Nested Refinements: A Logic for Duck Typing *

Ravi Chugh Patrick M. Rondon Ranjit Jhala
University of California, San Diego
{rchugh, prondon, jhala}@cs.ucsd.edu

Abstract

Programs written in dynamic languages make heavy use of features — run-time type tests, value-indexed dictionaries, polymorphism, and higher-order functions — that are beyond the reach of type systems that employ either purely syntactic or purely semantic reasoning. We present a core calculus, System D, that merges these two modes of reasoning into a single powerful mechanism of *nested refinement types* wherein the typing relation is itself a predicate in the refinement logic. System D coordinates SMT-based logical implication and syntactic subtyping to automatically typecheck sophisticated dynamic language programs. By coupling nested refinements with McCarthy's theory of finite maps, System D can precisely reason about the interaction of higher-order functions, polymorphism, and dictionaries. The addition of type predicates to the refinement logic creates a circularity that leads to unique technical challenges in the metatheory, which we solve with a novel stratification approach that we use to prove the soundness of System D.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs – Logics of Programs

General Terms Languages, Verification

Keywords Refinement Types, Dynamic Languages

1. Introduction

So-called dynamic languages like JavaScript, Python, Racket, and Ruby are popular as they allow developers to quickly put together scripts without having to appease a static type system. However, these scripts quickly grow into substantial code bases that would be much easier to maintain, refactor, evolve and compile, if only they could be corralled within a suitable static type system.

The convenience of dynamic languages comes from their support of features like run-time type testing, value-indexed finite maps (*i.e.* dictionaries), and duck typing, a form of polymorphism where functions operate over any dictionary with the appropriate keys. As the empirical study in [18] shows, programs written in dynamic languages make heavy use of these features, and their safety relies on invariants which can only be established by sophisticated reasoning about the flow of control, the run-time types of values, and the contents of data structures like dictionaries.

The following code snippet, adapted from the popular Dojo Javascript framework [36], illustrates common dynamic features:

```
let onto callbacks f obj =
  if f = null then
    new List(obj, callbacks)
  else
    let cb = if tag f = "Str" then obj[f] else f in
    new List(fun () -> cb obj, callbacks)
```

The function `onto` is used to register callback functions to be called after the DOM and required library modules have finished loading. The author of `onto` went to great pains to make it extremely flexible in the kinds of arguments it takes. If the `obj` parameter is provided but `f` is not, then `obj` is the function to be called after loading. Otherwise, both `f` and `obj` are provided, and either: (a) `f` is a string, `obj` is a dictionary, and the (function) value corresponding to key `f` in `obj` is called with `obj` as a parameter after loading; or (b) `f` is a function which is called with `obj` as a parameter after loading. To verify the safety of this program, and dynamic code in general, a type system must reason about dynamic type tests, control flow, higher-order functions, and heterogeneous, value-indexed dictionaries.

Current automatic type systems are not expressive enough to support the full spectrum of reasoning required for dynamic languages.[1] *Syntactic* systems use advanced type-theoretic constructs like structural types [3], row types [38], intersection types [17], and union types [18, 39] to track invariants of individual values. Unfortunately, such techniques cannot reason about value-dependent relationships *between* program variables, as is required, for example, to determine the specific types of the variables `f` and `obj` in `onto`. *Semantic* systems like [4, 6, 14, 23, 31, 35] support such reasoning by using logical predicates to describe invariants over program variables. Unfortunately, such systems require a clear (syntactic) distinction between *complex* values that are typed with arrows, type variables, *etc.*, and *base* values that are typed with predicates. Hence, they cannot support the interaction of complex values and value-indexed dictionaries that is ubiquitous in dynamic code, for example in `onto`, which can take as a parameter a dictionary containing a function value.

Our Approach. We present System D, a core calculus that supports fully automatic checking of dynamic idioms. In System D *all* values are described uniformly by formulas drawn from a decidable, quantifier-free refinement logic. Our first key insight is that to reason precisely about complex values (*e.g.* higher-order functions) nested deeply inside structures (*e.g.* dictionaries), we require

* This work was supported by NSF Grants CCF-0644361, CNS-0964702, and generous gifts from Microsoft Research.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

[1] Full dependent type systems like Coq [5] can support the necessary reasoning, but are not automatic as the programmer must provide explicit proofs to discharge type checking obligations.

a single new mechanism called *nested refinements* wherein syntactic types (resp. the typing relation) may be nested as special *type terms* (resp. *type predicates*) inside the refinement logic. Formally, the refinement logic is extended with atomic formulas of the form $x :: U$ where U is a type term, "::" (read "has type") is a binary, *uninterpreted* predicate in the refinement logic, and where the formula states that the value x "has the type" described by the term U. This unifying insight allows to us to express the invariants in idiomatic dynamic code like `onto` — including the interaction between higher-order functions and dictionaries — while staying within the boundaries of decidability.

Expressiveness. The nested refinement logic underlying System D can express complex invariants between base values and richer values. For example, we may disjoin two tag-equality predicates

$$\{\nu \mid tag(\nu) = \text{“}\texttt{Int}\text{”} \vee tag(\nu) = \text{“}\texttt{Str}\text{”}\}$$

to type a value ν that is either an integer or a string; we can then track control flow involving the dynamic type tag-lookup function `tag` to ensure that the value is safely used at either more specific type. To describe values like the argument `f` of the `onto` function we can combine tag-equality predicates with the type predicate. We can give `f` the type

$$\{\nu \mid \nu = \texttt{null} \vee tag(\nu) = \text{“}\texttt{Str}\text{”} \vee \nu :: Top \to Top\}$$

where Top is an abbreviation for $\{\nu \mid true\}$, which is a type that describes all values. Notice the uniformity — the types *nested* within this refinement formula are themselves refinement types.

Our second key insight is that dictionaries are finite maps, and so we can precisely type dictionaries with refinement formulas drawn from the (decidable) theory of finite maps [25]. In particular, McCarthy's two operators — $sel(x, a)$, which corresponds to the contents of the map x at the address a, and $upd(x, a, v)$, which corresponds to the new map obtained by updating x at the address a with the value v — are precisely what we need to describe reads from and updates to dictionaries. For example, we can write

$$\{\nu \mid tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge tag(sel(\nu, \texttt{y})) = \text{“}\texttt{Int}\text{”}\}$$

to type dictionaries ν that have (at least) an integer field `y`, where `y` is a program variable that dynamically stores the key with which to index the dictionary. Even better, since we have nested function types into the refinement logic, we can precisely specify, for the first time, combinations of dictionaries and functions. For example, we can write the following type for `obj`

$$\{\nu \mid tag(\texttt{f}) = \text{“}\texttt{Str}\text{”} \Rightarrow sel(\nu, \texttt{f}) :: Top \to Top\}$$

to describe the second portion of the `onto` specification, all while staying within a decidable refinement logic. In a similar manner, we show how nested refinements support polymorphism, datatypes, and even a form of bounded quantification.

Subtyping. The huge leap in expressiveness yielded by nesting types inside refinements is accompanied by some unique technical challenges. The first challenge is that because we nest complex types (*e.g.* arrows) as uninterpreted terms in the logic, subtyping (*e.g.* between arrows) cannot be carried out solely via the usual syntactic decomposition into SMT queries [6, 16, 31]. (A higher-order logic (*e.g.* [5]) would solve this problem, but that would preclude algorithmic checking; we choose the uninterpreted route precisely to relieve the SMT solver of higher-order reasoning!) We surmount this challenge with a novel decomposition mechanism where subtyping between types, syntactic type terms, and refinement formulas are defined *inter-dependently*, thereby using the logical structure of the refinement formulas to divide the labor of subtyping between the SMT solver for ground predicates (*e.g.* equality, uninterpreted functions, arithmetic, maps, *etc.*) and classical syntactic rules for type terms (*e.g.* arrows, type variables, datatypes).

Soundness. The second challenge is that the inter-dependency between the refinement logic and the type system renders the standard proof techniques for (refinement) type soundness inapplicable. In particular, we illustrate how uninterpreted type predicates break the usual substitution property and how nesting makes it difficult to define a type system that is well-defined and enjoys this property. We meet this challenge with a new proof technique: we define an infinite family of *increasingly precise* systems and prove soundness of the family, of which System D is a member, thus establishing the soundness of System D.

Contributions. To sum up, we make the following contributions:

- We show how nested refinements over the theory of finite maps encode function, polymorphic, dictionary and constructed data types within refinements and permit dependent structural subtyping and a form of bounded quantification.
- We develop a novel algorithmic subtyping mechanism that uses the structure of the refinement formulas to decompose subtyping into a collection of SMT and syntactic checks.
- We illustrate the technical challenges that nesting poses to the metatheory of System D and present a novel stratification-based proof technique to establish soundness.
- We define an algorithmic version of the type system with local type inference that we implement in a prototype checker.

Thus, by carefully orchestrating the interplay between syntactic- and SMT-based subtyping, the nested refinement types of System D enable, for the first time, the automatic static checking of features found in idiomatic dynamic code.

2. Overview

We start with a series of examples that give an overview of our approach. First, we show how by encoding types using logical refinements, System D can reason about control flow and relationships between program variables. Next, we demonstrate how nested refinements enable precise reasoning about values of complex types. After that, we illustrate how System D uses refinements over the theory of finite maps to analyze value-indexed dictionaries. We conclude by showing how these features combine to analyze the sophisticated invariants in idiomatic dynamic code.

Notation. We use the following abbreviations for brevity.

$$\begin{aligned}
Top(x) &\triangleq true \\
Int(x) &\triangleq tag(x) = \text{“}\texttt{Int}\text{”} \\
Bool(x) &\triangleq tag(x) = \text{“}\texttt{Bool}\text{”} \\
Str(x) &\triangleq tag(x) = \text{“}\texttt{Str}\text{”} \\
Dict(x) &\triangleq tag(x) = \text{“}\texttt{Dict}\text{”} \\
IorB(x) &\triangleq Int(x) \vee Bool(x)
\end{aligned}$$

We abuse notation to use the above as abbreviations for refinement types; for each of the unary abbreviations T defined above, an occurrence *without* the parameter denotes the refinement type $\{\nu \mid T(\nu)\}$. For example, we write Int as an abbreviation for $\{\nu \mid tag(\nu) = \text{“}\texttt{Int}\text{”}\}$. Recall that function values are also described by refinement formulas (containing type predicates). We often write arrows outside refinements to abbreviate the following:

$$x{:}T_1 \to T_2 \quad \triangleq \quad \{\nu \mid \nu :: x{:}T_1 \to T_2\}$$

We write $T_1 \to T_2$ when the return type T_2 does not refer to x.

2.1 Simple Refinements

To warm up, we show how System D describes all types through refinement formulas, and how, by using an SMT solver to discharge

the subtyping (implication) queries, System D makes short work of value- and control flow-sensitive reasoning [18, 39].

