hat{\sigma}^c_\kappa, c)$, and (c) Lemma 5.8, as $\text{domain}(\hat{\sigma}^c_\kappa) = \{\kappa\}$. □

Proof. (Theorem 5.9) Let $c' = \text{elim1}(\kappa, c) = \text{elim}^*(\hat{\sigma}^c_\kappa, c)$. We prove the two directions separately.

Case: c' is satisfiable implies c is satisfiable

$$\text{Assume } c' \text{ is satisfiable with} \quad \sigma' \models c'$$

$$\text{(by Lemma 4.5)} \quad \sigma' \models \text{flat}(c')$$

$$\text{(by Lemma A.1)} \quad \sigma' \models \hat{\sigma}^c_\kappa(c \uparrow \kappa)$$

$$\text{(by Lemma 4.4)} \quad \sigma' \cdot \hat{\sigma}^c_\kappa \models c \uparrow \kappa \tag{38}$$

$$\text{(by Theorem 5.5, Lemma 4.3)} \quad \sigma' \cdot \hat{\sigma}^c_\kappa \models c \downarrow \kappa \tag{39}$$

$$\text{By 38, 39, Lemma 4.7} \quad \sigma' \cdot \hat{\sigma}^c_\kappa \models c \quad \textit{i.e. } c \text{ is satisfiable.}$$

Case: c is satisfiable implies c' is satisfiable

Assume c is satisfiable with	$\sigma \models c$	
by Theorem 5.6	$\sigma \cdot \hat{\sigma}^c_\kappa \models c$	
by Lemma 4.7	$\sigma \cdot \hat{\sigma}^c_\kappa \models c \uparrow \kappa$	
by Lemma 4.4	$\sigma \models \hat{\sigma}^c_\kappa(c \uparrow \kappa)$	
by Lemma A.1	$\sigma \models \mathrm{flat}(c')$	*i.e.* by Lemma 4.5 c' is satisfiable.

□