UNIVERSITY OF CALIFORNIA, SAN DIEGO

Nested Refinement Types for JavaScript

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

in

Computer Science

by

Ravi Chugh

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel R. Buss
Professor Alin Deutsch
Professor Cormac Flanagan
Professor Sorin Lerner

2013

The Dissertation of Ravi Chugh is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2013

EPIGRAPH

No grammatical rules have sufficient authority
to control the firm and established usage of language.

Joseph M. Williams
"Style: Ten Lessons in Clarity and Grace"

TABLE OF CONTENTS

LIST OF FIGURES

PREFACE

The language considered in this work is essentially a small, untyped programming language. The firm and established usage of this language is taken to be the common patterns in so-called dynamic languages. And the grammatical rules proposed, typing rules, are intended not to reject large subsets of established usage and, hence, wield undue authority.

ACKNOWLEDGEMENTS

I have been extremely fortunate to meet many great friends and colleagues over the last six years.

I am hugely indebted to my advisor, Ranjit Jhala, for his guidance and support in research matters and otherwise. His incisive questions and insights, combined with his enthusiasm and good nature, have always helped lead me to work on exciting, fulfilling, and productive projects. Many thanks to Sorin Lerner, who I was lucky to work with and who was always quick to offer encouragement. Together, Sorin and Ranjit set the tone for an energetic programming languages group at UCSD. I also had the pleasure of working with Nikhil Swamy and Dave Herman, who mentored me during fun and productive internships. Shriram Krishnamurthi has been generous with his time and advice, and I have thoroughly enjoyed the spirited discussions with him about both research and teaching.

I am grateful to have been a part of the CSE department here, which has been filled with wonderful people. Thanks to Jan Voung, for his quick wit and good cheer. To Pat Rondon for sharing his wisdom and humor. Collaborating with Pat on the first part of this dissertation was tremendously fun and provided a jolt of excitement at just the right moment in those late-middle years. To Zach Tatlock for fueling countless discussions and debates. To Panos Vekris for bravely battling with DJS to help build on the work on this dissertation. To Jason Oberg for introducing me to pour-over coffee and for his persistence in getting me to play more golf. To Alden King for introducing me to sour beer. And to many other friends, officemates, and foosball, tennis, ultimate, and swimming companions over the years.

Thanks to many of my favorite coffee shops in and around San Diego, Seattle, Minneapolis, San Francisco, and Mountain View for stimulating plenty of ideas — not all of which were completely bad — with tasty coffee and lively atmosphere.

Lastly, I am deeply grateful to my mother, father, and brother for always demonstrating the virtues of hard work and determination. And to Hanna for providing me with an endless source of support, inspiration, and joy.

Prior Publications. The dissertation author was the principal investigator on several publications, listed below, that have been adapted in this dissertation. The following publications contain material that has been incorporated into Chapter 2, Chapter 3, and Chapter 8.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Dependent Dynamic Dictionaries.** `arXiv:1103.5055v1 [cs.PL]`, March 2011.
- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing (Technical Appendix).** `arXiv:1103.5055v2 [cs.PL]`, September 2011.
- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing.** In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2012.

The following publications contain material that has been incorporated into Chapter 5, Chapter 6, Chapter 7, and Chapter 8.

- Ravi Chugh and Ranjit Jhala. **Dependent Types for JavaScript.** `arXiv:1112.4106v1 [cs.PL]`, December 2011.
- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript (Technical Appendix).** `arXiv:1112.4106v3 [cs.PL]`, August 2012.
- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

VITA

2007	Bachelor of Science, University of Pennsylvania
2007	Master of Science, University of Pennsylvania
2013	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing**. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2012.

Juan Chen, Ravi Chugh, and Nikhil Swamy. **Type-preserving Compilation for End-to-end Verification of Security Enforcement.** In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

Nikhil Swamy, Juan Chen, and Ravi Chugh. **Enforcing Stateful Authorization and Information Flow Policies in Fine.** In *Proceedings of the European Symposium on Programming (ESOP)*, Paphos, Cyprus, March 2010.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. **Staged Information Flow for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. **Dataflow Analysis for Concurrent Programs using Datarace Detection.** In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, June 2008.

ABSTRACT OF THE DISSERTATION

Nested Refinement Types for JavaScript

by

Ravi Chugh

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor Ranjit Jhala, Chair

Decades of research on type systems have led to advances in the kinds of programming features that can be reasoned about and the kinds of errors that can be prevented. Nevertheless, the programming idioms in untyped, or dynamic, languages make heavy use of features — run-time type tests, mutable objects indexed by dynamically computed keys, prototype inheritance, and higher-order functions — that are beyond the reach of prior type systems. Because of the rising popularity of languages like JavaScript and Python, as well as the addition of dynamic features to statically typed languages, techniques for reasoning about untyped languages are potentially very valuable.

In this dissertation, we present Dependent JavaScript, a statically typed dialect of the imperative, object-oriented, dynamic language. Our system builds on *refinement types*, which augment traditional *syntactic types* with logical predicates to enable finer-grained reasoning. We make several contributions to overcome limitations in prior refinement type systems that prevent precise reasoning about idiomatic code in dynamic languages. First, we present *nested refinement types*, where the typing relation is itself a predicate in the refinement logic in order to specify types for programs that manipulate objects with dynamically computed keys. By carefully coordinating SMT-based logical implication with syntactic techniques, nested refinement subtyping enables reasoning about such programs without resorting to a complex refinement logic. Second, we present a formulation of *flow-sensitive refinement types* to retain the precision of refinement types despite the presence of mutable variables. Finally, we present a refinement type encoding called *heap unrolling* to reason about the semantics of prototype-based inheritance, again, without resorting to a complex refinement logic. To demonstrate how these novel techniques combine, we implement a type checker for Dependent JavaScript and evaluate its expressiveness on a set of small, but challenging, JavaScript examples adapted from several benchmarks.

Part I

Prologue

Chapter 1

Introduction

Writing programs that do not crash at run-time and, furthermore, behave as intended is difficult in any language. Type systems are designed to rule out the possibility of certain run-time errors — such as "field-not-found errors" and "null-pointer exceptions" — from occurring. Decades of research on type systems have produced numerous advances in terms of the kinds of programming features that can be reasoned about and the kinds of errors that can be prevented. Many of these techniques have made tremendous impact on widely-used languages such as C++, Java, and C#, as well as on less popular but cutting-edge languages such as Haskell. Despite this success, untyped, or so-called dynamic, languages like LISP, Python, and JavaScript have always been popular, increasingly so in the last two decades, fueling endless debates about "catching simple bugs early" in languages with type systems or "rapidly prototyping and innovating" in those without them. But one irrefutable aspect of this debate is that current type systems are unable to reason about programs in dynamic languages with sufficient precision and usability.

In this dissertation, we carve out a new point in the design space of type systems that enables safety verification for a larger class of programming styles than in existing approaches. We set the stage for our work, in this Introduction, as follows. First, we review a set of features that are ubiquitous in programming languages. Next, we provide some background on techniques in existing type systems that rule out classes of run-time errors in programs that use these features. Then, we describe how idiomatic programs in popular, modern, dynamic languages like JavaScript, Python, and Ruby fall beyond the reasoning capabilities of existing type systems. Finally, we outline our novel techniques that, although general, are particularly well-suited to reason about these challenging idioms.

1.1 An Untyped λ-calculus

Many widely-used programming languages feature functions, objects, and pointers. Although languages often differ on the concrete syntax and sometimes on the semantics for programs that manipulate these kinds of values, it is useful to study these "core" features in a way that is largely independent of a particular manifestation.

In seminal work, Alonzo Church [18, 20] introduced the λ-calculus (pronounced "lambda-calculus") as a minimal, elegant, and general formulation of the nature of computation, where the only the mechanisms are functions (also called "lambdas") and function application. All other programming constructs can be encoded in terms of the pure λ-calculus. Although equal in expressive power to Turing machines, the λ-calculus offers a higher level of abstraction for specifying and executing computation than the relatively low-level nature of Turing machines. The λ-calculus, however, is too low-level still for structuring complex, human-readable code, so researchers have enriched the λ-calculus with additional features, like the ones mentioned above, rather than relying on their encodings in the pure λ-calculus. These extensions allow the "essence" of full-fledged languages to be modeled faithfully, giving traction to humans and tools that interact with these programs, while avoiding overly-specific and unnecessary details.

We define the (abstract) syntax of a λ-calculus extended with records (*i.e.* objects) and references (*i.e.* pointers) in Figure 1.1, following the style of standard presentations (*e.g.* [77]), to model a set of core features found in many widely-used languages. We will describe, informally, the semantics (often referred to as "dynamic semantics") usually associated with these constructs, with a focus on the kinds of errors that may arise at run-time. To set up our discussion of type systems that rule out classes of errors, we will make a distinction between *simple errors*, or *stuck states*, and more complex errors, which we will refer to as *exceptions*. Unless otherwise noted, we assume that a language implementation terminates with an error message when a program encounters a stuck state or an exception.

Base Values and Primitive Functions. Programming languages typically provide a set of base values — like numbers n, booleans b, and strings s — and a set of primitive functions c that operate on them. For example: the primitive function `&&` may compute the boolean negation of two boolean arguments or become *stuck* if one of the arguments is not a boolean; the primitive function + may compute the sum of two numeric arguments or become stuck if one of the arguments is not a number; and the primitive function ++ may compute the concatenation of two strings of become stuck if one of the arguments is not a string.

Instead of providing disjoint operations for different kinds of values, a language might *overload* some operators. For example, instead of the semantics for + above, the primitive function may be defined to operate on either pairs of numbers or strings, computing addition in the former case and concatenation in the latter, and result in a stuck state for all other pairs of arguments.

Further still, a language may incorporate *automatic*, or *implicit*, *conversion* to provide yet more flexible semantics for primitive functions. For example, + might be defined to compute the sum of two numbers, the concatenation of two strings, and otherwise perform the concatenation of the string representation of the two arguments, assuming the existence of another primitive `toString` that produces a string representation for any argument. Under this semantics, calling

v	$::=$		**Values**
	$\mid$	$n \mid b \mid s$	base value (number, boolean, string, *etc.*)
	$\mid$	c	primitive function (`==`, `&&`, `+`, *etc.*)
	$\mid$	$\lambda x.e$	function
	$\mid$	$\{\overline{s} = \overline{v}\}$	record (a.k.a. struct)
	$\mid$	r	reference (a.k.a. pointer)
e	$::=$		**Expressions**
	$\mid$	v	value
	$\mid$	x	variable
	$\mid$	$e_1\ e_2$	function application
	$\mid$	$\{\}$	empty record
	$\mid$	$\{e_1\ \texttt{with}\ e_2 = e_3\}$	record update (a.k.a. field write)
	$\mid$	$e[e']$	record projection (a.k.a. field read)
	$\mid$	$\texttt{ref}\ e$	heap reference allocation
	$\mid$	$\texttt{deref}\ e$	dereference (a.k.a. heap read)
	$\mid$	$\texttt{setref}\ e_1\ e_2$	set-reference (a.k.a. heap assignment)
	$\mid$	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	if-expression

Figure 1.1. An Untyped λ-calculus with References and Records

the + function would never lead to a stuck state.

In addition to the ways in which primitive functions may lead to simple errors (*i.e.* become stuck), sometimes exceptions may be raised. For example, the division function / is usually defined to divide two numbers except when the second is zero, in which case a "divide-by-zero" exception is raised. In some languages, like C, this exception is *unchecked* and does not terminate the program. The nonsensical output can lead to subsequent errors, however, so the choice to silently proceed with evaluation can make it hard to reason about the root cause of a run-time error.

User-Defined Functions. Function definitions allow the programmer to build computations on top of the base values and primitive functions provided by the language. In *higher-order languages*, function definitions may appear arbitrarily in programs — as in the core language of Figure 1.1 as well as in full-fledged languages like LISP, OCaml, and Haskell — and are referred to as "first-class" citizens of the language, in the sense that functions can be passed to and returned from other functions and stored in data structures. *First-order languages* like C, on the other hand, restrict the syntax of programs to be a sequence of top-level function definitions, including a "main" function that serves as the entry-point, which themselves do not contain any nested function definitions. In either case, a lambda $\lambda x.e$ is used to describe a function that takes a single argument (without loss of generality) named by the *formal parameter* x which the function body (the return expression) e may refer to.

The function application expression $e_1\ e_2$, used to call primitive and user-defined functions, proceeds in one of three ways depending on how e_1 evaluates. If e_1 evaluates to a primitive function c, the application expression evaluates according to the semantics of the primitive function, as described before. If e_1 evaluates to a lambda $\lambda x.e$ and e_2 evaluates to some value v, the function body e is evaluated along with a mapping from the variable x to v. The subsequent evaluation produces v when x is referred to. When trying to evaluate a variable that has not been defined by (*i.e.* is not the formal parameter of) any enclosing function definition, evaluation becomes stuck with a "variable-not-found" error. If e_1 evaluates to anything other than a primitive or user-defined function, then evaluation becomes stuck.

Records. Programming languages provides facilities to build up complex data structures composed of base values and other data structures. One such mechanism, which we define in Figure 1.1, is a *record* that maps a sequence $\overline{s}$ of string *keys*, or *fields*, s_1 through s_n to values v_1 through v_n, respectively. Variations of records, also called structures or objects in different settings, appear in a wide variety of languages, from purely functional languages like OCaml and Haskell to imperative languages like C to object-oriented languages like C#, Java, and JavaScript. Values are retrieved from records using the projection expression $e[e']$: if e evaluates to a record, e' evaluates to a string s, and the record has an s key, then the value of that binding is retrieved; otherwise, evaluation becomes stuck with a "field-not-found error." The record update expression $\{e_1 \texttt{ with } e_2 = e_3\}$ evaluates e_1 to a record, evaluates e_2 to a string s, evaluates e_3 to a value v, and produces a new record value that is just like the old one except that it maps s to v, rather than the binding of s in the original record, if any.

Many languages with records provide an *inheritance* mechanism that allows records to implicilty inherit keys from a parent record. For example, one way to encode inheritance is to equip record values with a special key "`parent`" that is intended to store a parent record. Then, when evaluating the projection $e[e']$, where e evaluates to a record and e' to a string s that is not bound by the record, rather than becoming stuck evaluation would recursively look for s in the record stored in its "`parent`" field. If no such "`parent`" key exists, then the projection would finally become stuck with a "field-not-found error."

Although arrays are often provided as a separate data structure mechanism, we can encode arrays of length $n+1$ as records that bind the string keys "`0`" through "`n`". Typically, a language with arrays raises an "array-out-of-bounds exception" when trying to retrieve (the string representation of) an integer key that is either negative or beyond the length of the array.

References. For better or for worse, many widely-used languages are *imperative* with a *heap* that is kept “off to the side” during execution that maps *references* r (sometimes called pointers, addresses, or locations) to values. The heap can be thought of as a global record of references that is visible throughout the entire program. The reference creation, or allocation, expression `ref` e creates a new reference r in the heap and initializes its contents to whatever e evaluates to. The dereference expression `deref` e evaluates e to a reference r and retrieves the value stored in the heap at r. If e does not evaluate to a reference, evaluation becomes stuck. The set-reference, or assignment, expression `setref` e_1 e_2 evaluates e_1 to a reference r, which must already be bound by the heap, and updates the corresponding heap binding to whatever e_2 evaluates to. The purpose of the assignment is to perform a “side-effect” on the heap, so the value produced by the assignment is of no consequence and is usually some dummy value. For example, using syntax typical of some imperative languages, the assignment `x := x + 1` to a mutable variable `x` corresponds to the expression `setref x (deref x + 1)` in our core language.

In *memory-unsafe* languages like C, “pointers” are simply integers, rather than a separate kind of value, and can be manipulated in all the ways that arbitrary integers can. Because pointers are used to index into structs and arrays, exceptions like “field-not-found” and “array-out-of-bounds” are unchecked and, hence, can lead to errors that are hard to reproduce and hard to diagnose.

Additional Constructs. The last kind of expression we define in Figure 1.1 is the expression `if` e_1 `then` e_2 `else` e_3 that directs control flow. If the *guard* expression e_1 evaluates to the boolean `true`, then e_2 is evaluated. If e_1 evaluates to `false`, then e_3 is evaluated. If e_1 evaluates to any other value, evaluation becomes stuck. Some languages have a facility to automatically convert arbitrary values to booleans, in which case arbitrary expressions may be used as guards.

Many more features found in practical programming languages — including local variables, recursion, iteration, user-defined exceptions, non-local control flow, and threads, among many others — have been formulated as extensions to the λ-calculus; several textbooks, including [77] and [54], provide thorough introductions to many of these approaches. The semantics of these features, as well as their implications for type system design, are largely orthogonal to our discussion in this chapter.

1.2 Type Systems

As programs grow in size and complexity, it becomes harder to ensure that they will not fail with simple errors or exceptions at run-time. Type systems are designed to statically identify *safe* programs, which are guaranteed *not* to fail at run-time due a particular set of errors. Typically, this set of errors is the set of stuck states, or simple errors, but more powerful type systems can rule out certain kinds of exceptions as well. There is an inherent tradeoff in the design of every type system: in exchange for ruling out a class of errors from occurring at run-time, some safe programs — and other "good" programs that may fail in a way that, for whatever reason, is acceptable in a particular setting — will perform computation that lays beyond the reasoning capabilities of the type system and, hence, conservatively rejected. In the extreme, a type system that rejects *every* program ensures safety but is hardly useful! Decades of fruitful research on type systems have improved this "sweet spot" — classifying as many of the safe programs as possible — in two interrelated, but generally distinct, ways: (i) ensuring the absence of simple errors from programs that use a wider *variety* of programming features and styles; and (ii) ensuring the absence of more *complex* errors, beyond just the stuck states, for a given set of programming features. In this section, we will survey several landmark techniques along each of these fronts, after a couple general remarks.

Semantics Before Types. The techniques we will discuss identify classes of safe programs drawn from untyped lambda-calculi, like the core language from the previous section, languages whose semantics are defined before any notion of types. This "semantics before types" approach to language definition has been the subject of much research with broad practical and theoretical implications. But there are also many useful languages where types are defined along with the syntax and semantics. Notable languages of this kind include class-based object-oriented languages like C++ and C# which provide, in addition to a *dynamically dispatched* projection expression $e[e']$ that is subject to traversal along an inheritance hierarchy as described before, a *statically dispatched* projection expression $(e@T)[e']$ that looks for the given key only in the record denoted by the type T. Languages where static types interact with the dynamic semantics have been studied in many contexts, but we will not describe them further in this report.

Inference vs. Annotations. An important factor in the "usability" of a type system is the burden of type annotations required by the programmer, as opposed to types that can be inferred automatically by the system, to prove that a program is safe. The annotation burden for the systems we discuss generally increases with the sophistication of the system, from zero annotations required for programs in the simplest systems to annotations that can be even larger than the programs themselves! In the survey that follows, we will not discuss the details of how much automatic inference each particular system performs.

1.2.1 Simple Types and Subtyping

There are several basic approaches for how type systems reason about base values, functions, records, and references, which we will describe in this section. As we will see, a single expression may be assigned multiple different types, in which case a *subtyping* relation on types, written $S <: T$, relates types when the former type is "better" or "more specific" than the second. That is, an expression e of type S can be used in any context where an expression of type T is required without compromising the safety of the program. We provide a representative syntax for simple types in Figure 1.2.

Base and Function Types. Base values can be classified by types that correspond to the sorts of values, such as *Num*, *Bool*, and *Str*, and by types that correspond to a specific subset of a sort of values, such as *Int* to describe those numbers that are also integers. Because integers can be assigned multiple types, the following subtyping rules relate *Int* with *Num* and all other base types only to themselves:

$$\frac{}{\mathit{Int} <: \mathit{Num}} \qquad\qquad \frac{}{B <: B}$$

In the *simply-typed lambda-calculus* [19, 24], a function type, or *arrow*, $T_1 \rightarrow T_2$ describes any function $\lambda x.e$ that, given an argument x of type T_1, either produces a value of type T_2 or does not return at all because it *diverges* (*i.e.* loops forever) or fails with a run-time error. By allowing partiality, function types enable reasoning about functions and function applications in a program despite not tracking precisely the exceptions that may be raised. For example, the division function is typically assigned the type $\mathit{Num} \rightarrow \mathit{Num} \rightarrow \mathit{Num}$ which hides the fact that it raises a divide-by-zero exception when the second argument is zero.

Subtyping between function types is subtle. Consider an expression that expects an argument g of function type $\mathit{Int} \rightarrow \mathit{Num}$. To understand what functions f are safe to pass to this context, consider the ways in which the context can interact with g: it may call g with expressions of type *Int*, so f must accept values of any supertype of *Int* (*Int* and *Num* in our setting); and it may use the values returned by g in contexts that requires *Num*s, so f must return values of any subtype of *Num* (*Int* and *Num*). The following distinctive subtyping rule for function types, where the relation on argument types is *contravariant* and on return types is *covariant*, captures this intuition:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Control Flow. The standard way to reason about the expression `if` e_1 `then` e_2 `else` e_3 is to require that e_1 has type *Bool* and that e_2 and e_3 have some common supertype T, in which case the entire expression is assigned type T. This abstraction preludes the type system from accepting safe expressions, such as `if false then "bad" ++ 17 else "good"`, which contain unsafe subexpressions that are never evaluated at run-time.

Record Types. Record types have been studied extensively, with early investigations in the 1980s and 1990s (*e.g.* [13, 103, 81, 14, 1, 82]) forming the basis for more powerful notions of objects with inheritance found in modern languages. Many formulations of records, or "simple objects," use the type $\{\overline{s}:\overline{T}\}$ to describe records that definitely have the keys, or fields, s_1 through s_n that

$$\begin{array}{rcll}
B & ::= & \mathit{Int} \;\mid\; \mathit{Num} \;\mid\; \mathit{Str} \;\mid\; \mathit{Bool} & \textbf{Base Types} \\
 & & & \\
S, T & ::= & & \textbf{Types} \\
 & \mid & B & \text{base type} \\
 & \mid & T_1 \rightarrow T_2 & \text{function type (arrow)} \\
 & \mid & \{\overline{s} : \overline{T}\} & \text{record type} \\
 & \mid & \mathit{Ref}\ T & \text{reference type}
\end{array}$$

Figure 1.2. Simple Types

bind values of type T_1 through T_n, respectively. To avoid reasoning about arbitrary computation, these systems require that record update $\{e \texttt{ with } \text{“}\texttt{f}\text{”} = e'\}$ and record projection $e[\text{“}\texttt{f}\text{”}]$ (often written $e.\texttt{f}$) use syntactic (string literal) keys rather than arbitrary (computed) string keys. Record subtyping, defined below, typically comprises three notions: *permutation*, which allows key-type bindings to appear in any order; *width subtyping*, which allows extra keys to be “forgotten” if not required; and *depth subtyping*, which allows the binding of a required key to be a subtype of that required:

$$\frac{\forall j.\ \exists i.\ s'_i = s''_j \text{ and } T'_i <: T''_j}{\{\overline{s'} : \overline{T'}\} <: \{\overline{s''} : \overline{T''}\}}$$

In a system that includes width subtyping, a record type says nothing about keys not mentioned in the type; other keys may or may not be bound by record values of that type.

Reference Types. To reason about heap-manipulating operations, the type *Ref T* describes heap references that store values of type T. All assignments to such locations must store values of type T (or any subtype) to maintain the invariant. Because a reference value is used both to store values (which can be regarded as calling a function that sets a heap location) and to read values (which can be regarded as calling a function that reads a heap location), subtyping on reference types must be both contra- and covariant, or *invariant*:

$$\frac{}{\mathit{Ref}\ T <: \mathit{Ref}\ T}$$

1.2.2 Richer Syntactic Types

Simple types provide a useful and solid foundation for preventing simple errors in programs that manipulate functions, records, and references. Next, we will describe several notable extensions that enable the static verification of larger sets of safe programs; the research literature is filled with countless others.

Parametric Polymorphism. Some functions — such as the identity function $\texttt{id} = \lambda x.x$ — are agnostic to the types of their arguments, but simple function types are constructed only from concrete base types. As such, `id` must be assigned a single type, for example, $\mathit{Num} \rightarrow \mathit{Num}$, which would allow the program to call `id` with numbers but not other kinds of values. As a remedy, the *polymorphic lambda-calculus* [46, 83], or System F, allows functions that are *parametric* in their arguments to be assigned *universal types*. For example, the `id` function can be assigned the type $\forall A.\ A \rightarrow A$ in System F, which allows different call sites to pass in different types of arguments by *instantiating* the type variable A as needed in each context. A form of parametric polymorphism [59, 72, 25], which is less expressive than in System F but enables better inference, forms the basis for the functional programming languages Standard ML, OCaml, and Haskell.

Bounded Quantification. Using only simple types, functions that operate on records are assigned types that are too imprecise for many common situations. For example, consider the function $\texttt{incN} = \lambda x.\{x \ \texttt{with}\ \text{“}\texttt{n}\text{”} = 1 + x.\texttt{n}\}$ which increments the number stored in the “n” field of its argument and returns the updated record. Using simple types, `incN` can be assigned the type $\{\text{“}\texttt{n}\text{”}{:}\mathit{Num}\} \rightarrow \{\text{“}\texttt{n}\text{”}{:}\mathit{Num}\}$ which specifies only that the output record has a numeric “n” field. Width subtyping allows records with additional fields to be passed to `incN`, but any such fields are lost by the return type of the function.

Matters are improved in systems with *bounded quantification* [13], where a polymorphic function type $\forall A <: T.\ A \rightarrow A$ is augmented with a *type bound* T that constrains what types can be used to instantiate A. By choosing the record type from before as the bound, `incN` can be assigned the type $\forall A <: \{\text{“}\texttt{n}\text{”}{:}\mathit{Num}\}.\ A \rightarrow A$ which is sufficient to verify the safety of the projection and addition operations without discarding information about any other fields present at each call site. The interaction of bounded quantification with recursive types is improved by *F-bounded polymorphism* [12], which forms the basis for *generics* and *wildcards* [100] in class-based languages like Java.

Classes. As described earlier, the semantics of records in many languages incorporate the notion of inheritance, where records implicitly inherit missing fields from their ancestors. Inheritance, which is unaccounted for by the simple record types we have seen so far, has been the source of many studies. One hallmark approach to records with inheritance (*i.e.* objects) is based on *classes*, where a class C is a record of fields that serves as the parent for all *instances* of C, which are themselves records. Another class C' may be declared as the *superclass* of C and acts as the parent of C'. An instance of C, therefore, inherits fields from C, C', and so on. Critical to this formulation is that the record types described by classes and their instances are not subject to width subtyping; that is, each of these record types describes only those fields which are definitely present at run-time. As a result, statically-defined class hierarchies can be traversed during type checking to determine how dynamically-dispatched projection expressions will behave.

Intersection and Union Types. An *intersection type* [76] $T_1 \wedge T_2$ describes expressions that satisfy both T_1 and T_2. A common application of intersection types is to describe overloaded operators. Recall that in some languages, the + operator may be defined to accept either pairs of numbers or strings. Using intersection types, the type $\mathit{Num} \to \mathit{Num} \to \mathit{Num} \wedge \mathit{Str} \to \mathit{Str} \to \mathit{Str}$ can be assigned, and when type checking a function application, the type system can perform the necessary "case-split" to select the appropriate component of the intersection type.

Dually, a *union type* $T_1 \vee T_2$ describes values that satisfy either T_1 or T_2. Union types have been incorporated into systems for various purposes, for example, describing the "shared" components of multiple object types [60], similar to the notion of an *interface* in object-oriented languages.

Occurrence Typing. Many languages offer primitive operators that allow the program to *introspect*, or *reflect*, on the sorts of values at run-time. For example, the functions `isNum`, `isStr`, and `isBool` may be provided to accept arbitrary values, returning `true` if the argument is a member of the specified sort and `false` otherwise. Using this facility, the function `negate` $= \lambda x.$ `if isNum` x `then 0 -` x `else not` x is defined to expect an argument x that is either a number or a boolean, distinguished by a call to `isNum`, and computes the numeric subtraction or boolean negation accordingly. Using union types, we might like to specify the type of `negate` as $\mathit{Num} \vee \mathit{Bool} \to \mathit{Num} \vee \mathit{Bool}$, but to verify this type the system must account for the *path-sensitive* reasoning, about the result of the call to `isNum`, on each branch.

Several approaches, including the recent *occurrence typing* approach in Typed Racket [99], support a limited form of path-sensitive reasoning in order to *narrow* a union type by eliminating some of its components to a more specific type along a particular branch. For example, the primitive function `isNum` might, informally, be assigned the type $\mathit{Num} \to \mathit{True} \wedge \neg\mathit{Num} \to \mathit{False}$ where the *singleton* type *True* (resp. *False*) describes only the value `true` (resp. `false`), and $\neg\mathit{Num}$ describes all types except *Num*. By manipulating the types *True* and *False* specially when type checking an if-expression, this approach can verify that `negate` satisfies the above function type by narrowing the type of x, which is $\mathit{Num} \vee \mathit{Bool}$, to *Num* on the then-branch and *Bool* on the else-branch, thus, verifying the safety of the primitive operations on both branches.

1.2.3 Refinement Types

So far, we have taken a tour of mechanisms for statically preventing simple errors from programs that use a variety of features and programming styles. Next, we will turn our attention to approaches that verify the absence of more complex errors, or exceptions. The following *dependent* type systems allow the type of one value to depend on the type of another, enabling the specification of a wide variety of fine-grained properties. We refer to non-dependent type systems, like the ones before, as *syntactic* systems. Dependent type systems are often simpler type systems integrated with more precise, heavyweight program analysis techniques like *verification* [56], *model*

B	$::=$	$Int \mid Num \mid Str \mid Bool$	**Base Types**
S,T	$::=$		**Types**
	$\mid$	$\{x\!:\!B \mid p\}$	refined base type
	$\mid$	$x\!:\!T_1 \to T_2$	dependent function type (arrow)
	$\mid$	$\{\overline{s}\!:\!\overline{T}\}$	record type
	$\mid$	$Ref\ T$	reference type
p,q	$::=$		**Refinement Formulas**
	$\mid$	$\cdots$	predicate
	$\mid$	$p \wedge q \mid p \vee q \mid \neg p$	logical connective

Figure 1.3. Refinement Types

checking [63], and *abstract interpretation* [22]; the type structure and basic invariants imposed by a simpler type system provide a foundation atop which more complex invariants are tracked.

We refer to *refinement* type systems as a lightweight form of dependent types, where the additional precision in the system is limited to constraints, or predicates, expressed in a logic composed of *decidable* theories. Variations of refinement types and their applications have been studied in many settings (*e.g.* [41, 104, 26, 31, 37, 7, 85, 94, 67]); the formulation that we describe here, the syntax of which is summarized in Figure 1.3, is most closely related to presentations by Flanagan *et al.* [37, 67].

Refined Base Types and Dependent Function Types. The syntax of refinement types differs from that of simple types (Figure 1.2) in two ways. First, a base type is now written $\{x\!:\!B \mid p\}$ to describe the subset of values x of type B for which the formula p is true. Refinement formulas are boolean combinations of predicates drawn from decidable theories — for example, the theories of linear arithmetic, uninterpreted functions, and equality — that are relevant to the particular system and applications. For example, the type $\{x\!:\!Int \mid x > 0\}$ describes the set of positive integers and $\{x\!:\!Int \mid x > y\}$ describes the set of integers greater than y, an integer variable defined earlier in the program.

Second, function types $x\!:\!T_1 \to T_2$ now include the formal parameter x, which the return type T_2 can refer to. For example, (assuming parametric polymorphism in the system) the identity function $\texttt{id} = \lambda x.x$ can be given the type $\forall A.\ x\!:\!A \to \{y\!:\!A \mid y = x\}$ which is not only polymorphic but, furthermore, encodes the fact that the return value is exactly equal to the argument.

Refinement Subtyping. Central to the expressiveness of a refinement type system is the way in which subtyping relates types. For function, record, and reference types, the subtyping rules are very much like the ones described for simple type systems, with some details specific to the presence of dependency. The subtyping rule for base types, however, appeals to the notion of *validity*, because base types are defined in terms of refinement formulas:

$$\frac{B_1 <: B_2 \qquad \mathit{Valid}(p \Rightarrow q)}{\{x\!:\!B_1 \,|\, p\} <: \{x\!:\!B_2 \,|\, q\}}$$

That is, subtyping on base types, which are sets of values, boils down to *implication* between their refinement formulas. For example, $\{x\!:\!\mathit{Int} \,|\, x = 1\} <: \{x\!:\!\mathit{Int} \,|\, x > 0\} <: \{x\!:\!\mathit{Int} \,|\, \mathit{true}\}$ because $x = 1 \Rightarrow x > 0 \Rightarrow \mathit{true}$. By using rich but decidable theories, a refinement system provides a precise notion of subtyping and can use decision procedures for Satisfiability Modulo Theories (SMT), as implemented in solvers such as Z3 [27], to discharge the validity queries that arise during type checking.

Control Flow-Based Reasoning. By combining precise specifications via refinements and path-sensitive reasoning, refinement type systems are able to statically check for expressions that may lead to run-time exceptions. The following demonstrate how numbers, primitive operations on numbers, and equality can be assigned very precise, sometimes exact, specifications:

$$\begin{array}{rcl} n & :: & \{x\!:\!\mathit{Int} \,|\, x = n\} \\ \texttt{(/)} & :: & x\!:\!\mathit{Num} \rightarrow y\!:\!\{y'\!:\!\mathit{Num} \,|\, \neg(y' = 0)\} \rightarrow \mathit{Num} \\ \texttt{(=)} & :: & \forall A.\ x\!:\!A \rightarrow y\!:\!A \rightarrow \{b\!:\!\mathit{Bool} \,|\, b = \texttt{true} \Leftrightarrow x = y\} \end{array}$$

Notice that the type for division requires the second argument to be non-zero, in order to rule out divide-by-zero exceptions. When verifying that the function $\lambda x.\ \texttt{if}\ x \texttt{=} 0\ \texttt{then null else}\ 17 \texttt{/} x$ has type $\mathit{Num} \rightarrow (\mathit{Num} \vee \mathit{Null})$ (assuming the presence of union types and a singleton type *Null* that describes the `null` value), the boolean b produced by the equality test in the guard has type $\{b\!:\!\mathit{Bool} \,|\, b = \texttt{true} \Leftrightarrow x = 0\}$. On the then-branch, the system tracks that $b = \texttt{true}$. On the false-branch, the system tracks that $b = \texttt{false}$, so the call to the division operator is statically verified as safe because x is guaranteed to be non-zero. This kind of path-sensitive reasoning has been applied to verify the absence of array-bounds-exceptions (*e.g.* [104, 85]). For example, using the following integer comparision and array-manipulating primitives

$$\begin{array}{rcl} \texttt{arrayLen} & :: & \forall A.\ a\!:\!\mathit{Arr}(A) \rightarrow \{n\!:\!\mathit{Int} \,|\, n = \mathit{length}(a)\} \\ \texttt{arrayGet} & :: & \forall A.\ a\!:\!\mathit{Arr}(A) \rightarrow i\!:\!\{j\!:\!\mathit{Int} \,|\, 0 \leq j \wedge j < \mathit{length}(a)\} \rightarrow A \\ \texttt{(<)} & :: & x\!:\!\mathit{Int} \rightarrow y\!:\!\mathit{Int} \rightarrow \{b\!:\!\mathit{Bool} \,|\, b = \texttt{true} \Leftrightarrow x < y\} \end{array}$$

a refinement system can verify that $\lambda a.\ \lambda i.$ `if` i `<` `arrayLen` a `then` `arrayGet` $a\ i$ `else` `null` has the type (among other more precise ones) $\forall A.\ Arr(A) \rightarrow \{i{:}Int \mid i > 0\} \rightarrow (A \vee Null)$ and does not fail with array-bounds exceptions.

Data Structure Invariants. Beyond verifying the absence of complex errors, refinement types can also be used to specify and enforce fine-grained invariants of data structures. So far, we have discussed records and classes as ways to implement data structures. To simplify the notation in this section, we consider a different mechanism for structuring data called *algebraic datatypes*, or *disjoint unions*, which are especially common in functional programming languages [77]. Specified as a (recursive) datatype, the type

$$IntList \ \mathrel{\hat{=}}\ \mu t.\ Nil + Cons\ (hd{:}Int,\ tl{:}t)$$

specifies the type of integer lists, each of which is either the label Nil with no other data or a pair labeled $Cons$, where the first component hd is an integer and the second component tl is another integer list (referred to by the recursive type t being defined by the recursive type operator μ). Using refinements, this coarse-grained type can be augmented with a predicate that describes each value in the list. For example, the type of lists of positive integers can be described as follows:

$$PosIntList \ \mathrel{\hat{=}}\ \mu t.\ Nil + Cons\ (hd{:}\{x{:}Int \mid x > 0\},\ tl{:}t)$$

Going further, refinements can be used to relate elements inside a recursive data structure of unknown size. For example, in the datatype definition

$$SortedIntList \ \mathrel{\hat{=}}\ \mu t.\ Nil + Cons\ (hd{:}Int,\ tl{:}t[hd \leq hd'])$$

the type $t[hd \leq hd']$ represents the fact that for any given $Cons$ node in the list, the head hd of the node is no greater than the head hd' of the tail of the node (*i.e.* the next element). Viewed globally, this invariant captures the fact that the entire list is sorted (in particular, non-decreasing). Refinement types have been used to statically verify fine-grained data structure properties — such as list sortedness, tree balancedness, binary search tree well-orderedness, and graph acyclicity — even with a large degree of automation using a technique called *liquid type inference* [65].

Security. The beauty of the refinement type architecture is that the language of predicates can be selected to match the needs of a particular application. The theory of uninterpreted functions, specifically, allows domain-specific properties to be encoded easily.

Refinement types have been used to specify and verify that reference monitors to sensitive resources, like file systems or password managers, are implemented correctly with respect to a security policy (*e.g.* [7, 94, 50]). For example, a primitive function `readFile` that accesses the file system can be assigned the type $u{:}User \rightarrow \{f{:}Str \mid CanRead(u, f)\} \rightarrow Str$ to require that each user accesses only those files for which the *CanRead* privilege (represented as an uninterpreted predicate) has been granted, defined by a security policy (*i.e.* formula) and manipulated by the types of the primitive operations exposed by the reference monitor.

Precise Heap Reasoning. In simple type systems, references into the heap are assigned types of the form *Ref T*, where all references of this type point to values in the heap that share the invariant type (*i.e.* supertype) T. In many programs — for example, those in languages like C with explicit memory management — more precise reasoning is required to track individual references into the heap and the particular values they store, which may change over time.

To this end, refinement type systems can incorporate a standard approach in verification for imperative programs where the heap (also knows as the *memory* or *store*) is treated as a record mapping references to values, encoded in logic using McCarthy's theory of arrays [71]. The McCarthy operator $\mathit{sel}(h,x)$ retrieves the value stored at reference x in the heap h and the operator $\mathit{upd}(h,x,y)$ is the result of updating h so that x points to y, overwriting the previous contents stored at x, if any. In addition, using a syntactic mechanism called an *affine type* that ensures that a value be "used" at most once, heap values can be threaded through a program in monadic style, making explicit the otherwise implicit effects of imperative operations on the heap [101].

A refinement type system can incorporate this approach (*e.g.* [94]) to reason about a *first-order store*, which binds only non-function values (for simplicity, assume only integers), by using primitive operators that explicitly manipulate heap values:

$$\begin{aligned}
\texttt{ref} &:: h{:}\mathit{Heap} \to y{:}\mathit{Int} \to \{(x,\ h') : (\mathit{Ref},\ \mathit{Affine}[\mathit{Heap}]) \mid h' = \mathit{upd}(h,x,y)\} \\
\texttt{deref} &:: h{:}\mathit{Heap} \to x{:}\mathit{Ref} \to \{(y,\ h') : (\mathit{Int},\ \mathit{Affine}[\mathit{Heap}]) \mid y = \mathit{sel}(h,x) \wedge h' = h\} \\
\texttt{setref} &:: h{:}\mathit{Heap} \to x{:}\mathit{Ref} \to y{:}\mathit{Int} \to \{h' {:} \mathit{Affine}[\mathit{Heap}] \mid h' = \mathit{upd}(h,x,y)\}
\end{aligned}$$

The reference operation returns a new reference x and a new affine heap h' that extends the input heap h with the new binding; the dereference operation returns the value y bound in the heap h and produces a new affine heap h' that is unchanged; and the assignment operation produces a new affine heap h' with the appropriate binding updated. The type system syntactically ensures that an affine heap of type $\{h{:}\mathit{Affine}[\mathit{Heap}] \mid p\}$ is converted to the type $\{h{:}\mathit{Heap} \mid p\}$ at most once, ensuring soundness.