Ad-Hoc Unions. Our first example illustrates the simplest dynamic idiom: programs which operate on *ad-hoc* unions. The function `negate` takes an integer or boolean and returns its negation:

```
let negate x =
  if tag x = "Int" then 0 - x else not x
```

In System D we can ascribe to this function the type

$$\texttt{negate} :: \mathit{IorB} \rightarrow \mathit{IorB}$$

which states that the function accepts an integer or boolean argument and returns either an integer or boolean result.

To establish this, System D uses the standard means of reasoning about control flow in refinement-based systems [31], namely strengthening the environment with the guard predicate when processing the then-branch of an if-expression and the negation of the guard predicate for the else-branch. Thus, in the then-branch, the environment contains the assumption that $\mathit{tag}(\texttt{x}) = \text{“}\texttt{Int}\text{”}$, which allows System D to verify that the expression $0 - \texttt{x}$ is well-typed. The return value has the type $\{\nu \mid \mathit{tag}(\nu) = \text{“}\texttt{Int}\text{”} \wedge \nu = 0 - \texttt{x}\}$. This type is a subtype of IorB as the SMT solver can prove that $\mathit{tag}(\nu) = \text{“}\texttt{Int}\text{”}$ and $\nu = 0 - \texttt{x}$ implies $\mathit{tag}(\nu) = \text{“}\texttt{Int}\text{”} \vee \mathit{tag}(\nu) = \text{“}\texttt{Bool}\text{”}$. Thus, the return value of the then-branch is deduced to have type IorB.

On the other hand, in the else-branch, the environment contains the assumption $\neg(\mathit{tag}(\texttt{x}) = \text{“}\texttt{Int}\text{”})$. By combining this with the assumption about the type of `negate`'s input, $\mathit{tag}(\texttt{x}) = \text{“}\texttt{Int}\text{”} \vee \mathit{tag}(\texttt{x}) = \text{“}\texttt{Bool}\text{”}$, the SMT solver can determine that $\mathit{tag}(\texttt{x}) = \text{“}\texttt{Bool}\text{”}$. This allows our system to type check the call to

$$\texttt{not} :: \mathit{Bool} \rightarrow \mathit{Bool},$$

which establishes that the value returned in the else branch has type IorB. Thus, our system determines that both branches return a value of type IorB, and thus that `negate` meets its specification.

Dependent Unions. System D's use of refinements and SMT solvers enable expressive *relational* specifications that go beyond previous techniques [18, 39]. While `negate` takes and returns ad-hoc unions, there is a relationship between its input and output: the output is an integer (resp. boolean) iff the input is an integer (resp. boolean). We represent this in System D as

$$\texttt{negate} :: x\!:\!\mathit{IorB} \rightarrow \{\nu \mid \mathit{tag}(\nu) = \mathit{tag}(x)\}$$

That is, the refinement for the output states that its tag is *the same as* the tag of the input. This function is checked through exactly the same analysis as before; the tag test ensures that the environment in the then- (resp. else-) branch implies that `x` and the returned value are both Int (resp. Bool). That is, in both cases, the output value has the same tag as the input.

2.2 Nested Refinements

So far, we have seen how old-fashioned refinement types (where the predicates refine base values [6, 23, 27, 31]) can be used to check ad-hoc unions over base values. However, a type system for dynamic languages must be able to express invariants about values of base and function types with equal ease. We accomplish this in System D by adding types (resp. the typing relation) to the refinement logic as nested *type terms* (resp. *type predicates*).

However, nesting raises a rather tricky problem: with the typing relation included in the refinement logic, subtyping can no longer be carried out entirely via SMT implication queries [6]. We solve this problem with a new subtyping rule that *extracts* type terms from refinements to enable syntactic subtyping for nested types.

Consider the function `maybeApply` which takes an integer `x` and a value `f` which is either `null` or a function over integers:

```
let maybeApply x f =
  if f = null then x else f x
```

In System D, we can use a refinement formula that combines a base predicate and a type predicate to assign `maybeApply` the type

$$\texttt{maybeApply} :: \mathit{Int} \rightarrow \{\nu \mid \nu = \texttt{null} \vee \nu :: \mathit{Int} \rightarrow \mathit{Int}\} \rightarrow \mathit{Int}$$

Note that we have *nested* a function type as a term in the refinement logic, along with an assertion that a value has this particular function type. However, to keep checking algorithmic, we use a simple first-order logic in which type terms and predicates are completely *uninterpreted*; that is, the types can be thought of as constant terms in the logic. Therefore, we need new machinery to check that `maybeApply` actually enjoys the above type, *i.e.* to check that (a) `f` is indeed a function when it is applied, (b) it can accept the input `x`, and (c) it will return an integer.

Type Extraction. To accomplish the above goals, we *extract* the nested function type for `f` stored in the type environment as follows. Let Γ be the type environment at the callsite `(f x)`. For each type term U occurring in Γ, we query the SMT solver to determine whether $[\![\Gamma]\!] \Rightarrow \texttt{f} :: U$ holds, where $[\![\Gamma]\!]$ is the embedding of Γ into the refinement logic where type terms and predicates are treated in a purely uninterpreted way. If so, we say that U *must flow to* (or just, flows to) the caller expression `f`. Once we have found the type terms that flow to the caller expression, we map the uninterpreted type terms to their corresponding type definitions to check the call.

Let us see how this works for `maybeApply`. The then-branch is trivial: the assumption that `x` is an integer in the environment allows us to deduce that the expression `x` is well-typed and has type Int. Next, consider the else-branch. Let U_1 be the type term $\mathit{Int} \rightarrow \mathit{Int}$. Due to the bindings for `x` and `f` and the else-condition, the environment Γ is embedded as

$$[\![\Gamma]\!] \doteq \mathit{tag}(\texttt{x}) = \text{“}\texttt{Int}\text{”} \wedge (\texttt{f} = \texttt{null} \vee \texttt{f} :: U_1) \wedge \neg(\texttt{f} = \texttt{null})$$

Hence, the SMT solver is able to prove that $\Gamma \Rightarrow \texttt{f} :: U_1$. This establishes that `f` is a function on integers and, since `x` is known to be an integer, we can verify that the else-branch has type Int and hence check that `maybeApply` meets its specification.

Nested Subtyping. Next, consider a client of `maybeApply`:

```
let _ = maybeApply 42 negate
```

At the call to `maybeApply` we must show that the actuals are subtypes of the formals, *i.e.* that the two subtyping relationships

$$\begin{aligned} \Gamma_1 \vdash \{\nu \mid \nu = 42\} &\sqsubseteq \mathit{Int} \\ \Gamma_1 \vdash \{\nu \mid \nu = \texttt{negate}\} &\sqsubseteq \{\nu \mid \nu = \texttt{null} \vee \nu :: U_1\} \end{aligned} \quad (1)$$

hold, where $\Gamma_1 \doteq \texttt{negate}\!:\!\{\nu \mid \nu :: U_0\}, \texttt{maybeApply}\!:\!\cdots$ and $U_0 = x\!:\!\mathit{IorB} \rightarrow \{\nu \mid \mathit{tag}(\nu) = \mathit{tag}(x)\}$. Alas, while the SMT solver can make short work of the first obligation, it cannot be used to discharge the second via implication; the “real” types that must be checked for subsumption, namely, U_0 and U_1, are embedded as totally unrelated terms in the refinement logic!

Once again, extraction rides to the rescue. We show that all subtyping checks of the form $\Gamma \vdash \{\nu \mid p\} \sqsubseteq \{\nu \mid q\}$ can be reduced to a finite number of sub-goals of the form:

$$\begin{aligned} \textit{(“type predicate-free”)}\ [\![\Gamma']\!] &\Rightarrow p' \\ \text{or } \textit{(“type predicate”)}\ [\![\Gamma']\!] &\Rightarrow x :: U \end{aligned}$$

The former kind of goal has no type predicates and can be directly discharged via SMT. For the latter, we use extraction to find the finitely many type terms U_i that *flow to* x. (If there are none, the

check fails.) For each U_i we use *syntactic* subtyping to verify that the corresponding type is subsumed by (the type corresponding to) U under Γ'.

In our example, the goal 1 reduces to proving either

$$[\![\Gamma_1']\!] \Rightarrow \nu = \texttt{null} \quad \text{or} \quad [\![\Gamma_1']\!] \Rightarrow \nu :: U_1$$

where $\Gamma_1' \triangleq \Gamma_1, \nu = \texttt{negate}$. The former implication contains no type predicates, so we attempt to prove it by querying the SMT solver. The solver tells us that the query is not valid, so we turn to the latter implication. The extraction procedure uses the SMT solver to deduce that, under Γ_1' the type term U_0 flows into ν. Thus, all that remains is to retrieve the definition of U_0 and U_1 and check

$$\Gamma_1' \vdash x : IorB \rightarrow \{\nu \mid tag(\nu) = tag(x)\} \sqsubseteq Int \rightarrow Int$$

which follows via standard syntactic refinement subtyping [16], thereby checking the client's call. Thus, by carefully interleaving SMT implication and syntactic subtyping, System D enables, for the first time, the nesting of rich types *within* refinements.

2.3 Dictionaries

Next, we show how nested refinements allow System D to precisely check programs that manipulate dynamic dictionaries. In essence, we demonstrate how structural subtyping can be done via nested refinement formulas over the theory of finite maps [13, 25]. We introduce two abbreviations for dictionaries.

$$Fld(x, y, Int) \triangleq Dict(x) \wedge Str(y) \wedge Int(sel(x, y))$$
$$Fld(x, y, U) \triangleq Dict(x) \wedge Str(y) \wedge sel(x, y) :: U$$

The second abbreviation states that the type of a field is a syntactic type term U (*e.g.* an arrow).

Dynamic Lookup. SMT-based structural subtyping allows System D to support the common idiom of dynamic field lookup and update, where the field name is a value computed at run-time. Consider the following function:

```
let getCount t c =
  if has t c then toInt (t[c]) else 0
```

The function `getCount` uses the primitive operation

$$\texttt{has} :: d : Dict \rightarrow k : Str \rightarrow \{\nu \mid Bool(\nu) \wedge \nu = \texttt{true} \Leftrightarrow has(d, k)\}$$

to check whether the key `c` exists in `t`. The refinement for the input d expresses the precondition that d is a dictionary, while the refinement for the key k expresses the precondition that k is a string. The refinement of the output expresses the postcondition that the result is a boolean value which is true if and only if d has a binding for the key k, expressed in our refinements using $has(d, k)$, a predicate in the theory of maps that is true if and only if there is a binding for key k in the map d [13, 25].

The *dictionary lookup* `t[c]` is desugared to `get t c` where the primitive operation `get` has the type

$$\texttt{get} :: d : Dict \rightarrow k : \{\nu \mid Str(\nu) \ \wedge \ has(d, k)\} \rightarrow \{\nu \mid \nu = sel(d, k)\}$$

and $sel(d, k)$ is an operator in the theory of maps that returns the binding for key k in the map d. The refinement for the key k expresses the precondition that it is a string value in the domain of the dictionary d. Similarly, the refinement for the output asserts the postcondition that the value is the same as the contents of the map at the given key.