Compared to this "global" encoding of the heap, the "local" reasoning approach to verification in *separation logic* [62, 84] builds from the notion that in typical programs, most addresses in the heap are not aliased. Heap specifications in separation logic are smaller, referring only to the "footprint" of a heap that is directly manipulated, making it more tractable to verify large heap-manipulating programs. Alias Types [91], a cousin to separation logic, incorporates this notion into a syntactic type system, which is further combined with refinement types in *Low-Level Liquid Lypes* (LTLL) [86] to verify memory safety in C, a first-order imperative language.

1.2.4 Higher-Order Dependent Types

To conclude our survey of type systems, we consider two general categories of what we refer to as *higher-order dependent* type systems that manipulate properties beyond the reach of decidable reasoning. Higher-order dependent type systems can be used to provide very strong guarantees about programs, but the expressive power comes at the cost of relying on heuristics to help discharge proof obligations, relying on the programmer to write additional annotations (beyond just the specifications) to help the proving process, or both.

Undecidable Refinement Logics. The general refinement type approach can incorporate more expressive, undecidable refinement logics at the cost of automation. For example, recall that the refinement encoding of heaps, discussed above, was limited to storing non-function values. To demonstrate the challenge of reasoning about a *higher-order store*, consider the program

$$\texttt{let}\ (\texttt{idRef},\ h_1) = (\texttt{ref}\ h_0\ \lambda x.x)\ \texttt{in}\ \texttt{let}\ (\texttt{f},\ h_2) = (\texttt{deref}\ h_1\ \texttt{idRef})\ \texttt{in}\ 17 + (\texttt{f}\ 42)$$

which stores and retrieves the identify function before applying it to an integer argument. The type of value produced by the call to `deref` is described by the predicate $\mathit{sel}(h_1,\texttt{idRef})$, which says nothing about its type structure. So, there is no way to syntactically verify that it is a function type, let alone an appropriate one such as $\mathit{Int} \to \mathit{Int}$. Therefore, for the refinement system to support function calls through the heap, function types must be encoded in the refinement logic (as in [95]). For example, rather than the refinement type we saw earlier, the `id` function would be assigned a type like

$$\{f\colon (\forall A.\ A \to A) \mid \forall x.\ \forall y.\ y = \mathit{apply}(f,x) \Rightarrow y = x\}$$

where the *apply* predicate is used to encode the semantics of function application. An approach built like this — using quantifiers in first-order logic, which is undecidable — can succeed in verifying complex programs that manipulate a higher-order store; first-order theorem provers, and even some SMT solvers like Z3 [27], employ heuristics that perform well on some classes of instances that arise in program verification. On the other hand, relying so heavily on the logic to perform reasoning about functions, function calls, and data structures eliminates some of the benefits of formulating a type system in the first place.

As an aside, it is worth noting that the contra- and covariance of function subtyping $S_1 \to S_2 <: T_1 \to T_2$ is captured directly by the logical encoding:

$$(\forall x.\ S_1(x) \Rightarrow S_2(\mathit{apply}(f,x))) \Rightarrow (\forall x.\ T_1(x) \Rightarrow T_2(\mathit{apply}(f,x)))$$

Logical Frameworks. Systems equipped with dependent types can serve as a foundation, often called a *logical framework*, for proving properties about programs. For example, unlike the partial specification that a function type in weaker system represents, a *total correctness* property guarantees that the program always terminates and satisfies a particular specification. Logical frameworks from the basis for several programming languages, including LF [55], Coq [8], and Hoare Type Theory [73], in which programs are "correct by construction." However, building complex systems in this style typically requires a significant manual annotation burden to assist the proof process. One remarkable system built in this style is CompCert, a verified compiler for a large subset of C [69].

1.3 "Dynamic" Languages

We have seen a glimpse of the significant progress made towards reasoning about the absence of various run-time errors in programs that manipulate functions, objects, and references. In addition to eliminating errors, the specifications tracked by type systems also provide information (*i.e.* abstractions) useful for documenting the intent of the programmers and for assisting in semantics-preserving program transformations such as refactoring and compilation.

Nevertheless, untyped, or dynamic languages, have always also been a popular parallel playground for programming languages. The costs of static typing — the safe programs that are rejected because of limits in expressiveness and any additional work "up front" by way of type annotations — evidently outweigh the benefits in many situations. However, were type systems able to reason more precisely about dynamic languages, there are likely situations (many, but not all) where more statically-typed dialects of these languages would provide value to programmers. Unfortunately, except for the ultra-expressive (and, hence, not very user-friendly or suitable for mainstream programming) higher-order dependent ones, existing type systems are unable to reason precisely about dynamic languages. This is not due to a preponderance of exotic features in dynamic languages, although there are some — such as the expressive macro system in LISP and its descendants; the subtle semantics for scope in Python where variables are declared implicitly by their use; and the arrays in JavaScript which are a curious mix of objects and traditional arrays. Rather, dynamic languages comprise core features very much like the ones we have discussed, but the *programming idioms* in which they are used pose obstacles to decidable, static reasoning. We outline several of the major challenges, using JavaScript syntax, in this section.

1.3.1 Untagged Unions and Path Sensitivity

Dynamic languages often provide a `typeof` operator as a lightweight way to reflect on the kinds of values at run-time. Unlike the primitives `isNum`, `isBool`, and `isStr` that we discussed earlier to serve a similar purpose, `typeof` returns an ordinary string, called a *tag*, that classifies the value. For example, a program expecting a value `x` that is either a number or boolean might discriminate `x` as follows:

```
if (typeof x == "number")        { return 0 - x; }
else /* typeof x == "boolean" */ { return !x;    }
```

Syntactic techniques, like occurrence typing [99], for tracking control flow are better suited to specialized syntactic type-test functions like `isNum`, *etc.*, especially as more complicated dynamic tests are built up from individual ones and, furthermore, abstracted into functions. Refinement types, however, are well-suited for this kind of path-sensitive reasoning, as we saw earlier, and can also track precise relationships, for example, that the tag of the return value above is the same as the tag of `x`.

1.3.2 Dictionary-Style Objects

A much more significant challenge is the notion of objects, which are much like records, from the core language we discussed, that maps string keys to values. Recall that existing approaches for type checking records employ record types of the form $\{\overline{s}{:}\overline{T}\}$, where $\overline{s}$ is a sequence of strings, and require that programs use program uses only syntactic (*i.e.* string literal) keys when retrieving and updating bindings. To implement records with dynamically computed keys in these statically-typed systems, the programmer must use a separate mechanism (*e.g.* classes or datatypes) to implement *dictionaries*, also known as *maps*, *hash tables*, or *associative arrays*. A dictionary would be assigned a type like $\mathit{Dict}[K,V]$ to describe mappings from unknown K-typed keys to V-typed values. The sharp distinction between (i) records with fixed, statically-known fields that map to values of different types and (ii) dictionaries with unknown keys that map to values of the same type is maintained to facilitate static reasoning.

In dynamic languages, however, there is only a single mechanism: records *are* dictionaries. As a result, a static type system must be able to specify and enforce invariants about dynamically computed keys that bind values of different types. For example, a run-time test may detect the presence of a key and only then use the binding at a particular type:

```
if (k in duck) { return 17 + duck[k]; }
```

Using refinement types over McCarthy's theory of arrays [71], we can specify the type of this dictionary as $\{d{:}\mathit{Dict} \mid \mathit{has}(d,\mathtt{k}) \Rightarrow \mathit{Num}(\mathit{sel}(d,\mathtt{k}))\}$. This approach, employed in the statically-typed, first-order, dynamic language Dminor [9], works well for dictionaries that store non-function values. But, of course, dictionaries might bind function values, as well, for example:

```
if ("quack" in duck) { return "Duck says " + duck.quack(); }
else                 { return "This duck cannot quack!";   }
```

The function type $\mathit{Unit} \rightarrow \mathit{Str}$ for the "`quack`" field could be encoded inside a refinement type, but would enter the territory of requiring a higher-order refinement logic.

1.3.3 Extensible Objects

To further complicate matters, objects in dynamic languages are not pure dictionaries, but *references* to dictionaries. As such, they are subject to mutation by adding and removing keys. For example, an object may be initialized to (a reference to) an empty dictionary and then extended with a method:

```
var duck   = {};
duck.quack = function () { return "Quack!"; };
```

Although simple examples like this one can be syntactically rewritten so that all keys are initialized at once, in general, keys can be added and removed arbitrarily far apart in the program, across function boundaries, and depending on run-time control flow. Therefore, to verify that `duck.quack` will not fail with a "field-not-found-error," static reasoning must support strong updates to the types of heap values. As mentioned earlier, LTLL [86] is a refinement type system that supports precise heap reasoning à la Alias Types [91], but does not support the dictionary-style objects and higher-order functions found in dynamic languages.

1.3.4 Prototype Inheritance

A third challenge of objects in JavaScript, in particular, is that, in addition to being mutable and indexed by arbitrary (computed) string keys, each object maintains an implicit link to the "prototype" object from which it derives. To resolve a key lookup from an object at run-time, JavaScript transitively follows its prototype links until either the key is found or the root is reached without success. Thus, unlike in class-based languages, inheritance relationships are computed at run-time, using the function `Object.create` below, rather than provided as declarative specifications.

```
var duckling = Object.create(duck);
duckling.quack();  // "Quack!"

duck.quack = function () { return "Oink?"; };
duckling.quack();  // "Oink?"
```

Notice that not only is the prototype relationship computed, it is not even "frozen" at initialization time; strong updates to the parent object `duck` are reflected in subsequent key lookups on the child object `duckling`. There have been several proposals for type checking prototype-based objects (*e.g.* [10, 45, 57]), but these either have no support for prototype hierarchies or only prototype hierarchies of a finite length described syntactically in the source program.

1.3.5 Dynamically Loaded Code

The final challenge we highlight is the presence of a primitive function `eval` that parses an arbitrary string to evaluate as code. Although this feature is notorious for misuse and provides an attack vector for adversaries, it is pervasive and essential for web applications that comprise and interact with data from multiple servers:

```
var str = getFromNetwork("..."); eval(str);
```

Dynamic loading is not unique to dynamic languages, of course. But the fine-grained way in which it appears in programs is hard to reason about, compared to classes and modules, which provide natural boundaries for static reasoning to track the pre- and post-conditions for dynamically loaded components. One common way to restrict the power of `eval` is to use a version (*e.g.* `parseJSON` in popular JavaScript libraries) that accepts only data without code. This restriction is not appropriate for all situations, however, and says little about the shape of the resulting data.

1.4 Hybrid Typing

To mitigate the inherent limitations of static type systems — due to expressiveness limits or dynamically loaded code — the *hybrid type checking* approach [37, 67] inserts run-time *casts*, or *contracts* [36], into the program for checking obligations that cannot be discharged at compile-time. Only plausible casts are inserted; casts that are guaranteed to fail represent type errors within the reasoning capabilities of the static system. Hybrid typing continues in the spirit of a long line of research efforts to mix typed and untyped programs, referred to variously as *dynamic typing* [2, 58], *soft typing* [15, 5], and *gradual typing* [88, 89]. Compiling casts efficiently and in a way that facilitates good error messages for debugging are active areas of research (*e.g.* [90]).

Hybrid typing blends static- and run-time checking, neither of which is precise enough or sufficient to guarantee correctness, within a single specification language. Unfortunately, because existing syntactic and refinement type systems are unable to reason precisely about the idioms in dynamic languages, incorporating them into hybrid type systems for real-world dynamic languages would result in an overwhelming number of safety checks deferred until run-time.

1.5 Contributions

The thesis of this dissertation is that refinement type-based techniques can reason about many of the programming idioms found in dynamic languages like JavaScript in order to verify the absence of run-time errors. We make the following contributions to support this claim:

1. We present a novel mechanism called *nested refinements*, which generalizes prior refinement type systems by describing all values (rather than just base values) using decidable refinement

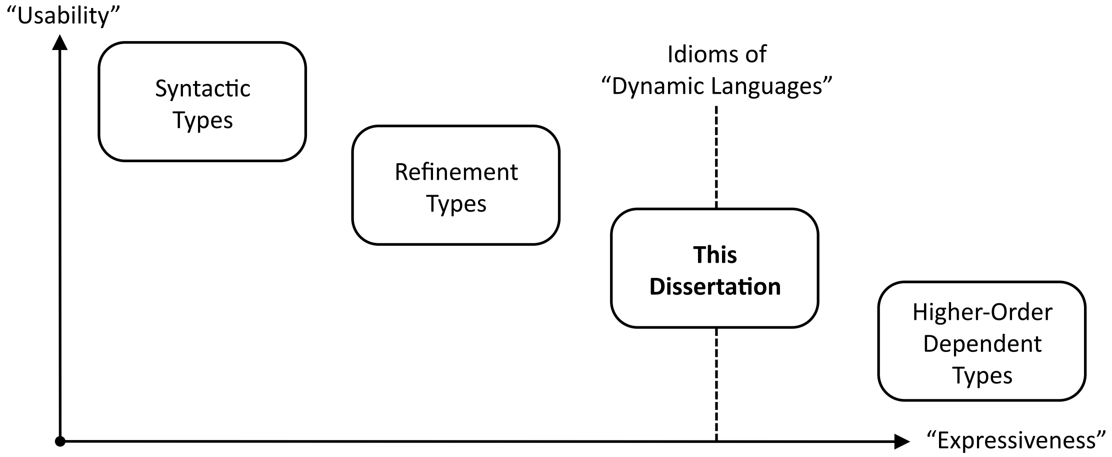

Figure 1.4. Spectrum of Syntactic and Dependent Type Systems

formulas, and a type system called System D that employs nested refinements to reason about dictionary objects with dynamic keys in a purely functional setting;

2. We present a new combination of (nested) refinement types and *flow-sensitive strong updates*, in a type system called System !D (pronounced "D-ref"), to enable reasoning about dictionary objects in the presence of mutable variables;

3. We present a refinement type encoding called *heap unrolling* to reason about prototype-based inheritance, which we incorporate, along with other JavaScript-specific encodings, in a type system called System DJS; and

4. We present and implement an explicitly typed language called Dependent JavaScript (or DJS for short), based heavily on the syntax and semantics of JavaScript, that translates to System DJS for type checking. We demonstrate the utility of our techniques by type checking (*i.e.* verifying the absence of run-time errors in) several small but challenging benchmarks.

Our contributions advance the state-of-the-art in refinement type systems that rely only on decidable, first-order theories, as opposed to the higher-order logics of more powerful dependent type systems. Although the mechanisms we develop are general, we bias our presentation towards their application to dynamic languages, because that serves as the inspiration for this work. Figure 1.4 shows a cartoon of where our work fits in between the kinds of syntactic, refinement, and dependent techniques outlined earlier.

Organization. We organize the technical material into two parts.

- In the first part, we study nested refinements in detail. We present System D in Chapter 2, starting with a series of examples that demonstrate the kinds of specifications that nested refinements enable for a purely functional language. Central to our approach is a novel subtyping algorithm that carefully factors work between SMT-decidable queries and more conventional syntactic rules in order to rely only on decidable theories. Nesting syntactic types inside refinements introduces a circularity that poses a challenge for proving type soundness. We address this using a stratification argument that allows us to prove the soundness of System D. In Chapter 3, we present a version of the system, called Algorithmic System D, that is suitable for implementation and forms the basis for our type checker for Dependent JavaScript. In Chapter 4, we investigate a few extensions that increase the expressive power of nested refinement subtyping, and we prove them sound in the context of System D.

- In the second part, we scale up the nested refinements technique to handle features in more full-fledged, real-world dynamic languages like JavaScript and Python, namely, mutable variables and inheritance. We extend System D to System !D in Chapter 5, adding support for precise reasoning about imperative updates, and we extend System !D to System DJS in Chapter 6, adding support for precise reasoning about prototype-based inheritance and other JavaScript-specific features. We choose to present System !D as an explicit, intermediate step in our progression, because the combination of refinement types and strong updates in a higher-order setting is novel and, we believe, will be useful in other settings beyond our application to JavaScript. We do not rigorously study the formal properties of System !D and System DJS as we do for System D, however, so this is an important direction for future work. Instead, we demonstrate the feasibility of our techniques, in Chapter 7, by implementing a type checker for System DJS and a translation from DJS to System DJS.

Afterwards, we conclude with discussions of future directions and related work. For a shorter, breadth-first overview of the techniques proposed in this dissertation, the reader may choose to read up through the end of the Overview sections of the System D, System !D, and System DJS chapters (§ 2.1, § 5.1, and § 6.1, respectively) before moving on to Chapter 7 to see how their combination enables static reasoning in Dependent JavaScript.

Part II

Nested Refinements

Chapter 2

System D

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 [52] 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 [96], illustrates common dynamic features:

```
let onto callbacks f obj =
  if f = null then
    List (obj, callbacks)
  else
    let cb = if tagof f = "string" then obj[f] else f in
    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.

Currently, the only systems that are expressive enough to support the full spectrum of reasoning required for dynamic languages are higher-order *dependent* type systems like Coq [8], which require that the programmer provide explicit proofs to discharge type checking obligations. Type systems that perform automatic checking of specifications, which we classify as *syntactic* and *refinement* systems, cannot support the necessary reasoning. Syntactic type systems use advanced type-theoretic constructs like structural types [6], row types [98], intersection types [42], and union types [99, 52] 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`. Refinement type systems like [31, 85, 67, 7, 94, 9] 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 for dynamic languages that requires explicit function type annotations but then supports automatic checking. 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 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. This is in stark contrast to prior refinement type systems where only base types (the "leaves" of more complex types like function types) are described using refinements. For example, in System D we specify the type of functions that map integers to boolean with the type

$$\{x \mid x :: \{x \mid tag(x) = \text{“}\texttt{integer}\text{”}\} \rightarrow \{x \mid tag(x) = \text{“}\texttt{boolean}\text{”}\}\}.$$

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

$$\{x \mid \mathit{tag}(x) = \text{“}\texttt{integer}\text{”} \vee \mathit{tag}(x) = \text{“}\texttt{string}\text{”}\}$$

to type a value x that is either an integer or a string; we can then track control flow involving the dynamic type tag-lookup function $\mathit{tag}(\cdot)$ 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

$$\{x \mid x = \texttt{null} \vee \mathit{tag}(x) = \text{“}\texttt{string}\text{”} \vee x :: \mathit{Any} \rightarrow \mathit{Any}\}$$

where *Any* is an abbreviation for $\{x \mid \mathit{true}\}$, 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 [71]. In particular, McCarthy's two operators — $\mathit{sel}(x,a)$, which corresponds to the contents of the map x at the address a, and $\mathit{upd}(x,a,v)$, 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

$$\{x \mid \mathit{tag}(x) = \text{“}\texttt{dictionary}\text{”} \wedge \mathit{tag}(\mathit{sel}(x,\texttt{y})) = \text{“}\texttt{integer}\text{”}\}$$

to type dictionaries x 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 combinations of dictionaries and functions. For example, we can write the following type for `obj`

$$\{x \mid \mathit{tag}(\texttt{f}) = \text{“}\texttt{string}\text{”} \Rightarrow \mathit{sel}(x,\texttt{f}) :: \mathit{Any} \rightarrow \mathit{Any}\}$$

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 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 [67, 85, 9]. (A higher-order logic (*e.g.* [8]) 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*, by 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. To meet this challenge, 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 summarize, we make several contributions in the first part of this dissertation:

- 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. (Chapter 2)
- We develop a novel subtyping mechanism that uses the structure of the refinement formulas to decompose subtyping into a collection of SMT and syntactic checks. (Chapter 2)
- We illustrate the technical challenges that nesting poses to the metatheory of System D and present a stratification-based proof technique to establish soundness. (Chapter 2)
- We define an algorithmic version of the type system with local type inference that we implement in a prototype checker. (Chapter 3)
- We present several extensions to nested refinement subtyping that further increase expressiveness of the system. (Chapter 4)

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

2.1 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 some sophisticated invariants in idiomatic dynamic code.

Notation. We use the following abbreviations for brevity.

$$\begin{array}{rcl rcl}
\mathit{Int}(x) & \doteq & \mathit{tag}(x) = \text{“}\texttt{integer}\text{”} & \mathit{Any}(x) & \doteq & \mathit{true} \\
\mathit{Str}(x) & \doteq & \mathit{tag}(x) = \text{“}\texttt{string}\text{”} & \mathit{Dict}(x) & \doteq & \mathit{tag}(x) = \text{“}\texttt{dictionary}\text{”} \\
\mathit{Bool}(x) & \doteq & \mathit{tag}(x) = \text{“}\texttt{boolean}\text{”} & \mathit{IntOrBool}(x) & \doteq & \mathit{Int}(x) \vee \mathit{Bool}(x)
\end{array}$$

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 $\{x \,|\, T(x)\}$. For example, we write *Int* as an abbreviation for $\{x \,|\, \mathit{tag}(x) = \text{“}\texttt{integer}\text{”}\}$. Recall that function values are also described by refinement formulas (containing type predicates). We often write arrows outside refinements to abbreviate the following:

$$x\!:\!T_1 \to T_2 \;\doteq\; \{y \,|\, y :: x\!:\!T_1 \to T_2\}$$

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

2.1.1 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 [99, 52].

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. In System D, the programmer explicitly annotates each function with a type inside a comment, demarcated by an additional `:` character, just before the definition. We typeset annotations in math mode for clarity, but the ASCII versions parsed by our type checker are quite similar.

```
(*: negate :: IntOrBool → IntOrBool *)
let negate x =
  if tagof x = "integer" then 0 - x else not x
```

The function type above states that the function accepts an integer or boolean argument and returns either an integer or boolean result.

To verify that the definition of `negate` satisfies the declared function type, System D uses the standard means of reasoning about control flow in refinement-based systems [85], 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}) = $ "`integer`", which allows System D to verify that the expression $0 - \texttt{x}$ is well-typed. The return value has the type $\{x \mid \mathit{tag}(x) = \text{“}\texttt{integer}\text{”} \wedge x = 0 - \texttt{x}\}$. This type is a subtype of *IntOrBool* since the SMT solver can prove that $\mathit{tag}(x) = \text{“}\texttt{integer}\text{”}$ and $x = 0 - \texttt{x}$ implies $\mathit{tag}(x) = \text{“}\texttt{integer}\text{”} \vee \mathit{tag}(x) = \text{“}\texttt{boolean}\text{”}$. Thus, the return value of the then-branch is deduced to have type *IntOrBool*.

On the other hand, in the else-branch, the environment contains the assumption that $\neg(\mathit{tag}(\texttt{x}) = \text{“}\texttt{integer}\text{”})$. By combining this with the assumption about the type of `negate`'s input, $\mathit{tag}(\texttt{x}) = \text{“}\texttt{integer}\text{”} \vee \mathit{tag}(\texttt{x}) = \text{“}\texttt{boolean}\text{”}$, the SMT solver can determine that $\mathit{tag}(\texttt{x}) = \text{“}\texttt{boolean}\text{”}$. This allows our system to type check the call to `not` :: *Bool* → *Bool*, which establishes that the value returned in the else branch has type *IntOrBool*. Thus, our system determines that both branches return a value of type *IntOrBool*, 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 [99, 52]. While `negate` takes and returns ad-hoc unions, there is a relationship between its input and output: the output is an integer (resp. boolean) if and only if the input is an integer (resp. boolean). We can represent this in System D by ascribing the following more precise function type to `negate`.

```
(*: negate :: x:IntOrBool → {y | tag(y) = tag(x)} *)
let negate x =
  if tagof x = "integer" then 0 - x else not 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.

2.1.2 Nested Refinements

So far, we have seen how old-fashioned refinement types (where the predicates refine base values [75, 67, 85, 9]) 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 [9]. 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. In System D, we can use a refinement formula that combines a base predicate and a type predicate to assign `maybeApply` the following type.

```
(*: maybeApply :: Num → {f | f = null ∨ f :: Num → Num} → Num *)
let maybeApply x f =
  if f = null then x else f x
```

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 satisfies 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 $\mathit{Embed}(\Gamma) \Rightarrow \texttt{f} :: U$ holds, where $\mathit{Embed}(\Gamma)$ 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 $\mathit{Int} \to \mathit{Int}$. Due to the bindings for `x` and `f` and the else-condition, the environment Γ_0 is embedded as

$$\mathit{Embed}(\Gamma_0) \;\doteq\; \mathit{tag}(\texttt{x}) = \text{“}\texttt{integer}\text{”} \wedge (\texttt{f} = \texttt{null} \vee \texttt{f} :: U_1) \wedge \neg(\texttt{f} = \texttt{null})$$

Hence, the SMT solver is able to prove that $\mathit{Embed}(\Gamma_0) \Rightarrow \texttt{f} :: U_1$. 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. That is, in the environment $\Gamma_1 \doteq \texttt{negate}{:}\{f \mid f :: U_0\}, \texttt{maybeApply}{:}\cdots$ where the function type assigned to `negate` is $U_0 \;\mathrel{\hat{=}}\; x{:}\mathit{IntOrBool} \rightarrow \{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}$, the following two subtyping relationships must hold:

$$\begin{aligned} \Gamma_1 \vdash \{x \mid x = 42\} &\sqsubseteq \mathit{Int} \\ \Gamma_1 \vdash \{f \mid f = \texttt{negate}\} &\sqsubseteq \{f \mid f = \texttt{null} \vee f :: U_1\} \end{aligned} \tag{2.1}$$

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, we use extraction to address the problem. We show that all subtyping checks of the form $\Gamma \vdash \{x \mid p\} \sqsubseteq \{x \mid q\}$ can be reduced to a finite number of sub-goals of the form:

$$\begin{array}{rl} \textit{("type predicate-free")} & \mathit{Embed}(\Gamma') \Rightarrow p' \\ \text{or } \textit{("type predicate")} & \mathit{Embed}(\Gamma') \Rightarrow x :: U \end{array}$$

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 (Equation 2.1) reduces to proving either

$$\mathit{Embed}(\Gamma_1') \Rightarrow f = \texttt{null} \quad \text{or} \quad \mathit{Embed}(\Gamma_1') \Rightarrow f :: U_1$$

where $\Gamma_1' \;\mathrel{\hat{=}}\; \Gamma_1, f = \texttt{negate}$. The former implication contains no type predicates, so we attempt to prove it by querying the SMT solver. The solver decides 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 to f. Thus, all that remains is to retrieve the definition of U_0 and U_1 and check

$$\Gamma_1' \vdash x{:}\mathit{IntOrBool} \rightarrow \{y \mid \mathit{tag}(y) = \mathit{tag}(x)\} \sqsubseteq \mathit{Int} \rightarrow \mathit{Int}$$

which follows via standard syntactic refinement subtyping [67], 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.

2.1.3 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 [71, 28]. In particular, we use the following primitives (denoted by `val` declarations) for manipulating dictionaries, where $x \mathrel{\mathit{iff}} p \;\mathrel{\hat{=}}\; \mathit{Bool}(x) \wedge (x = \texttt{true} \Leftrightarrow p)$:

```
val  {} :: (*: {d | d = empty} *)
val mem :: (*: d:Dict → k:Str → {b | b iff has(d,k)} *)
val get :: (*: d:Dict → k:{s | Str(s) ∧ has(d,s)} → {x | x = sel(d,k)} *)
val set :: (*: d:Dict → k:Str → x:Any → {d′ | d′ = upd(d,k,x)} *)
```

In the examples that follow, we often use traditional record and array syntax to abbreviate calls to the above dictionary primitives. For example:

$$\begin{aligned}
\texttt{x[k]} &\mathrel{\mathring{=}} \texttt{get x k} \\
\texttt{x.f} &\mathrel{\mathring{=}} \texttt{get x ``f''} \\
\texttt{\{``f'' = 17; ``g'' = true\}} &\mathrel{\mathring{=}} \texttt{set (set \{\} ``f'' 17) ``g'' true}
\end{aligned}$$

We also introduce two abbreviations for dictionary types:

$$\begin{aligned}
\mathit{Fld}(x,y,\mathit{Int}) &\mathrel{\mathring{=}} \mathit{Dict}(x) \wedge \mathit{Str}(y) \wedge \mathit{Int}(\mathit{sel}(x,y)) \\
\mathit{Fld}(x,y,U) &\mathrel{\mathring{=}} \mathit{Dict}(x) \wedge \mathit{Str}(y) \wedge \mathit{sel}(x,y) :: U
\end{aligned}$$

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:

```
(*: getCount :: Dict → Str → Int *)
let getCount t c =
  if mem t c then toInt t[c] else 0
```

The function `getCount` uses the primitive operation `mem` 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 $\mathit{has}(d,k)$, 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 [71, 28].

The *dictionary lookup* `t[c]` is desugared to `get t c` where the type of the primitive operation `get`, defined above, uses $\mathit{sel}(d,k)$, 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` :: $\mathit{Any} \rightarrow \mathit{Int}$. Note that the if-guard strengthens the environment under which the lookup appears with the fact $\mathit{has}(\texttt{t},\texttt{c})$, 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 declared type.

Dynamic Update. Dually, to allow dynamic updates, System D includes a primitive `set`, declared above, that produces a new dictionary, where $\mathit{upd}(d,k,x)$ is an operator in the theory of maps that denotes the functional update of d extended with a binding from k to x. The following illustrates how the `set` primitive can be used:

```
(*: incCount :: d:Dict → c:Str → {d′ | EqMod(d′,d,{c}) ∧ Int(sel(d′,c))} *)
let incCount t c =
  let newcount = 1 + getCount t c in
  let res      = set t c newcount in res
```

In the declared type, $\mathit{EqMod}(d_1,d_2,K)$ abbreviates a predicate that stipulates that d_1 is identical to d_2 at all keys *except* for those in the set of keys K. The output type of `getCount` allows System D to conclude that `newcount` $::$ *Int*. From the output type of `set`, System D deduces that `res` $:: \{d' \mid d' = \mathit{upd}(\texttt{t},\texttt{c},\texttt{newcount})\}$, which is a subtype of the output type of `incCount`. Next, consider the following:

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

System D verifies the following two types and, hence, concludes that the field lookups return *Int*s that can be safely added:

$$\begin{aligned} \texttt{d0} &:: \{x \mid \mathit{Fld}(x,\text{“}\texttt{files}\text{”},\mathit{Int})\} \\ \texttt{d1} &:: \{x \mid \mathit{Fld}(x,\text{“}\texttt{files}\text{”},\mathit{Int}) \wedge \mathit{Fld}(x,\text{“}\texttt{dirs}\text{”},\mathit{Int})\} \end{aligned}$$

2.1.4 Type Constructors

Next, we use nesting and extraction to enrich System D with data structures. In general, System D supports arbitrary user-defined datatypes, but to keep the current discussion simple, let us consider a single type constructor $\mathit{List}[T]$ for representing unbounded sequences of T-values. Informally, an expression of type $\mathit{List}[T]$ is a dictionary with a “`hd`” key of type T and a “`tl`” key of type $\mathit{List}[T]?$, which is an optional type as defined below. As for arrows, we allow list types to be written outside of refinements:

$$T? \ \doteq\ \{x \mid x = \texttt{null} \vee T(x)\} \qquad\qquad \mathit{List}[T] \ \doteq\ \{x \mid x :: \mathit{List}[T]\}$$

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

```
(*: concat :: Str → List[Any]? → Str *)
let rec concat sep xs =
  if xs = null then "" else
    let hd = xs["hd"] in
    let tl = xs["tl"] in
    if tagof hd != "string"
      then concat sep tl
      else hd ^ sep ^ concat sep tl
```

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 $\mathit{List}[\cdot]$ as a type term in the refinement logic. The following illustrates an idiom where an argument is either a single value or a list of values:

```
(*: runTest :: Str → {x | Int(x) ∨ x :: List[Int]?} → Bool *)
let runTest cmd fail_codes =
  let status = syscall cmd in
  if tagof fail_codes = "integer" then
    not (status = fail_codes)
  else
    not (listMem status fail_codes)
```

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

2.1.5 Polymorphism

Similarly, we can add polymorphic type variables to System D by simply treating type variables A, B, *etc.* as (uninterpreted) type terms in the logic. Furthermore, we attach a (possibly-empty) sequence of type variables to each function type term, and we write the following abbreviation. As before, we use the following notation to write (polymorphic) function types and type variables outside of refinements:

$$\forall\overline{A}.\ x{:}T_1 \to T_2 \ \circeq\ \{f \mid f :: \forall\overline{A}.\ x{:}T_1 \to T_2\}$$

$$A \ \circeq\ \{x \mid x :: A\}$$

Generic Containers. We can compose type constructors in the ways familiar to functional programming. For example, consider list `map` in System D.

```
(*: map :: ∀A,B. (A → B) → List[A]? → List[B]? *)
let rec map f xs =
  if xs = null then null
  else List (f xs["hd"], map f xs["tl"])
```

(Of course, we could add pattern matching to improve matters, but instead we are trying to demonstrate how much can be achieved with dictionaries alone.) By combining extraction with the reasoning used for `concat`, it is easy to verify the declared type.

Predicate Functions. Consider the following list filter function.

```
(*: filter :: ∀A,B. (x:A → {y | y = true ⇒ x :: B}) → List[A]? → List[B]? *)
let rec filter f xs =
  if xs = null      then null
  else if f xs["hd"] then List (xs["hd"], filter f xs["tl"])
  else                    filter f xs["tl"]
```

The predicate function `f` returns `true` for values of type A that also satisfy the possibly more precise type B. As a result, System D verifies that `filter` takes a list `xs` of A-typed values but returns a list of B-typed values. Thus, the general mechanism of nested refinements subsumes the kind of reasoning performed by specialized techniques like latent predicates [99].

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

```
(*: dispatch :: ∀A,B. d:{x | Dict(x) ∧ x :: A} → f:{x | Fld(d,x,A → B)} → B *)
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. Note that there is no need for explicit type bounds; all that is required is the conjunction of the appropriate nested refinements.

2.1.6 All Together Now

Unions, Generic Dispatch, and Polymorphism. We now have everything we need to type check the motivating example from earlier, `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` as follows.

```
(*: onto ::
    $\forall A.\ callbacks\!:\!List[Any \rightarrow Any]?$
     $\rightarrow f\!:\!\{x \mid x = \texttt{null} \lor Str(x) \lor x :: A \rightarrow Any\}$
     $\rightarrow obj\!:\!\{x \mid x :: A \land (f = \texttt{null} \Rightarrow x :: Any \rightarrow Any) \land (Str(f) \Rightarrow Fld(x,f,A \rightarrow Any))\}$
     $\rightarrow List[Any \rightarrow Any]?$ *)
```

Using reasoning similar to that for previous examples, System D checks that `onto` satisfies this specification, where the type of *obj* uses the form 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 check 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` [97].

```
(*: toXML :: $Any \rightarrow Str$ *)
let rec toXML x =
  if tagof x = "boolean" then
    element (if x then "true" else "false") null
  else if tagof x = "integer" then
    element "integer" (intToStr x)
  else if tagof x = "string" then
    element "string" x
  else if tagof x = "dictionary" then
    let ks = keys x in
    let vs = map (*: $[\{s \mid Str(s) \land has(x,s)\};\ Str]$ *)
      (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 $Str \rightarrow \{y \mid y = \texttt{null} \lor Str(y)\} \rightarrow Str$, we can verify the declared type for `toXML`. 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 following primitive operation

```
val keys :: (*: $d\!:\!Dict \rightarrow List[\{s \mid Str(s) \land has(d,s)\}]$ *)
```

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

$$\begin{array}{rrcl}
\textbf{Values} & v & ::= & \lambda x.e \;\mid\; x \;\mid\; c \;\mid\; v_1[v_2 \mapsto v_3] \;\mid\; C(\overline{v}) \\
\textbf{Expressions} & e & ::= & v \;\mid\; [\overline{T}]\; v_1\; v_2 \;\mid\; \texttt{if}\; v \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \;\mid\; \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \\
\textbf{Types} & T & ::= & \{x \mid p\} \\
\textbf{Formulas} & p,q & ::= & P(\overline{w}) \;\mid\; w :: U \;\mid\; p \wedge q \;\mid\; p \vee q \;\mid\; \neg p \\
\textbf{Logical Values} & w & ::= & v \;\mid\; F(\overline{w}) \\
\textbf{Type Terms} & U & ::= & \forall \overline{A}.\; x{:}T_1 \to T_2 \;\mid\; A \;\mid\; C[\overline{T}] \\
\textbf{Metavariables} & x,y,z & \in & \mathit{ValueIdentifiers} \\
 & A,B & \in & \mathit{TypeVarIdentifiers} \\
 & C & \in & \mathit{TypeConstructorIdentifiers} \\
 & c & \in & \mathit{Constants} \supset \{\texttt{true}, \texttt{null}, \mathit{17}, \text{“}\texttt{hanna}\text{”}, \texttt{(=)}, \texttt{tagof}, \texttt{get}, \texttt{fix}\} \\
 & F & \in & \mathit{FunctionSymbols} \supset \{\mathit{tag}, \mathit{sel}, \mathit{upd}, +\} \\
 & P & \in & \mathit{PredicateSymbols} \supset \{=, <\}
\end{array}$$

Figure 2.1. Syntax of System D Expressions and Types

2.2 Syntax and Semantics

We begin with the syntax and evaluation semantics of System D. Figure 2.1 shows the syntax of values, expressions, and types.

Values. Values v include functions, variables, constants, 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 w are all values and applications of primitive function symbols F, such as addition $+$ and dictionary selection *sel*, to logical values. The constant `tagof` allows introspection on the type tag of a value at run-time. For example,

$$\begin{array}{rclrcl}
\texttt{tagof}(3) & \mathrel{\hat{=}} & \text{“}\texttt{integer}\text{”} & \texttt{tagof}(\texttt{true}) & \mathrel{\hat{=}} & \text{“}\texttt{boolean}\text{”} \\
\texttt{tagof}(\text{“}\texttt{john}\text{”}) & \mathrel{\hat{=}} & \text{“}\texttt{string}\text{”} & \texttt{tagof}(\lambda x.e) & \mathrel{\hat{=}} & \text{“}\texttt{function}\text{”} \\
\texttt{tagof}(\mathit{List}(1, \texttt{null})) & \mathrel{\hat{=}} & \text{“}\texttt{dictionary}\text{”} & \texttt{tagof}(\{\}) & \mathrel{\hat{=}} & \text{“}\texttt{dictionary}\text{”}
\end{array}$$

Dictionaries. A dictionary $v_1[v_2 \mapsto v_3]$ extends the dictionary v_1 with the binding from string key v_2 to value v_3. For example, the dictionary mapping “x” to 3 and “y” to `true` is written $\{\}[\text{“x”} \mapsto 3][\text{“y”} \mapsto \texttt{true}]$. 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 as:

$$\begin{array}{l}
\texttt{get}\; (v[\text{“x”} \mapsto v_x])\; \text{“x”} \mathrel{\hat{=}} v_x \\
\texttt{get}\; (v[\text{“y”} \mapsto v_y])\; \text{“x”} \mathrel{\hat{=}} \texttt{get}\; v\; \text{“x”}
\end{array}$$

Notice that $\texttt{get}\ \{\}\ k$ is undefined for all keys k. The function `mem` tests for the presence of a field, and the function `set` updates the value bound to a key; these are defined as:

$$\begin{aligned}
\texttt{mem}\ (v[\text{“}\texttt{y}\text{”} \mapsto v_y])\ \text{“}\texttt{x}\text{”} &\mathrel{\circeq} \texttt{mem}\ v\ \text{“}\texttt{x}\text{”} \\
\texttt{mem}\ (v[\text{“}\texttt{x}\text{”} \mapsto v_x])\ \text{“}\texttt{x}\text{”} &\mathrel{\circeq} \texttt{true} \\
\texttt{mem}\ \{\}\ \text{“}\texttt{x}\text{”} &\mathrel{\circeq} \texttt{false} \\
\texttt{set}\ d\ k\ v &\mathrel{\circeq} d[k \mapsto v]
\end{aligned}$$

Expressions. We use an A-normal form (ANF) presentation [38] so that we need only define substitution of values (not arbitrary expressions) into types. The set of expressions e consists of values, function applications, if-then-else expressions, and let-bindings. Function application $[\overline{T}]\ v_1\ v_2$ applies the function v_1 to type arguments $\overline{T}$ and value argument v_2; we often write $v_1\ v_2$ when there are no type arguments. We use tuple syntax $(v_1, \ldots, v_n)$ as sugar for the dictionary with fields “`1`” through “`n`” bound to the component values.

Types. In System D, a type T is a *refinement type* of the form $\{x \mid p\}$, where p is a *refinement formula*, and is read “x such that p.” The values of this type are all values v such that the formula $p[v/x]$ “is true.” What this means, formally, is core to our approach and will be considered in detail in §2.4. The choice of binders for refinement types may vary, so types are equivalent up to alpha-renaming. Similar to the syntax for expression tuples, we use $(T_1, \ldots, T_n)$ as sugar for the dictionary type with fields “`0`” through “`n`” with the corresponding types. We define application $T(w)$ of types to logical values using substitution as follows:

$$\{x \mid p\}(w) \mathrel{\circeq} p[w/x]$$

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 $\{x \mid \mathit{tag}(x) = \text{“}\texttt{integer}\text{”}\}$, which we abbreviate to *Int*. The type of positive integers is $\{x \mid \mathit{tag}(x) = \text{“}\texttt{integer}\text{”} \wedge x > 0\}$ and the type of dictionaries with an integer field “`f`” is $\{x \mid \mathit{tag}(x) = \text{“}\texttt{dictionary}\text{”} \wedge \mathit{tag}(\mathit{sel}(x, \text{“}\texttt{f}\text{”})) = \text{“}\texttt{integer}\text{”}\}$.