The function `getCount` first tests that the dictionary `t` has a binding for the key `c`; if so, it is read and its contents are converted to an integer using the function `toInt`, of type $Top \rightarrow Int$. Note that the if-guard strengthens the environment under which the lookup appears with the fact $has(\texttt{t}, \texttt{c})$, ensuring the safety of the lookup. If `t` does not contain the key `c`, the default value `0` is returned. Both branches are thus verified to have type Int, so System D verifies that `getCount` has the type $\texttt{getCount} :: Dict \rightarrow Str \rightarrow Int$.

Dynamic Update. Dually, to allow dynamic updates, System D includes a primitive

$$\texttt{set} :: d : Dict \rightarrow k : Str \rightarrow x : Top \rightarrow \{\nu \mid \nu = upd(d, k, x)\}$$

that produces a new dictionary, where $upd(d, k, x)$ is an operator in the theory of maps that denotes d updated (or extended) with a binding from k to x. The following illustrates how the `set` primitive can be used:

```
let incCount t c =
  let newcount = 1 + getCount t c in
  let res      = set t c newcount in res
```

We give the function `incCount` the type

$$d : Dict \rightarrow c : Str \rightarrow \{\nu \mid EqMod(\nu, d, \{c\}) \wedge Fld(\nu, c, Int)\}$$

where $EqMod(d_1, d_2, K)$ abbreviates a predicate that stipulates that d_1 is identical to d_2 at all keys *except* for those in K. The output type of `getCount` allows System D to conclude that $\texttt{newcount} :: Int$. From the type of `set`, System D deduces

$$\texttt{res} :: \{\nu \mid EqMod(\nu, \texttt{t}, \{\texttt{c}\}) \wedge sel(\nu, \texttt{c}) = \texttt{newcount}\}$$

which is a subtype of the output type of `incCount`. Next, consider

```
let d0 = {"files" = 42 }
let d1 = incCount d0 "dirs"
let _  = d1["files"] + d1["dirs"]
```

System D verifies that

$$\texttt{d0} :: \{\nu \mid Fld(\nu, \text{“}\texttt{files}\text{”}, Int)\}$$
$$\texttt{d1} :: \{\nu \mid Fld(\nu, \text{“}\texttt{files}\text{”}, Int) \wedge Fld(\nu, \text{“}\texttt{dirs}\text{”}, Int)\}$$

and, hence, the field lookups return Ints that can be safely added.

2.4 Type Constructors

Next, we use nesting and extraction to enrich System D with data structures, thereby allowing for very expressive specifications. In general, System D supports arbitrary user-defined datatypes, but to keep the current discussion simple, let us consider a single type constructor $List[T]$ for representing unbounded sequences of T-values. Informally, an expression of type $List[T]$ is either a special `null` value or a dictionary with a “`hd`” key of type T and a “`tl`” key of type $List[T]$. As for arrows, we use the following notation to write list types outside of refinements:

$$List[T] \triangleq \{\nu \mid \nu :: List[T]\}$$

Recursive Traversal. Consider a textbook recursive function that takes a list of arbitrary values and concatenates the strings:

```
let rec concat sep xs =
  if xs = null then "" else
    let hd = xs["hd"] in
    let tl = xs["tl"] in
    if tag hd != "Str" then concat sep tl
    else if tl != null then hd ^ sep ^ concat sep tl
    else hd
```

We ascribe the function the type $\texttt{concat} :: Str \rightarrow List[Top] \rightarrow Str$. The null test ensures the safety of the “`hd`” and “`tl`” accesses and the tag test ensures the safety of the string concatenation using the techniques described above.

Nested Ad-Hoc Unions. We can now define ad-hoc unions over constructed types by simply nesting $List[\cdot]$ as a type term in the refinement logic. The following illustrates a common Python idiom when an argument is either a single value or a list of values:

```
let runTest cmd fail_codes =
  let status = syscall cmd in
  if tag fail_codes = "Int" then
    not (status = fail_codes)
  else
    not (listMem status fail_codes)
```

Here, `listMem` :: $Top \rightarrow List[Top] \rightarrow Bool$ and `syscall` :: $Str \rightarrow Int$. The input `cmd` is a string, and `fail_codes` is either a single integer or a list of integer failure codes. Because we nest $List[\cdot]$ as a type term in our logic, we can use the same kind of type extraction reasoning as we did for `maybeApply` to ascribe `runTest` the type

$$\texttt{runTest} :: Str \rightarrow \{\nu \mid Int(\nu) \vee \nu :: List[Int]\} \rightarrow Bool$$

2.5 Parametric Polymorphism

Similarly, we can add parametric polymorphism to System D by simply treating type variables $A, B, etc.$ as (uninterpreted) type terms in the logic. As before, we use the following notation to write type variables outside of refinements.

$$A \triangleq \{\nu \mid \nu :: A\}$$

Generic Containers. We can compose the type constructors in the ways we all know and love. Here is list `map` in System D:

```
let rec map f xs =
  if xs = null then null
  else new List(f xs["hd"], map f xs["tl"])
```

(Of course, pattern matching would improve matters, but we are merely trying to demonstrate how much can be — and is! — achieved with dictionaries.) By combining extraction with the reasoning used for `concat`, it is easy to check that

$$\texttt{map} :: \forall A, B.\ (A \rightarrow B) \rightarrow List[A] \rightarrow List[B]$$

Note that type abstractions are automatically inserted where a function is ascribed a polymorphic type.

Predicate Functions. Consider the list filter function:

```
let rec filter f xs =
  if xs = null then null
  else if not (f xs["hd"]) then filter f (xs["tl"])
  else new List(xs["hd"], filter f xs["tl"])
```

In System D, we can ascribe `filter` the type

$$\forall A, B.\ (x{:}A \rightarrow \{\nu \mid \nu = \texttt{true} \Rightarrow x :: B\}) \rightarrow List[A] \rightarrow List[B],$$

Note that the return type of the predicate, `f`, tells us what type is satisfied by values `x` for which `f` returns `true`, and the return type of `filter` states that the items `filter` returns all have the type implied by the predicate `f`. Thus, the general mechanism of nested refinements subsumes the kind of reasoning performed by specialized techniques like latent predicates [39].

Bounded Quantification. Nested refinements enable a form of bounded quantification. Consider the function

```
let dispatch d f = d[f] d
```

The function `dispatch` works for any dictionary `d` of type A that has a key `f` bound to a function that maps values of type A to values of type B. We can specify this via the dependent signature

$$\forall A, B.\ d{:}\{\nu \mid Dict(\nu) \wedge \nu :: A\} \rightarrow \{\nu \mid Fld(d, \nu, A \rightarrow B)\} \rightarrow B$$

Note that there is no need for explicit type bounds; all that is required is the conjunction of the appropriate nested refinements.

2.6 All Together Now

With the tools we've developed in this section, System D is now capable of type checking sophisticated code from the wild. The original source code for the following can be found in a technical report [8].

Unions, Generic Dispatch, and Polymorphism. We now have everything we need to type the motivating example from the introduction, `onto`, which combined multiple dynamic idioms: dynamic fields, tag-tests, and the dependency between nested dictionary functions and their arguments. Nested refinements let us formalize the flexible interface for `onto` given in the introduction:

$$\begin{aligned}
&\forall A.\ callbacks{:}List[Top \rightarrow Top] \\
&\quad \rightarrow f{:}\{\nu \mid \nu = \texttt{null} \vee Str(\nu) \vee \nu :: A \rightarrow Top\} \\
&\quad \rightarrow obj{:}\{\nu \mid \nu :: A \wedge (f = \texttt{null} \Rightarrow \nu :: Top \rightarrow Top) \\
&\qquad\qquad \wedge (Str(f) \Rightarrow Fld(\nu, f, A \rightarrow Top))\} \\
&\quad \rightarrow List[Top \rightarrow Top]
\end{aligned}$$

Using reasoning similar to that used in the previous examples, System D checks that `onto` enjoys the above type, where the specification for obj is enabled by the kind of bounded quantification described earlier.

Reflection. Finally, to round off the overview, we present one last example that shows how all the features presented combine to allow System D to statically type programs that introspect on the contents of dictionaries. The function `toXML` shown below is adapted from the Python 3.2 standard library's `plistlib.py` [37]:

```
let rec toXML x =
  if tag x = "Bool" then
    if x then element "true" null
    else      element "false" null
  else if tag x = "Int" then
    element "integer" (intToStr x)
  else if tag x = "Str" then
    element "string" x
  else if tag x = "Dict" then
    let ks = keys x in
    let vs = map {v| Str(v) and has(x,v)} Str
      (fun k -> element "key" k ^ toXML x[k]) ks in
    "<data>" ^ concat "\n" vs ^ "</data>"
  else element "function" null
```

The function takes an arbitrary value and renders it as an XML string, and illustrates several idiomatic uses of dynamic features. If we give the auxiliary function `intToStr` the type $Int \rightarrow Str$ and `element` the type $Str \rightarrow \{\nu \mid \nu = \texttt{null} \vee Str(\nu)\} \rightarrow Str$, we can verify that

$$\texttt{toXML} :: Top \rightarrow Str$$

Of especial interest is the dynamic field lookup `x[k]` used in the function passed to `map` to recursively convert each binding of the dictionary to XML. The primitive operation `keys` has the type

$$\texttt{keys} :: d{:}Dict \rightarrow List[\{\nu \mid Str(\nu) \wedge has(d, \nu)\}]$$

that is, it returns a list of string keys that belong to the input dictionary. Thus, `ks` has type $List[\{\nu \mid Str(\nu) \wedge has(\texttt{x}, \nu)\}]$, which enables the call to `map` to typecheck, since the body of the argument is checked in an environment where `k` :: $\{\nu \mid Str(\nu) \wedge has(\texttt{x}, \nu)\}$, which is the type that A is instantiated with. This binding suffices to prove the safety of the dynamic field access. The control flow reasoning described previously uses the tag tests guarding the other cases to prove each of them safe.

w	::=		**Values**
	|	x	variable
	|	c	constant
	|	w_1 ++ $\{w_2 \mapsto w_3\}$	dictionary extension
	|	$\lambda x.\, e$	function
	|	$\lambda A.\, e$	type function
	|	$C(\overline{w})$	constructed data
e	::=		**Expressions**
	|	w	value
	|	$w_1\ w_2$	function application
	|	$w\ [T]$	type function application
	|	if w then e_1 else e_2	if-then-else
	|	let $x = e_1$ in e_2	let-binding
td	::=	type $C[\overline{\theta A}]\{\overline{f : T}\}$	**Datatype Definitions**
prg	::=	$\overline{td}; e$	**Programs**
lw	::=		**Logical Values**
	|	w	value
	|	$F(\overline{lw})$	logical function application
p, q, r	::=		**Refinement Formulas**
	|	$P(\overline{lw})$	predicate
	|	$lw :: U$	type predicate
	|	$p \wedge q \mid p \vee q \mid \neg p$	logical connective
T	::=	$\{\nu \mid p\}$	**Refinement Types**
U	::=		**Type Terms**
	|	$x : T_1 \to T_2$	arrow
	|	A	type variable
	|	$C[\overline{T}]$	constructed type
	|	$Null$	null
S	::=	$T \mid \forall A.\, S$	**Type Schemes**

Figure 1. Syntax of System D

3. Syntax and Semantics

We begin with the syntax and evaluation semantics of System D. Figure 1 shows the syntax of values, expressions, and types.