Nesting: Has-Type Predicates and Type 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, $w :: U$, called a *has-type predicate*, where U is a *type term*. The type term $x{:}T_1 \rightarrow T_2$ describes values that have a dependent function type, *i.e.* functions that accept arguments v of type T_1 and return values of type $T_2[v/x]$, where x is bound in T_2. We write $T_1 \rightarrow T_2$ when x does not appear in T_2. Furthermore, we write the function type term $\forall \overline{A}.\ x{:}T_1 \rightarrow T_2$ to describe a function type that is parameterized

Abbreviations

$$
\begin{array}{lcl}
V & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Str} \mid \mathit{Dict} \mid \mathit{Fun} \\
 & \mid & \mathit{IntOrBool} \mid \mathit{IntOrStr} \mid \cdots \\
 & \mid & \mathit{Any}
\end{array}
$$

Syntactic Sugar $\boxed{(\!|\cdot|\!) = p}$

$$
\begin{array}{rcl}
(\!|\mathit{Any}(x)|\!) & = & \mathit{true} \\
(\!|\mathit{Int}(x)|\!) & = & \mathit{tag}(x) = \text{“}\texttt{integer}\text{”} \\
(\!|\mathit{Bool}(x)|\!) & = & \mathit{tag}(x) = \text{“}\texttt{boolean}\text{”} \\
(\!|\mathit{Str}(x)|\!) & = & \mathit{tag}(x) = \text{“}\texttt{string}\text{”} \\
(\!|\mathit{Dict}(x)|\!) & = & \mathit{tag}(x) = \text{“}\texttt{dictionary}\text{”} \\
(\!|\mathit{Fun}(x)|\!) & = & \mathit{tag}(x) = \text{“}\texttt{function}\text{”} \\
(\!|\mathit{IntOrBool}(x)|\!) & = & (\!|\mathit{Int}(x)|\!) \vee (\!|\mathit{Bool}(x)|\!) \\
(\!|\mathit{if}\ p\ \mathit{then}\ q_1\ \mathit{else}\ q_2|\!) & = & (p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2) \\
(\!|\mathit{ite}\ p\ q_1\ q_2|\!) & = & (\!|\mathit{if}\ p\ \mathit{then}\ q_1\ \mathit{else}\ q_2|\!) \\
(\!|x\ \mathit{iff}\ p|\!) & = & \mathit{Bool}(x) \wedge (x = \texttt{true} \Leftrightarrow p)
\end{array}
$$

Figure 2.2. Syntactic Sugar for System D

by the (possibly-empty) sequence of type variables $\overline{A}$. Type variables A, B, *etc.* are themselves type terms. 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 integer successor function has type $\{f \mid f :: x{:}\mathit{Int} \rightarrow \{y \mid \mathit{tag}(y) = \text{“}\texttt{integer}\text{”} \wedge y = x + 1\}\}$, dictionaries where the value at key “$\texttt{f}$” maps Int to Int have type $\{d \mid \mathit{tag}(d) = \text{“}\texttt{dictionary}\text{”} \wedge \mathit{has}(d, \text{“}\texttt{f}\text{”}) \wedge \mathit{sel}(d, \text{“}\texttt{f}\text{”}) :: \mathit{Int} \rightarrow \mathit{Int}\}$, and the constructed record $\mathit{List}(\texttt{1,null})$ can be assigned the type $\{x \mid x :: \mathit{List}[\mathit{Int}]\}$.

Syntactic Sugar. Throughout our discussion, we defined several syntactic abbreviations to make types concise. We summarize these and other notational conveniences in Figure 2.2 and Figure 2.3.

Datatype Definitions. To simplify the presentation in subsequent sections, we assume the presence of a global table Ψ of datatypes definitions (rather than including them in the syntax of expressions) as specified in Figure 2.5. A datatype definition of C defines a named, possibly recursive type that 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, which will be described shortly, 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 $\overline{\theta A}$

Syntactic Sugar $\boxed{(\!| \cdot |\!) = T}\ \boxed{(\!| \cdot |\!) = x\!:\!T}\ \boxed{(\!| \cdot |\!) = U}$

$$
\begin{array}{rcl}
(\!| V |\!) & = & \{x \mid (\!| V(x) |\!)\} \\
(\!| V\{p\} |\!) & = & \{x \mid (\!| V(x) |\!) \wedge p\} \\
(\!| U |\!) & = & \{x \mid x :: U\} \\
(\!| T? |\!) & = & \{x \mid T(x) \vee x = \texttt{null}\} \\
(\!| \{f\!:\!T_1;\ g\!:\!T_2\} |\!) & = & \{x \mid (\!| \mathit{Dict}(x) |\!) \wedge T_1(\mathit{sel}(x, \text{“}\texttt{f}\text{”})) \wedge T_2(\mathit{sel}(x, \text{“}\texttt{g}\text{”}))\} \\
(\!| (x_1\!:\!T_1,\ x_2\!:\!T_2) |\!) & = & \{x \mid (\!| \mathit{Dict}(x) |\!) \wedge \pi(T_1(x_1)) \wedge \pi(T_2(x_2))\} \\
 & & \quad \text{where } \pi = [\mathit{sel}(x, \text{“}\texttt{1}\text{”})/x_1][\mathit{sel}(x, \text{“}\texttt{2}\text{”})/x_2] \\
 & & \\
(\!| T |\!) & = & \_\!:\!T \\
(\!| x\!:\!V |\!) & = & x\!:\!\{x \mid (\!| V(x) |\!)\} \\
(\!| x\!:\!V\{p\} |\!) & = & x\!:\!\{x \mid (\!| V(x) |\!) \wedge p\} \\
(\!| x\!:\!U |\!) & = & x\!:\!\{x \mid (\!| U |\!)\} \\
 & & \\
(\!| x\!:\!T_1 \to y\!:\!T_2 |\!) & = & x\!:\!T_1 \to \{y \mid T_2(y)\} \\
(\!| (x_1\!:\!T_1,\ x_2\!:\!T_2) \to T |\!) & = & (\!| (x_1\!:\!T_1,\ x_2\!:\!T_2) |\!) \to \pi T \\
 & & \quad \text{where } \pi = [\mathit{sel}(x, \text{“}\texttt{1}\text{”})/x_1][\mathit{sel}(x, \text{“}\texttt{2}\text{”})/x_2]
\end{array}
$$

Figure 2.3. Syntactic Sugar for System D (continued)

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.

Semantics. The small-step operational semantics of System D is standard for a call-by-value, polymorphic λ-calculus, and is shown in Figure 2.4. Following standard practice, the semantics is parameterized by a function δ that assigns meaning to primitive functions c, including dictionary operations like `mem`, `get`, and `set`. Because expressions are A-normalized, there is a single congruence rule, E-Compat. Our implementation [80] desugars more palatable syntax into ANF.

2.3 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 SMT-based reasoning. We summarize the syntax of environments and typing judgements in Figure 2.5. The indices n on typing judgements should be ignored throughout this section (and, thus, we will usually omit them); we will discuss their significance in §2.4.

Operational Semantics $\boxed{e_1 \hookrightarrow e_2}$

$$\frac{}{[\overline{T}]\ (\lambda x.e)\ v \hookrightarrow e[v/x]}\ [\textsc{E-App}] \qquad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x]}\ [\textsc{E-Let}]$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1}\ [\textsc{E-IfTrue}] \qquad \frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2}\ [\textsc{E-IfFalse}]$$

$$\frac{\delta(c,v)\ \text{is defined}}{c\ v \hookrightarrow \delta(c,v)}\ [\textsc{E-Delta}] \qquad \frac{e_1 \hookrightarrow e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2}\ [\textsc{E-Compat}]$$

Figure 2.4. Semantics of System D

$$\begin{array}{rll}
\textbf{Datatypes} & \Psi & ::= \ - \ \mid\ \Psi, \texttt{type}\ C[\overline{\theta A}] = \{\overline{f}:\overline{T}\} \\
\textbf{Type Environments} & \Gamma & ::= \ - \ \mid\ \Gamma, x{:}T \ \mid\ \Gamma, A \ \mid\ \Gamma, p \\
 & & \\
\textbf{Typing} & \Gamma \vdash_n e :: T & \\
\textbf{Subtyping} & \Gamma \vdash_n T_1 \sqsubseteq T_2 & \\
\textbf{Syntactic Subtyping} & \Gamma \vdash_n U_1 <: U_2 & \\
\textbf{Implication} & \Gamma \Rightarrow_n p & \\
\textbf{Well-Formedness} & \vdash \Psi \quad \vdash \Gamma \quad \Gamma \vdash T \quad \Gamma \vdash p \quad \Gamma \vdash U \quad \Gamma \vdash w &
\end{array}$$

Figure 2.5. Syntax of System D Judgements

Environments. Type environments Γ include bindings that record either: the derived type T for a variable x; a type variable A introduced in the scope of a 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. We assume for clarity that Ψ is fixed and globally visible, and elide it from the judgements. In the sequel, we assume that Ψ contains at least the following definition:

$$\texttt{type}\ \mathit{List}[+A] = \{\ \text{“}\texttt{hd}\text{”}{:}A;\ \text{“}\texttt{tl}\text{”}{:}\mathit{List}[A]?\ \}$$

2.3.1 Well-Formedness

Figure 2.6 defines the well-formedness relations.

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. When checking the well-formedness of a refinement

Well-Formed Types $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma, x{:}Any \vdash p}{\Gamma \vdash \{x \mid p\}}$$

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

$$\frac{\Gamma \vdash w \qquad \Gamma \vdash U}{\Gamma \vdash w :: U} \qquad \frac{\Gamma \vdash \overline{w}}{\Gamma \vdash P(\overline{w})} \qquad \frac{\Gamma \vdash p \qquad \Gamma \vdash q}{\Gamma \vdash p \wedge q} \qquad \frac{\Gamma \vdash p \qquad \Gamma \vdash q}{\Gamma \vdash p \vee q} \qquad \frac{\Gamma \vdash p}{\Gamma \vdash \neg p}$$

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

$$\frac{\Gamma, \overline{A} \vdash T_1 \qquad \Gamma, \overline{A}, x{:}T_1 \vdash T_2}{\Gamma \vdash \forall \overline{A}.\ x{:}T_1 \to T_2} \qquad \frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{C \in dom(\Psi) \qquad \Gamma \vdash \overline{T}}{\Gamma \vdash C[\overline{T}]}$$

Well-Formed Logical Values (selected rules) $\boxed{\Gamma \vdash w}$

$$\frac{}{\Gamma \vdash c} \qquad \frac{x \in dom(\Gamma)}{\Gamma \vdash x} \qquad \frac{\Gamma \vdash \overline{w}}{\Gamma \vdash F(\overline{w})}$$

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

$$\frac{\vdash \Gamma \qquad x \notin dom(\Gamma) \qquad \Gamma \vdash T}{\vdash \Gamma, x{:}T} \qquad \frac{\vdash \Gamma \qquad A \notin \Gamma}{\vdash \Gamma, A} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash p}{\vdash \Gamma, p} \qquad \frac{}{\vdash -}$$

Well-Formed Datatypes $\boxed{\vdash \Psi}$

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

Figure 2.6. Well-Formedness for System D

type $\{x \mid p\}$, we check that p is well-formed in the environment extended with $x{:}Any$. 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, so we omit the definition; 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 the procedure *VarianceOk*, defined as

$$\begin{array}{ll} \mathit{VarianceOk}(A,\texttt{+},\overline{T}) & \text{if } (\cup_j\ \mathit{Polarity}(A,T_j)) \subseteq \{\texttt{+}\} \\ \mathit{VarianceOk}(A,\texttt{-},\overline{T}) & \text{if } (\cup_j\ \mathit{Polarity}(A,T_j)) \subseteq \{\texttt{-}\} \\ \mathit{VarianceOk}(A,\texttt{=},\overline{T}) & \text{always} \end{array}$$

that records whether type variables occur in positive or negative positions within the types of the fields. The helper procedure *Polarity* (defined in Appendix A) walks the structure of types, formulas, *etc.* to observe the occurrences of type variables. $\mathit{Polarity}(A,T)$ computes a subset of $\{\texttt{+},\texttt{-}\}$ that includes + (resp. -) if A occurs in at least one positive (resp. negative) position inside T. For each type variable, these polarities are computed across all field types in the definition and then checked against its variance annotation.

After successfully checking that a type definition is well-formed, it is added to the globally-available type definition environment Ψ. For example, when checking the well-formedness of the type term $C[\overline{T}]$, we make sure that C is defined by testing for its presence in Ψ.

2.3.2 Typing

The expression typing judgement $\Gamma \vdash e :: T$, defined in Figure 2.7 and Figure 2.8, verifies that e has type T 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:

```
val      1 :: (*: {x | x = 1} *)
val "john" :: (*: {x | x = “john”} *)
val   true :: (*: {x | x = true} *)
val  false :: (*: {x | x = false} *)
```

Value Typing $\boxed{\Gamma \vdash_n v :: T}$

$$\frac{}{\Gamma \vdash_n c :: ty(c)}\ \text{[T-CONST]} \qquad \frac{x \in dom(\Gamma)}{\Gamma \vdash_n x :: \{y \,|\, y = x\}}\ \text{[T-VAR]}$$

$$\frac{\Gamma \vdash_n v_1 :: Dict \qquad \Gamma \vdash_n v_2 :: Str \qquad \Gamma \vdash_n v_3 :: T}{\Gamma \vdash_n v_1[v_2 \mapsto v_3] :: \{x \,|\, x = v_1[v_2 \mapsto v_3]\}}\ \text{[T-EXTEND]}$$

$$\frac{\Gamma, \overline{A} \vdash T_1 \qquad \Gamma, \overline{A}, x{:}T_1 \vdash_n e :: T_2}{\Gamma \vdash_n \lambda x.e :: \{y \,|\, y = \lambda x.e \wedge y :: \forall \overline{A}.\ x{:}T_1 \to T_2\}}\ \text{[T-FUN]}$$

$$\frac{\Psi \ni (\texttt{type}\ \texttt{C}[\overline{\theta A}] = \{\overline{f{:}T'}\}) \qquad \forall i.\ \Gamma \vdash T_i \qquad \forall j.\ \Gamma \vdash_n v_j :: Inst(T'_j, \overline{A}, \overline{T})}{\Gamma \vdash_n C(\overline{v}) :: \{x \,|\, Fold(x, C, \overline{T}, \overline{v})\}}\ \text{[T-FOLD]}$$

Figure 2.7. Value Typing for System D

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.

```
val   (+) :: (*: x:Int → y:Int → {z | Int(z) ∧ z = x + y} *)
val   not :: (*: x:Bool → {b | b iff x = false} *)
val   (=) :: (*: x:Any → y:Any → {b | b iff x = y} *)
val   fix :: (*: ∀A. (A → A) → A *)
val tagof :: (*: x:Any → {s | Str(s) ∧ s = tag(x)} *)
```

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, defined in §2.1.3, are given refinement types over McCarthy's theory of arrays [71] 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 [28]. The types of our dictionary primitives {}, `mem`, `get`, and `set` use the constant *empty* to denote the empty dictionary, and the predicate $has(d,k)$ to abbreviate $sel(d,k) \neq bot$. To relate two dictionaries, we use the abbreviation

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

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 [11] by reduction to an equisatisfiable quantifier-free formula. The *EqMod* abbreviation is useful for dictionary updates where we do

Expression Typing $\boxed{\Gamma \vdash_n e :: T}$

$$\frac{\Gamma \vdash \overline{T} \qquad \Gamma \vdash_n v_1 :: \{y \mid y :: \forall \overline{A}.\ x{:}T_{11} \to T_{12}\} \qquad \Gamma \vdash_n v_2 :: \mathit{Inst}(T_{11},\overline{A},\overline{T})}{\Gamma \vdash_n [\overline{T}]\ v_1\ v_2 :: \mathit{Inst}(T_{12},\overline{A},\overline{T})[v_2/x]}\ \text{[T-App]}$$

$$\frac{\Gamma \vdash_n v :: \mathit{Bool} \qquad \Gamma, v = \texttt{true} \vdash_n e_1 :: T \qquad \Gamma, v = \texttt{false} \vdash_n e_2 :: T}{\Gamma \vdash_n \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: T}\ \text{[T-If]}$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash_n e_1 :: T_1 \qquad \Gamma, x{:}T_1 \vdash_n e_2 :: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: T_2}\ \text{[T-Let]}$$

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

$$\frac{\Gamma \vdash_n e :: \{x \mid x :: C[\overline{T}]\}}{\Gamma \vdash_n e :: \{x \mid \mathit{Unfold}(x, C, \overline{T})\}}\ \text{[T-Unfold]}$$

Figure 2.8. Expression Typing for System D

not know the *exact* value being stored, but do know some abstraction thereof, *e.g.* its type. For example, in `incCounter` (from § 2.1.3) 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. If x is bound 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 T to an if-expression if the condition v is a boolean-valued expression, the then-branch expression e_1 has type T under the assumption that v evaluates to `true`, and the else-branch expression e_2 has type T under the assumption that v evaluates to `false`. The T-App rule is standard, but notice that the arrow type of v_1 is nested inside a refinement type. The procedure *Inst*, which instantiates a type variable with a type, is defined recursively on formulas, type terms, and types, where the only non-trivial cases of the definition involve type predicates with type variables:

$$\mathit{Inst}(w :: A, A, \{x \mid p\}) = p[w/x] \qquad \mathit{Inst}(w :: B, A, T) = w :: B$$

We write $Inst(T,\overline{A},\overline{T})$ to denote the result of applying $Inst$ to T with the type variables and type arguments in succession. In T-Let, the type T_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 T if e has type T' and T' is a subtype of T.

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 v_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

$$Fold(x,C,\overline{T},\overline{v}) \;\mathrel{\hat{=}}\; tag(x) = \text{“}\texttt{dictionary}\text{”} \wedge x :: C[\overline{T}] \wedge (\wedge_j \; sel(x,f_j) = v_j)$$

specifies that 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 constructed data as a dictionary, using the formula

$$Unfold(x,C,\overline{T}) \;\mathrel{\hat{=}}\; tag(x) = \text{“}\texttt{dictionary}\text{”} \wedge (\wedge_j \; T'_j(sel(x,f_j)))$$

where $T'_j = Inst(T_j,\overline{A},\overline{T})$ for each j. For example, $Unfold(x,List,Int)$ expands to the formula $tag(x) = \text{“}\texttt{dictionary}\text{”} \wedge tag(sel(x,\text{“}\texttt{hd}\text{”})) = \text{“}\texttt{integer}\text{”} \wedge sel(x,\text{“}\texttt{tl}\text{”}) :: List[Int]?$.

2.3.3 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 [37, 85, 67]. 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 judgement $\Gamma \vdash \{x \mid p_1\} \sqsubseteq \{x \mid p_2\}$. We cannot simply use the SMT query

$$Embed(\Gamma) \Rightarrow (p_1 \Rightarrow p_2) \tag{2.2}$$

since 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 Equation 2.2 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 2.9.

Step 1: Split query into subgoals. The rule I-Cnf starts by converting $p_1 \Rightarrow p_2$ into an equivalent formula in conjunctive normal form, where each clause is written as an implication $q_i \Rightarrow Q'_i$, where

Q_i' is a disjunction (or set) of positive simple or type predicates, by rearranging literals and adding negations as necessary. The goal (Equation 2.2) is reduced to the following collection of subgoals, which requires that for each clause, there exists some predicate that can be discharged:

$$\forall i.\ \exists q_i'.\ \Gamma, q_i \Rightarrow q_i'$$

Step 2: Discharge subgoals. When the goal is a simple predicate, or a type predicate for which the environment contains an assumption, the subgoal is of the form

$$\text{(\textit{“type predicate-free”})} \qquad \Gamma, q_i \Rightarrow q_i'$$

which rule I-Valid handles by SMT. Otherwise, the subgoal is of the form

$$\text{(\textit{“type predicate”})} \qquad \Gamma, q_i \Rightarrow w :: U$$

which rule I-HasType handles via type extraction followed by a use of syntactic subtyping. In particular, the rule tries to establish $w :: U$ by searching for a type term U' that (i) *flows to* w, *i.e.* for which we can deduce via SMT that $Embed(\Gamma) \wedge q_i \Rightarrow w :: U'$ is valid, and (ii) is a syntactic subtype of U (written $\Gamma, q_i \vdash U' <: U$). The rule I-Valid-n is not intended for use in type checking source programs and is included solely for the metatheory, which we discuss in §2.4.

The rules U-Datatype and U-Arrow establish syntactic (refinement) subtyping, by (recursively) establishing that subtyping holds for the matching components [37, 85, 9]. Because syntactic subtyping recursively refers to subtyping, the S-Refine rule uses fresh variables to avoid duplicate bindings in the environment.

Formula Implication. In each SMT implication query $Embed(\Gamma) \Rightarrow p$, the operator $Embed(\cdot)$ describes the embedding of types and environments into the logic as follows:

$$\begin{aligned} Embed(\Gamma, x{:}T) &\doteq Embed(\Gamma) \wedge T(x) \\ Embed(\Gamma, p) &\doteq Embed(\Gamma) \wedge p \\ Embed(\Gamma, A) &\doteq Embed(\Gamma) \\ Embed(-) &\doteq true \end{aligned}$$

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.)

Refinement Logic. The SMT architecture allows individual decidable theories to be combined in a decidable way as long as they refer to disjoint function symbols of distinct sorts [74]. In

Subtyping $\boxed{\Gamma \vdash_n T_1 \sqsubseteq T_2}$

$$\frac{y \text{ fresh} \qquad \Gamma \Rightarrow_n (p_1[y/x_1] \Rightarrow p_2[y/x_2])}{\Gamma \vdash_n \{x_1 \mid p_1\} \sqsubseteq \{x_2 \mid p_2\}} \ \text{[S-Refine]}$$

Implication $\boxed{\Gamma \Rightarrow_n p}$

$$\frac{CNF(p) = \wedge_i \ (q_i \Rightarrow Q'_i) \qquad \forall i.\ \exists q' \in Q'_i.\ \Gamma, q_i \Rightarrow_n q'}{\Gamma \Rightarrow_n p} \ \text{[I-Cnf]}$$

$$\frac{Valid(Embed(\Gamma) \Rightarrow p)}{\Gamma \Rightarrow_n p} \ \text{[I-Valid]} \qquad \frac{Valid(Embed(\Gamma) \Rightarrow w :: U') \qquad \Gamma \vdash_n U' <: U}{\Gamma \Rightarrow_n w :: U} \ \text{[I-HasType]}$$

$$\frac{\mathcal{I}_n \models Embed(\Gamma) \Rightarrow p}{\Gamma \Rightarrow_n p} \ \text{[I-Valid-n]}$$

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

$$\frac{}{\Gamma \vdash_n A <: A} \ \text{[U-Var]} \qquad \frac{\Gamma, \overline{A} \vdash_n T_{21} \sqsubseteq T_{11} \qquad \Gamma, \overline{A}, x\!:\!T_{21} \vdash_n T_{12} \sqsubseteq T_{22}}{\Gamma \vdash_n \forall \overline{A}.\ x\!:\!T_{11} \to T_{12} <: \forall \overline{A}.\ x\!:\!T_{21} \to T_{22}} \ \text{[U-Arrow]}$$

$$\frac{\begin{array}{c}\Psi \ni (\texttt{type}\ C[\overline{\theta A}] = \{\overline{f\!:\!T'}\}) \\ \forall i.\ \text{if } \theta_i \in \{\texttt{+},\texttt{=}\} \text{ then } \Gamma \vdash_n T_{1i} \sqsubseteq T_{2i} \qquad \forall i.\ \text{if } \theta_i \in \{\texttt{-},\texttt{=}\} \text{ then } \Gamma \vdash_n T_{2i} \sqsubseteq T_{1i}\end{array}}{\Gamma \vdash_n C[\overline{T_1}] <: C[\overline{T_2}]} \ \text{[U-Datatype]}$$

Figure 2.9. Subtyping for System D

System D, however, we require that all values be part of a *single* sort, in particular, the disjoint union of integers, strings, booleans, dictionaries, *etc.* Therefore, we require a background theory to axiomatize our refinement of this single sort. To start, we define *tag* to be an uninterpreted function symbol and require the following axioms:

$$\forall w.\ tag(w) = \begin{cases} \text{“}\texttt{boolean}\text{”} & \text{if } w \in \{\texttt{true},\texttt{false}\} \\ \text{“}\texttt{integer}\text{”} & \text{if } w \text{ is an integer} \\ \text{“}\texttt{string}\text{”} & \text{if } w \text{ is a string} \\ \text{“}\texttt{function}\text{”} & \text{if } w = \lambda x.e \text{ or } w \text{ is a primitive function} \\ \text{“}\texttt{dictionary}\text{”} & \text{if } w = empty \text{ or } w = upd(w_1, w_2, w_3) \\ \text{“}\texttt{null}\text{”} & \text{if } w = \texttt{null} \\ \text{“}\texttt{bot}\text{”} & \text{if } w = bot \end{cases}$$

For dictionaries, we define *sel* and *upd* also as uninterpreted functions, and require the following "wrapper" axioms which correspond to McCarthy's theory of arrays extended with a default element [71, 28]:

$$\forall d,k,k',x. \quad tag(d) = \text{“}\texttt{dictionary}\text{”} \wedge\ k = k' \ \Rightarrow\ sel(upd(d,k,x),k') = x$$
$$\forall d,k,k',x. \quad tag(d) = \text{“}\texttt{dictionary}\text{”} \wedge\ k \neq k' \ \Rightarrow\ sel(upd(d,k,x),k') = sel(d,k')$$
$$\forall k. \quad sel(empty,k) = bot$$

Continuing in this style, we can encode the function symbols and axioms of any decidable theory that we wish to use (*e.g.* linear arithmetic) by defining corresponding uninterpreted function symbols and wrapper axioms in our background theory to interpret them.

Although our background theory uses quantified axioms, we are yet able to ask only decidable queries of the SMT solver; because we have a finite number of axioms and a finite number of bindings in any environment Γ, we can explicitly instantiate all axioms with all bindings when embedding Γ as a formula. We omit this expansion from our definition of $Embed(\Gamma)$ above. In this way, all we require from the SMT solver is the ability to reason about the (decidable) theory of uninterpreted functions. In our implementation [80], however, we explicitly axiomatize the above background theory using quantifiers rather than instantiating it for each query, because modern SMT solvers like Z3 [27] have quantifier-instantiation heuristics that tend to perform well on the kinds of queries generated by System D and other SMT-based verification systems.

Recap. Recall that our goal is to type check programs that use value-indexed dictionaries which may contain functions as values. On 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 that arise during in type checking 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 dynamic language programs that manipulate complex data, including dictionaries which map keys to function values.

2.4 Type Soundness

In this section, we discuss type soundness for System D. Unfortunately, the presence of nested refinements means that the standard proof techniques for proving preservation and progress are 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_n for which we establish preservation and progress properties. The soundness of System D follows, as it is a special case of the stratified System D_n.

2.4.1 The Problem: Substitution

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. The standard substitution property in a refinement type system requires the following:

Proposition (Substitution)**.**
If $x\!:\!T, \Gamma \vdash e :: T'$ *and* $\vdash v :: T$*, then* $\Gamma[v/x] \vdash e[v/x] :: T'[v/x]$*.*

The following snippet shows why System D lacks this property:

```
(*: foo :: f:(Int → Int) → {y | f :: Int → Int} *)
let foo f = 0 in
foo (fun n -> n + 1)
```

The return type of the function annotation 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 judgement must be derivable:

$$f\!:\!\mathit{Int} \rightarrow \mathit{Int} \vdash 0 :: \{y \mid f :: \mathit{Int} \rightarrow \mathit{Int}\} \tag{2.3}$$

By T-Const, T-Sub, S-Refine, I-Cnf, and I-Valid, the judgement reduces to the implication

$$\mathit{true} \wedge f :: \mathit{Int} \rightarrow \mathit{Int} \wedge \mathit{ty}(0)(y) \Rightarrow f :: \mathit{Int} \rightarrow \mathit{Int}.$$

which is trivially valid, thereby deriving Equation 2.3, and showing that `foo` does indeed have the ascribed type. If the substitution property is to hold, then the following judgement obtained by substituting `(fun n -> n + 1)` for f in Equation 2.3 must be derivable:

$$\vdash 0 :: \{y \mid \texttt{(fun x -> x + 1)} :: \mathit{Int} \rightarrow \mathit{Int}\}.$$

By T-Const, T-Sub, S-Refine, I-Cnf, and I-Valid, the judgement reduces to the implication

$$\mathit{true} \wedge \mathit{ty}(0)(y) \Rightarrow \texttt{(fun x -> x + 1)} :: \mathit{Int} \rightarrow \mathit{Int} \tag{2.4}$$

which is *invalid* as type predicates are *uninterpreted* in our refinement logic! Thus, the function body of `foo` before and after the call (via substitution) do not have the same type in System D, which illustrates the crux of the problem: the I-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

$$\mathcal{I} \models \lambda x.e :: x{:}T_1 \to T_2 \quad \text{if} \quad x{:}T_1 \vdash e :: T_2.$$

and we can replace I-Valid with the following rule:

$$\frac{\mathcal{I} \models \mathit{Embed}(\Gamma) \Rightarrow p}{\Gamma \Rightarrow p}\ [\textsc{I-Valid-Interpreted}]$$

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 Equation 2.4. Although the rule solves the problem with substitution, it introduces a new problem: a circular dependence between the typing judgements and the interpretation $\mathcal{I}$. Since our refinement logic includes negation, the type system corresponding to the set of rules outlined earlier combined with I-Valid-Interpreted is not necessarily well-defined.

2.4.2 The Solution: Stratified System D

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. In this section, we formalize this intuition and state the important lemmas and theorems. The full definitions and proofs may be found in Appendix A.

To achieve stratification, first, we index interpretations ($\mathcal{I}_n$) as well as typing, subtyping, and implication judgements ($\vdash_n n$); recall that we ignored these subscripts in our prior discussion. We call these the level-n interpretations and judgements, respectively. Second, we allow level-n judgements to use the I-Valid-n rule from Figure 2.9

$$\frac{\mathcal{I}_n \models \mathit{Embed}(\Gamma) \Rightarrow p}{\Gamma \Rightarrow_n p}\ [\textsc{I-Valid-n}]$$

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

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

Finally, we write

$$\Gamma \vdash_{*} e :: T \quad \text{if} \quad \exists n.\, \Gamma \vdash_{n} e :: T.$$

The derivations in System D_n consist of the level-n derivations. Derivations in System D_0, which we abbreviate to System D, may use all rules *except* I-VALID-N; these are the derivations used for type checking closed expressions. In the sequel, we often omit the index from a typing judgement when its index is 0.

The following "lifting" lemma states that the derivations at each level include the derivations at all lower levels:

Lemma (Lifting Derivations)**.**
If $\Gamma \vdash_n e :: T$*, then* $\Gamma \vdash_{n+1} e :: T$*.*

Stratification snaps the circularity knot and enables the proof of the following stratified substitution lemma:

Lemma (Stratified Substitution)**.**
If $x{:}T, \Gamma \vdash_n e :: T'$ *and* $\vdash_n v :: T$*, then* $\Gamma[v/x] \vdash_{n+1} e[v/x] :: T'[v/x]$*.*

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

Lemma (Satisfiable Typing)**.**
If $\vdash_n v :: T$*, then* $\mathcal{I}_{n+1} \models T(v)$*.*

Stratified substitution enables the proof of the following preservation result:

Theorem (Stratified Preservation)**.**
If $\vdash_n e :: T$*, and* $e \hookrightarrow e'$ *then* $\vdash_{n+1} e' :: T$*.*

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

Theorem (Stratified System D_n Type Soundness)**.**
If $\vdash_{*} e :: T$*, then either* e *is a value or* $e \hookrightarrow e'$ *and* $\vdash_{*} e' :: T$*.*

The soundness of System D follows as a corollary. We flesh out the details of our proof strategy in Appendix A.

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing.** In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2012.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing (Technical Appendix).** `arXiv:1103.5055v2 [cs.PL]`, September 2011.
- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Dependent Dynamic Dictionaries.** `arXiv:1103.5055v1 [cs.PL]`, March 2011.

Chapter 3

Algorithmic Typing

We established the expressiveness and soundness of nested refinements in the previous chapter, but that *declarative* presentation leaves several challenges to address in order to implement a nested refinement type checker for System D. In this chapter, we discuss these challenges and present an *algorithmic* version of the type system, which we refer to as Algorithmic System D, that employs local type inference [78, 31] and several techniques unique to our setting. The techniques described in this chapter form the basis for our implementation of DJS [80].

Syntax. As we saw in the examples of the previous section, we require that the programmer explicitly annotates function definitions. In Figure 3.1, we extend the syntax of System D to allow annotated let-expressions. We also introduce a new syntactic category of types, *prenex-quantified existential types* S of the form $\exists x{:}T.\,S$, which are useful for type checking let- and if-expressions, as discussed in the sequel. Prenex types may *not* appear in source-level annotations; only the type checker can generate and manipulate them. We often write $\exists \overline{x}{:}\overline{T}.\,T'$ to at once describe the (zero or more) existential quantifiers of a prenex type. The syntax of type environments Γ remains the same, since we will always explicitly manipulate existential types before adding them to environments.

3.1 Algorithmic Subtyping

In this section, we describe two challenges in implementing *type extraction*, a central mechanism in System D subtyping, and how we address them with our algorithmic subtyping rules, defined in Figure 3.2.

Ensuring Termination. The System D subtyping, implication, and syntactic subtyping relations (defined in Figure 2.9) are mutually recursive: after we extract type terms from the environment during the implication phase, we recursively invoke syntactic subtyping and, hence, the top-level subtyping relation, which completes the loop. As it turns out, a straightforward implementation of those rules do not terminate because invocations of syntactic subtyping do not satisfy the usual

$$
\begin{array}{rcll}
\textbf{Values} & v & ::= & \cdots \\
\textbf{Expressions} & e & ::= & \cdots \mid \texttt{let}\ x : T_1 = e_1\ \texttt{in}\ e_2 \\
\textbf{Types} & T & ::= & \cdots \\
\textbf{Prenex Types} & S & ::= & \exists x{:}T.\, S \mid T \\
\textbf{Formulas} & p,q & ::= & \cdots \\
\textbf{Logical Values} & w & ::= & \cdots \\
\textbf{Type Terms} & U & ::= & \cdots \\
\\
\textbf{Type Synthesis} & \multicolumn{3}{l}{\Gamma \vdash e \rhd S} \\
\textbf{Type Conversion} & \multicolumn{3}{l}{\Gamma \vdash e \lhd T} \\
\textbf{Subtyping} & \multicolumn{3}{l}{\Gamma;\, \mathfrak{U} \vdash S \sqsubseteq T} \\
\textbf{Syntactic Subtyping} & \multicolumn{3}{l}{\Gamma;\, \mathfrak{U} \vdash U_1 <: U_2} \\
\textbf{Implication} & \multicolumn{3}{l}{\Gamma;\, \mathfrak{U} \Rightarrow p}
\end{array}
$$

Figure 3.1. Syntax of Algorithmic System D

guarantee that they refer to syntactically smaller terms! To understand why, consider the environment $\Gamma_0 \circeq f{:}Any, f'{:}\{f'' \mid f'' = f \wedge f'' :: U_0\}$ where $U_0 \circeq x{:}\{y \mid y :: x{:}\{z \mid z = f\} \rightarrow Any\} \rightarrow Any$ and suppose we wish to check that

$$\Gamma_0 \Rightarrow f :: x{:}\{z \mid z = f\} \rightarrow Any. \tag{3.1}$$

I-Valid cannot derive this judgement, since the implication $Embed(\Gamma_0) \Rightarrow f :: x{:}\{z \mid z = f\} \rightarrow Any$ is not valid. Thus, we must derive Equation 3.1 by I-HasType. Type extraction (the first premise of I-HasType) derives that $f :: U_0$ in Γ_0, so the remaining obligation is

$$\Gamma \vdash U_0 <: x{:}\{z \mid z = f\} \rightarrow Any.$$

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

$$\Gamma \vdash \{z \mid z = f\} \sqsubseteq \{y \mid y :: x{:}\{z \mid z = f\} \rightarrow Any\}.$$

After S-Refine substitutes a fresh variable, say yz, for both refinement binders, this reduces to the implication obligation

$$\Gamma, yz = f \Rightarrow yz :: x{:}\{z \mid z = f\} \rightarrow Any.$$

Alas, this is essentially Equation 3.1, so we are stuck in an infinite loop! We will again extract the type U_0 for f (aliased to yz here) and repeat the process ad infinitum.

This situation arises only if we are allowed to invoke the rule I-HasType infinitely many times. In this case, I-HasType extracts a single type term from the environment infinitely often, since there are only finitely many in the environment. In the algorithmic system, we cut the loop with a modest restriction: along any branch of a subtyping derivation, we allow a particular 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 define the algorithmic subtyping relations to take a set $\mathfrak{U}$ of "already-used" type terms as an additional parameter, which cannot be extracted by the (algorithmic analog to) rule I-HasType. This strategy prevents Algorithmic System D from deriving certain infinite System D derivations, of the pathological kind just discussed, but we are not aware of any interesting finite derivations that are precluded by this strategy.

Type Extraction. Nearly all the declarative subtyping rules presented in Figure 2.9 are directed by the structure of the judgement being derived. The sole exception is I-HasType, whose first premise requires us to synthesize an unspecified type term U' such that the SMT solver can prove $w :: U'$. 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 the following procedure $\mathit{MustFlow}(\Gamma, T, \mathfrak{U})$ that uses the SMT solver to compute the set $\mathfrak{U}'$ of type terms out of all possible type terms mentioned in Γ except those that have been previously extracted (to ensure termination), such that for all values x, the binding $x\!:\!T$ implies that $x :: U'$.

$$\begin{aligned}\mathit{MustFlow}(\Gamma, T, \mathfrak{U}) \;\mathrel{\hat{=}}\; &\{\, U' \in \mathfrak{U}' \mid \mathit{Valid}(\mathit{Embed}(\Gamma, x\!:\!T) \Rightarrow x :: U') \,\}\\ &\text{where } \; \mathfrak{U}' = \mathit{TypeTermsOf}(\Gamma) \setminus \mathfrak{U} \text{ and } x \text{ is fresh}\end{aligned}$$

The procedure *TypeTermsOf* (not shown) simply traverses the environment and collects the top-level type terms in formulas. The rule IA-HasType calls $\mathit{MustFlow}(\Gamma, \{x \mid x = w\}, \mathfrak{U})$ to compute the set $\mathfrak{U}'$ of type terms that might be needed by the second premise. Since the declarative rule cannot possibly refer to a type term not in Γ, this strategy guarantees that $U' \in \mathfrak{U}'$ and, thus, that IA-HasType does not forfeit precision. After type extraction, the set $\mathfrak{U}'$ is added to the set of already-extracted type terms when checking that U' is a syntactic subtype of U.

Definitions. In Figure 3.2, we define algorithmic subtyping rules that correspond to the declarative rules in Figure 2.9 (besides I-Valid-n), as well as an additional rule to manipulate prenex types. Compared to the presentation of the declarative system, the algorithmic relations maintain a set $\mathfrak{U}$ of "already-used" type terms to ensure termination. Also, notice that although the IA-Valid and IA-Cnf rules overlap, there is no need for backtracking, since we can simply try to discharge the obligation $\Gamma \Rightarrow p$ first with IA-Valid and then IA-Cnf upon failure.