Values. Values w include variables constants, functions, type functions, dictionaries, and records created by type constructors. The set of constants c includes base values like integer, boolean, and string constants, the empty dictionary $\{\}$, and `null`. Logical values lw are all values and applications of primitive function symbols F, such as addition $+$ and dictionary selection sel, to logical values. The constant `tag` allows introspection on the type tag of a value at run-time. For example,

$$\begin{array}{rcl@{\qquad}rcl} \texttt{tag}(3) & \triangleq & \text{“}\texttt{Int}\text{”} & \texttt{tag}(\texttt{true}) & \triangleq & \text{“}\texttt{Bool}\text{”} \\ \texttt{tag}(\text{“john”}) & \triangleq & \text{“}\texttt{Str}\text{”} & \texttt{tag}(\lambda x.\, e) & \triangleq & \text{“}\texttt{Fun}\text{”} \\ \texttt{tag}(\{\}) & \triangleq & \text{“}\texttt{Dict}\text{”} & \texttt{tag}(\lambda A.\, e) & \triangleq & \text{“}\texttt{TFun}\text{”} \end{array}$$

Dictionaries. A dictionary w_1 ++ $\{w_2 \mapsto w_3\}$ extends the dictionary w_1 with the binding from string w_2 to value w_3. For example, the dictionary mapping “`x`” to 3 and “`y`” to `true` is written

$$\{\}\ \texttt{++}\ \{\text{“}\texttt{x}\text{”} \mapsto 3\}\ \texttt{++}\ \{\text{“}\texttt{y}\text{”} \mapsto \texttt{true}\}.$$

The set of constants also includes operations for extending dictionaries and accessing their fields. The function `get` is used to access dictionary fields and is defined

$$\begin{aligned} \texttt{get}\ (w\ \texttt{++}\ \{\text{“}\texttt{x}\text{”} \mapsto w_x\})\ \text{“}\texttt{x}\text{”} &\triangleq w_x \\ \texttt{get}\ (w\ \texttt{++}\ \{\text{“}\texttt{y}\text{”} \mapsto w_y\})\ \text{“}\texttt{x}\text{”} &\triangleq \texttt{get}\ w\ \text{“}\texttt{x}\text{”} \end{aligned}$$

The function `has` tests for the presence of a field and is defined

$$\begin{aligned} \texttt{has}\ (w\ \texttt{++}\ \{\text{“}\texttt{y}\text{”} \mapsto w_y\})\ \text{“}\texttt{x}\text{”} &\triangleq \texttt{has}\ w\ \text{“}\texttt{x}\text{”} \\ \texttt{has}\ (w\ \texttt{++}\ \{\text{“}\texttt{x}\text{”} \mapsto w_x\})\ \text{“}\texttt{x}\text{”} &\triangleq \texttt{true} \\ \texttt{has}\ \{\}\ \text{“}\texttt{x}\text{”} &\triangleq \texttt{false} \end{aligned}$$

The function `set` updates the value bound to a key and is defined

$$\texttt{set}\ d\ k\ w \triangleq d\ \texttt{++}\ \{k \mapsto w\}$$

Expressions. The set of expressions e consists of values, function applications, type instantiations, if-then-else expressions, and let-bindings. We use an A-normal presentation so that we need only define substitution of values (not arbitrary expressions) into types.

Types. We stratify types into monomorphic types T and polymorphic type schemes $\forall A.\ S$. In System D, a type T is a *refinement type* of the form $\{\nu \mid p\}$, where p is a *refinement formula*, and is read “ν such that p.” The values of this type are all values w such that the formula $p[w/\nu]$ “is true.” What this means, formally, is core to our approach and will be considered in detail in section 5.

Refinement Formulas. The language of *refinement formulas* includes predicates P, such as the equality predicate and dictionary predicates has and sel, and the usual logical connectives. For example, the type of integers is $\{\nu \mid tag(\nu) = \text{“}\texttt{Int}\text{”}\}$, which we abbreviate to Int. The type of positive integers is

$$\{\nu \mid tag(\nu) = \text{“}\texttt{Int}\text{”} \wedge \nu > 0\}$$

and the type of dictionaries with an integer field “`f`” is

$$\{\nu \mid tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge tag(sel(\nu, \text{“}\texttt{f}\text{”})) = \text{“}\texttt{Int}\text{”}\}.$$

We refer to the binder ν in refinement types as “the value variable.”

Nesting: Type Predicates and Terms. To express the types of values like functions and dictionaries containing functions, System D permits types to be nested within refinement formulas. Formally, the language of refinement formulas includes a form, $lw :: U$, called a *type predicate*, where U is a *type term*. The type term $x : T_1 \to T_2$ describes values that have a dependent function type, *i.e.* functions that accept arguments w of type T_1 and return values of type $T_2[w/x]$, where x is bound in T_2. We write $T_1 \to T_2$ when x does not appear in T_2. Type terms A, B, *etc.* correspond to type parameters to polymorphic functions. The type term $Null$ corresponds to the type of the constant value `null`. The type term $C[\overline{T}]$ corresponds to records constructed with the C type constructor instantiated with the sequence of type arguments $\overline{T}$. For example, the type of the (integer) successor function is

$$\{\nu \mid \nu :: x : Int \to \{\nu \mid tag(\nu) = \text{“}\texttt{Int}\text{”} \wedge \nu = x + 1\}\},$$

dictionaries where the value at key “`f`” maps Int to Int have type

$$\{\nu \mid tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge sel(\nu, \text{“}\texttt{f}\text{”}) :: Int \to Int\},$$

and the constructed record $List(\texttt{1}, \texttt{null})$ can be assigned the type $\{\nu \mid \nu :: List[Int]\}$.

Datatype Definitions. A datatype definition of C defines a named, possibly recursive type. A datatype definition includes a sequence $\overline{\theta A}$ of type parameters A paired with *variance annotations* θ. A variance annotation is either `+` (covariant), `-` (contravariant), or `=` (invariant). The rest of the definition specifies a sequence $\overline{f : T}$ of field names and their types. The types of the fields may refer to the type parameters of the declaration. A well-formedness check,

Operational Semantics $\boxed{e \hookrightarrow e'}$

$$\frac{\text{if } \delta(c, w) \text{ is defined}}{c\ w \hookrightarrow \delta(c, w)}\ \text{[E-DELTA]}$$

$$(\lambda x.\ e)\ w \hookrightarrow e[w/x]\ \text{[E-APP]}$$

$$\text{let } x = w \text{ in } e \hookrightarrow e[w/x]\ \text{[E-LET]}$$

$$(\lambda A.\ e)\ [T] \hookrightarrow e\ \text{[E-TAPP]}$$

$$\text{if } \texttt{true} \text{ then } e_1 \text{ else } e_2 \hookrightarrow e_1\ \text{[E-IFTRUE]}$$

$$\text{if } \texttt{false} \text{ then } e_1 \text{ else } e_2 \hookrightarrow e_2\ \text{[E-IFFALSE]}$$

$$\frac{e_1 \hookrightarrow e_1'}{\text{let } x = e_1 \text{ in } e_2 \hookrightarrow \text{let } x = e_1' \text{ in } e_2}\ \text{[E-COMPAT]}$$

Figure 2. Evaluation semantics of System D

which will be described in section 4, ensures that occurrences of type parameters in the field types respect their declared variance annotations. By convention, we will use the subscript i to index into the sequence $\overline{\theta A}$ and j for $\overline{f : T}$. For example, θ_i refers to the variance annotation of the i^{th} type parameter, and f_j refers to the name of the j^{th} field.

Programs. A program is a sequence of datatype definitions $\overline{td}$ followed by an expression e. Requiring all datatype definitions to appear first simplifies the subsequent presentation.

Semantics. The small-step operational semantics of System D is standard for a call-by-value, polymorphic lambda calculus, and is shown in Figure 2. Following standard practice, the semantics is parameterized by a function δ that assigns meaning to primitive functions c, including dictionary operations like `has`, `get`, and `set`. Because expressions are A-normalized, there is a single congruence rule, E-COMPAT. Our implementation desugars more palatable syntax into A-normal form.

4. Type Checking

In this section, we present the System D type system, comprising several well-formedness relations, an expression typing relation, and, at the heart of our approach, a novel subtyping relation which discharges obligations involving nested refinements through a combination of syntactic and semantic, SMT-based reasoning. We first define environments for type checking.

Environments. Type environments Γ are of the form

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, x : S \mid \Gamma, A \mid \Gamma, p$$

where bindings either record the derived type S for a variable x, a type variable A introduced in the scope of a type function, or a formula p that is recorded to track the control flow along branches of an if-expression. A type definition environment Ψ records the definition of each constructor type C. As type definitions appear at the beginning of a program, we assume for clarity that Ψ is fixed and globally visible, and elide it from the judgments. In the sequel, we assume that Ψ contains at least the definition

$$\text{type } \mathit{List}[+A]\{\text{“}\texttt{hd}\text{”} : \{\nu \mid \nu :: A\};\ \text{“}\texttt{tl}\text{”} : \{\nu \mid \nu :: \mathit{List}[A]\}\}.$$

4.1 Well-formedness

Figure 3 defines the well-formedness relations.

Well-Formed Type Schemes $\boxed{\Gamma \vdash S}$

$$\frac{x \text{ fresh} \qquad \Gamma, x : \mathit{Top} \vdash p[x/\nu]}{\Gamma \vdash \{\nu \mid p\}} \qquad \frac{\Gamma, A \vdash S}{\Gamma \vdash \forall A.\ S}$$

Well-Formed Formulas $\boxed{\Gamma \vdash p}$

$$\frac{\Gamma \vdash \mathit{lw} \qquad \Gamma \vdash U}{\Gamma \vdash \mathit{lw} :: U} \qquad \frac{\forall i.\ \Gamma \vdash \mathit{lw}_i}{\Gamma \vdash P(\overline{\mathit{lw}})} \qquad \frac{\Gamma \vdash p \qquad \Gamma \vdash q}{\Gamma \vdash p \wedge q}$$

Well-Formed Type Terms $\boxed{\Gamma \vdash U}$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x : T_1 \vdash T_2}{\Gamma \vdash x : T_1 \to T_2} \qquad \frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{}{\Gamma \vdash \mathit{Null}} \qquad \frac{C \in \mathit{Dom}(\Psi) \quad \forall i.\ \Gamma \vdash T_i}{\Gamma \vdash C[\overline{T}]}$$

Well-Formed Type Environments $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset} \qquad \frac{x \notin \mathit{Dom}(\Gamma) \quad \vdash \Gamma \quad \Gamma \vdash S}{\vdash \Gamma, x : S} \qquad \frac{\vdash \Gamma \quad A \notin \Gamma}{\vdash \Gamma, A} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash p}{\vdash \Gamma, p}$$

Well-Formed Type Definitions $\boxed{\vdash td}$

$$\frac{\forall j.\ \overline{A} \vdash T_j \qquad \forall i.\ \mathsf{VarianceOk}(A_i, \theta_i, \overline{T})}{\vdash \text{type } C[\overline{\theta A}]\{\overline{f : T}\}}$$

Figure 3. Well-formedness for System D

Formulas, Types and Environments. We require that types be *well-formed* within the current type environment, which means that formulas used in types are boolean propositions and mention only variables that are currently in scope. By convention, we assume that variables used as binders throughout the program are distinct and different from the special value variable ν, which is reserved for types. Therefore, ν is never bound in Γ. When checking the well-formedness of a refinement formula p, we substitute a fresh variable x for ν and check that $p[x/\nu]$ is well-formed in the environment extended with $x : \mathit{Top}$, to the environment, where $\mathit{Top} = \{\nu \mid \mathit{true}\}$. We use fresh variables to prevent duplicate bindings of ν.

Note that the well-formedness of formulas does *not* depend on type checking; all that is needed is the ability to syntactically distinguish between terms and propositions. Checking that values are well-formed is straightforward; the important point is that a variable x may be used only if it is bound in Γ.

Datatype Definitions. To check that a datatype definition is well-formed, we first check that the types of the fields are well-formed in an environment containing the declared type parameters. Then, to enable a sound subtyping rule for constructed types in the sequel, we check that the declared variance annotations are respected within the type definition. For this, we use a procedure $\mathsf{VarianceOk}$ (defined in a technical report [8]) that recursively walks formulas to record whether type variables occur in positive or negative positions within the types of the fields.

4.2 Expression Typing

The expression typing judgment $\Gamma \vdash e :: S$, defined in Figure 4, verifies that expression e has type scheme S in environment Γ. We highlight the important aspects of the typing rules.

Constants. Each primitive constant c has a type, denoted by $ty(c)$, that is used by T-CONST. Basic values like integers, booleans, *etc.* are given singleton types stating that their value equals the corresponding constant in the refinement logic. For example:

$$\begin{array}{rlrl} \texttt{1} :: & \{\nu \mid \nu = \texttt{1}\} & \texttt{true} :: & \{\nu \mid \nu = \texttt{true}\} \\ \text{“john”} :: & \{\nu \mid \nu = \text{“john”}\} & \texttt{false} :: & \{\nu \mid \nu = \texttt{false}\} \end{array}$$

Arithmetic and boolean operations have types that reflect their semantics. Equality on base values is defined in the standard way, while equality on function values is physical equality.

$$\begin{array}{l} \texttt{+} :: x{:}\mathit{Int} \to y{:}\mathit{Int} \to \{\nu \mid \mathit{Int}(\nu) \wedge \nu = x + y\} \\ \texttt{not} :: x{:}\mathit{Bool} \to \{\nu \mid \mathit{Bool}(\nu) \wedge x = \texttt{true} \Leftrightarrow \nu = \texttt{false}\} \\ \texttt{=} :: x{:}\mathit{Top} \to y{:}\mathit{Top} \to \{\nu \mid \mathit{Bool}(\nu) \wedge \nu = \texttt{true} \Leftrightarrow x = y\} \\ \texttt{fix} :: \forall A.\ (A \to A) \to A \\ \texttt{tag} :: x{:}\mathit{Top} \to \{\nu \mid \nu = \mathit{tag}(x)\} \end{array}$$

The constant `fix` is used to encode recursion, and the type for the tag-test operation uses an axiomatized function in the logic.

The operations on dictionaries are given refinement types over McCarthy's theory of arrays extended with a default element, which we write as bot, that is different from all program values. The extended theory is shown to be decidable in [13].

$$\begin{array}{l} \{\} :: \{\nu \mid \nu = \mathit{empty}\} \\ \texttt{has} :: d{:}\mathit{Dict} \to k{:}\mathit{Str} \to \{\nu \mid \mathit{Bool}(\nu) \wedge \nu = \texttt{true} \Leftrightarrow \mathit{has}(d,k)\} \\ \texttt{get} :: d{:}\mathit{Dict} \to k{:}\{\nu \mid \mathit{Str}(\nu) \wedge\ \mathit{has}(d,\nu)\} \to \{\nu \mid \nu = \mathit{sel}(d,k)\} \\ \texttt{set} :: d{:}\mathit{Dict} \to k{:}\mathit{Str} \to x{:}\mathit{Top} \to \{\nu \mid \nu = \mathit{upd}(d,k,x)\} \\ \texttt{keys} :: d{:}\mathit{Dict} \to \mathit{List}[\{\nu \mid \mathit{Str}(\nu) \wedge \mathit{has}(d,\nu)\}] \end{array}$$

The types above use the constant empty to denote the empty dictionary, and the predicate $\mathit{has}(d,k)$ to abbreviate $\mathit{sel}(d,k) \neq \mathit{bot}$. To relate two dictionaries, we use $\mathit{EqMod}(d_1,d_2,K)$ to abbreviate

$$\forall k'.\ (\wedge_{k \in K}\ k \neq k') \Rightarrow \mathit{sel}(d_1,k') = \mathit{sel}(d_2,k')$$

which states that the dictionaries d_1 and d_2 are identical *except* at the keys in K. This expansion falls into the array property fragment, shown to be decidable in [7] by reduction to an equisatisfiable quantifier-free formula. The EqMod abbreviation is particularly useful for dictionary updates where we do not know the *exact* value being stored, but do know some abstraction thereof, *e.g.* its type. For example, in `incCounter` (from section 2) we do not know what value is stored in the count field `c`, only that it is an integer. Thus, we say that the new dictionary is the same as the old except at `c`, where the binding is an integer. A more direct approach would be to use an existentially quantified variable to represent the stored value and say that the resulting dictionary is the original dictionary updated to contain this quantified value. Unfortunately, that would take the formulas outside the decidable fragment of the logic, thereby precluding SMT-based logical subtyping.

Standard Rules. We briefly identify several typing rules that are standard for lambda calculi with dependent refinements. T-VAR and T-VARPOLY assign types to variable expressions x. If x is bound to a (monomorphic) refinement type in Γ, then T-VAR assigns x the singleton type that says that the expression x evaluates to the same value as the variable x. T-IF assigns the type scheme S to an if-expression if the condition w is a boolean-valued expression, the then-branch expression e_1 has type scheme S under the assumption that w evaluates to `true`, and the else-branch expression e_2 has type scheme S under the assumption that w evaluates to `false`. The T-APP rule is standard, but notice that the arrow type of w_1 is nested inside a refinement type. In T-LET, the type scheme S_2 must be well-formed in Γ, which prevents the variable x from escaping its scope. T-SUB allows expression e to be used with type S if e has type S' and S' is a subtype of S.

Type Checking $\boxed{\Gamma \vdash e :: S}$

$$\frac{}{\Gamma \vdash c :: \mathit{ty}(c)}\ \text{[T-CONST]}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :: \{\nu \mid \nu = x\}}\ \text{[T-VAR]} \qquad \frac{\Gamma(x) = \forall A.\ S}{\Gamma \vdash x :: \forall A.\ S}\ \text{[T-VARPOLY]}$$

$$\frac{\Gamma \vdash w_1 :: \mathit{Dict} \qquad \Gamma \vdash w_2 :: \mathit{Str} \qquad \Gamma \vdash w_3 :: S}{\Gamma \vdash w_1 \texttt{ ++ } \{w_2 \mapsto w_3\} :: \{\nu \mid \nu = w_1 \texttt{ ++ } \{w_2 \mapsto w_3\}\}}\ \text{[T-EXTEND]}$$

$$\frac{\begin{array}{c}\Gamma \vdash w :: \mathit{Bool} \\ \Gamma, w = \texttt{true} \vdash e_1 :: S \qquad \Gamma, w = \texttt{false} \vdash e_2 :: S\end{array}}{\Gamma \vdash \text{if } w \text{ then } e_1 \text{ else } e_2 :: S}\ \text{[T-IF]}$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, x{:}T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.\ e :: \{\nu \mid \nu = \lambda x.\ e \wedge \nu :: x{:}T_1 \to T_2\}}\ \text{[T-FUN]}$$

$$\frac{\Gamma \vdash w_1 :: \{\nu \mid \nu :: x{:}T_{11} \to T_{12}\} \qquad \Gamma \vdash w_2 :: T_{11}}{\Gamma \vdash w_1\ w_2 :: T_{12}[w_2/x]}\ \text{[T-APP]}$$

$$\frac{A \notin \Gamma \qquad \Gamma, A \vdash e :: S}{\Gamma \vdash \lambda A.\ e :: \forall A.\ S}\ \text{[T-TFUN]}$$

$$\frac{\Gamma \vdash T \qquad \Gamma \vdash w :: \forall A.\ S}{\Gamma \vdash w\ [T] :: \mathsf{Inst}(S,A,T)}\ \text{[T-TAPP]}$$

$$\frac{\begin{array}{c}\forall i.\ \Gamma \vdash T_i \qquad \Psi(C) = [\overline{\theta A}]\{\overline{f{:}T'}\} \\ \forall j.\ \Gamma \vdash w_j :: \mathsf{Inst}(T'_j, \overline{A}, \overline{T})\end{array}}{\Gamma \vdash C(\overline{w}) :: \{\nu \mid \mathsf{Fold}(C, \overline{T}, \overline{w})\}}\ \text{[T-FOLD]}$$

$$\frac{\Gamma \vdash e :: \{\nu \mid \nu :: C[\overline{T}]\}}{\Gamma \vdash e :: \{\nu \mid \mathsf{Unfold}(C, \overline{T})\}}\ \text{[T-UNFOLD]}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash e_1 :: S_1 \qquad \Gamma, x{:}S_1 \vdash e_2 :: S_2 \qquad \Gamma \vdash S_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: S_2}\ \text{[T-LET]}$$

$$\frac{\Gamma \vdash e :: S' \qquad \Gamma \vdash S' \sqsubseteq S \qquad \Gamma \vdash S}{\Gamma \vdash e :: S}\ \text{[T-SUB]}$$

Figure 4. Type checking for System D

Type Instantiation. The T-TAPP rules uses the procedure Inst to instantiate a type variable with a (monomorphic) type. Inst is defined recursively on formulas, type terms, and types, where the only non-trivial case involves type predicates with type variables:

$$\begin{array}{c} \mathsf{Inst}(\mathit{lw} :: A, A, \{\nu \mid p\}) = p[\mathit{lw}/\nu] \\ \mathsf{Inst}(\mathit{lw} :: B, A, T) = \mathit{lw} :: B \end{array}$$

We write $\mathsf{Inst}(S, \overline{A}, \overline{T})$ to mean the result of applying Inst to S with the type variables and type arguments in succession.