As we will discuss in the next section, algorithmic typing derives prenex types for certain

Algorithmic Subtyping $\boxed{\Gamma;\, \mathfrak{U} \vdash S \sqsubseteq T}$

$$\frac{y \text{ fresh} \qquad \Gamma;\, \mathfrak{U} \Rightarrow (p_1[y/x_1] \Rightarrow p_2[y/x_2])}{\Gamma;\, \mathfrak{U} \vdash \{x_1 \mid p_1\} \sqsubseteq \{x_2 \mid p_2\}}\ [\text{SA-R}\textsc{efine}] \qquad \frac{\Gamma,\, x\!:\!T;\, \mathfrak{U} \vdash S_1 \sqsubseteq T_2}{\Gamma;\, \mathfrak{U} \vdash \exists x\!:\!T.\, S_1 \sqsubseteq T_2}\ [\text{SA-E}\textsc{xists}]$$

Algorithmic Implication $\boxed{\Gamma;\, \mathfrak{U} \Rightarrow p}$

$$\frac{\mathit{CNF}(p) = \wedge_i\ (q_i \Rightarrow Q_i') \qquad \forall i.\ \exists q' \in Q_i'.\ \Gamma, q_i;\, \mathfrak{U} \Rightarrow q'}{\Gamma;\, \mathfrak{U} \Rightarrow p}\ [\text{IA-C}\textsc{nf}]$$

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow p)}{\Gamma;\, \mathfrak{U} \Rightarrow p}\ [\text{IA-V}\textsc{alid}] \qquad \frac{\begin{array}{c} T = \{x \mid x = w\} \qquad \mathit{MustFlow}(\Gamma, T, \mathfrak{U}) = \mathfrak{U}' \\ \exists U' \in \mathfrak{U}'.\ \Gamma;\, \mathfrak{U} \cup \mathfrak{U}' \vdash U' <: U \end{array}}{\Gamma;\, \mathfrak{U} \Rightarrow w :: U}\ [\text{IA-H}\textsc{as}\text{T}\textsc{ype}]$$

Algorithmic Syntactic Subtyping (omitted) $\boxed{\Gamma;\, \mathfrak{U} \vdash U_1 <: U_2}$

Figure 3.2. Subtyping for Algorithmic System D

kinds of expressions but ensures that they will only appear on the left side of subtyping obligations. To handle such an obligation, the SA-Exists rule transfers existential bindings from a prenex type to the type environment, following the approach in [66]. Since we do not need to discharge subtyping queries where prenex types are on the right side, we avoid the need to "guess witnesses" that is algorithmically problematic [66].

Incremental Environments. A noteworthy, but unsurprising, optimization in our implementation [80] compared to the algorithmic system presented here is that we maintain the environment of logical assumptions incrementally. We add and remove assertions to and from the environment whenever the type system manipulates the type environment, so that by the time the IA-Valid rules needs to check $\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow p)$, the formula $\mathit{Embed}(\Gamma)$ is already in the background assumptions of the environment; only $\mathit{Valid}(p)$ needs to be discharged. For clarity, we refrain from threading this stack of background facts through all the typing and subtyping relations.

3.2 Bidirectional Type Checking

So that most expressions except function definitions may remain unannotated, we define a *bidirectional* type checking algorithm, following work on local type inference [78, 31], that comprises two parts: a *type conversion* judgement $\Gamma \vdash e \lhd T$ that checks whether e has type T in the environment Γ, where T is required either by a type checking context or a source-level

Type Synthesis (selected rules) $\boxed{\Gamma \vdash e \rhd S}$

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow \mathit{false})}{\Gamma \vdash e \rhd \{x \,|\, \mathit{false}\}}\;[\textsc{TS-False}] \qquad \frac{}{\Gamma \vdash c \rhd \mathit{ty}(c)}\;[\textsc{TS-Const}]$$

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash x \rhd \{y \,|\, y = x\}}\;[\textsc{TS-Var}] \qquad \frac{\Gamma + x\!:\!\mathit{Any} \vdash e \rhd T_2}{\Gamma \vdash \lambda x.e \rhd \{y \,|\, y :: x\!:\!\mathit{Any} \to T_2\}}\;[\textsc{TS-Fun}]$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \rhd T_1 \qquad \mathit{MustFlow}(\Gamma, T_1, \emptyset) = \mathfrak{U} \\ \mathfrak{U} = \{\forall \overline{A}.\ x\!:\!T_{11} \to T_{12}\} \qquad \Gamma \vdash v_2 \lhd \mathit{Inst}(T_{11}, \overline{A}, \overline{T'})\end{array}}{\Gamma \vdash [\overline{T'}]\ v_1\ v_2 \rhd \mathit{Inst}(T_{12}, \overline{A}, \overline{T})[v_2/x]}\;[\textsc{TS-App}]$$

$$\frac{\Gamma \vdash e_1 \rhd \exists \overline{y}\!:\!\overline{T'}.\, T_1 \qquad \Gamma + \overline{y}\!:\!\overline{T'} + x\!:\!T_1 \vdash e_2 \rhd S_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rhd \exists \overline{y}\!:\!\overline{T'}.\, \exists x\!:\!T_1.\, S_2}\;[\textsc{TS-Let}]$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash e_1 \lhd T_1 \qquad \Gamma + x\!:\!T_1 \vdash e_2 \rhd S_2}{\Gamma \vdash \texttt{let}\ x : T_1 = e_1\ \texttt{in}\ e_2 \rhd \exists x\!:\!T_1.\, S_2}\;[\textsc{TS-Let}']$$

$$\frac{\Gamma \vdash v \lhd \mathit{Bool} \qquad \Gamma, v = \texttt{true} \vdash e_1 \rhd S_1 \qquad \Gamma, v = \texttt{false} \vdash e_2 \rhd S_2}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd \mathit{Join}(v, S_1, S_2)}\;[\textsc{TS-If}]$$

Figure 3.3. Type Synthesis for Algorithmic System D

annotation; and a *type synthesis* judgement $\Gamma \vdash e \rhd S$ that derives the (prenex) type S for e when there is no expected type for e. The type checking relations are shown in Figure 3.3 and Figure 3.4. As usual, uses of T-Sub from the declarative system are factored into other typing rules. In this section, we highlight the novel aspects of bidirectional type checking in our setting.

Inconsistent Environments. Recall that the type extraction procedure collects the type terms U such that $\mathit{Valid}(\mathit{Embed}(\Gamma, x\!:\!T) \Rightarrow x :: U)$. If the environment $\Gamma, x\!:\!T$ happens to be inconsistent (*e.g.* due to an unreachable branch of an if-expression), then all such implications will be valid. As we will see, our typing rules for function application depend on type extraction returning exactly one syntactic arrow, which is not the case in an inconsistent environment. To avoid this situation, the synthesis rule TS-False and conversion rule TC-False first check whether the environment is inconsistent, and if it is, they trivially succeed, avoiding the need for any subsequent type extractions. This approach is sound because when the environment is inconsistent, all implication obligations can be discharged by the SMT solver anyway.

Type Conversion (selected rules) $\boxed{\Gamma \vdash e \lhd T}$

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow \mathit{false})}{\Gamma \vdash e \lhd T}\;[\text{TC-}\textsc{False}] \qquad \frac{\Gamma \vdash c \rhd S \quad \Gamma; \varnothing \vdash S \sqsubseteq T}{\Gamma \vdash c \lhd T}\;[\text{TC-}\textsc{Const}]$$

$$\frac{\Gamma \vdash x \rhd S \quad \Gamma; \varnothing \vdash S \sqsubseteq T}{\Gamma \vdash x \lhd T}\;[\text{TC-}\textsc{Var}] \qquad \frac{\Gamma, \overline{A} + x{:}T_1 \vdash e \lhd T_2}{\Gamma \vdash \lambda x.e \lhd \{y \mid y :: \forall \overline{A}.\ x{:}T_1 \to T_2\}}\;[\text{TC-}\textsc{Fun}]$$

$$\frac{\Gamma \vdash [\overline{T'}]\ v_1\ v_2 \rhd S \quad \Gamma; \varnothing \vdash S \sqsubseteq T}{\Gamma \vdash [\overline{T'}]\ v_1\ v_2 \lhd T}\;[\text{TC-}\textsc{App}]$$

$$\frac{\Gamma \vdash e_1 \rhd \exists \overline{y}{:}\overline{T'}.\, T_1 \quad \Gamma + \overline{y}{:}\overline{T'} + x{:}T_1 \vdash e_2 \lhd T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \lhd T_2}\;[\text{TC-}\textsc{Let}]$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash e_1 \lhd T_1 \quad \Gamma + x{:}T_1 \vdash e_2 \lhd T}{\Gamma \vdash \texttt{let}\ x : T_1 = e_1\ \texttt{in}\ e_2 \lhd T_2}\;[\text{TC-}\textsc{Let}']$$

$$\frac{\Gamma \vdash v \lhd \mathit{Bool} \quad \Gamma, v = \texttt{true} \vdash e_1 \lhd T \quad \Gamma, v = \texttt{false} \vdash e_2 \lhd T}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \lhd T}\;[\text{TC-}\textsc{If}]$$

Figure 3.4. Type Conversion for Algorithmic System D

Unfolding. The declarative rule T-Unfold is not syntax-directed, and we cannot predict where it will be needed since we do not have pattern matching to determine exactly when to unfold type definitions, as in languages like ML. Instead, in the algorithmic system, we eagerly unfold type definitions to anticipate all situations in which (one level of) unfolding might be required. In particular, whenever the declarative system extends a type environment with a new binding, using $\Gamma, x{:}T$, the algorithmic system uses the operation $\Gamma + x{:}T$ that, in addition to adding the binding, checks whether x satisfies any constructed types $C[\overline{T}]$, and if so, unfolds the corresponding type definitions and records them in the environment.

$$\Gamma + x{:}T \ \stackrel{\circ}{=}\ \Gamma, x{:}T, \wedge_{C[\overline{T'}] \in \mathfrak{U}}\ \mathit{Unfold}(x, C, \overline{T'}) \text{ where } \mathfrak{U} = \mathit{MustFlow}(\Gamma, \{y \mid y = x\}, \varnothing)$$

Values. For constants and variables, the synthesis rules TS-Const and TS-Var are similar to the declarative typing rules, and the conversion rules TC-Const and TC-Var invoke synthesis and then call into subtyping to check the synthesized type against the goal. The rules for dictionaries and constructed data (not shown) are similar. When the binder x of a lambda expression is not annotated, TS-Fun simply checks the body assuming that x has type *Any*. When converting a function to type T, TC-Fun requires that T syntactically have the form $\{f \mid f :: U\}$ where U is a function type. We can relax this restriction if necessary, for example, to allow intersections of function types.

Function Application. To synthesize a type for $[\overline{T'}]\ v_1\ v_2$, TS-App first synthesizes a type T_1 for the function v_1 and uses type extraction to convert T_1 to a syntactic function type $\forall\overline{A}.\ x : T_{11} \to T_{12}$. To avoid the need for backtracking in the type checker, TS-App requires that there is *exactly one* syntactic function type that flows to v_1. If there are multiple, we report an error for the application expression. Again, we can relax this restriction to allow intersections of function types, if necessary. From this point, the rule proceeds like the declarative rule for function application. The type conversion rule TC-App first synthesizes the application and then performs a subtyping check against the expected type.

Let-expressions. When deriving the type T_2 for a let-expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, the declarative T-Let rule checks that x is not mentioned in T_2, since otherwise T_2 would not be well-formed in the surrounding environment. When both e_1 and the entire let-expression are annotated with expected types, the TC-Let′ rule ensures that the type T_2 is well-formed (assuming that all source-level annotations are checked for well-formedness and that the type system generates only well-formed conversion checks, both of which are indeed true). More difficult are the cases where at least one of these annotations is missing, because the type derived for the expression body e_2 will usually *not* be well-formed due to the variables inserted inside formulas by the TS-Var rule. We would like to allow most intermediate let-expressions to remain unannotated, however.

To resolve this issue, we use existential types to deal with the binder x that must go out of scope. For example, the TS-Let′ synthesizes the type $\exists x : T_1.\ S_2$ for the let-expression $\texttt{let}\ x : T_1 = e_1\ \texttt{in}\ e_2$, where T_1 is the successfully verified type annotation for the expression e_1, and S_2 is the derived type for e_2 in an environment extended with the let-bound variable x. Because the type S_2 of e_2 is guaranteed to be in prenex form, $\exists x : T_1.\ S_2$ is, too. The rules TS-Let and TC-Let handle unannotated let-expressions using existentials in similar fashion.

If-expressions. The type conversion rule TC-If follows the corresponding declarative rule. To synthesize a precise type for an if-expression, we combine the synthesized types of both branches by guarding them with the appropriate branch conditions. A first attempt at this approach is the following, where the synthesized type is an exact join of the branches.

$$\frac{\Gamma \vdash v \lhd \mathit{Bool} \qquad \Gamma, v = \texttt{true} \vdash e_1 \rhd \{x \mid p_1\} \qquad \Gamma, v = \texttt{false} \vdash e_2 \rhd \{x \mid p_2\}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd \{x \mid (v = \texttt{true} \Rightarrow p_1) \land (v = \texttt{false} \Rightarrow p_2)\}}$$

This rule applies when both branches have refinement types, but type synthesis might derive a prenex type for one or both branches. To handle the general case, the rule TS-If uses the operator

Join(v, S_1, S_2), defined below, to first move the existential binders for each branch to the top-level, ensuring that the result is still in prenex form. Rearranging variables in this way is sound because we assume that, by convention, all let-bound variables in a program are distinct.

$$\begin{aligned}
\mathit{Join}(v, (\exists \overline{x}\!:\!\overline{T'}.\, T_1), S_2) &= \exists \overline{x}\!:\!(\overline{\mathit{Join}(v, T', \mathit{Any})}).\, \mathit{Join}(v, T_1, S_2) \\
\mathit{Join}(v, T_1, (\exists \overline{x}\!:\!\overline{T'}.\, T_2)) &= \exists \overline{x}\!:\!(\overline{\mathit{Join}(v, \mathit{Any}, T')}).\, \mathit{Join}(v, T_1, T_2) \\
\mathit{Join}(v, T_1, T_2) &= \{x \mid (v = \mathtt{true} \Rightarrow T_1(x)) \wedge (v = \mathtt{false} \Rightarrow T_2(x))\}
\end{aligned}$$

Type Parameter Inference. There are two sources of type variable polymorphism in System D. For polymorphic function applications, we use the standard "greedy" approach [78] (*i.e.* unification) to infer type instantiations when possible. For constructed data expressions of polymorphic type, 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:

$$\mathtt{type}\ \mathit{List}[\mathtt{+}A] = \{\ \text{“}\mathtt{hd}\text{”}\!:\!A;\ \text{“}\mathtt{tl}\text{”}\!:\!\mathit{List}[*A]?\ \}$$

Due to the presence of the marker $*$ in the type of the "`tl`" field, local type inference will use the type of v_2 (rather than the type of v_1, which will often synthesize to a more specific type than that satisfied by all elements of the list) to infer the omitted type parameter in *List*$(v_1,\ v_2)$. We elide both of these strategies from the rules.

3.3 Soundness

Each of the techniques in the algorithmic system are designed to be sound with respect to the declarative system, so that the properties below hold. The procedure *Erase* removes type annotations from annotated let-expressions to match the syntax of the declarative system. Notice that for type synthesis judgements, the resulting prenex type $S = \exists \overline{x}\!:\!\overline{T}.\, T'$ must be manipulated so that the existentially quantified variable bindings $\overline{x}\!:\!\overline{T}$ are moved to the typing environment and that the underlying refinement type T' is the goal for the declarative judgement. Also notice that the property for subtyping mentions only refinement types T_1 and T_2, not prenex types.

Proposition (Sound Algorithmic Typing)**.**

1. *If* $\Gamma;\, \mathfrak{U} \Rightarrow p$*, then* $\Gamma \Rightarrow p$*.*
2. *If* $\Gamma;\, \mathfrak{U} \vdash U_1 <: U_2$*, then* $\Gamma \vdash U_1 <: U_2$*.*
3. *If* $\Gamma;\, \mathfrak{U} \vdash T_1 \sqsubseteq T_2$*, then* $\Gamma \vdash T_1 \sqsubseteq T_2$*.*
4. *If* $\Gamma \vdash e \rhd \exists \overline{x}\!:\!\overline{T}.\, T'$*, then* $\Gamma, \overline{x}\!:\!\overline{T} \vdash \mathit{Erase}(e) :: T'$*.*
5. *If* $\Gamma \vdash e \lhd T$*, then* $\Gamma \vdash \mathit{Erase}(e) :: T$*.*

Proof sketch. We provide the intuition for why the above properties should hold, but we do not present a detailed proof. To show that algorithmic implication is sound with respect to declarative implication, consider IA-HasType and its type extraction procedure. It is easy to see that uses of *MustFlow* can be converted into derivations by I-Valid, since it depends on the validity of logical implications. Showing that algorithmic subtyping and syntactic subtyping are sound with respect to their declarative counterparts goes by induction on their derivation rules, which correspond one-to-one (except for SA-Exists, which is inapplicable for property (3)).

To show that type synthesis and type conversion are sound with respect to declarative typing, there are several key aspects to consider. First, the initial checks for an inconsistent type environment by TS-False and TC-False are sound because the I-Valid can check arbitrary implications given an inconsistent environment. Second, T-Unfold can be used in places where unfolded type definitions are eagerly added to environments. Third, the subtyping premises used in the algorithmic rules can be replaced with uses of T-Sub. Finally, the synthesis rules for let- and if-expressions derive prenex types, whose quantifiers can be moved to the environment to match the structure of proposition (4).

Incompleteness. As discussed earlier, the algorithmic subtyping relations limit the type terms that may be extracted to ensure termination. As a result, there are judgements derivable in System D that are not derivable Algorithmic System D. We conjecture that the difference in expressiveness does not preclude useful judgements, but it would be interesting to study the difference in detail in future work. Furthermore, System D derivations that require more than one level of unfolding for an expression are not derivable with our unfolding heuristic.

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing.** In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2012.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing (Technical Appendix).** `arXiv:1103.5055v2 [cs.PL]`, September 2011.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Dependent Dynamic Dictionaries.** `arXiv:1103.5055v1 [cs.PL]`, March 2011.

Chapter 4

Extensions to Subtyping

Having presented both declarative and algorithmic versions of System D, we now describe a few extensions to subtyping that increase expressiveness. We prove the extensions sound and discuss how to incorporate them into the algorithmic system.

Nested Refinement Subtyping. Recall the following two rules, which are central to System D subtyping, that normalize formulas into implication queries and combine SMT and syntactic reasoning to derive has-type predicates:

$$\frac{\mathit{CNF}(p) = \wedge_i \ (q_i \Rightarrow Q'_i) \qquad \forall i.\ \exists q' \in Q'_i.\ \Gamma, q_i \Rightarrow_n q'}{\Gamma \Rightarrow_n p} \ [\text{I-}\textsc{Cnf}]$$

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow w :: U') \qquad \Gamma \vdash_n U' <: U}{\Gamma \Rightarrow_n w :: U} \ [\text{I-}\textsc{HasType}]$$

By extracting syntactic type constructors from refinement types, System D can manipulate syntactic types in the premises of typing rules in familiar ways. For example, the T-App rule, which is used to type check the function application $v_1\ v_2$, requires that $v_1 :: \{y \,|\, y :: x\!:\!T_{11} \rightarrow T_{12}\}$; to satisfy this obligation, the I-HasType rule is responsible for proving that $v_1 :: x\!:\!T_{11} \rightarrow T_{12}$.

4.1 Joins

Let us consider a situation in which the approach for deriving implications above falls short. Consider the function type terms

$$U_1 \ \mathrel{\hat{=}}\ \mathit{Any} \rightarrow \{y \,|\, \mathit{Int}(y) \wedge y \geq 0\} \qquad\qquad U_2 \ \mathrel{\hat{=}}\ \mathit{Any} \rightarrow \{y \,|\, \mathit{Int}(y) \wedge y \leq 0\}$$

and a value `getInt` $:: \{x \,|\, x :: U_1 \vee x :: U_2\}$ that could, for example, be the result of an if-expression that chooses between functions of the more specific types. Intuitively, `getInt` has type $\mathit{Any} \rightarrow \mathit{Int}$, so we might expect the application `getInt` “`unit`” to be well-typed, but the T-App and I-HasType rules cannot derive the required function type. In order to treat a value x as having syntactic type U, the first premise of I-HasType requires that it *definitely* have some syntactic subtype U' that the SMT solver can prove. When the it can prove only that x has *either* type U_1 or U_2, I-HasType does not provide the means to establish $x :: U$, even when both are syntactic subtypes of U.

To handle such situations, we can make the following two additions to System D: a *join* operator, defined in Figure 4.1, that combines syntactic type terms U_1 and U_2 into a single overapproximate type term U; and the following rule which generalizes I-HasType:

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow w :: U_1 \vee \cdots \vee w :: U_k) \qquad \Gamma \vdash_n (U_1 \sqcup \cdots \sqcup U_k) <: U}{\Gamma \Rightarrow_n w :: U}\ \text{[I-HasType-Join]}$$

Although many syntactic systems with subtyping include a join operator (see [77], for example), there are several interesting aspects in our setting. First, the join of function types uses logical disjunction and conjunction — rather than recursively using join and a separate meet operator — to implement the typical contravariance of argument types and covariance of return types. Second, we include the join operator in our declarative system — rather than only in our algorithmic system, as is usually the case, where they are helpful for computing minimal types [77] — which is not surprising given the “algorithmic flavor” of subtyping in System D, where we manipulate the structure of formulas. Lastly, when joining two instantiations $C[T_1]$ and $C[T_2]$ of an invariant type constructor (the fourth equation in the definition of join), we define type equality

$$\{x \mid p\} \equiv \{x \mid q\} \ \mathrel{\hat{=}}\ \mathit{Valid}(p \Leftrightarrow q)$$

via equisatisfiability rather than syntactic equality to determine equivalence. The combination of I-HasType-Join and T-App (without modification) allows the problematic application `getInt` “`unit`” to type check, because the join of U_1 and U_2 is $\mathit{Any} \to \mathit{Int}$. For simplicity, the definition of join on constructed types in Figure 4.1 shows only the case for type constructors with exactly one type argument; the general case is similar.

4.2 Meets

In the previous section, we added joins to handle situations where the SMT solver can prove that a value x satisfies *one of* a set of syntactic type terms $\mathfrak{U}$. Now we present a similar extension to handle situations where x satisfies *all of* $\mathfrak{U}$. For example, consider a function `getZero` :: $\{x \mid x :: U_1 \wedge x :: U_2\}$ and the application `getZero` “`unit`” There are (at least) two possibilities for the type of the resulting value, either $\{y \mid \mathit{Int}(y) \wedge y \geq 0\}$ or $\{y \mid \mathit{Int}(y) \wedge y \leq 0\}$. But since `getZero` satisfies *both* function types, we might want the more precise result type $\{y \mid \mathit{Int}(y) \wedge y \geq 0 \wedge y \leq 0\}$ — that is, $\{y \mid y = 0\}$ — which is not derivable in the system to this point.

Joins and Meets $\boxed{U_1 \sqcup U_2}\ \boxed{U_1 \sqcap U_2}$

$$
\begin{array}{rrcrcl}
 & A & \sqcup & A & = & A \\
(\texttt{type}\ C[\texttt{+}A] = \{\ \cdots\ \}) & C[T_1] & \sqcup & C[T_2] & = & C[T_1 \vee T_2] \\
(\texttt{type}\ C[\texttt{-}A] = \{\ \cdots\ \}) & C[T_1] & \sqcup & C[T_2] & = & C[T_1 \wedge T_2] \\
(\texttt{type}\ C[\texttt{=}A] = \{\ \cdots\ \}) & C[T_1] & \sqcup & C[T_2] & = & C[T_1]\ \ \text{if } T_1 \equiv T_2 \\
 & \forall\overline{A}.\ x\!:\!T_{11} \to T_{12} & \sqcup & \forall\overline{A}.\ x\!:\!T_{21} \to T_{22} & = & \forall\overline{A}.\ x\!:\!(T_{11} \wedge T_{21}) \to (T_{12} \vee T_{22})
\end{array}
$$

$$
\begin{array}{rrcrcl}
 & A & \sqcap & A & = & A \\
(\texttt{type}\ C[\texttt{+}A] = \{\ \cdots\ \}) & C[T_1] & \sqcap & C[T_2] & = & C[T_1 \wedge T_2] \\
(\texttt{type}\ C[\texttt{-}A] = \{\ \cdots\ \}) & C[T_1] & \sqcap & C[T_2] & = & C[T_1 \vee T_2] \\
(\texttt{type}\ C[\texttt{=}A] = \{\ \cdots\ \}) & C[T_1] & \sqcap & C[T_2] & = & C[T_1]\ \ \text{if } T_1 \equiv T_2 \\
 & \forall\overline{A}.\ x\!:\!T_{11} \to T_{12} & \sqcap & \forall\overline{A}.\ x\!:\!T_{21} \to T_{22} & = & \forall\overline{A}.\ x\!:\!(T_{11} \vee T_{21}) \to T_2 \\
 & & & & & \quad T_2 = \{y \mid (T_{11}(x) \Rightarrow T_{12}(y)) \\
 & & & & & \qquad\qquad \wedge\ (T_{21}(x) \Rightarrow T_{22}(y))\}
\end{array}
$$

Union and Intersection Types $\boxed{T_1 \vee T_2}\ \boxed{T_1 \wedge T_2}$

$$\{x \mid p\} \vee \{x \mid q\} = \{x \mid p \vee q\} \qquad\qquad \{x \mid p\} \wedge \{x \mid q\} = \{x \mid p \wedge q\}$$

Figure 4.1. Joins and Meets for System D

We make two additions to handle situations like this: a *meet* operator, defined in Figure 4.1, that combines U_1 and U_2 into a common syntactic subtype U; and the following analog to I-HasType-Join:

$$\frac{\mathit{Valid}(\mathit{Embed}(\Gamma) \Rightarrow w :: U_1 \wedge \cdots \wedge w :: U_k) \qquad \Gamma \vdash_n (U_1 \sqcap \cdots \sqcap U_k) <: U}{\Gamma \Rightarrow_n w :: U}\ \text{[I-HasType-Meet]}$$

Like the join operator, meet is defined with logical connectives rather than recursively using meet and join. Notice that when computing the meet of two function types, the return type predicates are guarded by their corresponding argument types to ensure soundness. With these extensions, System D is able to derive the type $\{y \mid y = 0\}$ for the application `getZero` “`unit`”.

4.3 Formula Normalization

The last extension we present addresses situations that involve mixing "all-of" and "one-of" has-type predicates. Consider the function types

$$\begin{array}{lcllcl} U_1 & \mathrel{\hat{=}} & \mathit{Any} \to \{y \mid y \geq 0\} & U_3 & \mathrel{\hat{=}} & \mathit{Any} \to \{y \mid y = 0\} \\ U_2 & \mathrel{\hat{=}} & \mathit{Any} \to \{y \mid y \leq 1\} & U_4 & \mathrel{\hat{=}} & \mathit{Any} \to \{y \mid y = 1\} \end{array}$$

and the following implication query:

$$\begin{gathered} \mathit{Embed}(\mathit{\Gamma}_0) \Rightarrow x :: \mathit{Any} \to \{y \mid y = 0 \vee y = 1\} \\ \text{where } \Gamma_0 \ \mathrel{\hat{=}} \ (x :: U_1 \wedge x :: U_2) \vee x :: U_3 \vee x :: U_4 \end{gathered}$$

The I-HasType-Meet rule does not apply, since there is no valid conjunction of type predicates. The I-HasType-Join rule does apply because we can derive the formula

$$\mathit{Valid}(\mathit{Embed}(\mathit{\Gamma}_0) \Rightarrow x :: U_1 \vee x :: U_2 \vee x :: U_3 \vee x :: U_4)$$

but the join of these four type terms is $\mathit{Any} \to \mathit{Int}$, not $\mathit{Any} \to \{y \mid y = 0 \vee y = 1\}$ as desired.

The problem is that the SMT solver cannot perform a precise case analysis involving type predicates. Analogous to the way I-Cnf converts the *right* side of formulas into *conjunctive* normal form to help *introduce* type predicates, we can add the following rule that converts the *left* side of formulas into *disjunctive* normal form to help *eliminate* type predicates:

$$\frac{\mathit{DNF}(\mathit{Embed}(\mathit{\Gamma})) = \vee_i\, q_i \qquad \forall i.\ q_i \Rightarrow_n p}{\Gamma \Rightarrow_n p}\ \text{[I-Dnf]}$$

For the example above, I-Dnf breaks the implication query into three subgoals, and the I-HasType-Meet rule can handle $x :: U_1 \wedge x :: U_2$ case and I-HasType-Join can handle the $x :: U_3$ and $x :: U_4$ cases. Normalizing the left side of formulas is quite powerful — in fact, it can be used to derive I-HasType-Join — but undesirable for an algorithmic system, as discussed in the sequel.

4.4 Soundness

In Appendix A, we augment our type soundness proofs for System D with the three extensions described in this chapter. In this section, we outline two of the important lemmas.

The following lemma states three properties of the meet operator. Only the first property is required for soundness (in particular, for the Satisfiable Typing lemma described in §2.4). The last two are similar to the usual properties one would expect of a meet operator: that it computes a lower bound and that it computes the greatest lower bound. Proving the last property depends on the presence of the I-Dnf rule, so the meet cannot be considered maximal without it.

Lemma (Sound Meet)**.** *Suppose* $U_1 \sqcap U_2 = U$.

1. *If* $\mathcal{I}_n \models w :: U_1$ *and* $\mathcal{I}_n \models w :: U_2$ *then* $\mathcal{I}_n \models w :: U$.
2. $\vdash_n U <: U_1$ *and* $\vdash_n U <: U_2$.
3. *If* $\vdash_n U' <: U_1$ *and* $\vdash_n U' <: U_2$, *then* $\vdash_n U' <: U$.

We also prove three properties of the join operator analogous to those for the meet operator. The proof that the join computes the least upper bound (the third property below) depends on the presence of I-Dnf in the system.

Lemma (Sound Join)**.** *Suppose* $U_1 \sqcup U_2 = U$.

1. *If* $\mathcal{I}_n \models w :: U_1$ *or* $\mathcal{I}_n \models w :: U_2$, *then* $\mathcal{I}_n \models w :: U$.
2. $\vdash_n U_1 <: U$ *and* $\vdash_n U_2 <: U$.
3. *If* $\vdash_n U_1 <: U'$ *and* $\vdash_n U_2 <: U'$, *then* $\vdash_n U <: U'$.

These lemmas, along with several others, allow us to prove type soundness for the extended system. The full details may be found in Appendix A.

4.5 Algorithmic Typing

Each of the three extensions is simple to add to the algorithmic system. We define the extensions in Figure 4.2. The rules IA-HasType-Meet, IA-HasType-Join, and IA-Dnf are straightforward analogs to their declarative counterparts.

Function Application. We also update how function applications are synthesized. Recall that the TS-App rule (Figure 3.3) requires that to call a function v_1, there must be exactly one function type that flows to it. Now the rule TS-App-Meet generalizes (and replaces) TS-App, by using the meet operator to combine the extracted set $\mathfrak{U}'$ of type terms. When *MustFlow* returns zero type terms, we now use the new rule TS-App-Join, where the procedure *CanFlow* tries increasingly larger sets $\mathfrak{U}$ of type terms to serve as an "upper bound" for v_1.

DNF. Using IA-Dnf is algorithmically undesirable, since the environment of assumptions can no longer be maintained incrementally (§ 3.1), leading to much larger SMT queries. Therefore, we intend that a type checker not use IA-Dnf by default but only for complicated examples that require its expressiveness.

Endnotes

In this chapter, we have shown how to extend the basic nested refinement subtyping algorithm to increase expressiveness by introducing notions of joins and meets for syntactic type constructors and by further manipulating the structure of formulas when discharging implications.

$$\frac{\mathit{MustFlow}(\Gamma,\{x \mid x = w\},\mathfrak{U}) = \mathfrak{U}' \qquad U_0 = \sqcap_{U' \in \mathfrak{U}'} U' \qquad \Gamma;\, \mathfrak{U} \cup \mathfrak{U}' \vdash U_0 <: U}{\Gamma;\, \mathfrak{U} \Rightarrow w :: U}\ \text{[IA-\textsc{HasType-Meet}]}$$

$$\frac{\mathit{CanFlow}(\Gamma,\{x \mid x = w\},\mathfrak{U}) = \mathfrak{U}' \qquad U_0 = \sqcup_{U' \in \mathfrak{U}'} U' \qquad \Gamma;\, \mathfrak{U} \cup \mathfrak{U}' \vdash U_0 <: U}{\Gamma;\, \mathfrak{U} \Rightarrow w :: U}\ \text{[IA-\textsc{HasType-Join}]}$$

$$\frac{\mathit{DNF}(\mathit{Embed}(\mathit{\Gamma})) = \vee_i\, q_i \qquad \forall i.\ q_i;\, \mathfrak{U} \Rightarrow p}{\Gamma;\, \mathfrak{U} \Rightarrow p}\ \text{[IA-\textsc{Dnf}]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \,\rhd\, T_1 \qquad \mathit{MustFlow}(\Gamma, T_1, \emptyset) = \mathfrak{U} \\ U = \sqcap_{U' \in \mathfrak{U}}\, U' \qquad U = \forall \overline{A}.\ x\!:\!T_{11} \rightarrow T_{12} \qquad \Gamma \vdash v_2 \,\lhd\, \mathit{Inst}(T_{11}, \overline{A}, \overline{T'})\end{array}}{\Gamma \vdash [\overline{T'}]\ v_1\ v_2 \,\rhd\, \mathit{Inst}(T_{12}, \overline{A}, \overline{T'})[v_2/x]}\ \text{[TS-\textsc{App-Meet}]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \,\rhd\, T_1 \qquad \mathit{CanFlow}(\Gamma, T_1, \emptyset) = \mathfrak{U} \\ U = \sqcup_{U' \in \mathfrak{U}}\, U' \qquad U = \forall \overline{A}.\ x\!:\!T_{11} \rightarrow T_{12} \qquad \Gamma \vdash v_2 \,\lhd\, \mathit{Inst}(T_{11}, \overline{A}, \overline{T'})\end{array}}{\Gamma \vdash [\overline{T'}]\ v_1\ v_2 \,\rhd\, \mathit{Inst}(T_{12}, \overline{A}, \overline{T'})[v_2/x]}\ \text{[TS-\textsc{App-Join}]}$$

Figure 4.2. Extensions to Algorithmic System D

In subsequent chapters, however, we exclude these extensions and use only the core subtyping mechanisms from before.

Part III

From System D to JavaScript

Chapter 5

System !D

Dynamic languages like JavaScript, Python, and Ruby are widely popular for building both client and server applications, in large part because they provide powerful sets of features — run-time type tests, mutable variables, extensible objects, and higher-order functions. But as applications grow, the lack of static typing makes it difficult to achieve reliability, security, maintainability, and performance. In response, several authors have proposed type systems which provide static checking for various subsets of dynamic languages [57, 43, 99, 52, 79, 9].

In the previous part of this dissertation, we developed a core calculus for dynamic languages that supports the above dynamic idioms but in a purely functional setting. The main insight in System D is to dependently type all values with formulas drawn from an SMT-decidable refinement logic. We use an SMT solver to reason about the properties it tracks well, namely, control-flow invariants and dictionaries with dynamic keys that bind scalar values. But to describe dynamic keys that bind rich values like functions, System D encodes function types as logical terms and *nests* the typing relation as an uninterpreted predicate within the logic. By dividing work between syntactic subtyping and SMT-based validity checking, the calculus supports automatic type checking of dynamic features like run-time type tests, value-indexed dictionaries, higher-order functions, and polymorphism.

In the second part of this dissertation, we scale up System D to Dependent JavaScript (abbreviated to DJS), an explicitly typed dialect of the imperative, object-oriented, dynamic language. We bridge the vast gap between System D and JavaScript in three steps.

Step 1: Imperative Updates. The types of variables in JavaScript are routinely "changed" either by assignment or by incrementally adding or removing fields to objects bound to variables. The presence of mutation makes it challenging to assign precise types to variables, and the standard method of assigning a single "invariant" reference type that overapproximates all values held by the variable is useless in the JavaScript setting. We overcome this challenge by extending our calculus with *flow-sensitive heap types* (in the style of [91, 29, 39, 3, 86]) which allow the system to precisely track the heap location each variable refers to as well as aliasing relationships, thereby enabling *strong updates* through mutable variables. Our formulation of flow-sensitive heaps combined with higher-order functions and refinement types is novel, and allows DJS to express precise pre- and post-conditions of heaps, similar to as in separation logic [44].

Step 2: Prototype Inheritance. Each JavaScript object maintains an implicit link to the "prototype" object from which it derives. To resolve a key lookup from an object at run-time, JavaScript *transitively* follows its prototype links until either the key is found or the root is reached without success. Thus, unlike in class-based languages, inheritance relationships are *computed* at run-time rather than provided as declarative specifications. The semantics of prototypes is challenging for static typing, because to track the type of a key binding, the system must statically reason about a potentially unbounded number of prototype links! In DJS, we solve this problem with a novel decomposition of the heap into a "shallow" part, for which we precisely track a finite number of prototype links, and a "deep" part, for which we do not have precise information, represented abstractly via a logical heap variable. We *unroll* prototype hierarchies in shallow heaps to precisely model the semantics of object operations, and we use *uninterpreted heap predicates* to reason abstractly about deep parts. In this way, we reduce the reasoning about unbounded, imperative, prototype hierarchies to the underlying decidable, first-order, refinement logic.

Step 3: Arrays. JavaScript arrays are simply objects whose keys are string representations of integers. Arrays are commonly used both as *heterogeneous* tuples (that have a fixed number of elements of different types) as well as *homogeneous* collections (that have an unbounded number of elements of the same type). The overloaded use of arrays, together with the fact that arrays are otherwise syntactically indistinguishable and have the same prototype-based semantics as non-array objects, makes it hard to statically reason about the very different ways in which they are used. In DJS, we use nested refinements to address the problem neatly by uniformly encoding tuples and collections with refinement predicates, and by using *intersection* types that simultaneous encode the semantics of tuples, collections, and objects.

JavaScript Semantics by Desugaring. Many corner cases of JavaScript are clarified by λ_{JS} [51], a syntax-directed translation, or *desugaring*, of JavaScript programs to a mostly-standard lambda-calculus with explicit references and records (like the one in discussed in Chapter 1). As λ_{JS} is a core language with well-understood semantics and proof techniques, the translation paves a path to a typed dialect of JavaScript: define a type system for the core language and then type check desugared JavaScript programs. We take this path by developing System DJS, a new statically typed calculus based on λ_{JS}. Although the operational semantics of System DJS is straightforward, the *dynamic* features of the language ensure that building a type system expressive enough to support desugared JavaScript idioms is not.

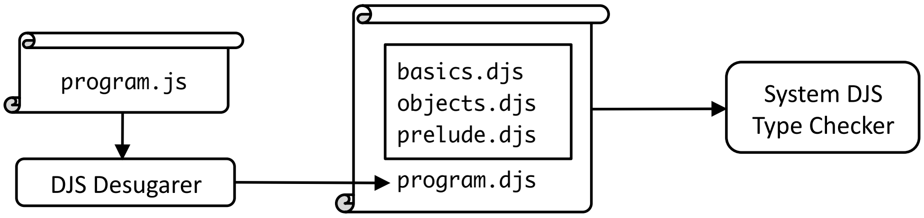

Figure 5.1. Architecture of Dependent JavaScript

Contributions. To sum up, we make several contributions in the second part of this dissertation:

- We extend System D with support for flow-sensitive strong updates, resulting in a type system called System !D (pronounced "D-ref"), to enable reasoning about dictionary objects in the presence of mutable variables. (Chapter 5)
- We extend System !D with encodings for prototype-based inheritance and JavaScript arrays, resulting in a type system called System DJS. (Chapter 6)
- We define an explicitly typed dialect called Dependent JavaScript (or DJS for short), which desugars to System DJS for type checking. Figure 5.1 depicts the architecture of our approach: we desugar a Dependent JavaScript (DJS) file `program.js` to the System DJS file `program.djs`, which is analyzed by the type checker along with a standard prelude comprising three files (`basics.djs`, `objects.djs`, and `prelude.djs`) that model JavaScript semantics. (Chapter 7)

We have implemented DJS [80] and demonstrated its expressiveness by checking a variety of properties found in small but subtle examples drawn from a variety of sources, including the popular book *JavaScript: The Good Parts* [23] and the SunSpider benchmark suite [93]. Our experiments show that several examples simultaneously require the gamut of features in DJS, but that many examples conform to recurring patterns that rely on particular aspects of the type system. We identify several ways in which future work can handle these patterns more specifically in order to reduce the annotation burden and performance for common cases, while falling back to the full expressiveness of DJS in general. Thus, we believe that DJS provides a significant step towards truly retrofitting JavaScript with a practical type system.

5.1 Overview

In this section, we present a series of examples to motivate and introduce the techniques we use in System !D to reason about mutable variables, mutable dictionary-based objects, and collections of objects. We continue to use the notational conveniences we introduced for System D.

Terminology. For the purposes of this chapter, we assume that JavaScript objects are simply references to pure dictionary values; in the next chapter, we eliminate this assumption and take into account the semantics of prototype-based objects. We refer to this simplified language as "JavaScript" (and the corresponding typed dialect as "DJS"), and we typeset examples with JavaScript syntax as follows.

```
var negate = function (x) {
  if (typeof x == "integer") { return 0 - x; }
  else                       { return !x;    }
};
```

5.1.1 Imperative Updates

JavaScript is an imperative language where variables can be reassigned arbitrary values. Consider the following DJS function that is like `negate` but first assigns the eventual result in the variable `x`:

```
//: also_negate :: $(x{:}\mathit{IntOrBool}) \to \{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}$
var also_negate = function (x) {
  if (typeof x == "integer") { x = 0 - x; }
  else                       { x = !x;    }
  return x;
};
```

The following is the translation of `also_negate` to System !D (ignore the comments for now):

```
(*: also_negate :: $(x{:}\mathit{IntOrBool}) \to \{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}$ *)
let also_negate (x) =                         (*: $\Gamma_1 = -$   $\Sigma_1 = $emp *)
  let _x = ref x in                           (*: $\Gamma_2 = $x$:\mathit{Any}$   $\Sigma_2 = (\ell_x \mapsto $x$)$ *)
  if tagof (deref _x) = "integer" then        (*: $\Gamma_3 = \Gamma_2, $_x$:\mathit{Ref}\,\ell_x$   $\Sigma_3 = \Sigma_2$ *)
    _x := (0 - deref _x)     (*: $\Gamma_4 = \Gamma_3, \mathit{Int}($x$)$   $\Sigma_4 = \exists x_4{:}\mathit{Int}.\,(\ell_x \mapsto x_4)$ *)
  else
    _x := not (deref _x)     (*: $\Gamma_6 = \Gamma_3, \neg\mathit{Int}($x$)$   $\Sigma_6 = \exists x_6{:}\mathit{Bool}.\,(\ell_x \mapsto x_6)$ *)
  ;           (*: $\Gamma_7 = \Gamma_3$   $\Sigma_7 = \exists z{:}\{y \mid \mathit{ite}\ (\mathit{Int}($x$))\ (y = x_4)\ (y = x_6)\}.\,(\ell_x \mapsto z)$ *)
  deref _x   (*: $\Gamma_8 = \Gamma_3$   $\Sigma_8 = \Sigma_7$ *)
in
let _also_negate = ref also_negate
```

Several aspects of the translation warrant attention. First, since the formal parameter `x`, like all JavaScript variables, is mutable, the translation of the function body begins with an *explicit reference* cell `_x` initialized with `x`, and each read of `x` is desugared to a dereference of `_x`. Presentations of imperative languages often model assignable variables directly rather than with explicit references. Both approaches are equivalent; we choose the latter to make the presentation more similar to

λ_{JS} [51] and System D. Second, notice that scalar constants like `0` and `true` and operators like `typeof` and `==` are translated directly to corresponding ones in System !D. Third, notice that each assignment to the variable `x` translates to a set reference (*i.e.* assignment) operation to update the contents of the heap cell. Finally, the translations stores the function value in a mutable variable called `_also_negate`. For System !D to verify that `also_negate` satisfies the function type specification, it must precisely reason about heap updates in addition to control flow, as for `negate` in the purely function System D setting (*cf.* §2.1.1).

Reference Types. The traditional way to handle references in the λ-calculus [77] is to (i) assign a reference cell some type *Ref* T, (ii) require that only values of type T be stored in it, and then (iii) conclude that dereferences produce values of type T. This approach supports so-called *weak updates*, because even if a stored value satisfies a stronger type T' than T (*i.e.* if T' is a subtype of T), subsequent dereferences produce values of the original, weaker type T. Put another way, this approach requires that the type assigned to a reference cell be a supertype of all the values written to the cell.

Unfortunately, weak updates would preclude System !D from verifying `also_negate`. The initialization of `_x` on line 2 stores the parameter `x` which has type *Any*, so `_x` would be assigned type *Ref Any*. The assignments on lines 4 and 6 type check because the updated values satisfy the trivial type *Any*, but the dereference on line 8 produces a value with type *Any*, which does not satisfy the specified return type. Thus, we need a way to reason more precisely about heap updates.

Strong Updates. Allowing assignment to change the type of a reference is called *strong update*, which is sound only when a reference is guaranteed to point to a *single* heap cell and when there are no accesses through other *aliases* that refer to the same cell. The *Alias Types* approach [91] provides a means of managing these concerns. Rather than *Ref* T, a reference type is written *Ref* ℓ, where ℓ is the (compile-time) name of a location in the heap, and a separate (compile-time) heap maps locations to types, for example, $(\ell \mapsto T)$. Strong updates are realized by allowing heaps to change *flow-sensitively*, and the aliasing problem is mitigated by maintaining the invariant that distinct location names ℓ and ℓ' do not alias.

System !D employs this approach by using a *type environment* Γ that grows and shrinks as usual during type checking but remains flow-insensitive, and a *heap environment* Σ that can be strongly updated per program point. We introduce a new syntactic type term, *Ref* ℓ, for reference types that can be nested inside refinements à la System D. Extending our notational conventions from System D, we allow reference types to be written outside refinements to abbreviate the following:

$$\mathit{Ref}\ \ell \ \mathrel{\hat{=}}\ \{y \,|\, y :: \mathit{Ref}\ \ell\}$$

The comments in the translated version of `also_negate` show how System !D verifies the function type annotation. The figure shows, at each line i, the type environment Γ_i used to check the expression on the line, and the heap environment Σ_i that exists *after* checking the expression. After starting with the empty heap $\Sigma_1 = \mathtt{emp}$, the allocation on line 2 creates a fresh location ℓ_x in the new heap $\Sigma_2 \mathrel{\hat{=}} \Sigma_1 \oplus (\ell_x \mapsto \mathtt{x})$ and adds $\mathtt{\_x}{:}\mathit{Ref}\ \ell_x$ to the type environment. We use the symbol $\oplus$ to construct unordered sets of heap bindings. To exploit the precision of dependent types, we map locations to *values* rather than types (*i.e.* $(\ell_x \mapsto \mathtt{x})$ rather than $(\ell_x \mapsto \mathit{IntOrBool})$).

When checking the if-expression guard on line 3, the dereference retrieves the initial value $\mathtt{x}$ from the heap Σ_2, so as a result of the tag-test, System !D adds $\mathit{Int}(\mathtt{x})$ to the type environment Γ_4 along the true-branch and $\neg\mathit{Int}(\mathtt{x})$ to Γ_6 along the false-branch. In the true-branch, the subtraction on line 4 is well-typed because $\mathit{Int}(\mathtt{x})$ and produces an integer x_4 that is stored in the heap Σ_4 at location ℓ_x. In the false-branch, $\mathtt{x}$ is negated on line 6, producing a boolean x_6 that is stored in the heap Σ_6 at location ℓ_x. System !D combines the branches by *joining* the heaps Σ_4 and Σ_6, producing Σ_7 that describes the heap no matter which branch is taken. The dereference on line 8 retrieves z, a value of type $\{y \mid \mathit{ite}\ (\mathit{Int}(x))\ (y = x_4)\ (y = x_6)\}$ (where $\mathit{ite}\ p\ q_1\ q_2 \mathrel{\hat{=}} (p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2)$), which is a subtype of the return type annotation $\{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}$.

In this way, System !D syntactically tracks strong updates to the heap, while reducing subtyping obligations to implication queries in a pure refinement logic (as in System D) that does *not* model imperative updates.

5.1.2 Mutable Objects

In functional dynamic languages, an object is a dictionary indexed by dynamically computed keys, and object update (realized via the `set` primitive in System D) produces a new (immutable) dictionary value. In an imperative language like JavaScript, however, object update *mutates* the existing object rather than producing a new one. In this section, we describe how System !D retains the precision of System D dictionary operations despite the presence of mutation.

System !D inherits the dictionary primitives from before; we also incorporate a primitive `del` for removing a key, which was omitted from the presentation of System D:

```
val  {} :: (*: {d | d = empty} *)
val mem :: (*: d:Dict → k:Str → {b | b iff has(d,k)} *)
val get :: (*: d:Dict → k:{s | Str(s) ∧ has(d,s)} → {x | x = sel(d,k)} *)
val set :: (*: d:Dict → k:Str → x:Any → {d′ | d′ = upd(d,k,x)} *)
val del :: (*: d:Dict → k:Str → {d′ | d′ = upd(d,k,bot)} *)
```

In the discussion that follows, we refer to the following example, where the JavaScript code on the left exercises several operations on objects and its translation to System !D on the right makes every implicit reference operation explicit:

```
var k = "g";
var x = {f: k};
var xf = x.f;
x[k] = 3;
assert (x[k] == 3);
delete x.f;
assert (!("f" in x));
var y = x;
y.f = "hi";
assert (x.f == "hi");
```

```
1  let _k = ref "g" in
2  let _x = ref (ref {"f" = deref _k}) in
3  let _xf = ref (get (deref (deref _x)) "f") in
4  deref _x := set (deref (deref _x)) (deref _k) 3;
5  assert (get (deref (deref _x)) (deref _k) = 3);
6  deref _x := del (deref (deref _x)) "f";
7  assert (not (mem (deref (deref _x)) "f"));
8  let _y = ref (deref _x) in
9  deref _y := set (deref (deref _y)) "f" "hi";
10 assert (get (deref (deref _x)) "f" = "hi");
```

As before, we write Σ_i to refer to the heap environment after processing the code on line i.

Scalar vs. Reference Values. As we saw with `also_negate`, variable definitions in DJS like `k` on line 1 above are translated to bindings to a fresh references, and uses are translated to dereferences. The translation of a value depends on whether it is a *scalar value* (*e.g.* integer, boolean, string) or a *reference value* (*e.g.* object), a distinction made in languages like JavaScript, Java, and C# but not in pure-object languages like Python and Ruby. Scalar values like "g" on line 1 are translated directly, while reference values like the object literal `{f: k}` on line 2 are wrapped in reference cells. This distinction allows functions in the translation to mutate arguments that are reference values. After processing the first two lines, we have the following type and heap environments:

$$\Gamma_3 \;\mathrel{\mathring{=}}\; \texttt{\_k}{:}\mathit{Ref}\ \ell_k,\ d{:}\{d_0 \,|\, d_0 = \mathit{upd}(\mathit{empty}, \text{“}\texttt{f}\text{”}, \text{“g”})\},\ \_d{:}\mathit{Ref}\ \ell_d,\ \texttt{\_x}{:}\mathit{Ref}\ \ell_x$$

$$\Sigma_2 \;\mathrel{\mathring{=}}\; (\ell_k \mapsto \text{“g”}) \oplus (\ell_d \mapsto d) \oplus (\ell_x \mapsto \_d)$$

Mutability and Dynamic Keys. The combination of nested refinements and strong updates allows us to precisely track objects with dynamic keys despite the presence of imperative updates. Consider the desugaring of our example above; we omit the assertions for clarity.

The dictionary d referred to by the DJS variable `x` on line 2 is stored via two levels of indirection in the heap Σ_2, so on line 3 it is dereferenced *twice* and then supplied to the primitive `get` that performs key lookup on the functional (immutable) dictionary d on the heap. The expression `deref (deref _x)` retrieves the dictionary d from Σ_2, which has an "`f`" binding, so the call to `get` type checks and produces a value of type $\{z \,|\, z = \mathit{sel}(d, \text{“}\texttt{f}\text{”})\}$, which in the current type environment is equivalent to $\{z \,|\, z = \text{“g”}\}$. Thus, for line 3 the type system derives `_xf` :: $\mathit{Ref}\ \ell_{xf}$ and the heap $\Sigma_3 \mathrel{\mathring{=}} \Sigma_2 \oplus (\ell_{xf} \mapsto \text{“g”})$.

System !D reasons about key membership, key update, and key deletion on mutable objects in similar fashion. On line 4, the DJS object is extended with a *dynamic key*, that is, an arbitrary program value and not just a string literal; this poses no problem, since System D enables the specification of dynamic dictionaries. As opposed to the functional setting where a new dictionary would be created, the extension *mutates* the existing dictionary on the heap, for which we use the reference update operator `:=` (also known as set-reference, or `setref`). In particular, the new heap is the result of *strongly updating* Σ_3, where a new dictionary d' of type $\{d_1 \,|\, d_1 = \mathit{upd}(d,\text{“g”},3)\}$ replaces the old one stored at location ℓ_d:

$$\Sigma_4 \;\mathrel{\dot{=}}\; (\ell_k \mapsto \cdots) \oplus (\ell_x \mapsto \cdots) \oplus (\ell_{xf} \mapsto \cdots) \oplus (\ell_d \mapsto d')$$

The system can, hence, prove that the call to `get` on line 5 is safe (because d' has a binding for “g”) and returns a value of type $\{n \,|\, n = 3\}$. Like object extension, the key deletion on line 6 results in a strong update, producing

$$\Sigma_6 \;\mathrel{\dot{=}}\; (\ell_k \mapsto \cdots) \oplus (\ell_x \mapsto \cdots) \oplus (\ell_{xf} \mapsto \cdots) \oplus (\ell_d \mapsto d'')$$

where d'' has type $\{d_2 \,|\, d_2 = \mathit{upd}(d',\text{“}\mathtt{f}\text{”},\mathit{bot})\}$. Thus, the system proves the assertion on line 7 that d'' does not have a binding for “`f`”.

Aliasing. Factoring references into flow-insensitive reference types and flow-sensitive heap types in the style of Alias Types [91] makes it easy to track strong updates in the presence of aliasing. After the declaration on line 8, `_y` :: *Ref* ℓ_y is added to the type environment and the heap environment is:

$$\Sigma_8 \;\mathrel{\dot{=}}\; \cdots \oplus (\ell_d \mapsto d'') \oplus (\ell_x \mapsto \_d) \oplus (\ell_y \mapsto \_d)$$

Because both ℓ_x and ℓ_y store $\_d$, a pointer to the *single* dictionary stored at ℓ_d, the update through `_y` on line 9 is reflected when reading through `_x` on line 10, and, hence, the assertion is proven.

5.1.3 Function Types

The only System !D function we have seen so far, `also_negate`, does not take any heap references as arguments and does not return any to the caller. In general, however, heap references may cross function boundaries, so the types of functions must describe values on the heap.

Input and Output Worlds. We define a *world* W, of the form $x\!:\!T \,/\, h$, to describe a value x of type T along with a *heap type* (or simply *heap*) h of the form $H \oplus h'$, where H is a single *heap variable* followed by a sequence of unordered location bindings. The bindings in h' are like those in a heap environment Σ but they map locations to binder-type pairs rather than values (*e.g.* $(\ell \mapsto y\!:\!T)$ rather than $(\ell \mapsto v)$). The variable H is used to refer to all other locations in a given heap not constrained by h'. We sometimes refer to portion of the heap described by H' as the “deep” heap and the portion by h' as the “shallow” heap, since we can “see into” the latter to inspect its locations. All of the binders in a world — namely, the binder x and all of the binders y in the heap — may refer to one another. As a result, a world can be thought of as a (dependent) tuple of values, some of which reside in the heap.

A function type in System !D, parameterized by a sequence of *type variables* A, *location variables* L, and heap variables H, comprises an input and output world as follows:

$$\forall \overline{A}, \overline{L}, \overline{H}.\ W_1 \to W_2 \ = \ \forall \overline{A}, \overline{L}, \overline{H}.\ x_1 \colon T_1 \,/\, h_1 \to x_2 \colon T_2 \,/\, h_2$$

This type describes a function that, given an argument x_1 of type T_1 in a calling context that satisfies the input heap $h_1 = H_1 \oplus h_1'$, produces an output value x_2 of type T_2 and a modified heap $h_2 = H_2 \oplus h_2'$. The input and output heap bindings h_1' and h_2' describe only those locations that are accessed by the function; all other locations are unaffected and bound to the heap variables H_1 and H_2 (these correspond variables to the "frame" from separation logic [44]). All of the binders in the input world W_1 are in scope in the output world W_2, which is useful for describing how a function mutates heap values. We often omit binders when they are not referred to.

To match the structure of function types, function applications must instantiate type and location variables. However, our implementation, described in Chapter 7, infers instantiations in many cases using standard local type inference techniques. When we write DJS examples in the sequel, we omit instantiations at applications wherever our current implementation infers them.

Location Polymorphism. Consider the following function adapted from System D (*cf.* §2.1.3) to the imperative setting. So that the following function works with references to *any* location, we write an annotation quantified by a location parameter:

```
//: getCount :: ∀L,H. (Ref L, Str) / H ⊕ (L ↦ d:Dict)
//:             → Int / H ⊕ (L ↦ d′:Dict{d′ = d})
var getCount = function (t, c) {
  if (c in t) { return toInt(t[c]); }
  else        { return 0;           }
};
```

Each caller must instantiate L appropriately to match the reference parameter that it passes in. With this signature, and a constant `toInt` :: $Any \to Int$, the type system can prove that the key lookup operation is safely guarded by the key membership test, that the output value is always an integer, and that the output heap is *exactly* the same as the input heap (*i.e.* unmodified).

Notation. We sweeten function type syntax with some sugar:

- A single heap variable H is implicitly added to a function type when it contains none, and H is added to both the input and output heaps.
- When used as an output heap, the token *same* refers to the sequence of locations in the corresponding input heap, where each binding records that the final value is exactly equal to the initial value.

- In an input world, a reference binding $x\,{:}\,\mathit{Ref}$ without a location introduces a location variable L that is quantified by the type, and x (a value of type $\mathit{Ref}\ L$) can be used as a location in heaps to refer to this variable L.

For example, the function type annotation for `also_negate` from before expands to the following:

$$\texttt{also\_negate} :: \forall H.\ (x\,{:}\,\mathit{IntOrBool})\,/\,H \rightarrow \{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}\,/\,H$$

And we can write the following concise type for `getCount` to abbreviate the previous one:

$$\texttt{getCount} :: (x\,{:}\,\mathit{Ref},\ \mathit{Str})\,/\,(x \mapsto d\,{:}\,\mathit{Dict}) \rightarrow \mathit{Int}\,/\,\mathit{same}$$

Effects. Neither of the previous functions has an observable effect on callers; although each function allocates local references, they are inaccessible from call sites. The following function (adapted from §2.1.3) *does* mutate the heaps of its callers:

```
//: incCount :: (t:Ref, c:Str) / (t ↦ d:Dict)
//:            → Any / (t ↦ d′:Dict{EqMod(d′,d,{c}) ∧ Int(sel(d′,c))})
var incCount = function (t, c) {
  var i = getCount(t, c);
  t[c] = 1 + i;
};
```

The function mutates the object argument rather than creating a copy of the object and extending it, so the updated object type is reflected in the output heap. The macro $\mathit{EqMod}(d',d,\{c\})$ encodes the fact that the dictionary d' is equal to d at all keys except c; additionally, the predicate $\mathit{Int}(\mathit{sel}(d',c))$ records the fact that d' has an integer c field.

5.1.4 Collections

As discussed in §5.1.1, strong updates are sound only for references that point to *exactly one* object, which is far too restrictive for all situations as real programs manipulate collections of objects. In this section, we describe *weak references* in DJS to refer to multiple objects, a facility that enables programming with arrays of mutable objects as well as recursive types.

Lists of References. In the following example, we iterate over a list of passenger objects and compute the sum of their weights; we use a default value `max_weight` when a passenger does not list his weight (ignore the annotations for now).

```
//: sumWeights ::
//:   ∀~L,~L′. (~L ↦ T_passenger) ⊕ (~L′ ↦ List[Ref ~L; ~L′]) ⇒ (Ref ~L′?) → Int
var sumWeights = function (passengers) {
  if (passengers == null) { return 0; }
5    else {
6      var p = passengers.hd;
7      var n = sumWeights(passengers.tl);
8      //: thaw p
9      if ("weight" in p) { return p.weight + n; }
10     else               { return n;            }
11     //: freeze p
12   }
13 }
```

We would like to specify an argument type that is a list of objects with the type

$$T_{passenger} \doteq \{d \mid Dict(d) \wedge has(d, \text{“}\texttt{weight}\text{”}) \Rightarrow Int(sel(d, \text{“}\texttt{weight}\text{”}))\}.$$

Using the mechanisms we have seen so far, we can assign the type

$$\texttt{sumWeights} :: \forall L.\ (List[Ref\ L]) \,/\, (L \mapsto T_{passenger}) \to Int \,/\, same \tag{5.1}$$

where *List* is a pre-defined recursive type constructor in the style of System D collections (§2.1.4). However, this type is not very useful as it requires that the argument be a list of references to a *single* object stored at the *single* location L.

Weak Locations. To refer to an *arbitrary number* (one or more) objects of the same type, we adopt the Alias Types [91] solution, which categorizes some locations as *weak* to describe an arbitrary number of locations that satisfy the same type, and syntactically ensures that weak locations are weakly updated. We extend the syntax of a function type in System !D to the form

$$\forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \to W_2$$

to describe a *weak heap* Ψ of bindings $(\tilde{\ell} \mapsto T)$, where $\tilde{\ell}$ is a weak location such that all objects that might reside at $\tilde{\ell}$ satisfy the type T and where M ranges over *strong location variables* L and *weak location variables* $\tilde{L}$. Unlike for a strong location, there is no heap binder for a weak location because there is not a single value to describe. We allow the weak heap to be omitted when it contains no bindings, writing $\forall \overline{A}, \overline{M}, \overline{H}.\ W_1 \to W_2$ similar to before.

Thaw and Freeze. Using the weak location facility, we would like to describe the type of the function above as follows:

$$\texttt{sumWeights} :: \forall \tilde{L}.\ (\tilde{L} \mapsto T_{passenger}) \Rightarrow (List[Ref\ \tilde{L}]) \to Int \tag{5.2}$$

Unfortunately, the existing mechanisms we have discussed are unable to verify this specification. Each (desugared) use of p on line 9 is a dictionary of type $T_{passenger}$, which is not sufficient to type check the addition operation. This is quite unsatisfying, however, because the conditional establishes that along the then-branch, `p` *does* possess the key and therefore should be assigned the more precise type $\{d \,|\, \mathit{Int}(\mathit{sel}(d,\text{“}\texttt{weight}\text{”}))\}$.

To solve this problem, we adopt a mechanism found in derivatives of Alias Types (*e.g.* [29, 39, 3, 86]) that allows a weak location to be *temporarily* treated as strong. A weak location $\tilde{}\ell$ is said to be *frozen* if all references $\mathit{Ref}\ \tilde{}\ell$ use the location only at its weak (invariant) type. The type system can *thaw* a location, producing a strong reference $\mathit{Ref}\ \ell_k$ (with a fresh name) that can be used to strongly update the type of the cell. While a location is thawed, the type system prohibits the use of weak references to the location, and does not allow further thaw operations. When the thawed (strong) reference is no longer needed, the type system checks that the original type has been restored, *re-freezes* the location, and discards the thawed location. Soundness of the approach depends on the invariant that each weak location has at most one corresponding thawed location at a time.

In our example, we do not need to temporarily violate the type of `p`, but the thaw/freeze mechanism does help us *relate* the two accesses to `p` on line 9. We can assign the following type to the function above, where *thaw state* annotations require that the weak location L that stores passengers objects must be frozen upon entry and exit of the function:

$$\texttt{sumWeights} :: \forall \tilde{}L.\ (\tilde{}L \mapsto T_{passenger}) \Rightarrow (\mathit{List}[\mathit{Ref}\ \tilde{}L])\,/\,(\tilde{}L \mapsto \texttt{frzn}) \rightarrow \mathit{Int}\,/\,(\tilde{}L \mapsto \texttt{frzn}) \qquad (5.3)$$

Notice that thaw state annotations for weak locations are placed inside *strong* heaps because they are subject to update. The thaw annotation on line 8 changes the type of `p` from a weak reference of type $\mathit{Ref}\ \tilde{}L$ to a strong reference $\mathit{Ref}\ \tilde{}\ell_1$ to a fresh thawed location, which stores a *particular* dictionary on the heap (named with a binder) that is retrieved by both subsequent uses of `p`. Thus, on line 9 we can relate the key membership test to the lookup, and track that `p.weight` produces an integer. The freeze annotation on line 11 restores the invariant that $\tilde{}L$ must be frozen before the function returns. We describe this technique further in §5.3.

Because weak locations are often frozen at function boundaries, we allow the thaw state annotation for a weak location $\tilde{}L$ to be omitted from a function type, in which case $(\tilde{}L \mapsto \texttt{frzn})$ is added to both the input and output worlds of the function. For example, Equation 5.3 is syntactic sugar for Equation 5.2.

Weak Locations for Recursive Types. Having addressed the issue of describing an arbitrary number of heap values with the same type, we now turn our attention to the mechanism for describing the list of references itself. Although we could adopt the recursive type constructors from System D to facilitate collections, instead, in System !D, we reuse the weak location mechanism to describe recursive types. For example, we can describe a weak location $(\tilde{}L_0 \mapsto \{\text{“}\texttt{hd}\text{”}{:}\mathit{Int};\ \text{“}\texttt{tl}\text{”}{:}(\mathit{Ref}\ \tilde{}L_0)?\})$ and use the type $\mathit{Ref}\ \tilde{}L_0$ to describe (an arbitrary number of) integer lists, each of which stores an integer and a possibly-null reference to another integer

$$\begin{array}{rrcl}
\textbf{Values} & v & ::= & \lambda x.e \mid x \mid c \mid v_1[v_2 \mapsto v_3] \mid r \\
\textbf{Expressions} & e & ::= & v \mid [\overline{T},\overline{m},\overline{h}]\; v_1\; v_2 \\
 & & \mid & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{ref}\ \ell\ v \mid \texttt{deref}\ v \mid \texttt{setref}\ v_1\ v_2 \\
 & & \mid & \texttt{freeze}\ \tilde{\ell}\ \theta\ v \mid \texttt{thaw}\ \ell\ v \\
\textbf{Types} & T & ::= & \{x \mid p\} \\
\textbf{Prenex Types} & S & ::= & \exists x{:}T.\,S \mid T \\
\textbf{Formulas} & p,q & ::= & P(\overline{w}) \mid w :: U \mid p \wedge q \mid p \vee q \mid \neg p \\
\textbf{Logical Values} & w & ::= & v \mid F(\overline{w}) \\
\textbf{Type Terms} & U & ::= & \forall \overline{A},\overline{M},\overline{H}.\ \Psi \Rightarrow W_1 \rightarrow W_2 \mid A \mid \mathit{Ref}\ m \\
\textbf{Weak Heaps} & \Psi & ::= & - \mid \Psi \oplus (\tilde{\ell} \mapsto T) \\
\textbf{(Strong) Heaps} & h & ::= & H \mid h \oplus (\ell \mapsto x{:}T) \mid h \oplus (\tilde{\ell} \mapsto \theta) \\
\textbf{Thaw States} & \theta & ::= & \texttt{frzn} \mid \texttt{thwd}\ \ell \\
\textbf{Worlds} & W & ::= & x{:}T\,/\,h \\
\textbf{Strong Locations} & \ell & ::= & a \mid L \\
\textbf{Weak Locations} & \tilde{\ell} & ::= & \tilde{a} \mid \tilde{L} \\
\textbf{Locations} & m & ::= & \ell \mid \tilde{\ell} \\
\textbf{Location Variables} & M & ::= & L \mid \tilde{L} \\
\textbf{Metavariables} & L & \in & \mathit{LocationVarIdentifiers} \\
 & H & \in & \mathit{HeapVarIdentifiers} \\
 & a & \in & \mathit{StaticLocationConstants} \\
 & r & \in & \mathit{DynamicLocationConstants}
\end{array}$$

Figure 5.2. Syntax of System !D

list. Using this approach and the abbreviation $\mathit{List}[A; \tilde{L}_0] \doteq \{\text{“}\texttt{hd}\text{”}{:}\,A;\ \text{“}\texttt{tl}\text{”}{:}\,(\mathit{Ref}\ \tilde{L}_0)?\}$ we can now specify and verify the following type (reproduced from line 2):

$$\texttt{sumWeights} :: \forall \tilde{L}, \tilde{L}'.\ (\tilde{L} \mapsto T_{passenger}) \oplus (\tilde{L}' \mapsto \mathit{List}[\mathit{Ref}\ \tilde{L}; \tilde{L}']) \Rightarrow (\mathit{Ref}\ \tilde{L}'?) \rightarrow \mathit{Int}$$

5.2 Syntax and Semantics

We now introduce the formal syntax of values, expressions, and types of System !D, defined in Figure 5.2. Metavariables not defined in Figure 5.2 are inherited from the syntax of System D (Figure 2.1).

Operational Semantics $\boxed{\Delta_1;\ e_1 \hookrightarrow e_2;\ \Delta_2}$

$$\textbf{Run-Time Heaps}\quad \Delta \quad ::= \quad - \ \mid\ \Delta[r \mapsto v]$$

$$\frac{}{\Delta;\ [\overline{T},\overline{m},\overline{h}]\ (\lambda x.e)\ v \hookrightarrow e[v/x];\ \Delta}\ \text{[E-App]} \qquad \frac{}{\Delta;\ \texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x];\ \Delta}\ \text{[E-Let]}$$

$$\frac{}{\Delta;\ \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1;\ \Delta}\ \text{[E-IfTrue]}$$

$$\frac{}{\Delta;\ \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2;\ \Delta}\ \text{[E-IfFalse]}$$

$$\frac{\delta(c,v)\ \text{is defined}}{\Delta;\ c\ v \hookrightarrow \delta(c,v);\ \Delta}\ \text{[E-Delta]} \qquad \frac{\Delta;\ e_1 \hookrightarrow e_1';\ \Delta'}{\Delta;\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2;\ \Delta'}\ \text{[E-Compat]}$$

$$\frac{r\ \text{fresh}}{\Delta;\ \texttt{ref}\ \ell\ v \hookrightarrow r;\ \Delta[r \mapsto v]}\ \text{[E-Ref]} \qquad \frac{r \in dom(\Delta)}{\Delta;\ \texttt{deref}\ r\ \hookrightarrow \Delta(r);\ \Delta}\ \text{[E-Deref]}$$

$$\frac{r \in dom(\Delta)}{\Delta;\ \texttt{setref}\ r\ v \hookrightarrow v;\ \Delta[r \mapsto v]}\ \text{[E-Setref]}$$

$$\frac{}{\Delta;\ \texttt{thaw}\ \ell\ r \hookrightarrow r;\ \Delta}\ \text{[E-Thaw]} \qquad \frac{}{\Delta;\ \texttt{freeze}\ \tilde{\ell}\ \theta\ r \hookrightarrow r;\ \Delta}\ \text{[E-Freeze]}$$

Figure 5.3. Semantics of System !D

Values. Values v include lambdas $\lambda x.e$, variables x, constants c, (functional) dictionaries $v_1[v_2 \mapsto v_3]$, and run-time heap locations r, which do not appear in source programs but arise during evaluation. The set of constants c is inherited from System D and includes base values (numbers, booleans, strings, the empty dictionary $\{\}$, `null`, *etc.*) and primitive functions (`tagof`, `get`, `(=)`, *etc.*). We use tuple syntax $(v_0,\ \ldots,\ v_n)$ as sugar for the dictionary with fields "`0`" through "`n`" bound to the component values. Logical values w are all values and applications of primitive function symbols F, such as addition $+$ and dictionary selection *sel*, to logical values.

Expressions. As in System D, we use an A-normal form (ANF) expression syntax so that we need only define substitution of values (not arbitrary expressions) into types. We use a more general syntax for examples throughout this paper, and our implementation desugars expressions into A-normal form. Expressions e include values, function application, if-expressions, let-bindings, and several reference-manipulating operations. Since function types will be parameterized by type, location, and heap variables, the syntax of function application requires that these be instantiated. Reference operations include reference allocation, dereference, and update, and the operational semantics, defined in Figure 5.3, maintains a separate heap that maps locations to values. The only difference compared to standard presentations of the λ-calculus extended with references (*e.g.* [77]) is that the syntax of reference allocation includes an explicit strong location ℓ, which is intended to be a compile-time abstraction of a set of run-time locations r' such that $\mathit{StrongLoc}(r') = \ell$ and that includes the freshly allocated run-time location r. The $\texttt{thaw}\ \ell\ v$ operation converts a weak reference to a strong one; $\texttt{freeze}\ \tilde{\ell}\ \theta\ v$ converts a strong reference to a weak one, where the thaw state θ is used by the type system for bookkeeping.

Types and Formulas. Values in System !D are described by *refinement types* of the form $\{x \mid p\}$ where the scope of x is p and prenex-quantified *existential types* $\exists x\!:\!T.\ S$ where the scope of x is S. Existential types do not appear in source programs; they are created only during type checking in a controlled fashion that does not preclude algorithmic type checking [66]. We often write $\exists \overline{x}\!:\!\overline{T}.\ T'$ to at once describe the (zero or more) existential quantifiers of a prenex type.

The language of *refinement formulas* includes predicates P, such as equality and the dictionary predicate *has*, and the usual logical connectives. Similar to the syntax for expression tuples, we use $(T_1, \ldots, T_n)$ as sugar for the dictionary type with fields "0" through "n" with the corresponding types.

As in System D, we use an uninterpreted *has-type predicate* $w :: U$ in formulas to describe values that have complex types, represented by *type terms* U, which includes function types, type variables, and reference types. A reference type names a strong or weak location in the heap, where a strong location ℓ is either a constant a or a variable L and a weak location $\tilde{\ell}$ is either a constant $\tilde{a}$ or variable $\tilde{L}$.

Function types, as discussed in §5.1.3, are parameterized over type, location, and heap variables and comprise a *weak heap* Ψ and input and output *worlds* W_1 and W_2. The type of a strong location is subject to strong update, so it may vary from input to output world. The type of a weak location, however, does not vary, so there is no output weak heap for a function type. But the *thaw state* of a weak location, either *frozen* or temporarily *thawed* to a strong location, may vary and is, thus, recorded in the strong input and output heaps.

Heap Types. A *strong heap type*, or simply *heap*, h is a single *heap variable* H followed by an ordered list of *heap bindings* h' concatenated with the $\oplus$ operator. The heap binding $(\ell \mapsto x\!:\!T)$ represents the fact that the value at location ℓ has type T; the binder x refers to this value in the types of other heap bindings. The binding $(\tilde{\ell} \mapsto \theta)$ records the current thaw state of weak location $\tilde{\ell}$, to help maintain the invariant that it has at most one thawed location at a time.

Syntactic Sugar. We freely use the syntactic abbreviations defined in Figure 2.2 and Figure 2.3 for System D as well as the ones introduced earlier in this chapter.

$$
\begin{array}{rlll}
\textbf{Type Environments} & \Gamma & ::= & - \mid \Gamma, x\!:\!T \mid \Gamma, p \\
 & & \mid & \Gamma, A \mid \Gamma, M \mid \Gamma, H \\
 & & \mid & \Gamma, (\tilde{\ell} \mapsto T) \\
\textbf{Heap Environments} & \Sigma & ::= & H \mid \Sigma \oplus (\ell \mapsto v) \mid \Sigma \oplus (\tilde{\ell} \mapsto \theta)
\end{array}
$$

$$
\begin{array}{rl}
\textbf{Value Typing} & \Gamma;\, \Sigma \vdash v :: T \\
\textbf{Expression Typing} & \Gamma;\, \Sigma \vdash e :: S\,/\,\Sigma' \\
\textbf{Subtyping} & \Gamma \vdash T_1 \sqsubseteq T_2 \\
\textbf{Syntactic Subtyping} & \Gamma \vdash U_1 <: U_2 \\
\textbf{Implication} & \Gamma \Rightarrow p \\
\textbf{World Subtyping} & \Gamma \vdash W_1 \sqsubseteq W_2;\ \pi \\
\textbf{World Satisfaction} & \Gamma \vdash S\,/\,\Sigma \models W;\ \pi \\
\textbf{Heap Matching} & h \sim h;\ \pi \\
\textbf{Heap Env. Matching} & \Sigma \sim h;\ \pi \\
\textbf{Well-Formedness} & \vdash \Gamma \quad \Gamma \vdash T \quad \Gamma \vdash p \quad \Gamma \vdash U \quad \Gamma \vdash w \\
 & \Gamma \vdash m \quad \Gamma \vdash W \quad \Gamma \vdash h \quad \Gamma \vdash \Psi
\end{array}
$$

Figure 5.4. Syntax of System !D Judgements

5.3 Type Checking

In this section, we discuss the well-formedness, typing, and subtyping relations of System !D. The type system reuses the System D subtyping algorithm to factor subtyping obligations between a first-order SMT solver and syntactic subtyping rules. The novel technical development in System !D is the formulation of flow-sensitive heap types in a higher-order, dependent setting. We summarize the syntax of environments and typing judgements in Figure 5.4.

Environments. The type checking relations make use of type environments Γ and heap environments Σ. A type environment binding records either: the derived type for a variable; a formula p to track control flow along a conditional branch; a polymorphic variable introduced by a function type; or the description of a weak location (which does not change flow-sensitively), namely, that every object stored at $\tilde{\ell}$ satisfies type T. A heap environment is just like a heap type, except that a strong location ℓ binds the value v it stores (as opposed to the type of v).

5.3.1 Well-Formedness

As usual in a refinement type system, we define well-formedness relations (Figure 5.5) that govern how values may be used inside formulas. The key intuition is that formulas are

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\Gamma, x{:}\mathit{Any} \vdash p}{\Gamma \vdash \{x \mid p\}} \qquad \frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash S}{\Gamma \vdash \exists x{:}T.\, S}$$

Well-Formed Worlds, Heaps, and Locations $\boxed{\Gamma \vdash W}$ $\boxed{\Gamma \vdash h}$ $\boxed{\Gamma \vdash \Psi}$ $\boxed{\Gamma \vdash m}$

$$\frac{\Gamma' = \Gamma, x{:}\mathit{Any}, \mathit{Binders}(h) \quad \Gamma' \vdash T \quad \Gamma' \vdash h \quad \text{no duplicate locations in } h}{\Gamma \vdash x{:}T\,/\,h}$$

$$\frac{}{\Gamma \vdash -} \qquad \frac{\Gamma \vdash h \quad \Gamma \vdash \ell \quad \Gamma \vdash T}{\Gamma \vdash h \oplus (\ell \mapsto x{:}T)} \qquad \frac{\Gamma \vdash h \quad \Gamma \vdash \tilde{\ell} \quad \Gamma \vdash \ell}{\Gamma \vdash h \oplus (\tilde{\ell} \mapsto \texttt{thwd}\ \ell)} \qquad \frac{\Gamma \vdash h \quad \Gamma \vdash \tilde{\ell}}{\Gamma \vdash h \oplus (\tilde{\ell} \mapsto \texttt{frzn})}$$

$$\frac{\Gamma \vdash \Psi \quad \Gamma \vdash \tilde{\ell} \quad \Gamma \vdash T}{\Gamma \vdash \Psi \oplus (\tilde{\ell} \mapsto T)} \qquad \frac{}{\Gamma \vdash a} \qquad \frac{}{\Gamma \vdash \tilde{a}} \qquad \frac{M \in \mathit{dom}(\Gamma)}{\Gamma \vdash M}$$

Well-Formed Type Terms (selected rules) $\boxed{\Gamma \vdash U}$

$$\frac{\Gamma \vdash m}{\Gamma \vdash \mathit{Ref}\ m} \qquad \frac{\Gamma' = \Gamma, \overline{A}, \overline{M}, \overline{H} \quad \Gamma' \vdash \Psi \quad \Gamma' \vdash W_1 \quad \Gamma', \mathit{Binders}(W_1) \vdash W_2}{\Gamma \vdash \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \rightarrow W_2}$$

Well-Formed Formulas and Logical Values (omitted) $\boxed{\Gamma \vdash p}$ $\boxed{\Gamma \vdash w}$

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

Figure 5.5. Well-Formedness for System !D

boolean propositions and mention only variables that are currently in scope. Locations in a heap type h must either be location constants or location variables bound by the type environment, and may not be bound multiple times. All of the binders in a world $x{:}T\,/\,h$, namely, x and all of the binders in h are in scope in T and all types in h. Thus, the values in a world can be regarded as a dependent tuple. For function types, the binders of the input world W_1 are in scope in the types of the output world W_2.