Fold and Unfold. The T-FOLD rule is used for records of data created with the datatype constructor C and type arguments $\overline{T}$. The rule succeeds if the argument w_j provided for each field f_j has the required type T'_j after instantiating all type parameters $\overline{A}$ with the type arguments $\overline{T}$. If these conditions are satisfied, the formula

Subtyping $\boxed{\Gamma \vdash S_1 \sqsubseteq S_2}$

$$\frac{x \text{ fresh} \quad p_1' = p_1[x/\nu] \quad p_2' = p_2[x/\nu] \quad \mathsf{Normalize}(p_2') = \wedge_i(q_i \Rrightarrow r_i) \quad \forall i.\, \Gamma, p_1' \vdash q_i \Rrightarrow r_i}{\Gamma \vdash \{\nu \,|\, p_1\} \sqsubseteq \{\nu \,|\, p_2\}} \text{ [S-MONO]}$$

$$\frac{\Gamma \vdash S_1 \sqsubseteq S_2}{\Gamma \vdash \forall A.\, S_1 \sqsubseteq \forall A.\, S_2} \text{ [S-POLY]}$$

Clause Implication $\boxed{\Gamma \vdash q \Rrightarrow r}$

$$\frac{\mathsf{Valid}([\![\Gamma]\!] \wedge q \Rightarrow r)}{\Gamma \vdash q \Rrightarrow r} \text{ [C-VALID]}$$

$$\frac{\exists j. \quad \mathsf{Valid}([\![\Gamma]\!] \wedge q \Rightarrow lw_j :: U) \quad \Gamma, q \vdash U <: U_j}{\Gamma \vdash q \Rrightarrow \vee_i\, lw_i :: U_i} \text{ [C-IMPSYN]}$$

Syntactic Subtyping $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{\Gamma \vdash T_{21} \sqsubseteq T_{11} \quad \Gamma, x{:}T_{21} \vdash T_{12} \sqsubseteq T_{22}}{\Gamma \vdash x{:}T_{11} \to T_{12} <: x{:}T_{21} \to T_{22}} \text{ [U-ARROW]}$$

$$\frac{}{\Gamma \vdash A <: A} \text{ [U-VAR]} \qquad \frac{}{\Gamma \vdash \mathit{Null} <: C[\overline{T}]} \text{ [U-NULL]}$$

$$\frac{\Psi(C) = [\overline{\theta A}]\{\cdots\} \quad \forall i.\ \text{if } \theta_i \in \{+, =\} \text{ then } \Gamma \vdash T_{1i} \sqsubseteq T_{2i} \quad \forall i.\ \text{if } \theta_i \in \{-, =\} \text{ then } \Gamma \vdash T_{2i} \sqsubseteq T_{1i}}{\Gamma \vdash C[\overline{T_1}] <: C[\overline{T_2}]} \text{ [U-DATATYPE]}$$

Figure 5. Subtyping for System D

returned by $\mathsf{Fold}(C, \overline{T}, \overline{w})$, defined as

$$\nu \neq \texttt{null} \wedge tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge \nu :: C[\overline{T}] \wedge (\wedge_j\, sel(\nu, f_j) = w_j)$$

records that the value is non-null, that the values stored in the fields are precisely the values used to construct the record, and that the value has a type corresponding to the specific constructor used to create the value. T-UNFOLD exposes the fields of non-null constructed data as a dictionary, using $\mathsf{Unfold}(C, \overline{T})$, defined as

$$\nu \neq \texttt{null} \Rightarrow (tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge (\wedge_j [\![T_j'']\!](sel(\nu, f_j))))$$

where $\Psi(C) = [\overline{\theta A}]\{\overline{f{:}T'}\}$, $[\![\{\nu \,|\, p\}]\!](lw) \triangleq p[lw/\nu]$, and for all j, $T_j'' = \mathsf{Inst}(T_j', \overline{A}, \overline{T})$. For example, $\mathsf{Unfold}(\mathit{List}, \mathit{Int})$ is

$$\nu \neq \texttt{null} \Rightarrow (tag(\nu) = \text{“}\texttt{Dict}\text{”} \wedge tag(sel(\nu, \text{“}\texttt{hd}\text{”})) = \text{“}\texttt{Int}\text{”} \wedge sel(\nu, \text{“}\texttt{tl}\text{”}) :: \mathit{List}[\mathit{Int}])$$

4.3 Subtyping

In traditional refinement type systems, there is a two-level hierarchy between types and refinements that allows a syntax-directed reduction of subtyping obligations to SMT implications [16, 23, 31]. In contrast, System D's refinements include uninterpreted type predicates that are beyond the scope of (first-order) SMT solvers.

Let us consider the problem of establishing the subtyping judgment $\Gamma \vdash \{\nu \,|\, p_1\} \sqsubseteq \{\nu \,|\, p_2\}$. We cannot use the SMT query

$$[\![\Gamma]\!] \wedge p_1 \Rightarrow p_2 \tag{2}$$

as the presence of (uninterpreted) type-predicates may conservatively render the implication invalid. Instead, our strategy is to massage the refinements into a normal form that makes it easy to factor the implication in (2) into a collection of subgoals whose consequents are either simple (non-type) predicates or type predicates. The former can be established via SMT and the latter by recursively invoking syntactic subtyping. Next, we show how this strategy is realized by the rules in Figure 5.

Step 1: Split query into subgoals. We start by converting p_2 into a *normalized conjunction* $\wedge_i(q_i \Rrightarrow r_i)$. Each conjunct, or *clause*, $q_i \Rrightarrow r_i$ is normalized such that its consequent is a *disjunction* of type predicates. We use the symbol $\Rrightarrow$ instead of the usual implication arrow $\Rightarrow$ to emphasize the normal structure of each clause. By splitting p_2 into its normalized clauses, rule S-MONO reduces the goal (2) to the equivalent collection of subgoals

$$\forall i.\, \Gamma, p_1 \vdash q_i \Rrightarrow r_i$$

Step 2: Discharge subgoals. The normalization ensures that the consequent of each subgoal above is a disjunction of type predicates. When the disjunction of a clause is *empty*, the subgoal is

$$\textit{(“type predicate-free”)} \quad \Gamma, p_1 \vdash q_i \Rrightarrow \mathit{false}$$

which rule C-VALID handles by SMT. Otherwise, the subgoal is

$$\textit{(“type predicate”)} \quad \Gamma, p_1 \vdash q_i \Rrightarrow lw_j :: U_j$$

which rule C-IMPSYN handles via type extraction followed by a use of of syntactic subtyping. In particular, the rule tries to establish *one of* the disjuncts $lw_j :: U_j$, by searching for a type term U that occurs in Γ that 1) *flows to* lw_j, *i.e.* for which we can deduce via SMT that

$$[\![\Gamma]\!] \wedge p_1 \wedge q_i \Rightarrow lw_j :: U$$

is valid and, 2) is a syntactic subtype of U_j in an appropriately strengthened environment (written $\Gamma, p_1, q_i \vdash U <: U_j$). The rules U-DATATYPE and U-ARROW establish syntactic (refinement) subtyping, by (recursively) establishing that subtyping holds for the matching components [6, 16, 31]. Because syntactic subtyping recursively refers to subtyping, the S-MONO rule uses fresh variables to avoid duplicate bindings of ν in the environment.

Formula Normalization. Procedure Normalize converts a formula p into a conjunction of clauses $\wedge_i(q_i \Rrightarrow r_i)$ as described above. The conversion is carried out by translating p to conjunctive normal form (CNF), and then for each CNF clause, rearranging literals and adding negations as necessary. For example,

$$\mathsf{Normalize}(\nu = \texttt{null}) \triangleq \neg(\nu = \texttt{null}) \Rrightarrow \mathit{false}$$
$$\mathsf{Normalize}(\nu = \texttt{null} \vee \nu :: U) \triangleq \neg(\nu = \texttt{null}) \Rrightarrow \nu :: U$$

Formula Implication. In each SMT implication query $[\![\Gamma]\!] \wedge p \Rightarrow q$, the operator $[\![\cdot]\!]$ describes the embedding of environments and types into the logic as follows:

$$\begin{array}{rclrcl} [\![\{\nu \,|\, p\}]\!] & \triangleq & p & [\![\Gamma, x{:}T]\!] & \triangleq & [\![\Gamma]\!] \wedge [\![T]\!][x/\nu] \\ [\![\emptyset]\!] & \triangleq & \mathit{true} & [\![\Gamma, x{:}\forall A.\, S]\!] & \triangleq & [\![\Gamma]\!] \\ [\![\Gamma, p]\!] & \triangleq & [\![\Gamma]\!] \wedge p & [\![\Gamma, A]\!] & \triangleq & [\![\Gamma]\!] \end{array}$$

When embedding values into the logic, we represent each lambda by a distinct uninterpreted constant. Thus, function equality is “physical equality,” so there is no concern about the equivalence of expressions. (Note that lambdas never need to appear inside refinement formulas in source programs, and are included in the grammar of formulas just for the metatheory.)

Ensuring Termination. An important concern remains: as we extract type terms from the environment and recursively invoke the subtyping relation on them, we do not have the usual guarantee that subtyping is recursively invoked on strictly syntactically smaller terms. Thus, it is not clear whether subtyping checks will terminate.

Indeed, if we are not careful, they may not! Consider the environment

$$\Gamma \triangleq y : Top,\ x : \{\nu \mid \nu = y \wedge \nu :: U\}$$
$$\text{where } U \triangleq a{:}\{\nu \mid \nu :: b{:}\{\nu \mid \nu = y\} \to Top\} \to Top$$

and suppose we wish to check that

$$\Gamma \vdash true \Rrightarrow y :: x{:}\{\nu \mid \nu = y\} \to Top. \tag{3}$$

C-VALID cannot derive this judgment, since the implication

$$[\![\Gamma]\!] \wedge true \Rightarrow y :: x{:}\{\nu \mid \nu = y\} \to Top$$

is not valid. Thus, we must derive Equation 3 by C-IMPSYN. Type extraction derives that $y :: U$ in Γ, so the remaining obligation is

$$\Gamma \vdash U <: x{:}\{\nu \mid \nu = y\} \to Top.$$

Because of the contravariance of function subtyping on the left-hand side of the arrow, the following judgment must be derivable:

$$\Gamma \vdash \{\nu \mid \nu = y\} \sqsubseteq \{\nu \mid \nu :: b{:}\{\nu \mid \nu = y\} \to Top\}.$$

After SA-MONO substitutes a fresh variable, say ν', for ν in both types, this reduces to the clause implication obligation

$$\Gamma, \nu' = y \vdash true \Rrightarrow \nu' :: b{:}\{\nu \mid \nu = y\} \to Top.$$

Alas, this is essentially Equation 3, so we are stuck in an infinite loop! We will again extract the type U for y (aliased to ν' here) and repeat the process *ad inifinitum*.