5.3.2 Value Typing

The value typing judgement $\Gamma;\, \Sigma \vdash v :: T$ (defined in Figure 5.6) verifies that the value v has type T in the given environments. Since values do not produce any effects, this judgement does not produce an output heap environment. Each primitive constant c has a type, denoted by $ty(c)$, that is used by T-Const. The standard T-Var rule assigns singleton or "selfified" types to

Value Typing $\boxed{\Gamma;\, \Sigma \vdash e :: T}$

$$\frac{}{\Gamma;\, \Sigma \vdash c :: ty(c)}\ [\text{T-Const}] \qquad \frac{x \in dom(\Gamma)}{\Gamma;\, \Sigma \vdash x :: \{y \mid y = x\}}\ [\text{T-Var}]$$

$$\frac{\Gamma;\, \Sigma \vdash (v_1,\, v_2,\, v_3) :: (Dict,\, Str,\, T)}{\Gamma;\, \Sigma \vdash v_1[v_2 \mapsto v_3] :: \{x \mid x = v_1[v_2 \mapsto v_3]\}}\ [\text{T-Extend}]$$

$$\frac{StrongLoc(r) = \ell \qquad m = \ell \text{ or } m = WeakLoc(\ell)}{\Gamma;\, \Sigma \vdash r :: \{x \mid x :: Ref\, m\}}\ [\text{T-Loc}]$$

$$\frac{\begin{array}{c} U = \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \to W_2 \qquad \Gamma \vdash U \qquad EnvOfWorld(W_1) = (x_0 : T_0,\ \overline{x} : \overline{T},\ \Sigma_1) \\ \Gamma_1 = \Gamma, \overline{A}, \overline{M}, \overline{H}, \Psi, (x_0 : T_0, \overline{x} : \overline{T}) \qquad \Gamma_1;\, \Sigma_1 \vdash e :: S \,/\, \Sigma_2 \qquad \Gamma_1 \vdash S \,/\, \Sigma_2 \models W_2;\ \pi \end{array}}{\Gamma;\, \Sigma \vdash \lambda x.e :: \{y \mid y :: U\}}\ [\text{T-Fun}]$$

World Satisfaction $\boxed{\Gamma \vdash S \,/\, \Sigma \models W;\ \pi}$

$$\frac{\Gamma' = \Gamma, \overline{x} : \overline{T} \qquad \Gamma' \vdash T' \sqsubseteq T'' \qquad \Sigma \sim h;\ \pi \qquad \Gamma', y : T' \Rightarrow \pi Embed(h)}{\Gamma \vdash \exists \overline{x} : \overline{T}.\ T' \,/\, \Sigma \models y : T'' \,/\, h;\ \pi}$$

Heap Environment Matching $\boxed{\Sigma \sim h;\ \pi}$

$$\frac{\Sigma \equiv \Sigma' \qquad h \equiv h' \qquad \Sigma' \sim h';\ \pi}{\Sigma \sim h;\ \pi} \qquad \frac{}{H \sim H;\ []}$$

$$\frac{\Sigma \sim h;\ \pi}{\Sigma \oplus (\ell \mapsto v) \sim h \oplus (\ell \mapsto x : T);\ \pi[v/x]} \qquad \frac{\Sigma \sim h;\ \pi}{\Sigma \oplus (\tilde{\ell} \mapsto \theta) \sim h \oplus (\tilde{\ell} \mapsto \theta);\ \pi}$$

Figure 5.6. Value Typing for System !D

variables. The T-Extend rule for dictionaries is the same as in System D. The T-Loc rule assigns run-time locations types based on their compile-time locations, as per previous discussion. The rule T-Fun uses the procedure *EnvOfWorld*, defined as

$$\begin{aligned}
\mathit{EnvOfWorld}(x_0 : T_0 \,/\, h) &= (x_0 : T_0,\ \overline{x} : \overline{T},\ \Sigma) \text{ where } \mathit{EnvOfHeap}(h) = (\overline{x} : \overline{T},\ \Sigma) \\
\mathit{EnvOfHeap}(h \oplus (\ell \mapsto x : T)) &= ((\Gamma, x : T),\ (\Sigma \oplus (\ell \mapsto x))) \text{ where } \mathit{EnvOfHeap}(h) = (\Gamma,\ \Sigma) \\
\mathit{EnvOfHeap}(h \oplus (\tilde{\ell} \mapsto \theta)) &= (\Gamma,\ (\Sigma \oplus (\tilde{\ell} \mapsto \theta))) \text{ where } \mathit{EnvOfHeap}(h) = (\Gamma,\ \Sigma) \\
\mathit{EnvOfHeap}(H) &= (-,\ H)
\end{aligned}$$

that takes a "snapshot" of the input world W_1 by collecting all of its binders to add to the type environment (as a dependent tuple) and producing a heap environment Σ_1 for type checking the body. Dually, after deriving type S and heap environment Σ_2 for the function body, the world satisfaction judgement $\Gamma_1 \vdash S \,/\, \Sigma_2 \models W_2;\ \pi$ checks that these satisfy W_2, modulo permutation of heap bindings. We write $h \equiv h'$ and $\Sigma \equiv \Sigma'$ for syntactic equality of heaps and heap environments modulo permutation. The substitution π maps binders from the heap type in W_2 to the values stored at corresponding locations in the heap environment Σ_2; the substitutions produced by world satisfaction are useful for type checking function applications, discussed shortly.

5.3.3 Expression Typing

The expression typing judgement $\Gamma;\ \Sigma \vdash e\ ::\ S \,/\, \Sigma'$ (defined in Figure 5.7) verifies that the evaluation of expression e produces a value of type S and a new heap environment Σ'.

Prenex Quantified Types. The T-Let rule uses an existential (like in the *algorithmic* presentation of System D in § 3.2) to describe the type T_1 of the variable x that goes out of scope after the body expression is checked. Alternatively, the more traditional approach (like in the *declarative* presentation of System D in § 2.3) requires that the variable be eliminated (*e.g.* via subsumption). But we use existentials here in the declarative presentation of System !D because it simplifies several other typing rules, in particular, T-Thaw and T-App which also derive prenex types. We could do without existentials in System !D if we changed the syntax of thaw operations and function applications so that each always appears as the equation of a let-binding, in which case we could use the traditional approach.

So that existentials appear only on the left side of subtyping obligations, we ensure that the typing rules derive prenex quantified types of the form $\exists \overline{x} : \overline{T}.\ T'$, where all the types $\overline{T}$ and T' are refinement types, not existential types. In particular, to combine the worlds of two branches, the *Join* operator (not shown) rearranges existentials in similar fashion as the *Join* operator in §3.2 to ensure that the resulting world is in prenex form. For example, for a conditional with guard b, the join of $(\exists x_1 : T_1.\ \mathit{Any} \,/\, (\ell \mapsto x_1))$ and $(\exists x_2 : T_2.\ \mathit{Any} \,/\, (\ell \mapsto x_2))$ is (equivalent to) $(\exists y : T_{12}.\ \mathit{Any} \,/\, (\ell \mapsto y))$ where $T_{12} \doteq \{z \mid \mathit{if}\ b = \mathtt{true}\ \mathit{then}\ T_1(z)\ \mathit{else}\ T_2(z)\}$.

Expression Typing $\boxed{\Gamma;\, \Sigma \vdash e \,::\, S \,/\, \Sigma'}$

$$\frac{\Gamma;\, \Sigma \vdash v \,::\, T}{\Gamma;\, \Sigma \vdash v \,::\, T \,/\, \Sigma}\;[\text{T-}\textsc{Val}]$$

$$\frac{\Gamma;\, \Sigma \vdash e_1 \,::\, \exists y\!:\!\overline{T'}.\, T_1 \,/\, \Sigma_1 \qquad \Gamma, \overline{y}\!:\!\overline{T'}, x\!:\!T_1;\, \Sigma_1 \vdash e_2 \,::\, S_2 \,/\, \Sigma_2}{\Gamma;\, \Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,::\, \exists \overline{y}\!:\!\overline{T'}.\, \exists x\!:\!T_1.\, S_2 \,/\, \Sigma_2}\;[\text{T-}\textsc{Let}]$$

$$\frac{\Gamma;\, \Sigma \vdash v \,::\, \mathit{Bool} \qquad \Gamma, v = \texttt{true};\, \Sigma \vdash e_1 \,::\, S_1 \,/\, \Sigma_1 \qquad \Gamma, v = \texttt{false};\, \Sigma \vdash e_2 \,::\, S_2 \,/\, \Sigma_2}{\Gamma;\, \Sigma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \,::\, \mathit{Join}(v, S_1 \,/\, \Sigma_1, S_2 \,/\, \Sigma_2)}\;[\text{T-}\textsc{If}]$$

$$\frac{\ell \notin \mathit{dom}(\Sigma) \qquad \Gamma;\, \Sigma \vdash v \,::\, T}{\Gamma;\, \Sigma \vdash \texttt{ref}\ \ell\ v \,::\, \{x \,|\, x :: \mathit{Ref}\ \ell\} \,/\, \Sigma \oplus (\ell \mapsto v)}\;[\text{T-}\textsc{Ref}]$$

$$\frac{\Gamma;\, \Sigma \vdash v \,::\, \{y \,|\, y :: \mathit{Ref}\ \ell\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v')}{\Gamma;\, \Sigma \vdash \texttt{deref}\ v \,::\, \{x \,|\, x = v'\} \,/\, \Sigma}\;[\text{T-}\textsc{Deref}]$$

$$\frac{\Gamma;\, \Sigma \vdash v_1 \,::\, \{y \,|\, y :: \mathit{Ref}\ \ell\} \qquad \Gamma;\, \Sigma \vdash v_2 \,::\, T \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v)}{\Gamma;\, \Sigma \vdash \texttt{setref}\ v_1\ v_2 \,::\, \{x \,|\, x = v_2\} \,/\, \Sigma_0 \oplus (\ell \mapsto v_2)}\;[\text{T-}\textsc{SetRef}]$$

$$\frac{\begin{array}{c}\Gamma;\, \Sigma \vdash v \,::\, \{y \,|\, y :: \mathit{Ref}\ \ell\} \qquad \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \theta) \oplus (\ell \mapsto v') \\ \theta = \texttt{frzn}\ \text{or}\ \theta = \texttt{thwd}\ \ell \qquad \Gamma(\tilde{\ell}) = T \qquad \Gamma;\, \Sigma \vdash v' \,::\, T\end{array}}{\Gamma;\, \Sigma \vdash \texttt{freeze}\ \tilde{\ell}\ \theta\ v \,::\, \{x \,|\, x :: \mathit{Ref}\ \tilde{\ell}\} \,/\, \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn})}\;[\text{T-}\textsc{Freeze}]$$

$$\frac{\Gamma;\, \Sigma \vdash v \,::\, \{z \,|\, z :: \mathit{Ref}\ \tilde{\ell}\} \qquad \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn}) \qquad \Gamma(\tilde{\ell}) = T}{\Gamma;\, \Sigma \vdash \texttt{thaw}\ \ell\ v \,::\, \exists x\!:\!T.\, \{y \,|\, y :: \mathit{Ref}\ \ell\} \,/\, \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{thwd}\ \ell) \oplus (\ell \mapsto x)}\;[\text{T-}\textsc{Thaw}]$$

$$\frac{\begin{array}{c}\Gamma;\, \Sigma \vdash v_1 \,::\, \{f \,|\, f :: \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \to W_2\} \qquad \Gamma;\, \Sigma \vdash v_2 \,::\, T'' \\ \Gamma \vdash [\overline{T} / \overline{A}] \qquad \Gamma \vdash [\overline{m} / \overline{M}] \qquad \Gamma \vdash [\overline{h} / \overline{H}] \qquad W_2' = \mathit{Freshen}(W_2) \\ (\Psi', W_1', W_2'') = \mathit{Inst}(\mathit{Inst}(\mathit{Inst}((\Psi, W_1, W_2'), \overline{A}, \overline{T}), \overline{M}, \overline{m}), \overline{H}, \overline{h}) \\ \Gamma \vdash \Psi' \qquad \Gamma \vdash W_1' \qquad \Gamma \vdash W_2'' \qquad \Psi' \subseteq \Gamma \qquad \Gamma \vdash T'' \,/\, \Sigma \models W_1';\ \pi \\ W_1' = y\!:\!\_ \,/\, \_ \qquad \pi' = \pi[v_2 / y] \qquad W_2''' = \pi' W_2'' \qquad \mathit{EnvOfWorld}(W_2''') = (x_0\!:\!T_0',\ \overline{x}\!:\!\overline{T'},\ \Sigma_2)\end{array}}{\Gamma;\, \Sigma \vdash [\overline{T}, \overline{m}, \overline{h}]\ v_1\ v_2 \,::\, \exists x_0\!:\!T_0'.\, \exists \overline{x}\!:\!\overline{T'}.\, \{z \,|\, z = x_0\} \,/\, \Sigma_2}\;[\text{T-}\textsc{App}]$$

Figure 5.7. Expression Typing for System !D

Imperative Operations. There are three rules for imperative operations on strong locations. To check the reference allocation `ref` ℓ v, the rule T-Ref ensures that ℓ is not already bound in the heap, and then adds a binding that records exactly the value being stored. The rule T-Deref checks that the given value is a reference to a strong location, and then retrieves the stored value; this is the imperative analog to the "selfifying" T-Var rule. The rule T-SetRef strongly updates a simple location.

In System !D, we use weak locations to describe collections of values, for which strong updates are not sound. Although we could define analogs of the previous three rules for weak locations, we require that *all* imperative operations go through strong locations because we provide a mechanism for temporarily thawing weak locations. To safely allow a weak location $\tilde{\ell}$ to be treated temporarily as strong, System !D ensures that $\tilde{\ell}$ has at most one corresponding *thawed* location at a time; if there is none, we say $\tilde{\ell}$ is *frozen*. The rule T-Thaw thaws $\tilde{\ell}$ to a strong location ℓ (which we syntactically require be distinct from all other thawed locations for $\tilde{\ell}$) and updates the heap environment with thaw state `thwd` ℓ to track the correspondence. Finally, the new heap also binds a value x of type T, the invariant for all values stored at $\tilde{\ell}$, and the output type introduces an existential so that x is in scope in the new heap.

The rule T-Freeze serves two purposes, to merge a strong location ℓ into a weak (frozen) location $\tilde{\ell}$ and to *re-freeze* a thawed (strong) location ℓ that originated from $\tilde{\ell}$, as long as the heap value stored at ℓ satisfies the invariant required by $\tilde{\ell}$. Compared to the presentation in [3], we combine freeze and re-freeze into a single `freeze` expression that includes an explicit thaw state θ.

Function Application. To type check $[\overline{T},\overline{m},\overline{h}]\ v_1\ v_2$, the T-App rule must perform some heavy lifting. Three well-formedness checks ensure that the number of type, location, and heap parameters must match the number of type, location, and heap variables of the function type, and that the sequence of locations $\overline{m}$ contains no duplicates to ensure the soundness of strong updates [91]. The procedure *Freshen* generates fresh binders for the output world so that bindings at different call sites do not collide.

The substitution of parameters for polymorphic variables proceeds in three steps. First, the type variables $\overline{A}$ inside has-type predicates are instantiated with the type parameters $\overline{T}$ using the procedure *Inst*. Second, the location variables $\overline{M}$ are replaced with the parameters $\overline{m}$ by substitution. Third, the heap variables $\overline{H}$ are instantiated with heap parameters. At this point, the polymorphic variables have been fully instantiated. The resulting weak heap, input world, and output world are checked for well-formedness to ensure that no locations appear multiple times.

Next, the argument type T'' and current heap environment Σ are checked to satisfy the input world W_1'. If so, the substitution π maps binders from the input heap in W_1' to the corresponding ones in the current heap Σ. The substitution is extended with a mapping to the argument v_2 and then applied to the output world. Then, like in the T-Fun rule, we use *EnvOfWorld* to collect the bindings of the output world and convert it to a heap environment Σ_2.

Subtyping (omitted) $\boxed{\Gamma \vdash T_1 \sqsubseteq T_2}$

Implication (omitted) $\boxed{\Gamma \Rightarrow p}$

Syntactic Subtyping (selected rules) $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{}{\Gamma \vdash \mathit{Ref}\ m <: \mathit{Ref}\ m}\ [\textsc{U-Ref}]$$

$$\frac{\Gamma \vdash W_{21} \sqsubseteq W_{11};\ \pi \qquad \Gamma, \mathit{Embed}(W_{21}) \vdash \pi W_{12} \sqsubseteq W_{22};\ \pi'}{\Gamma \vdash \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_{11} \rightarrow W_{12} <: \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_{21} \rightarrow W_{22}}\ [\textsc{U-Arrow}]$$

World Subtyping $\boxed{\Gamma \vdash W_1 \sqsubseteq W_2;\ \pi}$

$$\frac{\Gamma \vdash T_1 \sqsubseteq T_2 \qquad h_1 \sim h_2;\ \pi \qquad \pi' = \pi[x_1/x_2] \qquad \Gamma, x_1 : T_1, \mathit{Embed}(h_1) \Rightarrow \pi' \mathit{Embed}(h_2)}{\Gamma \vdash x_1 : T_1 \,/\, h_1 \sqsubseteq x_2 : T_2 \,/\, h_2;\ \pi'}$$

Heap Matching $\boxed{h_1 \sim h_2;\ \pi}$

$$\frac{h_1 \equiv h_1' \qquad h_2 \equiv h_2' \qquad h_1' \sim h_2';\ \pi}{h_1 \sim h_2;\ \pi} \qquad \frac{}{H \sim H;\ []}$$

$$\frac{h_1 \sim h_2;\ \pi}{h_1 \oplus (\ell \mapsto x_1 : T) \sim h_2 \oplus (\ell \mapsto x_2 : T);\ \pi[x_1/x_2]} \qquad \frac{h_1 \sim h_2;\ \pi}{h_1 \oplus (\tilde{\ell} \mapsto \theta) \sim h_2 \oplus (\tilde{\ell} \mapsto \theta);\ \pi}$$

Figure 5.8. Subtyping for System !D

Finally, the derived type uses existentials to describe the values in the output world.

Location Polymorphism. To simplify the presentation of System !D, we offer only a single mechanism — namely, universal quantification — to abstract over locations. As a result, functions must be quantified over all simple locations inserted by desugaring (to model imperative JavaScript variables), which clutters function types and, worse, requires explicit declaration and instantiation of locations that are "internal" to the desugaring translation and *not* accessible in the original DJS program. Instead, in the next chapter, we discuss how to use existential quantification in the output types of functions to describe these internal locations, and use universal quantification in the input types of functions to describe only those locations which are visible in DJS.

5.3.4 Subtyping

Several relations, defined in Figure 5.8, comprise subtyping.

Subtyping and Implication. As in System D, subtyping on refinement types reduces to implication of refinement formulas, which is discharged by a combination of uninterpreted, first-order reasoning and syntactic subtyping.

Syntactic Subtyping. As in Alias Types [91], we enforce the invariant that distinct strong locations do not alias, so references to them are *never* related by subtyping. The U-Ref relates each location only to itself. The U-Arrow rule for function types is familiar, treating input worlds contravariantly and output worlds covariantly.

Worlds. In order to check world subtyping, the judgement $\Gamma \vdash x_1 : T_1 \,/\, H_1 \oplus h_1 \sqsubseteq x_2 : T_2 \,/\, H_2 \oplus h_2$ checks that T_1 is a subtype of T_2 and that the heaps agree on the "deep" part (that is, if $H_1 = H_2$. Then, it checks that the structure of the "shallow" parts match — using a heap matching relation that uses a $\equiv$ operator that permutes bindings as necessary — and creates a substitution π of binders from h_2 to h_1. Finally, the heap bindings, which can be thought of as dependent tuples, are embedded as formulas and checked by implication.

Embedding. We write $\mathit{Embed}(T)$ for the *embedding* of a type as a formula, a straightforward definition in § 2.3 that lifts to environments $\mathit{Embed}(\Gamma)$, heap bindings $\mathit{Embed}(h)$, and worlds $\mathit{Embed}(W)$. Because the binders in a world may refer to each other in any order (recall that a world can be thought of as a dependent tuple, where each component is named with a binder), the embedding of a world starts by inserting dummy bindings so that all binders in scope for the type of each heap binding. For example:

$$\begin{aligned} W_0 &\doteq x_0 : T_0 \,/\, H \oplus (\ell_1 \mapsto x_1 : T_1) \oplus (\ell_2 \mapsto x_2 : T_2) \\ \mathit{Embed}(W_0) &= \mathit{Embed}(x_0{:}\mathit{Any}, x_1{:}\mathit{Any}, x_2{:}\mathit{Any}, T_0(x_0), T_1(x_1), T_2(x_2)) \end{aligned}$$

5.3.5 Type Soundness

We intend System !D to satisfy progress and preservation theorems, but we do not prove them. The process will likely be tedious but not require new proof techniques. Unlike System !D, which introduced the problematic nesting of syntactic types inside uninterpreted formulas, System !D does not introduce any new mechanisms in the refinement logic. Furthermore, several variations of Alias Types [91, 102, 57, 3, 39], even in a dependent setting [86], have been proven sound, and we expect to reuse their techniques to prove the soundness of System !D. Alternatively, rather than adopting these prior syntactic techniques, other proof strategies to consider, in future work, are to translate System !D into a more expressive verification system like Hoare Type Theory [73] or F* [95], or even in world-passing style to System D, which can model the heap as a dictionary indexed by references.

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript (Technical Appendix).** `arXiv:1112.4106v3 [cs.PL]`, August 2012.

- Ravi Chugh and Ranjit Jhala. **Dependent Types for JavaScript.** `arXiv:1112.4106v1 [cs.PL]`, December 2011.

Chapter 6

System DJS

In the previous chapter, we showed how to combine nested refinements with support for flow-sensitive heaps in order to reason precisely about mutable objects. In this chapter, we extend our techniques with a refinement type encoding called *heap unrolling* to reason precisely about mutable objects that feature prototype-based inheritance. In addition, we provide refinement type encodings for JavaScript primitive operators and arrays, which pose several challenges. Finally, we describe how to reason about the loops and control flow operators in an imperative language like JavaScript. The resulting system, called System DJS, serves as the target for translating and reasoning about Dependent JavaScript in the next chapter.

6.1 Overview

Terminology. JavaScript has a long history and an evolving specification. Throughout the rest of this dissertation, we say "JavaScript" to roughly mean ECMAScript Edition 3, the standard version of the language for more than a decade [61]. We say "ES5" to refer to Edition 5 of the language, recently released by the JavaScript standards committee [32]; Edition 4 was never standardized. We say "ES6" to refer to features proposed for the next version of the language, scheduled to be finalized soon. DJS includes a large set of core features common to all editions.

6.1.1 Base Types and Primitive Operators

The base values of JavaScript differ slightly from those we have presented in System !D. Like before, JavaScript has booleans, strings, and functions with tags "`boolean`", "`string`", and "`function`", respectively. Unlike System !D, however, JavaScript has only a single number type, for both integers and double-precision floating point numbers, with tag "`number`"; we included only integers in System !D, with tag "`integer`", omitting floats for simplicity. Furthermore, the tag of `null` in JavaScript is "`object`" (not "`null`"), as is the tag of every object reference. Finally, there is an `undefined` value with tag "`undefined`".

Consider the following JavaScript function, annotated in DJS, based on a familiar example from previous chapters. Notice that unlike the System !D function `also_negate` (discussed in §5.1.1), the tag-test checks for "`number`" rather than "`integer`" and, more importantly, that the argument type annotation is *Any* rather than *NumOrBool*.

```
//: negateAny :: (x:Any) → {y | if Num(x) then Num(y) else Bool(y)}
var negateAny = function (x) {
  if (typeof x == "number") { x = 0 - x; }
  else                      { x = !x;    }
  return x;
};
```

If the input to `negateAny` is a number, so is the return value. If not, the function uses an interesting feature of JavaScript, namely, that *all* values have a boolean interpretation. The values `false`, `null`, `undefined`, the empty string "", `0`, and the "not-a-number" value `NaN` are considered *falsy*, and evaluate to `false` when used in a boolean context; all other values are *truthy*. The operator `!` inverts "truthiness," so the else-branch returns a boolean no matter what the type of `x` is. The ability to treat arbitrary values as booleans is commonly used, for example, to guard against `null` values. We use the following abbreviations to capture this notion:

$$\begin{aligned} \mathit{falsy}(x) &\doteq x \in \{\texttt{false} \vee \texttt{null} \vee \texttt{undefined} \vee \text{``''} \vee 0 \vee \texttt{NaN}\} \\ \mathit{truthy}(x) &\doteq \neg \mathit{falsy}(x) \end{aligned}$$

Primitives. Using refinements, we assign precise, and sometimes exact, types to System DJS primitive functions, defined in the file `basics.djs` (Figure 6.1). The type of the `not` operator (which the DJS negation operator `!` desugars to) inverts truthiness. The types of the operators `&&` and `||` are interesting, because as in JavaScript, they do not necessarily return booleans. The "guard" operator `&&` returns its second operand if the first is truthy, which enables the idiom `if (x && x.f) { ... }` that checks whether the object `x` and its "`f`" field are non-`null`. Dually, the "default" operator `||` returns its second operand if the first is falsy, which enables the idiom `x = x || default` to specify a default value.

The + operator is specified as an *intersection* of function types and captures the fact that it performs both string concatenation and numerical addition, but does *not* type check expressions like $3 + \text{``}\texttt{hi}\text{''}$ that rely on the *implicit coercion* in JavaScript. We choose types for System !D primitives that prohibit implicit coercions since they often lead to subtle programming errors.

By using refinement types, we have the flexibility to decide how much we want to restrict the underlying semantics of the language. The reader may wish to compare the types of operators in `basis.djs` to the original ones in System D (and, hence, System !D) defined in §2.3.2.

Equality. JavaScript provides two equality operators: == implicitly coerces the second operand if its tag differs from the first, and strict equality === does not perform any coercion. To avoid

```
val tagof :: (*: x:Any → y:Str{y = tag(x)} *)
val   not :: (*: x:Any → b:Bool{b iff falsy(x)} *)
val  (||) :: (*: x:Any → y:Any → z:Any{if falsy(x) then z = y else z = x} *)
val  (&&) :: (*: x:Any → y:Any → z:Any{if truthy(x) then z = y else z = x} *)
val (===) :: (*: x:Any → y:Any{tag(y) = tag(x)} → b:Bool{b iff (x = y ∧ x ≠ NaN)} *)
val  (==) :: (*: x:Any → y:Any → b:Bool{tag(x) = tag(y) ⇒ b iff (x = y ∧ x ≠ NaN)} *)
val   (+) :: (*: x:Num → y:Num → z:Num{(Int(x) ∧ Int(y)) ⇒ (Int(z) ∧ z = x + y)} *)
val   (+) :: (*: Str → Str → Str *)
val   fix :: (*: ∀A. (A → A) → A *)
```

Figure 6.1. Excerpt from System DJS file `basics.djs`

reasoning about implicit coercions, we give a relatively weaker type to ==, where the boolean result relates its operands *only if* they have the same tag.

Integers. JavaScript provides a single number type that has no minimum or maximum value. However, programmers and optimizing JIT compilers [53] often distinguish integers from arbitrary numbers. In System !D, we describe integers with the following abbreviation:

$$Int(x) \;\overset{\circ}{=}\; Num(x) \wedge integer(x)$$

We introduce the uninterpreted predicate $integer(x)$ in the types of integer literals. Numeric functions like + propagate "integer-ness" where possible, and they use the (decidable) theory of linear arithmetic to precisely reason about integers, which is important for dealing with arrays.

Control Flow and Imperative Updates. System DJS inherits the System !D mechanisms for reasoning about control flow along branches and about strong updates to mutable variables. Thus, after desugaring the DJS function `negateAny` to System DJS in the same style described in the previous chapter, DJS verifies that the function satisfies the type annotation.

6.1.2 Prototype-Based Objects

JavaScript sports a special form of inheritance, where each base object is equipped with a link to its *prototype object*. This link is set when the base object is created and cannot be changed or accessed by the program. When trying to retrieve a key `k` not stored in an object `x` itself, JavaScript *transitively* searches the *prototype chain* of `x` until it either finds `k` or it reaches the root of the object hierarchy without finding `k`. The prototype chain does not play a role in the semantics of key update, addition, or deletion. (Many implementations, however, expose the prototype of an object `x` with a non-standard `x.__proto__` property, and prototypes do affect key update in ES5. We discuss these issues further in §7.4.)

For example, consider the initially empty object `child` created by the function `beget` (described in the sequel) with prototype object `parent`. The prototype of object literals, like `parent`, is the object stored in `Object.prototype` (note that the "`prototype`" key of `Object` is *not* the same as its prototype object). Thus, all keys in `parent` and `Object.prototype` are transitively accessible via `child`.

```
var parent = {"last": " Doe"};
var child = beget(parent);
child.first = "John";
assert (child.first + child.last == "John Doe");
assert ("last" in child == true);
assert (child.hasOwnProperty("last") == false);
```

The JavaScript operator `k in x` tests for the presence of `k` *anywhere* along the prototype chain of `x`, whereas the native function `Object.prototype.hasOwnProperty` tests only the "own" object itself. Keys routinely resolve through prototypes, so a static type system must precisely track them. Unfortunately, we cannot encode prototypes directly within the framework of refinement types and strong update, as the semantics of transitively traversing mutable and unbounded prototype hierarchies is beyond the reach of decidable, first-order reasoning.

Shallow and Deep Heaps. We solve this problem by syntactically reducing reasoning about prototype-based objects to the refinement logic. Our key insight is to decompose the heap into a "shallow" part, the bounded portion of the heap for which we have explicit locations, and a "deep" part, which is the potentially unbounded portion which we can represent by uninterpreted heap variables H. We explicitly track prototype links in the shallow heap by using bindings of the form $(\ell \mapsto \langle d, \ell' \rangle)$, where the prototype of the object at ℓ is stored at ℓ'. We cannot track prototype links explicitly in the deep heap, so instead we summarize information about deep prototype chains by using the abstract (uninterpreted) heap predicate $\mathit{HeapHas}(H,\ell,k)$ to encode the proposition that the object stored at location ℓ in H transitively has the key k, and the abstract (uninterpreted) heap function $\mathit{HeapSel}(H,\ell,k)$ to represent the corresponding value retrieved by lookup.

As an example, recall the `child` object and its prototype `parent`. Suppose that the prototype of `parent` is an unknown object `grandpa`, rather than `Object.prototype` as written. If `child`, `parent`, and `grandpa` are stored at locations ℓ_1, ℓ_2, and ℓ_3 with underlying "own" dictionary values d_1, d_2, and d_3, then we write the following heap:

$$\ell_1 \mapsto \langle d_1, \ell_2 \rangle \oplus \ell_2 \mapsto \langle d_2, \ell_3 \rangle \oplus \ell_3 \mapsto \langle d_3, \ell_4 \rangle \oplus H$$

Despite not knowing what value is the prototype of `grandpa`, we name its location ℓ_4 that is somewhere in the deep part of the heap H.

Function Types. Before we proceed, we pause to discuss function types in System DJS, which are of the form

$$\forall \overline{A}, \overline{M}, \overline{H}.\ W_1 \rightarrow W_2 \ = \ \forall \overline{A}, \overline{M}, \overline{H}.\ x_1 \,{:}\, T_1 \,/\, h_1 \rightarrow x_2 \,{:}\, T_2 \,/\, h_2$$

just like in System !D (§5.1.3). We re-iterate the notational conveniences introduced for System !D and we add two new ones in System DJS to accommodate prototype links and the fact that heap variables H can now be mentioned inside formulas:

- When used as an output heap, the token *same* refers to the sequence of locations in the corresponding input heap, where each binding records that the final value is exactly equal to the initial value.
- In an input world, a reference binding $x\,{:}\,\mathit{Ref}$ without a location introduces a location variable L that is quantified by the type, and x (a value of type $\mathit{Ref}\ L$) can be used as a location in heaps to refer to this variable L. Further, the dotted variable $\dot{x}$ introduces a location parameter L' corresponding to the prototype of x, and $\dot{x}$ can be used as a location in heaps to refer to this variable L'.
- A heap variable H is implicitly added to a function type when it contains none, and H is added to both the input and output heaps; this variable corresponds to the "frame" from separation logic [44]. In this case, the token *cur* refers to H.

Key Membership and Lookup. When describing simple objects in System !D, we used the original System D primitives (`mem` and `get`) to desugar key membership and lookup operations. In System DJS, however, to account for the transitive semantics of key membership and lookup facilitated by prototype links, we introduce new primitives `hasPropObj` and `getPropObj` defined in `objects.djs` (Figure 6.2). These primitives differ from their purely functional System D counterparts in two ways: each operation goes through a *reference* to a dictionary on the heap, and the abstract predicates *ObjHas* and *ObjSel* are used in place of *has* and *sel*. These abstract predicates are defined over the *disjoint union* of the shallow and deep heaps as follows and, intuitively, summarize whether an object transitively has a key and, if so, the value it binds.

$$\begin{aligned} \mathit{ObjHas}(d,k,H,\dot{x}) \ &\mathrel{\dot{=}}\ \mathit{has}(d,k) \vee \mathit{HeapHas}(H,\dot{x},k) \\ x = \mathit{ObjSel}(d,k,H,\dot{x}) \ &\mathrel{\dot{=}}\ \mathit{if}\ \mathit{has}(d,k)\ \mathit{then}\ x = \mathit{sel}(d,k)\ \mathit{else}\ x = \mathit{HeapSel}(H,\dot{x},k) \end{aligned}$$

Transitive Semantics via Unrolling. Let us return to the example of the `child`, `parent`, and `grandpa` prototype chain to understand how unrolling captures the semantics of transitive lookup. Below, we show how DJS key membership tests and key lookup operations (on the left) desugar to System DJS (on the right):

```
val hasPropObj :: (*: (x:Ref, k:Str) / (x ↦ ⟨d:Dict, ẋ⟩)
                      → b:Bool{b iff ObjHas(d,k,cur,ẋ)} / same *)
val getPropObj :: (*: (x:Ref, k:Str) / (x ↦ ⟨d:Dict, ẋ⟩)
                      → x:Any{x = ObjSel(d,k,cur,ẋ)} / same *)
val setPropObj :: (*: (x:Ref, k:Str, y:Any) / (x ↦ ⟨d:Dict, ẋ⟩)
                      → z:Any{z = y} / (x ↦ ⟨d′:Dict{d′ = upd(d,k,y)}, ẋ⟩) *)
val delPropObj :: (*: (x:Ref, k:Str) / (x ↦ ⟨d:Dict, ẋ⟩)
                      → Bool / (x ↦ ⟨d′:Dict{d′ = upd(d,k,bot)}, ẋ⟩) *)

val  getIdxArr :: (*: ∀A. (x:Ref, i:Int) / (x ↦ ⟨a:Arr(A), ẋ⟩)
                      → y:Any{arrGetIdx(y,a,i,A)} / same *)
val  getLenArr :: (*: ∀A. (x:Ref, k:Str{k = "length"}) / (x ↦ ⟨a:Arr(A), ẋ⟩)
                      → n:Int{if packed(a) then n = len(a) else true} / same *)
val getPropArr :: (*: ∀A. (x:Ref, k:Str{k ≠ "length"}) / (x ↦ ⟨a:Arr(A), ẋ⟩)
                      → y:Any{arrGetProp(y,cur,ẋ,k)} / same *)

val  setIdxArr :: (*: ∀A. (x:Ref, i:Int, y:A) / (x ↦ ⟨a:Arr(A), ẋ⟩)
                      → z:Any{z = y} / (x ↦ ⟨a′:Arr(A){arrSetIdx(a′,a,i)}, ẋ⟩) *)

val    getElem :: (and (type getPropObj)
                       (type getIdxArr) (type getLenArr) (type getPropArr))
```

Figure 6.2. Excerpt from System DJS file `objects.djs`

`k in child`	1	`hasPropObj (deref _child, deref _k)`
`child[k]`	2	`getPropObj (deref _child, deref _k)`

Assuming that the mutable reference `_k` stores a value k, the result of the System DJS function call on line 1 has the following type, which we expand by unrolling *ObjHas*:

$$\begin{aligned} &\{x \mid x \textit{ iff } \mathit{ObjHas}(d_1,k,(\ell_2 \mapsto \langle d_2,\ell_3\rangle) \oplus (\ell_3 \mapsto \langle d_3,\ell_4\rangle) \oplus H,\ell_2)\} \\ =\ &\{x \mid x \textit{ iff } \mathit{has}(d_1,k) \vee \mathit{has}(d_2,k) \vee \mathit{has}(d_3,k) \vee \mathit{HeapHas}(H,\ell_4,k)\} \end{aligned}$$

The first three disjuncts correspond to looking for k in the shallow heap, and the last is the uninterpreted predicate that summarizes whether k exists in the deep heap. Similarly, we unroll *ObjSel* in the type of the System DJS call on line 2 as follows:

$$\begin{aligned} \{x \mid\ &\textit{if } \mathit{has}(d_1,k) \textit{ then } x = \mathit{sel}(d_1,k) \textit{ else} \\ &\textit{if } \mathit{has}(d_2,k) \textit{ then } x = \mathit{sel}(d_2,k) \textit{ else} \\ &\textit{if } \mathit{has}(d_3,k) \textit{ then } x = \mathit{sel}(d_3,k) \textit{ else} \\ &\textit{if } \mathit{HeapHas}(H,\ell_4,k) \textit{ then } x = \mathit{HeapSel}(H,\ell_4,k) \textit{ else } x = \texttt{undefined}\ \} \end{aligned}$$

Notice that we use the constant `undefined` in the last case because, unlike in stricter languages

```
//: __hasOwn :: (this:Ref, k:Str) / (x ↦ ⟨d:Dict, ẋ⟩) → b:Bool{b iff has(d,k)} / same
//: __hasOwn :: ∀A. (this:Ref, k:Str) / (this ↦ ⟨a:Arr(A), ṫhis⟩)
//:                 → b:Bool{b iff k = "length"} / same
//: __hasOwn :: ∀A. (this:Ref, i:Int) / (this ↦ ⟨a:Arr(A), ṫhis⟩)
//:                 → b:Bool{if packed(a) then b iff 0 ≤ i < len(a) else true} / same
var __hasOwn = "#extern";

function Object() { ... }
Object.prototype = { "hasOwnProperty": __hasOwn, ... };

//: __push :: ∀A. (this:Ref, x:A) / (this ↦ ⟨a:Arr(A), ṫhis⟩)
//:                 → Int / (this ↦ ⟨a′:Arr(A){arrSetIdx(a′,a,1)}, ṫhis⟩)
var __push = "#extern";

//:  __pop :: ∀A. (this:Ref, x:A) / (this ↦ ⟨a:Arr(A), ṫhis⟩)
//:                 → y:Any{if packed(a) then y :: A else Undef(y)}
//:                 / (this ↦ ⟨a′:Arr(A){arrSetIdx(a′,a,−1)}, ṫhis⟩)
var  __pop = "#extern";

function Array() { ... }
Array.prototype = {"push": __push, "pop": __pop, ... };
```

Figure 6.3. Excerpt from DJS file `prelude.js`

like System !D, the semantics of JavaScript returns `undefined` when an unbound key is retrieved. (Attempting to retrieve a key from `undefined`, however, *does* produce a run-time error.) Thus, our technique of decomposing the heap into shallow and deep parts, followed by heap unrolling, captures the *exact* semantics of prototype-based object operations modulo the unknown portion of the heap. Thus, System DJS precisely tracks objects in the presence of mutation and prototypes.

Additional Primitives. The new update and deletion primitives `setPropObj` and `delPropObj` (Figure 6.2) affect only the "own" object, since the prototype chain does not participate in the semantics. We model native JavaScript functions like `Object.prototype.hasOwnProperty` with type annotations in the file `prelude.js` (Figure 6.3). Notice that the function type for objects (the first in the intersection) checks only the "own" object for the given key.

Constructors. JavaScript provides the expression form `new Foo(args)` as a second way of *constructing* objects, in addition to object literals whose prototypes are set to `Object.prototype`. The semantics are straightforward, but quite different than the traditional `new` syntax suggests. Here, if `Foo` is any function (object), then a fresh, empty object is created with prototype object `Foo.prototype`, and `Foo` is called with the new object bound to `this` (along with the remaining arguments) to finish its initialization. We desugar constructors and `new` with standard objects and functions (following λ_{JS} [51]) without adding any special System DJS constructs or primitive functions. We describe the desugaring of constructors in detail in the next chapter.

Inheritance. Several inheritance relationships, including ones that simulate traditional classes, can be encoded with the construction mechanism, as shown in the popular book *JavaScript: The Good Parts* [23]. Here, we examine the "prototypal pattern," a minimal abstraction which wraps construction to avoid the unusual syntax and semantics that leads to common errors; we discuss the rest in §7.3. The function `beget` (the basis for `Object.create` in ES5) returns a fresh empty object with prototype `o`.

```
1 //: beget :: ∀L. (o:Ref) / (o ↦ ⟨d:Dict, ȯ⟩)
2 //:          → Ref L / (L ↦ ⟨{x | x = empty}, o⟩) ⊕ (o ↦ same)
3 var beget = function (o) {
4   /*: ctor F :: this:Ref → {x | x = this} */
5   function F() { return this; };
6   F.prototype = o;
7   return new /*: L */ F();
8 };
```

The DJS token `ctor` on line 4 instructs desugaring to: initialize the function object with a "`prototype`" key that stores an empty object literal (since it will be called as a constructor); and expand the type annotation provided as follows to require that `this` initially be an empty dictionary, as is common for all constructors.

$$\mathit{this}\!:\!\mathit{Ref} \,/\, (\mathit{this} \mapsto \langle \{d \mid d = \mathit{empty}\}, \dot{\mathit{this}} \rangle) \to \{x \mid x = \mathit{this}\} \,/\, \mathit{same}$$

The assignment on line 6 strongly updates `Foo.prototype` (overwriting its initial empty object) with the argument `o`. Thus, the object constructed (at location L) on line 7 has prototype `o`, so `beget` has the ascribed type. In most cases, `new` can be used without a location annotation and a fresh one is chosen by our implementation. In this case, however, we annotate line 7 with L (from the type of `beget`), which our implementation does not infer because there is no input corresponding to L.

6.1.3 Arrays

The other workhorse data structure of JavaScript is the array, which is really just an object with integer "indices" converted to ordinary string keys. However, arrays pose several tricky challenges as they are commonly used both as finite tuples as well as unbounded collections.

```
var arr = [17, "hi", true];
arr[3] = 3; arr.push(4);
assert (arr.length == 5 && arr[5] == undefined);
```

As for any object, retrieving a non-existent key returns `undefined` rather than raising an "out-of-bounds" exception. Like other objects, arrays are extensible simply by writing "past the end." Array literal objects have prototype `Array.prototype`, which includes a `push` (resp. `pop`) function for adding an element to (resp. removing an element from) the end of an array.

Loops are used to iterate over arrays of unknown size. But since lookups may return `undefined`, it is important to track when an access is "in-bounds." JavaScript bestows upon arrays an unusual "`length`" property, rather than a method, to help. Reading it returns the largest integer key of the array, which is *not* necessarily its "size" because it may contain "holes" or even non-integer keys. Furthermore, assigning a number `n` to the "`length`" of an array either truncates it if `n` is less than its current length, or extends it (by padding with holes) if it is greater. Despite the unusual semantics, programmers commonly use arrays as if they are traditional "packed" arrays with integer "indices" zero to "size" minus one. The type system must reconcile this discrepancy.

Array Types. We introduce a new syntactic type term $\mathit{Arr}(T)$ and maintain the following four properties for every value a that satisfies the has-type predicate $a :: \mathit{Arr}(T)$. We refer to strings that do not coerce to integers as "safe," and we use an uninterpreted predicate *safe* to describe such strings. For example, $\mathit{safe}(\text{“}\texttt{foo}\text{”})$ whereas $\neg\mathit{safe}(\text{“}\texttt{17}\text{”})$.

(A1) a contains the special "`length`" key.

(A2) All other "own" keys of a are (strings that coerce to) integers.

(A3) For all integers i, either a maps the key i to a value of type T, or it has no binding for i.

(A4) All inherited keys of a are safe (*i.e.* non-integer) strings.

An array can have arbitrary objects in its prototype chain, so to ensure (A4), we require that *all* non-array objects bind only safe strings. This sharp distinction between between array objects (that bind integer keys) and non-array objects (that bind safe string keys) allows System DJS to avoid reasoning about string coercions, and does not significantly limit expressiveness because, in our experience, programs typically conform to this division anyway. To enforce this restriction, the type for keys manipulated by primitives in `objects.djs` and `prelude.js` is actually *SafeStr*, rather than *Str* as shown in Figure 6.2 and Figure 6.3, where $\mathit{SafeStr} \;\mathrel{\dot{\hat{=}}}\; \{s \,|\, \mathit{Str}(s) \wedge \mathit{safe}(s)\}$. We discuss an alternative approach in §7.4 that allows non-array objects to bind unsafe strings.

Packed Arrays. Arrays a that additionally satisfy the uninterpreted predicate $\mathit{packed}(a)$ enjoy the following property, where $\mathit{len}(a)$ is an uninterpreted function symbol.

(A5) For all integers i, if i is between zero and $\mathit{len}(a)$ minus one, then a maps i to a value of type T. Otherwise, a has no binding for i.

We introduce the following notation for packed arrays:

$$a :: \mathit{Array}(T) \;\mathrel{\hat{=}}\; a :: \mathit{Arr}(T) \wedge \mathit{packed}(a)$$

Tuple Arrays. Using additional predicates, System DJS gives precise types to array literals, which are often used as finite tuples in idiomatic code. For example, we can describe pairs as follows:

$$\begin{aligned}
[\mathit{Int},\ \mathit{Int}] \;&\mathrel{\hat{=}}\; \{a \,|\, a :: \mathit{Array}(\mathit{Int}) \wedge \mathit{len}(a) = 2\} \\
[\mathit{Bool},\ \mathit{Str}] \;&\mathrel{\hat{=}}\; \{a \,|\, a :: \mathit{Array}(\mathit{Any}) \wedge \mathit{len}(a) = 2 \wedge \mathit{Str}(\mathit{sel}(a,0)) \wedge \mathit{Bool}(\mathit{sel}(a,1))\}
\end{aligned}$$

Thus, the technique of nested refinements allows us to smoothly reason about arrays both as packed homogeneous collections as well as heterogeneous tuples.

Array Primitives. We define several array-manipulating primitives in `objects.djs` (some of which we show in Figure 6.2) that maintain and use the array invariants above. For key lookup on arrays, we define three primitives:

- `getIdxArr` looks for the integer key i on the own object a using the abbreviation

  $$\begin{aligned}
  \mathit{arrGetIdx}(y,a,i,A) \;\mathrel{\hat{=}}\;& \mathit{if}\ \neg\mathit{packed}(a)\ \mathit{then}\ (y :: A \vee \mathit{Undef}(y))\ \mathit{else} \\
  & (\mathit{if}\ 0 \leq i < \mathit{len}(a)\ \mathit{then}\ y :: A\ \mathit{else}\ \mathit{Undef}(y))
  \end{aligned}$$

  and ignores the prototype chain of a because (A4) guarantees that a will not inherit i, and returns a value subject to the properties (A3) and (A5) that govern its integer key bindings;

- `getLenArr` handles the special case when the string key k is "`length`", which (A1) guarantees is bound by a, and returns the the true length of the array only if it is packed; and

- `getPropArr` deals with all other (safe) string keys k by reading from the prototype chain of the array, re-using the heap unrolling mechanism and the macro

  $$\mathit{arrGetProp}(y,H,\ell,k) \;\mathrel{\hat{=}}\; \mathit{if}\ \mathit{HeapHas}(H,\ell,k)\ \mathit{then}\ y = \mathit{HeapSel}(H,\ell,k)\ \mathit{else}\ \mathit{Undef}(y)$$

  ignoring its own bindings because of (A2).

For array updates, we define `setIdxArr` that uses the following macros to preserve packedness (A5) when possible.

$$\begin{aligned}
\mathit{arrSetIdx}(a',a,i) \;&\mathrel{\hat{=}}\; \mathit{if}\ 0 \leq i < \mathit{len}(a)\ \mathit{then}\ \mathit{arrSize}(a',a,0)\ \mathit{else} \\
&\phantom{\mathrel{\hat{=}}}\;\ \mathit{if}\ i = \mathit{len}(a)\ \mathit{then}\ \mathit{arrSize}(a',a,1)\ \mathit{else}\ \mathit{true} \\
\mathit{arrSize}(a',a,n) \;&\mathrel{\hat{=}}\; \mathit{packed}(a) \Rightarrow (\mathit{packed}(a') \wedge \mathit{len}(a') = \mathit{len}(a) + n)
\end{aligned}$$

In particular, the updated array a' is packed if (1) the original array a is packed, and (2) the updated index i is either within the bounds of a (in which case, the length of a' is the same as a)

or just past the end (so the length of a' is one greater than a). In similar fashion, we specify the remaining primitives for update and deletion to maintain the array invariants, and the ones for key membership to use them, but we do not show them in Figure 6.2.

In `prelude.js` (Figure 6.3), we use precise types to model the native `push` and `pop` methods of `Array.prototype` (which maintain packedness, as above), as well as the behavior of `Object.prototype.hasOwnProperty` on arrays (the last two cases of the intersection type). Thus, the precise dependent types we ascribe to array-manipulating operations maintain invariants (A1) through (A5) and allow System DJS to precisely track array operations.

Desugaring. It may seem that we need to use separate primitive functions for array and non-array object operations, even though they are syntactically indistinguishable in JavaScript. Nevertheless, we are able to desugar DJS based purely on expression syntax (and not type information) by unifying key lookup within a single primitive `getElem` and giving it a type that is the *intersection* of the (three) array lookup primitives and the (one) non-array lookup primitive `getPropObj`. We define `getElem` in Figure 6.2, where we specify the intersection type using `and` and `type` as syntactic sugar to refer to the previous type annotations. We define similar unified primitives for `setElem`, `hasElem`, and `delElem` (not shown in Figure 6.2). Desugaring uniformly translates object operations to these unified general primitives, and type checking of function calls ensures that the appropriate cases of the intersection type apply.

6.1.4 Null References

The object and array primitives we have shown require (non-`null`) strong references as arguments, which statically rules out the possibility of null-dereference exceptions. To facilitate idiomatic programming, however, we choose to adopt types for these primitives that allow `null` references to be passed as arguments. For example, we modify the type of `hasPropObj` below, using the abbreviation $\mathit{Ref}? \circeq \{x \mid x :: \mathit{Ref}\ L \vee x = \texttt{null}\}$, where L is the location variable that the function type is implicitly quantified over. Notice that we add the predicate $x \neq \texttt{null}$ to the *output* type, because if `hasPropObj` evaluates without raising an exception, then x is guaranteed to be non-`null`, a fact which may help discharge subsequent obligations in the program.

```
val hasPropObj :: (*: (x:Ref?, k:Str) / (x ↦ ⟨d:Dict, ẋ⟩)
                      → b:Bool{x ≠ null ∧ b iff ObjHas(d,k,cur,ẋ)} / same *)
```

We modify the types for all other object and array primitives in `objects.djs` and `prelude.js` in similar fashion, but elide these changes from the presentation.

6.1.5 Collections

We reuse the weak location mechanism from System !D to describe recursive data structures in System DJS. Consider the following adapted from the SunSpider [93] benchmark `access-binary-trees.js`, annotated in DJS.

```
1  //: type TreeNode[~L] = {“i”:Num; “l”:(Ref ~L)?; “r”:(Ref ~L)?}
2
3  //: ctor TreeNode :: ∀~L. (~L ↦ ⟨TreeNode[~L], a_TreeNodeProto⟩) ⇒
4  //:                     (this:Ref, Num, Ref ~L?, Ref ~L?) → Ref ~L
5  function TreeNode(item, left, right) {
6    this.i = item; this.l = left; this.r = right;
7    //: freeze this
8    return this;
9  }
10
11 //: itemCheck :: ∀~L. (~L ↦ ⟨TreeNode[~L], a_TreeNodeProto⟩) ⇒ (Ref ~L?) → Num
12 TreeNode.prototype.itemCheck = function f() {
13   if (this.l == null) { return this.item; }
14   else {
15     return this.i + f.apply(this.l) + f.apply(this.r);
16   }
17 };
```

The source-level macro on line 1 introduces *TreeNode* to abbreviate the type of `TreeNode`s, using traditional record type syntax instead of the underlying McCarthy operators. This abbreviation is parameterized by a weak location $\tilde{L}$ so that disjoint sets of trees can be described with different weak reference types. In the constructor annotation on line 3, the prototype object for new instances is specified to be the predictable location $a_{TreeNodeProto}$ created by desugaring for the object `TreeNode.prototype`. The return type $Ref\ \tilde{L}$ declares that the output is a reference to one of these recursive objects, which System DJS verifies by checking that on line 6 the appropriate fields are added to the strong, initially-empty object `this` before it is frozen and returned.

Recursive Traversal. There are two differences in the `itemCheck` function above compared to the original version, which cannot be type checked in DJS. First, we name the function being defined (notice the `f` on line 12), a JavaScript facility for recursive definitions. Second, we write `f.apply(this.r)` instead of `this.r.itemCheck()` as in the original, where the native JavaScript function `apply` allows a caller to explicitly supply a receiver argument. The trouble with the original call is that it goes through the heap (in particular, the prototype chain of `this`) to resolve the recursive function being defined. This function will be stored in a strong object, and we have no facility (*e.g.* mu-types) for strong objects with recursive types; our only mechanism is for weak objects. If we write `f.apply(this.r)`, however, the recursive function `f` is syntactically manifest, and we can translate the definition with a call to the standard `fix` primitive (Figure 6.1). In Chapter 7, we describe how we handle a limited form of `apply` that is sufficient for our idiomatic recursive definitions in DJS. We expect that we can add a more powerful mechanism for recursive types that supports the original code as written, but we leave this to future work.

Loops. The System !D and System DJS examples with weak locations, which we have seen in this section and in §5.1.4, use recursion to iterate over elements of collections. We support loops in DJS by translation to recursive functions. As such, a loop requires an annotation like any other function. Consider the following function adapted from the previous chapter:

```
 1  //: type Passenger = {d | Dict(d) ∧ has(d,"weight") ⇒ Int(sel(d,"weight"))}
 2  //: sumWeights ::
 3  //:    ∀~L,L. (~L ↦ ⟨Passenger, L⟩) ⇒
 4  //:        (ps:Ref, Int) / (ps ↦ ⟨a:Array(Ref ~L), ṗs⟩) → Int / same
 5  var sumWeights = function (ps, max_weight) {
 6    var sum = 0;
 7    //: (~L ↦ frzn) ⊕ (ps ↦ ⟨a′:Any{a′ = a}, ṗs⟩) → same
 8    for (var i=0; i < ps.length; i++) {
 9      var p = ps[i];
10      //: thaw p
11      if (p.weight) { sum += p.weight;   }
12      else          { sum += max_weight; }
13      //: freeze p
14    }
15    return sum;
16  }
```

Line 7 provides a loop annotation, which specifies an input (resp. output) heap type that must be satisfied before (resp. that is guaranteed after) each iteration of the loop. This annotation says that the weak location is frozen before and after every iteration of the loop, and that the array stored at location *ps* is unmodified (*i.e.* the array value a' is exactly equal to the initial array a from the entry point of the `sumWeights` function). The latter fact is important for tracking that the length of the array is the same across each iteration and, thus, that the array lookup on line 9 is within bounds. Our implementation automatically inserts loop annotations for common patterns like this, which we will discuss in the next chapter.

6.1.6 Return Statements

Throughout our presentation so far, we have written only JavaScript functions that have either a single `return` statement or a `return` statement along every control flow path. In general, however, `return` statements can appear in arbitrary positions. For example, consider the following variation of the `negate` function, and for the purposes of this discussion, let us assume that we choose a type for the negation operator that accepts *only* boolean arguments:

```
//: anotherNegate :: (x:NumOrBool) → {y | tag(y) = tag(x)}
var anotherNegate = function (x) {
  if (typeof x == "number") {
    return 0 - x;
  }
  return !x;
};
```

In System DJS, we handle arbitrary `return` statements by using *break* and *label* expressions, following the approach of λ_{JS} [51]. The following is the desugared function in System DJS:

```
1  let anotherNegate (x) =
2    let _x = ref x in
3    @return {
4      let tmp =
5        if (tagof _x == "number") then
6          break @return (0 - deref _x)
7        else
8          undefined in
9      break @return (not (deref _x))
10   } in
11 let _anotherNegate = ref anotherNegate
```

Notice that the entire function body is wrapped in an expression labeled `@return`, and the two JavaScript `return` statements are translated to break expressions (on lines 6 and 9) that complete the evaluation of the expression labeled `@return`, producing the specified value. Because the if-statement in the JavaScript function did not specify an else-case, the desugared function produces `undefined` on the else-branch. The expression labeled `@return` is the final expression of the function body, so System DJS must check that it satisfies the return type annotation, namely, $\{y \mid \mathit{tag}(y) = \mathit{tag}(x)\}$. In particular, *every* break expression that terminates the `@return` expression must produce a value of this type.

The break expression on line 6 appears in the then-branch of the conditional, so $\mathit{Num}(x)$ holds. Furthermore, the value produced by the subtraction is also a number, so this break expression satisfies the expected return type. Let us now consider the break expression on line 9, which appears *after* the if-expression, which returned only on the then-branch. To verify the safety of the boolean negation on line 9 and to verify that the result satisfies the expected type, we need a way to track that execution reaches line 9 *only if* x is not a number. In System DJS, we assign the type $\{z \mid \mathit{false}\}$ to each break expression `break` $@x\ v$ (in addition to checking that v satisfies the expected type for $@x$) to capture the fact that code after the break is unreachable. As a result, by *joining* the types of the branches, we obtain the following type for the if-expression

$$\texttt{tmp} :: \{z \mid \mathit{if}\ \mathit{Num}(x)\ \mathit{then}\ \mathit{false}\ \mathit{else}\ z = \texttt{undefined}\}$$

$$\begin{array}{rcll}
\textbf{Values} & v & ::= & \lambda x.e \mid x \mid c \mid v_1[v_2 \mapsto v_3] \mid r \\
\textbf{Expressions} & e & ::= & v \mid [\overline{T},\overline{m},\overline{h}]\; v_1\; v_2 \\
 & & \mid & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ e \mid \texttt{ref}\ \ell\ v \mid \texttt{deref}\ v \mid \texttt{setref}\ v_1\ v_2 \\
 & & \mid & \texttt{newobj}\ \ell\ v\ v' \mid \texttt{freeze}\ \tilde{\ell}\ \theta\ v \mid \texttt{thaw}\ \ell\ v \\
 & & \mid & @x\texttt{\{}\,e\,\texttt{\}} \mid \texttt{break}\ @x\ v \\
\textbf{Types} & T & ::= & \{x \mid p\} \\
\textbf{Prenex Types} & S & ::= & \exists x{:}T.\,S \mid T \\
\textbf{Formulas} & p,q & ::= & P(\overline{w}) \mid w :: U \mid p \wedge q \mid p \vee q \mid \neg p \mid \mathit{HeapHas}(H,\ell,w) \\
\textbf{Logical Values} & w & ::= & v \mid F(\overline{w}) \mid \mathit{HeapSel}(H,\ell,w) \\
\textbf{Type Terms} & U & ::= & \forall \overline{A},\overline{M},\overline{H}.\ \Psi \Rightarrow W_1 \rightarrow W_2 \mid A \mid \mathit{Ref}\,m \mid \mathit{Arr}(T) \\
\textbf{Weak Heaps} & \Psi & ::= & - \mid \Psi \oplus (\tilde{\ell} \mapsto T) \mid \Psi \oplus (\tilde{\ell} \mapsto \langle T, \ell \rangle) \\
\textbf{(Strong) Heaps} & h & ::= & H \mid h \oplus (\ell \mapsto x{:}T) \mid h \oplus (\ell \mapsto \langle x{:}T, \ell' \rangle) \mid h \oplus (\tilde{\ell} \mapsto \theta) \\
\textbf{Thaw States} & \theta & ::= & \texttt{frzn} \mid \texttt{thwd}\ \ell \\
\textbf{Worlds} & W & ::= & x{:}T\,/\,h \\
\textbf{Metavariables} & @x & \in & \mathit{Labels}
\end{array}$$

Figure 6.4. Syntax of System DJS

which captures the fact that evaluation terminates (or diverges) on the then-branch. This type for `tmp`, together with the binding $x{:}\mathit{NumOrBool}$, allows System DJS to deduce that $\mathit{Bool}(x)$ on line 9. Thus, the boolean negation is well-typed and, furthermore, produces a value that satisfies the expected return type.

6.2 Syntax and Semantics

We now introduce the formal syntax of values, expressions, and types of System DJS, defined in Figure 6.4. The syntax and semantics of System DJS is based heavily on System !D (§5.2), so we limit our discussion in this section to the differences. The metavariables not defined in Figure 6.4 are inherited from Figure 2.1 and Figure 5.2.

Simple vs. Object Locations. In System DJS, we use two kinds of location bindings in strong heaps: a *simple location* binding $(\ell \mapsto x{:}T)$ describes the binding of a non-dictionary value x of type T, and an *object location* binding $(\ell \mapsto \langle x{:}T, \ell' \rangle)$ describes the binding of a dictionary value x of type T along with a prototype link to the location ℓ'. As in System !D, we use weak locations to describe collections of values. Analogous to strong heap locations, there are weak simple locations and weak object locations. The weak heap binding $(\tilde{\ell} \mapsto T)$ describes an arbitrary number (one or more) values of type T, and $(\tilde{\ell} \mapsto \langle T, \ell \rangle)$ describes an arbitrary number of dictionary values of type T, all of which have the prototype link ℓ. As we will see in the next chapter, the desugaring of DJS to System DJS requires only weak locations for objects, but the mechanism works for both kinds of locations.

Expressions. The syntax of expressions differs from System !D in four ways. First, we introduce an additional expression form `let` $x =$ `ref` v `in` e for allocating simple references. We syntactically pair reference allocation with a let-binding to facilitate a new typing rule for references. The expression forms for dereference and assignment are unchanged. Second, the expression `newobj` $\ell\ v\ v'$ stores the value v at a fresh location r — where the name ℓ is a compile-time abstraction of a set of run-time location names that includes r — with its prototype link set to the object location referred to by v'. Whereas `ref` allocates references to simple locations, `newobj` allocates references to object locations. Unlike simple locations, `deref` and `setref` cannot be used to manipulate object locations. Instead, the primitive dictionary and array functions `getPropObj`, `getIdxArr`, `getElem`, `setElem`, *etc.* are the only means for manipulating object locations; the semantics of these operations were discussed earlier. Third, the expression @x{ e } labels the enclosed expression e. Finally, the break expression `break` @x v terminates execution of the innermost expression labeled @x within the function currently being evaluated and produces the result v. If no such labeled expression is found, evaluation becomes stuck. Label and break expressions are included to translate the control flow operations of DJS.

The operational semantics of System DJS is based on λ_{JS} [51] — a λ-calculus extended with references, prototype-based records, and control-flow operators — with only minor differences. For example, in λ_{JS} each dictionary binds a reference to its prototype object in a distinguished field "`__proto__`". In System DJS, however, to facilitate heap unrolling in the type system, we add a second binding form (for object locations) as follows to make prototype links manifest in the syntax of run-time heaps:

$$\textbf{Run-Time Heaps} \quad \Delta \quad ::= \quad - \;\mid\; \Delta[r \mapsto v] \;\mid\; \Delta[r \mapsto \langle v, r' \rangle]$$

Furthermore, rather than including explicit syntactic forms for object operations as in λ_{JS}, we desugar these operations to calls to `getElem`, `setElem`, *etc.* Because the operational semantics of System DJS is mostly standard, we refer the reader to [51] for the full details.

Array Types. Compared to System !D, we add a new syntactic type term $Arr(T)$ to describe dictionaries indexed by integer keys, each of which binds either `undefined` or a value of type T.

Uninterpreted Heap Symbols. As in System !D, each strong heap h, of the form $H \oplus h'$, comprises a "shallow" part h' that describes the types of certain locations and a "deep" part H that summarizes all other locations. To describe invariants about the deep part of a heap, System DJS introduces two uninterpreted *heap symbols* that can appear in refinement formulas. The predicate $\mathit{HeapHas}(H,\ell,k)$ represents the fact that the chain of objects in H starting with ℓ has the key k. Similarly, the function symbol $\mathit{HeapSel}(H,\ell,k)$ refers to the value retrieved when looking up key k in the heap H starting with ℓ.

Syntactic Sugar. We introduced syntactic sugar in Figure 2.3 to write traditional record type syntax in terms of refinements. Because of inheritance in System DJS, it is now useful to define a variation that specifies the entire set of keys bound by a dictionary. We use the abbreviation

$$\mathit{Keys}(d,K) \;\doteq\; \wedge_{k\in K}\, \mathit{has}(d,k) \wedge (\forall k'.\ (\wedge_{k\in K}\ k \neq k') \Rightarrow \neg \mathit{has}(d,k'))$$

to capture the fact that *only* the keys K are bound in d, using a formula that falls into the decidable array property fragment [11]. We extend the syntactic sugar for record types to use this predicate, for example, as in the second abbreviation below:

$$\begin{array}{rcl} (\!|\{f\!:\!T_1;\ g\!:\!T_2\}|\!) & = & \{x \mid (\!|\mathit{Dict}(x)|\!) \wedge T_1(\mathit{sel}(x,\text{“}\mathtt{f}\text{”})) \wedge T_2(\mathit{sel}(x,\text{“}\mathtt{g}\text{”}))\} \\ (\!|\{f\!:\!T_1;\ g\!:\!T_2;\ \_\!:\!\mathit{Bot}\}|\!) & = & \{x \mid (\!|\{f\!:\!T_1;\ g\!:\!T_2\}|\!) \wedge \mathit{Keys}(x,\{\text{“}\mathtt{f}\text{”},\text{“}\mathtt{g}\text{”}\})\} \end{array}$$

6.3 Type Checking

We now turn our attention to the well-formedness, typing, and subtyping relations of System DJS. The type system reuses the System D subtyping algorithm to factor subtyping obligations between a first-order SMT solver and syntactic subtyping rules and reuses the System !D formulation of heaps to support strong updates. The novel technical developments here are: the use of uninterpreted heap symbols to regain precision in the presence of imperative, prototype-based objects; the encoding of array primitives to support idiomatic use of JavaScript arrays; and the use of refinement types to assign precise types to JavaScript operators. Type checking for System DJS is based heavily on System !D (§5.3), so our discussion in this section will highlight the differences.

We summarize the syntax of environments and typing judgements in Figure 6.5. The changes to type and heap environments in System DJS compared to System !D mirror the changes to strong and weak heap types. Expression typing in System DJS refers to a label environment Ω that records the world W that the expression labeled $@x$ is expected to satisfy; all other judgement forms remain the same as in System !D (summarized in Figure 5.4).

6.3.1 Well-Formedness

It is straightforward to extend System !D well-formedness to deal with the extensions to the syntax of types in System DJS, namely, object locations in strong and weak heaps, and heap predicates. We elide these definitions.

$$
\begin{array}{rlcl}
\textbf{Type Environments} & \Gamma & ::= & - \mid \Gamma, x{:}T \mid \Gamma, p \\
 & & \mid & \Gamma, A \mid \Gamma, M \mid \Gamma, H \\
 & & \mid & \Gamma, (\tilde{\ell} \mapsto T) \mid \Gamma, (\tilde{\ell} \mapsto \langle T, \ell \rangle) \\
\textbf{Heap Environments} & \Sigma & ::= & H \mid \Sigma \oplus (\ell \mapsto v) \mid \Sigma \oplus (\ell \mapsto \langle v, \ell' \rangle) \mid \Sigma \oplus (\tilde{\ell} \mapsto \theta) \\
\textbf{Label Environments} & \Omega & ::= & - \mid \Omega, @x{:}W
\end{array}
$$

$$\textbf{Expression Typing} \quad \Gamma;\ \Sigma;\ \Omega \vdash e :: S \,/\, \Sigma'$$

Figure 6.5. Syntax of System DJS Judgements

Subtyping (omitted) $\boxed{\Gamma \vdash T_1 \sqsubseteq T_2}$

Implication (omitted) $\boxed{\Gamma \Rightarrow p}$

Syntactic Subtyping (selected rules) $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{T_1 \equiv T_2}{\Gamma \vdash \mathit{Arr}(T_1) <: \mathit{Arr}(T_2)}\ [\text{U-Array}]$$

World Subtyping (omitted) $\boxed{\Gamma \vdash W_1 \sqsubseteq W_2;\ \pi}$

Heap Matching (omitted) $\boxed{h_1 \sim h_2;\ \pi}$

Figure 6.6. Subtyping for System DJS

6.3.2 Subtyping

We extend System !D subtyping with an array subtyping rule U-Array (Figure 6.6). Arrays are invariant in their type parameter (using type equality $\{x \,|\, p\} \equiv \{x \,|\, q\} \mathrel{\overset{\circ}{=}} \mathit{Valid}(p \Leftrightarrow q)$), but can be related with additional predicates. For example, $\{a \,|\, a :: \mathit{Arr}(\mathit{Num}) \wedge \mathit{packed}(a) \wedge \mathit{len}(a) = 2\}$ is a subtype of $\{a \,|\, a :: \mathit{Arr}(\mathit{Num})\}$.

6.3.3 Value Typing

The definition of the value typing judgement $\Gamma;\ \Sigma \vdash v :: T$ in System DJS (Figure 6.7) differs from System !D in two ways. First, although the T-Const rule remains the same, the set of constants c and their types $\mathit{ty}(c)$ have changed, as discussed in §6.1. In our implementation, the function $\mathit{ty}(c)$ is defined by the standard prelude files (`basics.djs`, `objects.djs`, and `prelude.js`). Second, because the expression typing judgement refers to a label environment, the

Value Typing (selected rules) $\boxed{\Gamma;\, \Sigma \vdash v :: T}$

$$\frac{}{\Gamma;\, \Sigma \vdash c :: ty(c)}\ [\text{T-Const}]$$

$$\frac{\begin{array}{c} U = \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \to W_2 \quad \Gamma \vdash U \quad \mathit{EnvOfWorld}(W_1) = (x{:}T_0,\ \overline{y}{:}\overline{T},\ \Sigma_1) \\ \Gamma_1 = \Gamma, \overline{A}, \overline{M}, \overline{H}, \Psi, x{:}T_0, \overline{y}{:}\overline{T} \quad \Gamma_1;\, \Sigma_1;\, - \vdash e :: S\,/\,\Sigma_2 \quad \Gamma_1 \vdash S\,/\,\Sigma_2 \models W_2;\ \pi \end{array}}{\Gamma;\, \Sigma \vdash \lambda x.e :: \{z \,|\, z :: U\}}\ [\text{T-Fun}]$$

World Satisfaction (omitted) $\boxed{\Gamma \vdash S\,/\,\Sigma \models W;\ \pi}$

Heap Environment Matching (omitted) $\boxed{\Sigma \sim h;\ \pi}$

Figure 6.7. Value Typing for System DJS

T-Fun rule type checks a function body with the empty label environment, because the semantics of break expressions prevents function boundaries from being crossed; all other premises of T-Fun are unchanged.

6.3.4 Expression Typing

In this section, we describe several noteworthy aspects of the System DJS expression typing judgement $\Gamma;\ \Sigma;\ \Omega \vdash e :: S\,/\,\Sigma'$, defined in Figure 6.8 and Figure 6.10.

If-Expressions. The T-If rule allows an arbitrary (non-boolean) value v as the guard for an if-expressions, and, hence, strengthens the type environment with $\mathit{truthy}(v)$ rather than $v = \texttt{true}$ (resp. $\mathit{falsy}(v)$ rather than $v = \texttt{false}$) when type checking the then-branch (resp. false-branch). The *Join* operation (not shown) for combining worlds is also modified to guard existentials with "truthiness" predicates, rather than boolean equality, in this way.

Labeled Expressions. Label environments Ω play a role when type checking break and label expressions; the typing rules for all other expression forms simply thread the label environment through their premises without modification. The T-Label rule for $@x\{\,e\,\}$ binds the label $@x$ to an expected world W in the label environment used to check e, and expects that all exit points of e produce a value and heap environment that satisfy the expected world. The exit points are all `break` $@x\ v$ expressions in e, as well as the "fall-through" of expression e for control flow paths that do not end with `break`; the T-Break rule handles the former cases, and the second and third premises of T-Label handle the latter. If all exit points satisfy the expected world, we use the *EnvOfWorld* procedure to convert the heap type into a heap environment, like in the T-App

Expression Typing (selected rules) $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash e\ ::\ S\,/\,\Sigma'}$

$$\frac{\Gamma;\ \Sigma \vdash v\ ::\ T}{\Gamma;\ \Sigma;\ \Omega \vdash v\ ::\ T\,/\,\Sigma}\ \text{[T-VAL]}$$

$$\frac{\Gamma;\ \Sigma \vdash v\ ::\ \mathit{Any} \qquad \Gamma, \mathit{truthy}(v);\ \Sigma;\ \Omega \vdash e_1\ ::\ S_1\,/\,\Sigma_1 \qquad \Gamma, \mathit{falsy}(v);\ \Sigma;\ \Omega \vdash e_2\ ::\ S_2\,/\,\Sigma_2}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ::\ \mathit{Join}(v, S_1\,/\,\Sigma_1, S_2\,/\,\Sigma_2)}\ \text{[T-IF]}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega, @x\!:\!W \vdash e\ ::\ S\,/\,\Sigma' \qquad \Gamma \vdash S\,/\,\Sigma' \models W;\ \pi \\ \mathit{EnvOfWorld}(W) = (x_0\!:\!T_0,\ \overline{y}\!:\!\overline{T},\ \Sigma'')\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash @x\texttt{\{}\,e\,\texttt{\}}\ ::\ \exists x_0\!:\!T_0.\ \exists\overline{y}\!:\!\overline{T}.\ \{z \mid z = x_0\}\,/\,\Sigma''}\ \text{[T-LABEL]}$$

$$\frac{\Omega(@x) = W \qquad \Gamma;\ \Sigma \vdash v\ ::\ T \qquad \Gamma \vdash T\,/\,\Sigma \models W;\ \pi}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{break}\ @x\ v\ ::\ \{x \mid \mathit{false}\}\,/\,\Sigma}\ \text{[T-BREAK]}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma \vdash v_1\ ::\ \{f \mid f :: \forall\overline{A},\overline{M},\overline{H}.\ \Psi \Rightarrow W_1 \to W_2\} \qquad \Gamma;\ \Sigma \vdash v_2\ ::\ T'' \\ \Gamma \vdash [\overline{T}/\overline{A}] \qquad \Gamma \vdash [\overline{m}/\overline{M}] \qquad \Gamma \vdash [\overline{h}/\overline{H}] \qquad W_2' = \mathit{Freshen}(W_2) \\ (\Psi', W_1', W_2'') = \mathit{Unroll}(\mathit{Inst}(\mathit{Inst}(\mathit{Inst}((\Psi, W_1, W_2'), \overline{A}, \overline{T}), \overline{M}, \overline{m}), \overline{H}, \overline{h})) \\ \Gamma \vdash \Psi' \qquad \Gamma \vdash W_1' \qquad \Gamma \vdash W_2'' \qquad \Psi' \subseteq \Gamma \qquad \Gamma \vdash T''\,/\,\Sigma \models W_1';\ \pi \\ W_1' = y\!:\!\_\,/\,\_ \qquad \pi' = \pi[v_2/y] \qquad W_2''' = \pi' W_2'' \qquad \mathit{EnvOfWorld}(W_2''') = (x_0\!:\!T_0',\ \overline{x}\!:\!\overline{T'},\ \Sigma_2)\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash [\overline{T},\overline{m},\overline{h}]\ v_1\ v_2\ ::\ \exists x_0\!:\!T_0'.\ \exists\overline{x}\!:\!\overline{T'}.\ \{z \mid z = x_0\}\,/\,\Sigma_2}\ \text{[T-APP]}$$

Figure 6.8. Expression Typing for System DJS

rule. Notice that T-BREAK derives the type $\{x \mid \mathit{false}\}$ because a `break` expression immediately completes the evaluation context, thus making the subsequent program point unreachable (*cf.* the `anotherNegate` example in §6.1.6).

Heap Unrolling. The T-APP rule for System DJS must do even more than the one in System !D due to the presence of the heap predicates $\mathit{HeapHas}(H,\ell,k)$ and $\mathit{HeapSel}(H,\ell,k)$ in formulas. Every premise of T-APP remains the same as in System !D except the one that instantiates type, location, and heap variables with the supplied parameters. As a result of the instantiations, occurrences of *HeapHas* and *HeapSel* may refer to arbitrary heap types h rather than just heap variables H, as required. These *pre-types* (and *pre-formulas*, *pre-heaps*, *etc.*) are expanded using the procedure *Unroll*, defined in Figure 6.9, that transitively follows prototype links in heap bindings, precisely matching the semantics of object key membership and lookup. Figure 6.9 shows only the rules for heap predicates; all other rules simply recursively traverse the syntax of types, formulas, heaps,

$$\mathit{Unroll}(\mathit{HeapHas}(H \oplus h, \ell, k)) \;=\; \mathit{UnrollHas}(H, h, \ell, k)$$

$$\mathit{UnrollHas}(H, h, \ell, k) \;=$$

$$\begin{cases} \mathit{has}(d,k) \vee \mathit{UnrollHas}(H,h,\ell',k) & \text{if } (\ell \mapsto \langle d\!:\!T, \ell' \rangle) \in h \\ \mathit{HeapHas}(H,\ell,k) & \text{else if } \ell \neq a_{root} \\ \mathit{false} & \text{else } (\textit{i.e. } \ell = a_{root}) \end{cases}$$

$$\mathit{Unroll}(\psi(\mathit{HeapSel}(H \oplus h, \ell, k))) \;=\; \mathit{UnrollSel}(\psi, H, h, \ell, k)$$

$$\mathit{UnrollSel}(\psi, H, h, \ell, k) \;=$$

$$\begin{cases} \mathit{if}\ \mathit{has}(d,k)\ \mathit{then}\ \psi(\mathit{sel}(d,k))\ \mathit{else}\ \mathit{UnrollSel}(\psi,H,h,\ell',k) & \text{if } (\ell \mapsto \langle d\!:\!T, \ell' \rangle) \in h \\ \psi(\mathit{HeapSel}(H,\ell,k)) & \text{else if } \ell \neq a_{root} \\ \psi(\texttt{undefined}) & \text{else } (\textit{i.e. } \ell = a_{root}) \end{cases}$$

Figure 6.9. Heap Unrolling

etc. We write the location constant a_{root} for the root of the prototype hierarchy. We use the notation $\psi(p)$ to refer to a *formula context* ψ, a formula with a hole, filled with p.

Object References. The rule T-NewObj stores the dictionary v_1 in the heap at location ℓ_1 along with a prototype link to the location ℓ_2 that v_2 refers to. The T-ThawObj and T-FreezeObj rules are analogs to T-Thaw and T-Freeze, inherited without change from System !D, to operate on object locations rather than simple locations. There are no rules for object locations analogous to T-Deref and T-SetRef, inherited from System !D. Instead, several primitive functions (`getElem`, `setElem`, *etc.*) facilitate reading from and writing to object locations.

Existentially-Quantified Locations. In System !D, we added universal quantification over location variables to the structure of function types. Being the only mechanism for location polymorphism in the system, this leads to function types with an excess of universally-quantified location variables. To see why, consider the following function that resembles the kind of System !D function that results from desugaring an identify function in JavaScript:

```
let id x =
  let _x = ref x in
  deref _x
```

System !D and its rule T-Ref requires that the syntax of `ref` mention some strong location ℓ, not allocated before, to use for this fresh reference; this location is omitted from the code listing above.

Expression Typing (selected rules) $\boxed{\Gamma;\, \Sigma;\, \Omega \vdash e :: S \,/\, \Sigma'}$

$$\frac{\ell \notin \mathit{dom}(\Sigma) \qquad \Gamma;\, \Sigma \vdash v :: T}{\Gamma;\, \Sigma \vdash \texttt{ref}\ \ell\ v :: \{x \,|\, x :: \mathit{Ref}\,\ell\} \,/\, \Sigma \oplus (\ell \mapsto v)}\ \text{[T-Ref]}$$

$$\frac{\begin{array}{c} L \text{ fresh} \qquad \Gamma_1 = \Gamma,\, L,\, x\!:\!(\{y \,|\, y :: \mathit{Ref}\,L\}) \qquad \Sigma_1 = \Sigma \oplus (L \mapsto v) \\ \Gamma;\, \Sigma \vdash v :: \mathit{Any} \qquad \Gamma_1;\, \Sigma_1;\, \Omega \vdash e :: S \,/\, \Sigma_2 \qquad \Sigma_2 \equiv \Sigma' \oplus (L \mapsto v') \\ S = \exists \overline{z}\!:\!\overline{T}.\, T' \qquad \Gamma, \overline{z}\!:\!\overline{T};\, \Sigma_2 \vdash v' :: T \qquad \mathit{CleanLoc}(x, L, T, S \,/\, \Sigma') = (S' \,/\, \Sigma'') \end{array}}{\Gamma;\, \Sigma;\, \Omega \vdash \texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ e :: S' \,/\, \Sigma''}\ \text{[T-LetRef]}$$

$$\frac{\Gamma;\, \Sigma \vdash (v_1,\, v_2) :: (\mathit{Dict},\, \{y \,|\, y :: \mathit{Ref}\,\ell_2\}) \qquad \Sigma \equiv \Sigma_0 \oplus (\ell_2 \mapsto \langle v', \ell_3 \rangle)}{\Gamma;\, \Sigma;\, \Omega \vdash \texttt{newobj}\ \ell_1\ v_1\ v_2 :: \{x \,|\, x :: \mathit{Ref}\,\ell_1\} \,/\, \Sigma \oplus (\ell_1 \mapsto \langle v_1, \ell_2 \rangle)}\ \text{[T-NewObj]}$$

$$\frac{\begin{array}{c} \Gamma;\, \Sigma \vdash v :: \{y \,|\, y :: \mathit{Ref}\,\ell\} \qquad \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \theta) \oplus (\ell \mapsto \langle v', \ell' \rangle) \\ \theta = \texttt{frzn} \text{ or } \theta = \texttt{thwd}\ \ell \qquad \Gamma(\tilde{\ell}) = \langle T, \ell' \rangle \qquad \Gamma;\, \Sigma \vdash v' :: T \end{array}}{\Gamma;\, \Sigma;\, \Omega \vdash \texttt{freeze}\ \tilde{\ell}\ \theta\ v :: \{x \,|\, x :: \mathit{Ref}\,\tilde{\ell}\} \,/\, \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn})}\ \text{[T-FreezeObj]}$$

$$\frac{\Gamma;\, \Sigma \vdash v :: \{z \,|\, z :: \mathit{Ref}\,\tilde{\ell}\} \qquad \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn}) \qquad \Gamma(\tilde{\ell}) = \langle T, \ell' \rangle}{\Gamma;\, \Sigma;\, \Omega \vdash \texttt{thaw}\ \ell\ v :: \exists x\!:\!T.\, \{y \,|\, y :: \mathit{Ref}\,\ell\} \,/\, \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{thwd}\ \ell) \oplus (\ell \mapsto \langle x, \ell' \rangle)}\ \text{[T-ThawObj]}$$

Figure 6.10. Expression Typing for System DJS (continued)

One choice is to pick some location constant a_x for `_x`, which results in the following type for `id`:

$$\texttt{id} :: x\!:\!\mathit{Any} \rightarrow \{y \,|\, y = x\} \,/\, (a_x \mapsto \{y \,|\, y = x\}) \qquad (6.1)$$

Notice that the output heap of the function *must* refer to a_x; heap subtyping in System !D and System DJS may not drop location bindings from strong heaps, which would allow the same location name to be used for multiple allocations and, thus, undermine the soundness of strong updates in the system.