This situation arises only if we are allowed to invoke the rule C-IMPSYN infinitely many times. In this case, C-IMPSYN extracts a single type term from the environment infinitely often, since there are only finitely many in the environment. We cut the loop with a modest restriction: along any branch of a subtyping derivation, we allow a type term to be extracted at most once. Since there are only finitely many type terms in the environment, this is enough to ensure termination. To implement this strategy, we augment the subtyping relations to take a set of "already-used" type terms as an additional parameter, which cannot be extracted by the rule C-IMPSYN. To keep the presentation in this paper simpler, we elide this restriction from the subtyping rules in Figure 5; the full definition can be found in a technical report [8]. The versions with and without this restriction may or may not coincide, but we are not particularly concerned with the outcome of this question because in our experience the kind of problematic subtyping obligation discussed in this section is a pathological corner case that does not arise in practice.

Recap. Recall that our goal is to typecheck programs which use value-indexed dictionaries which may contain functions as values. On the one hand, the theory of finite maps allows us to use logical refinements to express and verify complex invariants about the contents of dictionaries. On the other, without resorting to higher-order logic, such theories cannot express that a dictionary maps a key to a value of function type.

To resolve this tension, we introduced the novel concept of *nested refinements*, where types are nested into the logic as uninterpreted terms and the typing relation is nested as an uninterpreted predicate. The logical validity queries arising in typechecking are discharged by rearranging the formula in question into an implication between a purely logical formula and a disjunction of type predicates. This implication is discharged using a novel combination of logical queries, discharged by an SMT solver, and syntactic subtyping. This approach enables the efficient, automatic type checking of sophisticated dynamic language programs that manipulate complex data, including dictionaries which map keys to function values.

5. Soundness

At this point in the proceedings, it is customary to make a claim about the soundness of the type system by asserting that it enjoys the standard preservation and progress properties. Unfortunately, the presence of nested refinements means this route is unavailable to us, as the usual substitution property does not hold! Next, we describe why substitution is problematic and define a *stratified* system System D* for which we establish the preservation and progress properties. The soundness of System D follows, as it is a special case of the stratified System D*.

5.1 The Problems

The key insight in System D is that we can use uninterpreted functions to nest types inside refinements, thereby unlocking the door to expressive SMT-based reasoning for dynamic languages. However, this very strength precludes the usual substitution lemma upon which preservation proofs rest.

Substitution. The standard substitution property requires that if $x{:}S, \Gamma \vdash e :: S'$ and $\vdash w :: S$, then $\Gamma[w/x] \vdash e[w/x] :: S'[w/x]$. The following snippet shows why System D lacks this property:

```
let foo f = 0 in foo (fun x -> x + 1)
```

Suppose that we ascribe to `foo` the type

$$\texttt{foo} :: f{:}(Int \to Int) \to \{\nu \mid f :: Int \to Int\}.$$

The return type of the function states that its argument f is a function from integers to integers and does not impose any constraints on the return value itself. To check that `foo` does indeed have this type, by T-FUN, the following judgment must be derivable:

$$f{:}Int \to Int \vdash \texttt{0} :: \{\nu \mid f :: Int \to Int\} \tag{4}$$

By T-CONST, T-SUB, S-MONO and C-VALID the judgment reduces to the implication

$$true \wedge f :: Int \to Int \wedge [\![ty(\texttt{0})]\!][\texttt{0}/\nu] \Rightarrow f :: Int \to Int.$$

which is trivially valid, thereby deriving (4), and showing that `foo` does indeed have the ascribed type.

Next, consider the call to `foo`. By T-APP, the result has type

$$\{\nu \mid \texttt{(fun x -> x + 1)} :: Int \to Int\}.$$

The expression `foo (fun x -> x + 1)` evaluates in one step to `0`. Thus, if the substitution property is to hold, `0` should also have the above type. In other words, System D must be able to derive

$$\vdash \texttt{0} :: \{\nu \mid \texttt{(fun x -> x + 1)} :: Int \to Int\}.$$

By T-CONST, T-SUB, S-MONO, and C-VALID, the judgment reduces to the implication

$$true \wedge [\![ty(\texttt{0})]\!][\texttt{0}/\nu] \Rightarrow \texttt{(fun x -> x + 1)} :: Int \to Int \tag{5}$$

which is *invalid* as type predicates are *uninterpreted* in our refinement logic! Thus, the call to `foo` and the reduced value do not have the same type in System D, which illustrates the crux of the problem: the C-VALID rule is not closed under substitution.

Circularity. Thus, it is clear that the substitution lemma will require that we define an interpretation for type predicates. As a first attempt, we can define an interpretation $\mathcal{I}$ that interprets type predicates involving arrows as:

$$\mathcal{I} \models \lambda x.\, e :: x{:}T_1 \to T_2 \quad \textit{iff} \quad x{:}T_1 \vdash e :: T_2.$$

Next, let us replace C-VALID with the following rule that restricts the antecedent to the above interpretation:

$$\frac{\mathcal{I} \models [\![\Gamma]\!] \wedge p \Rightarrow q}{\Gamma \vdash p \Rrightarrow q}\ \text{[C-VALID-INTERPRETED]}$$

Notice that the new rule requires the implication be valid in the particular interpretation $\mathcal{I}$ instead of in all interpretations. This allows the logic to "hook back" into the type system to derive types for closed lambda expressions, thereby discharging the problematic implication query in (5). While the rule solves the problem with substitution, it does not take us safely to the shore — it introduces a circular dependence between the typing judgments and the interpretation $\mathcal{I}$. Since our refinement logic includes negation, the type system corresponding to the set of rules outlined earlier combined with C-VALID-INTERPRETED is not necessarily well-defined.

5.2 The Solution: Stratified System D*

Thus, to prove soundness, we require a well-founded means of interpreting type predicates. We achieve this by *stratifying* the interpretations and type derivations, requiring that type derivations at each level refer to interpretations at the same level, and that interpretations at each level refer to derivations at strictly lower levels. Next, we formalize this intuition and state the important lemmas and theorems. The full proofs may be found in a technical report [8].

Formally, we make the following changes. First, we index typing judgments ($\vdash_n$) and interpretations ($\mathcal{I}_n$) with a natural number n. We call these the level-n judgments and interpretations, respectively. Second, we allow level-n judgments to use the rule

$$\frac{\mathcal{I}_n \models [\![\Gamma]\!] \wedge p \Rightarrow q}{\Gamma \vdash_n p \Rrightarrow q} \text{ [C-VALID-N]}$$

and the level-n interpretations to use lower-level type derivations:

$$\mathcal{I}_n \models \lambda x.\, e :: x{:}T_1 \to T_2 \quad \textit{iff} \quad x{:}T_1 \vdash_{n-1} e :: T_2.$$

Finally, we write

$$\Gamma \vdash_* e :: S \quad \textit{iff} \quad \exists n.\, \Gamma \vdash_n e :: S.$$

The derivations in System D* consist of the derivations at all levels. The following "lifting" lemma states that the derivations at each level include the derivations at all lower levels:

Lemma (Lifting Derivations)**.**

1. *If* $\Gamma \vdash e :: S$*, then* $\Gamma \vdash_* e :: S$.
2. *If* $\Gamma \vdash_n e :: S$*, then* $\Gamma \vdash_{n+1} e :: S$.

The first clause holds since the original System D derivations cannot use the C-VALID-N rule, *i.e.* $\Gamma \vdash e :: S$ exactly when $\Gamma \vdash_0 e :: S$. The second clause follows from the definitions of $\vdash_n$ and $\mathcal{I}_n$. Stratification snaps the circularity knot and enables the proof of the following stratified substitution lemma:

Lemma (Stratified Substitution)**.**
If $x{:}S, \Gamma \vdash_n e :: S'$ *and* $\vdash_n w :: S$*,*
then $\Gamma[w/x] \vdash_{n+1} e[w/x] :: S'[w/x]$.

The proof of the above depends on the following lemma, which captures the connection between our typing rules and the logical interpretation of formulas in our refinement logic:

Lemma (Satisfiable Typing)**.**
If $\vdash_n w :: T$*, then* $\mathcal{I}_{n+1} \models [\![T]\!][w/x]$.

Stratified substitution enables the following preservation result:

Theorem (Stratified Preservation)**.**
If $\vdash_n e :: S$*, and* $e \hookrightarrow e'$ *then* $\vdash_{n+1} e' :: S$.

From this, and a separate progress result, we establish the type soundness of System D*:

Theorem (System D* Type Soundness)**.**
If $\vdash_* e :: S$*, then either* e *is a value or* $e \hookrightarrow e'$ *and* $\vdash_* e' :: S$.

By coupling this with Lifting, we obtain the soundness of System D as a corollary.

6. Algorithmic Typing

Having established the expressiveness and soundness of System D, we establish its practicality by implementing a type checker and applying it to several interesting examples. The declarative rules for type checking System D programs, shown in section 4, are not syntax-directed and thus unsuitable for implementation. We highlight the problematic rules and sketch an *algorithmic* version of the type system that also performs local type inference [29]. Our prototype implementation [8] verifies all of the examples in this paper and in [39], using Z3 [12] to discharge SMT obligations. A more detailed discussion of the algorithmic system may be found in a technical report [8].

6.1 Algorithmic Subtyping

Nearly all the declarative subtyping rules presented in Figure 5 are non-overlapping and directed by the structure of the judgment being derived. The sole exception is C-IMPSYN, whose first premise requires us to synthesize a type term U such that the SMT solver can prove $lw_j :: U$ for some j, where U is used in the second premise. We note that, since type predicates are uninterpreted, the only type terms U that can satisfy this criterion must come from the environment Γ. Thus, we define a procedure $\mathsf{MustFlow}(\Gamma, T)$ that uses the SMT solver to compute the set of type terms U', out of *all* possible type terms mentioned in Γ, such that for all values x, $x{:}T$ implies that $x :: U'$. To implement C-IMPSYN, we call $\mathsf{MustFlow}(\Gamma, \{\nu \mid \nu = lw_j\})$ to compute the set $\mathcal{U}$ of type terms that might be needed by the second premise. Since the declarative rule cannot possibly refer to a type term U not in Γ, this strategy guarantees that $U \in \mathcal{U}$ and, thus, does not forfeit precision.

6.2 Bidirectional Type Checking

We extend the syntax of System D with optional type annotations for binding constructs and constructed data, and, following work on local type inference [29], we define a *bidirectional* type checking algorithm. In the remainder of this section, we highlight the novel aspects of our bidirectional type system.

Function Applications. To typecheck an application $w_1\ w_2$, we must synthesize a type T_1 for the function w_1 and use type extraction to convert T_1 to a syntactic arrow. Since the procedure $\mathsf{MustFlow}$ can return an arbitrary number of type terms, we must decide how to proceed in the event that T_1 can be extracted to multiple different arrow types. To avoid the need for backtracking in the type checker, and to provide a semantics that is simple for the programmer, we synthesize a type for w_1 only if there is *exactly one* syntactic arrow that is applicable to the given argument w_2.

Remaining Rules. We will now briefly summarize some of the other algorithmic rules presented in a technical report [8]. Uses of T-SUB can be factored into other typing rules. However, uses of T-UNFOLD cannot, since we cannot syntactically predict where it is needed. Since we do not have pattern matching to determine exactly when to unfold type definitions, as in languages like ML, we eagerly unfold type definitions to anticipate all situations in which unfolding might be required. For let-expressions, to handle the fact that synthesized types might refer to variables that are about to go out of scope, making them ill-formed, we use several simple heuristics to eliminate occurrences of local variables. In all of the examples we have tested, the annotations provided on top-level let-bindings are sufficient to allow synthesizing well-formed types for all unannotated inner let-expressions. Precise types are synthesized for if-expressions by synthesizing the types of both

branches, guarding them by the appropriate branch conditions, and conjoining them. For constructed data expressions, we allow the programmer to provide hints in type definitions that help the type checker decide how to infer type parameters that are omitted. For example, suppose the $List$ definition is updated as follows:

$$\text{type } List[+A]\{\text{“}\texttt{hd}\text{”} : \{\nu \mid \nu :: A\}; \text{“}\texttt{tl}\text{”} : \{\nu \mid \nu :: List[*A]\}\}$$

Due to the presence of the marker $*$ in the type of the “`tl`” field, local type inference will use the type of w_2 to infer the omitted type parameter in $List(w_1, w_2)$. Finally, although the techniques in [29] would allow us to, for simplicity we do not attempt to synthesize parameters to type functions.

Soundness. We write $\Gamma \vdash e \lhd S$ for the algorithmic type checking judgment, which verifies e against the given type S, and $\Gamma \vdash e \rhd S$ for the algorithmic type synthesis judgment, which produces a type S for expression e. Each of the techniques employed in this section are sound with respect to the declarative system, so we can show the following property, where we use a procedure erase to remove type annotations from functions, let-bindings, and constructed data because the syntax of the declarative system does not permit them:

Proposition (Sound Algorithmic Typing)**.**
If $\Gamma \vdash e \rhd S$ *or* $\Gamma \vdash e \lhd S$, *then* $\Gamma \vdash \mathsf{erase}(e) :: S$.

7. Related Work

In this section, we highlight related approaches to statically verifying features of dynamic languages. For a thorough introduction to contract-based and other hybrid approaches, see [15, 23, 34].

Dynamic Unions and Control Flow. Among the earliest attempts at mixing static and dynamic typing was adding the special type `dynamic` to a statically-typed language like ML [1]. In this approach, an arbitrary value can be injected into `dynamic`, and a typecase construct allows inspecting its precise type at run-time. However, one cannot guarantee that a particular `dynamic` value is of one of a subset of types (cf. `negate` from section 2). Several researchers have used union types and tag-test sensitive control-flow analyses to support such idioms. Most recently, λ_{TR} [39] and λ_S [18] feature values of (untagged) union types that can be used at more precise types based on control flow. In the former, each expression is assigned two propositional formulas that hold when the expression evaluates to either true or false; these propositions are strengthened by recording the guard of an if-expression in the typing environment when typing its branches. Typechecking proceeds by solving propositional constraints to compute, for each value at each program point, the set of tags it may correspond to. The latter shows how a similar strategy can be developed in an imperative setting, by coupling a type system with a data flow analysis. However, both systems are limited to ad-hoc unions over basic and function values. In contrast, System D shows how, by pushing all the information about the value (resp. reasoning about flow) into expressive, but decidable refinement predicates (resp. into SMT solvers), one can statically reason about significantly richer idioms (related tags, dynamic dictionaries, polymorphism, *etc.*).

Records and Objects. There is a large body of work on type systems for objects [22, 28]. Several early advances incorporate records into ML [30], but the use of records in these systems is unfortunately unlikely to be flexible enough for dynamic dictionaries. In particular, record types cannot be joined when they disagree on the type of a common field, which is crucially enabled by the use of the theory of finite maps in our setting. Recent work includes type systems for JavaScript and Ruby. [3] presents a rich type system and inference algorithm for JavaScript, which uses row-types and width subtyping to model dictionaries (objects). The system does not support unions, and uses fixed field names. This issue is addressed in [38], which models dictionaries using row types labeled by singletons indexed by string constants, and depth subtyping. A recent proposal [41] incorporates an initialization phase during which object types can be updated. However, these systems preclude truly dynamic dictionaries, which require dependent types, and moreover lack the control flow analysis required to support ad-hoc unions. DRuby [17] is a powerful type system designed to support Ruby code that mixes intersections, unions, classes, and parametric polymorphism. DRuby supports “duck typing,” by converting from nominal to structural types appropriately. However, it does not support ad-hoc unions or dynamic dictionary accesses.

Dependent Types for First-Order Programs. The observation that ad-hoc unions can be checked via dependent types is not new. [24] develops a dependent type system called guarded types that is used to describe records and ad-hoc unions in legacy Cobol programs that make extensive use of tag-tests, where the “tag” is simply the first few bytes of a structure. [21] presents an SMT-based system for statically inferring dependent types that verify the safety of ad-hoc unions in legacy C programs. [10] describes how type-checking and property verification are two sides of the same coin for C (which is essentially uni-typed.) It develops a precise logic-based type system for C and shows how SMT solvers can be used for type-checking. This system contains a `hastype(x,T)` which is similar to ours except that `T` ranges over a fixed set of type constants as opposed to arbitrary types. Thus, one cannot use their `hastype` to talk about complex values (*e.g.* dependent functions, duck-typed records with only some fields) nested within dictionaries in their system. Finally, the system supports function pointers but does not fully support higher-order functions. Dminor [6] uses refinement types to formalize similar ideas in a first-order functional data description language with fixed-key records and run-time tag-tests. The authors show how unions and intersections can be expressed in refinements (and even collections, via recursive functions), and hence how SMT solvers can wholly discharge all subtyping obligations. However, the above techniques apply only to first-order languages, with static keys and dictionaries over base values.

Refinement Types for Higher-Order Programs. The key novelty of System D is the introduction of *nested* refinement types, which are a generalization of the refinement types introduced by the long line of work pioneered by Xi and Pfenning [40] and further studied in [4, 11, 14, 23, 31, 35]. The main difficulty in applying these classical refinement type systems to dynamic languages is that they require a distinction between base values that are typed with refinement predicates and complex values that are typed with syntactic constructors. In particular, dynamic languages contain dependent dictionaries, which require refinements (over the theory of arrays) to describe keys but syntactic types to describe the values bound to keys. This combination is impossible with earlier refinement types systems but is enabled by nesting types within refinements.

Combining Decision Procedures. Our approach of combining logical reasoning by SMT solvers and syntactic reasoning by subtyping is reminiscent of work on combining decision procedures [26, 33]. However, such techniques require the theories being combined to be disjoint; since our logic includes type terms which themselves contain arbitrary terms, our theory of syntactic types cannot be separated from the other theories in our system, so these techniques cannot be directly applied.

8. Conclusions and Future Work

We have shown how, by nesting type predicates within refinement formulas and carefully interleaving syntactic- and SMT-based subtyping, System D can statically type check dynamic programs that

manipulate dictionaries, polymorphic higher-order functions and containers. Thus, we believe that System D can be a foundation for two distinct avenues of research: the addition of heterogeneous dictionaries to static languages like C#, Java, OCaml and Haskell, or dually, the addition of expressive static typing to dynamic languages like Clojure, JavaScript, Racket, and Ruby.

We anticipate several concrete lines of work that are needed to realize the above goals. First, we need to add support for references and imperative update, features common to most popular dynamic languages. Since every dictionary operation in an imperative language goes through a reference, we will need to extend the type system with flow-sensitive analyses, as in [32] and [18], to precisely track the values stored in reference cells at each program point. Furthermore, to precisely track updates to dictionaries in the imperative setting, we will likely need to introduce some flow-sensitivity to the type system itself, adopting strong update techniques as in [19] and [41]. Second, our system treats strings as atomic constants. Instead, it should be possible to incorporate modern decision procedures for strings [20] to support logical operations on keys, which would give even more precise support for reflective metaprogramming. Third, we plan to extend our local inference techniques to automatically derive polymorphic instantiations [29] and use Liquid Types [31] to globally infer refinement types. Finally, for dynamic languages, it would be useful to incorporate some form of staged analysis to support dynamic code generation [2, 9].

Acknowledgements. The authors wish to thank Jeff Foster, Ming Kawaguchi, Sorin Lerner, Todd Millstein, Zachary Tatlock, David Walker, and the anonymous reviewers for their detailed feedback on drafts of this paper.

References

[1] M. Abadi, L. Cardelli, B. C. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL*, 1989.

[2] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *POPL*, 2011.

[3] C. Anderson, S. Drossopoulou, and P. Giannini. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, June 2005.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[5] Y. Bertot and P. Castéran. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions, 2004.

[6] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an smt solver. In *ICFP*, 2010.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[8] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. `http://arxiv.org/abs/1103.5055v2`.

[9] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of PLDI 2009*, pages 50–62, 2009.

[10] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, 2009.

[11] R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[13] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.

[14] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[15] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[16] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[17] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.

[18] A. Guha, C. Softoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *ESOP*, 2011.

[19] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP*, pages 200–224, 2010.

[20] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI*, pages 248–262, 2011.

[21] R. Jhala, R. Majumdar, and R.-G. Xu. State of the union: Type inference via craig interpolation. In *TACAS*, 2007.

[22] A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *FOOL-WOOD*, 2007.

[23] K. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2), 2010.

[24] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *TACAS*, pages 157–173, 2005.

[25] J. McCarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[26] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1979.

[27] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[28] J. Palsberg and M. I. Schwartzbach. *OO Type Systems*. Wiley, 1994.

[29] B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.

[30] D. Rémy. Type checking records and variants in a natural extension of ml. In *POPL*, 1989.

[31] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[32] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, pages 131–144, 2010.

[33] R. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[34] J. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[35] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP*, 2010.

[36] The Dojo Foundation. Dojo toolkit. `http://dojotoolkit.org/`.

[37] The Python Software Foundation. Python 3.2 standard library. `http://python.org/`.

[38] P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.

[39] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, pages 117–128, 2010.

[40] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

[41] T. Zhao. Type inference for scripting languages with implicit extension. In *FOOL*, 2010.