Although the type above is correct, its use is rather limited: because it refers to a single location constant a_x, the function may not be called twice as the second call would try to allocate the location a_x, which would already have been allocated by the first call. The natural recourse for this situation is to quantify over the location instead:

$$\texttt{id} :: \forall L_x.\; x\!:\!\mathit{Any} \rightarrow \{y \,|\, y = x\} \,/\, (L_x \mapsto \{y \,|\, y = x\}) \qquad (6.2)$$

With this type, each call site would instantiate the location parameter L_x with a different location constant, allowing the function to be called arbitrarily many times.

Relying on universal quantification for every reference allocation is unfortunate, however, since there are many such allocations in programs that result from desugaring JavaScript. Even though many of these abstractions and instantiations can be mechanically inserted, we might like to extend the system with *existential locations* to avoid the hassle. The following rule would introduce a new strong location variable L for each reference allocation; notice that the syntax of the `ref` expression does not name a strong location:

$$\frac{\Gamma;\, \Sigma \vdash v \, :: \, T}{\Gamma;\, \Sigma;\, \Omega \vdash \texttt{ref}\ v \, :: \, \exists L.\, \{x \,|\, x :: \mathit{Ref}\ L\} \,/\, \Sigma \oplus (L \mapsto v)}\ \text{[T-}\textsc{Ref-Exists}\text{]}$$

We would then extend the syntax of function types to include a sequence of existentially-quantified strong locations in the output world for the reference cells allocated by the function body:

$$\forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \rightarrow \exists \overline{L}.\, W_2$$

It would be up to the function application rule, rather than the syntax of each function application expression, to instantiate each of these variables with fresh location constants. With this approach, we would write the following type:

$$\texttt{id} \, :: \, x \!:\! \mathit{Any} \rightarrow \exists L_x.\, \{y \,|\, y = x\} \,/\, (L_x \mapsto \{y \,|\, y = x\}) \tag{6.3}$$

This is better, but we would still rather that the location not be mentioned at all:

$$\texttt{id} \, :: \, x \!:\! \mathit{Any} \rightarrow \{y \,|\, y = x\} \tag{6.4}$$

But, as we said before, we cannot simply drop locations from heap types, because other types — like those for functions that close over these reference cells — may depend on these constraints. Consider the following example:

```
let makeCounter () =
  let _n = ref 0 in
  fun () -> setref _n (1 + deref _n); deref _n in
let next = makeCounter () in
assert (next () == 1);
assert (next () == 2);
```

Using existentials, we can give the following precise type:

$$\begin{aligned}\texttt{makeCounter} \, :: \, \mathit{Any} \rightarrow \exists L.\, &(\mathit{Any} \,/\, (L \mapsto j \!:\! \mathit{Int}) \rightarrow \{k \,|\, k = j + 1\} \,/\, (L \mapsto \{k \,|\, k = j + 1\}))\\ &/\ (L \mapsto \{i \,|\, i = 0\})\end{aligned} \tag{6.5}$$

This type records the fact that given any argument, `makeCounter` allocates some location L initialized to 0 and returns a function that, given any argument in an environment where the heap maps L to some integer j, returns the integer $j+1$ and modifies the heap such that L maps to $j+1$. (We can assign an equivalently precise type using universal quantification by moving the location variable to the input world as before.) In this case, it is clear that we cannot simply drop the existential location L because the type of the closure refers to it! However, instead, if we were to assign the less precise type

$$\begin{aligned}\texttt{makeCounter} \;::\; & \mathit{Any} \to \exists L.\,(\mathit{Any}\,/\,(L \mapsto \mathit{Int}) \to \mathit{Int}\,/\,(L \mapsto \mathit{Int})) \\ & \qquad\qquad /\ (L \mapsto \mathit{Int})\end{aligned} \tag{6.6}$$

then, as for the type of `id`, we would like to drop the existential location from the type completely and write the following:

$$\texttt{makeCounter} \;::\; \mathit{Any} \to \mathit{Any} \to \mathit{Int} \tag{6.7}$$

Dropping Locations. In System DJS, we provide a rule T-LetRef — in addition to the T-Ref inherited from System !D — that allows locations to be dropped from subsequent types and heaps. For the expression `let` $x =$ `ref` v `in` e, creates a "temporary" existential location L for the reference cell that is only in scope while checking the let-body e. After type checking e, T-LetRef drops all occurrences of L from the types and heaps if two conditions are met: (i) that location L stores a value v' that satisfies some type $T = \{x \mid p\}$, where the only variable p refers to is x; and (ii) that any heap bindings of L in the derived type for e are syntactically of the form $(L \mapsto T)$ where there is *no* binder to name the value. These two conditions are sufficient to ensure that values stored at L will *always* satisfy T, and that no other type will depend on a *particular* value stored at L. Hence, the location need not be specifically tracked any longer. These restrictions are stronger than absolutely necessary to justify that the location L be forgotten, but they seem to be a reasonable heuristic for common cases (like the types for `id` and `makeCounter` above) where we want specifications that do not require the full expressive power of dependent types and existential locations. The T-LetRef uses the procedure *CleanLoc* to check the two conditions above and remove any occurrences of L from the derived type and heap; we elide its definition.

Because System DJS inherits from System !D the ability to use universal quantification for very precise specifications, in System DJS we include only the limited form of existential quantification that T-LetRef provides, rather than also including T-Ref-Exists and its associated changes as discussed earlier.

6.3.5 Type Soundness

Many standard stuck states are not stuck in JavaScript: a function can be applied with any number of arguments; an operator can be used with any values because of implicit coercion;

and, all property lookups succeed (possibly producing `undefined`). Nonetheless, several (non-exceptional) stuck states remain, including applying a non-function value and retrieving a property for a non-object value. System DJS is designed to ensure that well-typed programs do not get stuck and can only fail with exceptions due to retrieving a property from `undefined` or `null`. We can also provide the stronger guarantees that only bound keys are retrieved and only non-`null` objects are accessed (thus ruling out the possibility of `null` dereference exceptions) simply by changing the types of object primitives appropriately. As with System !D, however, we do not prove progress and preservation theorems for System DJS. Many of the new techniques in System DJS beyond those in System !D do not introduce any new features to the refinement logic. Proving the soundness of the T-LetRef rule for dropping locations in certain situations, however, will likely require an imperative, higher-order program logic such as Hoare Type Theory [73].

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript (Technical Appendix).** `arXiv:1112.4106v3 [cs.PL]`, August 2012.

- Ravi Chugh and Ranjit Jhala. **Dependent Types for JavaScript.** `arXiv:1112.4106v1 [cs.PL]`, December 2011.

Chapter 7

Dependent JavaScript

The techniques we developed in the previous chapters have culminated in System DJS, a refinement type system that can reason about mutable, prototype-based objects with dynamically computed keys, untagged unions and control flow, and the specific details of JavaScript arrays. As such, we are now ready to define Dependent JavaScript, a statically typed dialect of a large JavaScript subset. First, we define the explicitly typed syntax of DJS along with its semantics via *desugaring* rules that translate DJS expressions to System DJS. Next, we describe several optimizations in our implementation to help reduce the annotation burden and improve the performance of the type checker. Then, we describe an evaluation of DJS on a series of small but challenging benchmarks. We conclude with a discussion of some limitations and features not currently supported in DJS.

7.1 Desugaring DJS to System DJS

In this section, we present the explicitly typed abstract syntax of DJS along with desugaring rules $[\![\,\texttt{e}\,]\!] = e$ that translate DJS expressions `e` to System DJS expressions e. Most of the desugaring rules borrow from the translation of JavaScript to λ_{JS} in [51], so we limit our discussion to the aspects most relevant to DJS; we refer the reader their work for more details. Our desugaring rules produce System DJS expressions that are not in A-normal form; our implementation converts desugared expressions into ANF before type checking. We use annotated let-expressions $\texttt{let}\ x : T_1 = e_1\ \texttt{in}\ e_2$ in System DJS to declare the expected type T_1 for e_1 in order to help facilitate bidirectional type checking, in the style described in Chapter 3. We define the System DJS ascription and assertion commands as follows:

$$e\ \texttt{as}\ T \ \mathrel{\circeq}\ \texttt{let}\ x : T = e\ \texttt{in}\ x$$

$$\texttt{assert}\ e\ T \ \mathrel{\circeq}\ \texttt{let}\ x = e\ \texttt{in}\ \texttt{let}\ \_ = x\ \texttt{as}\ T\ \texttt{in}\ x$$

We use the metavariable $I ::= [\overline{T},\overline{m},\overline{h}]$ to range over instantiation parameters for function application. Most instantiation parameters are inferred by the type checker (§7.2).

Constants. Desugaring constants is straightforward. For example:

$$\begin{array}{rcl}
\llbracket\ \texttt{typeof e}\ \rrbracket & = & \texttt{tagof}\ \llbracket\texttt{e}\rrbracket \\
\llbracket\ \texttt{!e}\ \rrbracket & = & \texttt{not}\ \llbracket\texttt{e}\rrbracket \\
\llbracket\ \texttt{e}_1\ \texttt{+}\ \texttt{e}_2\ \rrbracket & = & \llbracket\texttt{e}_1\rrbracket\ \texttt{+}\ \llbracket\texttt{e}_2\rrbracket \\
\llbracket\ \texttt{"September"}\ \rrbracket & = & \texttt{"September"} \\
\llbracket\ \texttt{2013}\ \rrbracket & = & \texttt{2013}
\end{array}$$

Variables. JavaScript variables are mutable, so they desugar to explicit references in System DJS. In the DS-LetRef rule below, we implicitly assume that $\texttt{e}_1$ is not a function value, a case which subsequent desugaring rules handle.

$$\begin{array}{rcll}
\llbracket\ \texttt{x}\ \rrbracket & = & \texttt{deref \_x} & \text{[DS-Deref]} \\
\llbracket\ \texttt{e}_1 = \texttt{e}_2\ \rrbracket & = & \texttt{setref}\ \llbracket\texttt{e}_1\rrbracket\ \llbracket\texttt{e}_2\rrbracket & \text{[DS-SetRef]} \\
\llbracket\ \texttt{var x} = \texttt{e}_1\texttt{;}\ \texttt{e}_2\ \rrbracket & = & \texttt{let \_x} = \texttt{ref}\ \llbracket\texttt{e}_1\rrbracket\ \texttt{in}\ \llbracket\texttt{e}_2\rrbracket & \text{[DS-LetRef]}
\end{array}$$

Sequencing and Conditionals. These are straightforward:

$$\begin{array}{rcll}
\llbracket\ \texttt{e}_1\texttt{;}\ \texttt{e}_2\ \rrbracket & = & \texttt{let \_} = \llbracket\texttt{e}_1\rrbracket\ \texttt{in}\ \llbracket\texttt{e}_2\rrbracket & \text{[DS-Seq]} \\
\llbracket\ \texttt{if}\ (\texttt{e}_1)\ \{\ \texttt{e}_2\ \}\ \texttt{else}\ \{\ \texttt{e}_3\ \}\ \rrbracket & = & \texttt{if}\ \llbracket\texttt{e}_1\rrbracket\ \texttt{then}\ \llbracket\texttt{e}_2\rrbracket\ \texttt{else}\ \llbracket\texttt{e}_3\rrbracket & \text{[DS-If]}
\end{array}$$

Objects. Object and array operations desugar to the primitive functions `hasElem`, `getElem`, `setElem`, and `delElem` described in §6.1. For desugaring object and array literals, we write

$$\mathit{pro}(e) \mathrel{\overset{\circ}{=}} \texttt{getPropObj}\ (\llbracket\texttt{e}\rrbracket,\ \texttt{"prototype"})$$

to set the prototypes of fresh object and array literals, translated to `newobj` which creates values with prototype links. Our implementation inserts a fresh location for object and array literals if none is provided.

$$\begin{array}{rcll}
\llbracket\ /{*}:I\,{*}/\ \texttt{e}_2\ \texttt{in}\ \texttt{e}_1\ \rrbracket & = & /{*}:I\,{*}/\ \texttt{hasElem}\ (\llbracket\texttt{e}_1\rrbracket,\ \llbracket\texttt{e}_2\rrbracket) & \text{[DS-HasElem]} \\
\llbracket\ /{*}:I\,{*}/\ \texttt{e}_1[\texttt{e}_2]\ \rrbracket & = & /{*}:I\,{*}/\ \texttt{getElem}\ (\llbracket\texttt{e}_1\rrbracket,\ \llbracket\texttt{e}_2\rrbracket) & \text{[DS-GetElem]} \\
\llbracket\ /{*}:I\,{*}/\ \texttt{e}_1[\texttt{e}_2] = \texttt{e}_3\ \rrbracket & = & /{*}:I\,{*}/\ \texttt{setElem}\ (\llbracket\texttt{e}_1\rrbracket,\ \llbracket\texttt{e}_2\rrbracket,\ \llbracket\texttt{e}_3\rrbracket) & \text{[DS-SetElem]} \\
\llbracket\ /{*}:I\,{*}/\ \texttt{delete}\ \texttt{e}_1[\texttt{e}_2]\ \rrbracket & = & /{*}:I\,{*}/\ \texttt{delElem}\ (\llbracket\texttt{e}_1\rrbracket,\ \llbracket\texttt{e}_2\rrbracket) & \text{[DS-DelElem]} \\
\llbracket\ /{*}:\ell\,{*}/\ \{\ \overline{\texttt{e}_\texttt{k}}\texttt{:}\overline{\texttt{e}_\texttt{v}}\ \}\ \rrbracket & = & \texttt{newobj}\ \ell\ \{\llbracket\texttt{e}_{\texttt{k}0}\rrbracket = \llbracket\texttt{e}_{\texttt{v}0}\rrbracket;\ \dots\ \}\ (\mathit{pro}(\texttt{Object})) & \text{[DS-ObjLit]} \\
\llbracket\ /{*}:\ell\,{*}/\ [\ \overline{\texttt{e}}\ ]\ \rrbracket & = & \texttt{newobj}\ \ell\ \{\texttt{"0"} = \llbracket\texttt{e}_0\rrbracket;\ \dots\ \}\ (\mathit{pro}(\texttt{Array})) & \text{[DS-ArrLit]}
\end{array}$$

Functions. To desugar a function value, we create mutable reference cells for all of its parameters, as well as an explicit `this` parameter, and wrap the entire (desugared) function body expression with the label *@return*. In DJS, as opposed to JavaScript, we choose to distinguish between recursive and non-recursive functions by the presence or absence of the name `f` in the syntax of the function value (just before the argument list). For simplicity, unlike in JavaScript, we do not package all arguments into an `arguments` array, but this would be easy to incorporate if desired.

$$\begin{aligned}
&\llbracket\ \texttt{function}\ (\texttt{x}_1,\ \ldots,\ \texttt{x}_n)\ \{\ \texttt{e}\ \}\ \rrbracket\ = && \text{[DS-F{\small UNCTION}]}\\
&\quad \lambda(\texttt{this},\ \texttt{x}_1,\ \ldots,\ \texttt{x}_n).\\
&\qquad \texttt{let}\ (\texttt{\_this},\ \texttt{\_x}_1,\ \ldots,\ \texttt{\_x}_n) = (\texttt{ref this},\ \texttt{ref}\ \texttt{x}_1,\ \ldots,\ \texttt{ref}\ \texttt{x}_n)\ \texttt{in}\\
&\qquad @\mathit{return}\{\ \llbracket \texttt{e} \rrbracket\ \}\\
&\llbracket\ /\!*\!:\texttt{f} :: T\ *\!/\ \texttt{var f}\ =\ \texttt{function}\ (\texttt{x}_1,\ \ldots,\ \texttt{x}_n)\ \{\ \texttt{e}\ \};\ \texttt{e}'\ \rrbracket\ = && \text{[DS-L{\small ET}F{\small UNC}]}\\
&\quad \texttt{let \_f} = \texttt{ref}\ (\llbracket \texttt{function}(\texttt{x}_1,\ldots,\texttt{x}_n)\{\texttt{e}\} \rrbracket\ \texttt{as}\ T)\ \texttt{in}\ \llbracket \texttt{e}' \rrbracket\\
&\llbracket\ /\!*\!:\texttt{f} :: T\ *\!/\ \texttt{var f}\ =\ \texttt{function f}(\texttt{x}_1,\ \ldots,\ \texttt{x}_n)\ \{\ \texttt{e}\ \};\ \texttt{e}'\ \rrbracket\ = && \text{[DS-L{\small ET}F{\small IX}F{\small UNC}]}\\
&\quad \texttt{let \_f} = \texttt{ref}\ ([T]\ \texttt{fix}\ (\lambda \texttt{f}.\ \llbracket \texttt{function}(\texttt{x}_1,\ldots,\texttt{x}_n)\{\texttt{e}\} \rrbracket)\ \texttt{in}\ \llbracket \texttt{e}' \rrbracket
\end{aligned}$$

The following two rules handle `return` and `this` according to the desugaring of functions:

$$\begin{aligned}
\llbracket\ \texttt{return e}\ \rrbracket\ &=\ \texttt{break}\ @\mathit{return}\ \llbracket \texttt{e} \rrbracket && \text{[DS-R{\small ETURN}]}\\
\llbracket\ \texttt{this}\ \rrbracket\ &=\ \texttt{deref \_this} && \text{[DS-T{\small HIS}]}
\end{aligned}$$

Function Calls. JavaScript has several different syntactic forms for function calls. The following two rules handle *direct calls* and *method calls*:

$$\begin{aligned}
\llbracket\ /\!*\!: I\ *\!/\ \texttt{e}(\texttt{e}_1,\ \ldots,\ \texttt{e}_n)\ \rrbracket\ &=\ /\!*\!: I\ *\!/\ \llbracket \texttt{e} \rrbracket\ (\texttt{null},\ \llbracket \texttt{e}_1 \rrbracket,\ \ldots,\ \llbracket \texttt{e}_n \rrbracket) && \text{[DS-D{\small IRECT}C{\small ALL}]}\\
\llbracket\ /\!*\!: I\ *\!/\ \texttt{e}[\texttt{e}'](\texttt{e}_1,\ \ldots,\ \texttt{e}_n)\ \rrbracket\ &=\ \texttt{let}\ \mathit{obj} = \llbracket \texttt{e} \rrbracket\ \texttt{in} && \text{[DS-M{\small ETHOD}C{\small ALL}]}\\
&\ \ \texttt{let}\ f = \texttt{getElem}\ (\mathit{obj},\ \llbracket \texttt{e}' \rrbracket)\ \texttt{in}\\
&\ \ /\!*\!: I\ *\!/\ f\ (\mathit{obj},\ \llbracket \texttt{e}_1 \rrbracket,\ \ldots,\ \llbracket \texttt{e}_n \rrbracket)
\end{aligned}$$

In DS-FuncCall, we supply `null` as the receiver rather than the "global object" `window` as in JavaScript, which is a source of subtle programming errors.

In JavaScript, all functions are desugared to objects. In DJS, we distinguish between *scalar* functions that desugar to scalar values, as above, and *constructor* functions that desugar to objects, as we will discuss shortly. The primary benefit of non-constructor functions as objects in JavaScript is that they inherit from `Function.prototype`, which stores two native functions `apply` and `call` that allow the caller to explicitly supply the receiver argument. We do not provide general support `apply` and `call` in DJS, because they require mechanisms beyond the scope of our (already-large) type system System DJS; for example, the latter accepts an arbitrary number of arguments. However, to support the recursive function idioms described in §6.1.5, we do provide limited support for `apply` that *syntactically* looks for "`apply`" and explicitly sets the receiver:

$$\llbracket\ /\!*\!: I\ *\!/\ \texttt{e.apply}(\texttt{e}_1,\ \ldots,\ \texttt{e}_n)\ \rrbracket\ =\ /\!*\!: I\ *\!/\ \llbracket \texttt{e} \rrbracket\ (\llbracket \texttt{e}_1 \rrbracket,\ \llbracket \texttt{e}_2 \rrbracket,\ \ldots,\ \llbracket \texttt{e}_n \rrbracket) \qquad \text{[DS-A{\small PPLY}C{\small ALL}]}$$

Because we do not desugar scalar functions to objects, there is no danger that the `apply` function can be "hijacked" by overwriting the "`apply`" property, since an attempt to write to any field of a scalar function is rejected by the System DJS type system.

Constructors. In JavaScript, any function can be used as a constructor, but sometimes the lack of distinction can lead to programming errors. In DJS, we use the token `ctor` in a function annotation to declare its intent as a constructor, and we desugar the function accordingly.

$$
\begin{array}{l}
[\![\ \texttt{/*:ctor F}::T\ \texttt{*/ function F}\ (\texttt{x}_1,\ \ldots,\ \texttt{x}_\texttt{n})\ \{\ \texttt{e}\ \};\ \texttt{e}'\ ]\!]\ = \qquad \text{[DS-LetConstructor]} \\
\qquad \texttt{let}\ \mathit{ctor} = [\![\texttt{function}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})\{\texttt{e}\}]\!]\ \texttt{as}\ \mathit{EmptyThis}(T)\ \texttt{in} \\
\qquad \texttt{let}\ \mathit{proto} = [\![\texttt{/*:}\,a_{\texttt{FProto}}\ \texttt{*/}\ \{\}]\!]\ \texttt{in} \\
\qquad \texttt{let}\ \mathit{foo} = \{\ \text{“}\texttt{\_\_code\_\_}\text{”} = \mathit{ctor};\ \text{“}\texttt{prototype}\text{”} = \mathit{proto}\ \}\ \texttt{in} \\
\qquad \texttt{let \_F} = \texttt{newobj}\ a_\texttt{F}\ \mathit{foo}\ (\mathit{pro}(\texttt{Function}))\ \texttt{in}\ [\![\texttt{e}']\!]
\end{array}
$$

In particular, the desugared System DJS expression is an object, allocated at the predictable location a_F that can be mentioned in subsequent source-level annotations, with two fields: a distinguished “`__code__`” field that stores the actual function value, desugared as before; and a “`prototype`” field initialized to an empty object at the predictable location $a_{\texttt{FProto}}$. As we will see next, constructor functions are *always* called with an empty object as the first parameter. To avoid this boilerplate in the types of constructor functions, *EmptyThis*(T) augments the user-supplied annotation T with this requirement. For example:

$$
\begin{array}{ll}
& \mathit{EmptyThis}((\mathit{this}\!:\!\mathit{Ref},\ T_1) \rightarrow \mathit{Ref}\ \mathit{this}\,/\,(\mathit{this} \mapsto \langle T_2, \dot{\mathit{this}}\rangle)) \\
\mathrel{\hat{=}} & (\mathit{this}\!:\!\mathit{Ref},\ T_1)\,/\,(\mathit{this} \mapsto \langle\{d \mid d = \mathit{empty}\}, \dot{\mathit{this}}\rangle) \rightarrow \mathit{Ref}\ \mathit{this}\,/\,(\mathit{this} \mapsto \langle T_2, \dot{\mathit{this}}\rangle)
\end{array}
$$

For calls to constructors, the DS-New rule creates a fresh object whose prototype is set to the object in the constructor object’s “`prototype`” field (which may or may not be the initial object created when the constructor function was defined), and calls the function in the “`__code__`” field to finish the initialization.

$$
\begin{array}{lcl}
[\![\ \texttt{new /*:}\,\ell\ I\ \texttt{*/ e}(\texttt{e}_1,\ \ldots,\ \texttt{e}_\texttt{n})\ ]\!] & = & \texttt{let}\ \mathit{foo} = [\![\texttt{e}]\!]\ \texttt{in} \qquad \text{[DS-NewCall]} \\
& & \texttt{let}\ \mathit{ctor} = \texttt{getElem}\ (\mathit{foo},\ \text{“}\texttt{\_\_code\_\_}\text{”})\ \texttt{in} \\
& & \texttt{let}\ \mathit{proto} = \texttt{getElem}\ (\mathit{foo},\ \text{“}\texttt{prototype}\text{”})\ \texttt{in} \\
& & \texttt{let}\ \mathit{obj} = \texttt{newobj}\ \ell\ \{\}\ \mathit{proto}\ \texttt{in} \\
& & \texttt{/*:}\,I\ \texttt{*/}\ \mathit{ctor}\ (\mathit{obj},\ [\![\texttt{e}_1]\!],\ \ldots,\ [\![\texttt{e}_\texttt{n}]\!])
\end{array}
$$

Loops. We translate iteration in DJS to recursion in System DJS. Because the “arguments” and “return values” of loops are trivial, we require that loop (function) annotations T provide only input and output heaps. In particular, loop annotations are of the following form which are expanded to ordinary function types:

$$h_1 \rightarrow h_2 \mathrel{\hat{=}} \forall H.\ (\mathit{Any})\,/\,H \oplus h_1 \rightarrow \mathit{Any}\,/\,H \oplus h_2$$

The desugaring of loops to recursive functions below is standard. In our presentation, we omit the facility for desugaring JavaScript `break` and `continue` expressions using label and break expression in System DJS; see [51] for this technique.

$$\begin{array}{ll} [\![\ /\!*\!: T *\!/\ \texttt{while}\ (\texttt{e}_{\texttt{cond}})\ \{\ \texttt{e}_{\texttt{body}}\ \}\ ]\!]\ = & \text{[DS-WHILE]} \\ \quad \texttt{letrec}\ \mathit{loop}\ :\ T = \lambda\_. & \\ \quad\quad \texttt{if}\ [\![\texttt{e}_{\texttt{cond}}]\!]\ \texttt{then}\ ([\![\texttt{e}_{\texttt{body}}]\!];\ \mathit{loop}\ \text{“}\texttt{unit}\text{”})\ \texttt{else}\ \texttt{undefined} & \\ \quad \texttt{in}\ \mathit{loop}\ \text{“}\texttt{unit}\text{”} & \\ [\![\ /\!*\!: T *\!/\ \texttt{for}\ (\texttt{e}_{\texttt{init}};\ \texttt{e}_{\texttt{cond}};\ \texttt{e}_{\texttt{incr}})\ \{\ \texttt{e}_{\texttt{body}}\ \}\ ]\!]\ = & \text{[DS-FOR]} \\ \quad \texttt{let}\ \_ =\ [\![\texttt{e}_{\texttt{init}}]\!]\ \texttt{in} & \\ \quad \texttt{letrec}\ \mathit{loop}\ :\ T = \lambda\_. & \\ \quad\quad \texttt{if}\ [\![\texttt{e}_{\texttt{cond}}]\!]\ \texttt{then}\ ([\![\texttt{e}_{\texttt{body}}]\!];\ [\![\texttt{e}_{\texttt{incr}}]\!];\ \mathit{loop}\ \text{“}\texttt{unit}\text{”})\ \texttt{else}\ \texttt{undefined} & \\ \quad \texttt{in}\ \mathit{loop}\ \text{“}\texttt{unit}\text{”} & \end{array}$$

Thaw and Freeze. These are straightforward:

$$\begin{array}{rcll} [\![\ /\!*\!: \texttt{thaw}\ \ell\ \texttt{x} *\!/\ ]\!] & = & \texttt{setref \_x}\ (\texttt{thaw}\ \ell\ (\texttt{deref \_x}\ )) & \text{[DS-THAW]} \\ [\![\ /\!*\!: \texttt{freeze}\ \tilde{}\ell\ \theta\ \texttt{x} *\!/\ ]\!] & = & \texttt{setref \_x}\ (\texttt{freeze}\ \tilde{}\ell\ \theta\ (\texttt{deref \_x}\ )) & \text{[DS-FREEZE]} \end{array}$$

Assertions. These are straightforward:

$$\begin{array}{rcll} [\![\ \texttt{assert(e)}\ ]\!] & = & \texttt{assert}\ [\![\texttt{e}]\!]\ \{x \mid x = \texttt{true}\} & \text{[DS-ASSERTTRUE]} \\ [\![\ \texttt{assert(}/\!*\!: T *\!/\ \texttt{e)}\ ]\!] & = & \texttt{assert}\ [\![\texttt{e}]\!]\ T & \text{[DS-ASSERTTYPE]} \end{array}$$

7.2 Implementation

We have implemented a desugarer and type checker for DJS [80] in OCaml, borrowing and extending the λ_{JS} [51] JavaScript parser and desugarer, and using the Z3 SMT solver [27] to discharge logical validity queries. We specify System DJS primitive functions in the files `basics.djs` and `objects.djs`, and JavaScript built-in functions like `Object.prototype.hasOwnProperty` in `prelude.js` (desugared to `prelude.djs`). These three files comprise a standard prelude included with every desugared DJS program for type checking. Next, we will discuss several optimizations that we have implemented to reduce the annotation burden on DJS source programs and to improve the performance of the System DJS type checker.

7.2.1 Improving Performance

Our DJS implementation extends the bidirectional type checking approach described in Chapter 3, requiring explicit function type annotations but inferring types for many intermediate

B^*	$::=$	$\mathit{Int} \mid \mathit{Num} \mid \mathit{Str} \mid \mathit{Bool}$	**Base Types**
T^*	$::=$		**Sugared Refinement Types**
	$\mid$	$\{x \mid p\}$	general type
	$\mid$	$\{x\!:\!(U^*) \mid p\}$	stylized type
U^*	$::=$		**Sugared Syntactic Types**
	$\mid$	$U_1 \cap \cdots \cap U_n$	intersection of syntactic type terms
	$\mid$	$B_1^* \cup \cdots \cup B_n^*$	union of base types
	$\mid$	$(T_1^*, \cdots, T_n^*)$	tuple type
	$\mid$	$\{f_1\!:\!T_1^*;\ \cdots;\ f_n\!:\!T_n^*\}$	record type (unknown domain)
	$\mid$	$\{f_1\!:\!T_1^*;\ \cdots;\ f_n\!:\!T_n^*;\ \_\!:\!\mathit{Bot}\}$	record type (known domain)
	$\mid$	$T^*?$	option type
	$\mid$	Null	null type

Figure 7.1. Sugared Types for DJS

expressions. In addition, we employ several optimizations to reduce the number of validity queries made to the SMT solver and, hence, reduce the running time of the type checker.

Fast Path. A key aspect of nested refinements is the ability to describe all values, base values as well as more complex ones like functions and dictionaries, using refinement predicates. With System DJS, we have shown how to build on this general mechanism to specify and reason about properties of programming patterns in dynamic languages. But although the expressiveness from describing all values with predicates is crucial, there are often specific patterns where syntactic handling of types are sufficient for verifying safety.

We implement a "fast path" in our type checker that tracks some information about types of values syntactically, resorting to the refinement-type based mechanisms we have discussed for the general case. Figure 7.1 shows the syntax of types that our implementation tracks, splitting types into general refinement types $\{x \mid p\}$, corresponding to the System DJS refinement types we have discussed, and more specific, stylized, syntactic types $\{x\!:\!(U^*) \mid p\}$, which keep some information about the type manifest with syntax rather than only logically with predicates. Syntactic types U^* include intersection, union, tuple, record, and option types and are used to help optimize some of the typing rules. For example, to implement the System DJS rule T-Deref (replicated below) in the algorithmic style described in Chapter 3, the type checker must extract a type term for the value v being dereferenced, an operation that requires possibly-many SMT queries. To avoid this expense, our implementation first attempts to derive a type for the dereference expression using the following rule TS-Deref*:

$$\frac{\Gamma;\,\Sigma \vdash v \;::\; \{y \,|\, y :: \mathit{Ref}\,\ell\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v')}{\Gamma;\,\Sigma;\,\Omega \vdash \texttt{deref}\ v \;::\; \{x \,|\, x = v'\} \,/\, \Sigma}\ \text{[T-Deref]} \qquad \frac{\Gamma;\,\Sigma \vdash v \rhd \{y\!:\!(\mathit{Ref}\,\ell) \,|\, p\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v')}{\Gamma;\,\Sigma;\,\Omega \vdash \texttt{deref}\ v \;\rhd\; \{x \,|\, x = v'\} \,/\, \Sigma}\ \text{[TS-Deref*]}$$

The first premise synthesizes a stylized reference type for v that syntactically mentions the location ℓ to be found in Σ, thus avoiding the need to query the SMT solver. If the expression cannot be type checked using this optimized rule, (an algorithmic version of) the general rule T-Deref is used instead. Similarly, we add optimized versions of other rules that manipulate syntactic types, for example, the T-App rule which requires the first argument to be a function type.

We also use sugared syntactic types to optimize subtyping queries before resorting to the general rules. For example, the optimized rules for base types and record types below resemble the traditional syntactic subtyping rules described in §1.2.1:

$$\frac{\Gamma \vdash U_1^* <: U_2^*}{\Gamma \vdash \{x\!:\!(U_1^*) \,|\, p\} \sqsubseteq \{y\!:\!(U_2^*) \,|\, \mathit{true}\}}\ \text{[SA-Sugar-Fast*]} \qquad \frac{}{\Gamma \vdash \mathit{Null} <: T^*?}\ \text{[SA-Null*]}$$

$$\frac{\exists i.\ B_i^* = \mathit{Int} \text{ or } B_i^* = \mathit{Num}}{\Gamma \vdash \cup_i\, B_i^* <: \mathit{Num}}\ \text{[SA-Num*]} \qquad \frac{\forall j.\ \exists i.\ f_{1i} = f_{2j} \text{ and } \Gamma \vdash T_{1i}^* \sqsubseteq T_{2j}^*}{\Gamma \vdash \{\overline{f_1}\!:\!\overline{T_1^*}\} <: \{\overline{f_2}\!:\!\overline{T_2^*}\}}\ \text{[SA-Recd*]}$$

When falling back to the general case with the rule below, stylized types are converted into refinement types with straightforward embeddings into the logic (for a reminder, see Figure 2.2, Figure 2.3, and the end of §6.2).

$$\frac{\Gamma \vdash \{x \,|\, (\!| U_1^* |\!)(x) \wedge p\} \sqsubseteq \{y \,|\, (\!| U_2^* |\!)(y) \wedge q\}}{\Gamma \vdash \{x\!:\!(U_1^*) \,|\, p\} \sqsubseteq \{y\!:\!(U_2^*) \,|\, q\}}\ \text{[SA-Sugar-Slow*]}$$

Selfification. At odds with the fast path rules above is the T-Var rule (replicated below) which assigns the "selfified" type $\{y \,|\, y = x\}$ to a variable x that is bound in the environment. Although this precision is crucial, the equality stymies the optimized handling of types because it "hides" the syntactic structure of types. Accordingly, we replace T-Var with a new rule TS-Var*

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma;\,\Sigma \vdash x \;::\; \{y \,|\, y = x\}}\ \text{[T-Var]} \qquad \frac{x \in \mathit{dom}(\Gamma)}{\Gamma;\,\Sigma \vdash x \rhd \mathit{Selfify}(\Gamma(x), x)}\ \text{[TS-Var*]}$$

where the helper procedure *Selfify* keeps syntactic type information, if any, intact:

$$\begin{aligned} \mathit{Selfify}(\{y \,|\, p\}, w) \;&\mathrel{\mathring{=}}\; \{y \,|\, y = w\} \\ \mathit{Selfify}(\{y\!:\!(U^*) \,|\, p\}, w) \;&\mathrel{\mathring{=}}\; \{y\!:\!(U^*) \,|\, y = w\} \end{aligned}$$

Notice that selfified types are still exact but also retain syntactic sugar. This strategy also applies to primitives whose output types involve equality predicates. For example, the following optimized rule uses sugared types to optimize the key lookup for an object (*i.e.* a dictionary on the heap):

$$\frac{\Gamma;\, \Sigma \vdash v \,\rhd\, \{y\,{:}\,(\mathit{Ref}\;\ell) \mid p\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto \langle d, \ell' \rangle) \qquad \Gamma(d) = \{y\,{:}\,(\{\text{“}\mathtt{f}\text{”}\,{:}\,T^*;\ \cdots\ \}) \mid q\}}{\Gamma;\, \Sigma;\, \Omega \vdash \mathtt{getElem}\ (v,\ \text{“}\mathtt{f}\text{”}) \,\rhd\, \mathit{Selfify}(T^*, \mathit{sel}(d, \text{“}\mathtt{f}\text{”}))\,/\,\Sigma}\ \text{[TS-GetElem-1*]}$$

The output type may be stylized, but it is still exact because it contains the predicate $\mathit{sel}(d, \text{“}\mathtt{f}\text{”})$.

Arrays and Safe Strings. In §6.1.3, we discussed how arrays in JavaScript inherit from arbitrary prototype chains, just like non-array objects. To help recover the notion of integer-indexed arrays in DJS, we introduced the notion of safe strings, which do not coerce to integers, and required that all non-array objects bind only safe string keys.

Instead, in our implementation, we restrict arrays so that they must have the particular prototype location $a_{\mathit{ArrayProto}}$, which is the initial object location allocated for `Array.prototype` in our prelude, rather than an arbitrary location. Furthermore, we require that this location always store the value $d_{\mathit{ArrayProto}}$, which is the dictionary value initialized by the prelude. For example, for the `getIdxArr` primitive, rather than the first type below (replicated from Figure 6.2), we assign the second type instead:

$$\forall A.\ (x\,{:}\,\mathit{Ref},\ i\,{:}\,\mathit{Int})\,/\,(x \mapsto \langle a\,{:}\,\mathit{Arr}(A), \dot{x} \rangle) \rightarrow y\,{:}\,\mathit{Any}\{\mathit{arrGetIdx}(y,a,i,A)\}\,/\,\mathit{same}$$

$$\begin{aligned}&\forall A.\ (x\,{:}\,\mathit{Ref},\ i\,{:}\,\mathit{Int})\,/\,(x \mapsto \langle a\,{:}\,\mathit{Arr}(A), a_{\mathit{ArrayProto}} \rangle) \oplus (a_{\mathit{ArrayProto}} \mapsto \langle \{d \mid d = d_{\mathit{ArrayProto}}\}, a_{\mathit{root}} \rangle)\\ &\qquad \rightarrow y\,{:}\,\mathit{Any}\{\mathit{arrGetIdx}(y,a,i,A)\}\,/\,\mathit{same}\end{aligned}$$

These restrictions prohibit user programs from extending or overwriting `Array.prototype`, which is useful in many situations. But in return, we avoid the need to reason about the safety of strings because these restrictions ensure that the prototype chain of every array, which binds only safe strings, is known.

Incremental Environments. As mentioned in Chapter 3, our implementation maintains environments of logical assumptions incrementally (using a backtracking facility in SMT solvers like Z3) to minimize the cost of embedding type environments as formulas.

7.2.2 Reducing the Annotation Burden

Our bidirectional type checker requires type annotations for functions but infers types for many intermediate expressions. Nevertheless, in DJS, function types can be quite large, and several expression forms require explicit type parameters. We incorporate several techniques in our implementation to reduce the annotation burden. These techniques supplement the many notational abbreviations we have employed throughout this dissertation for refinement types (many of which are summarized in Figure 2.2 and Figure 2.3) and for polymorphic, flow-sensitive function types (described in §5.1.3 and §6.1.2).

Polymorphic Instantiation Inference. The syntax of function application $[\overline{T},\overline{m},\overline{h}]\ v_1\ v_2$ requires type, location, and heap parameters to instantiate the universally-quantified variables of the function type $\forall \overline{A},\overline{M},\overline{H}.\ \Psi \Rightarrow W_1 \rightarrow W_2$ of v_1. Type and location parameters are inferred using the standard "greedy" approach employed in many local type inference algorithms [78], where we syntactically pattern-match type and location variables against the (stylized) type of the argument v_2. We also look for type and location arguments in the current heap environment to match corresponding type and location variables in the input heap type of a function. When type and location variables appear only in the output world of the function type, we are unable to infer instantiations; in these cases, the syntax of the expression must explicitly provide instantiations. In our benchmarks, we are able to omit most type and location arguments.

To infer heap arguments to instantiate the deep part H of an input heap type $H \oplus h$, we collect each heap location ℓ that is not mentioned in the shallow part h and infer the binding $(\ell \mapsto \{x \,|\, x = y\})$ — or $(\ell \mapsto \langle \{x \,|\, x = y\}, \ell' \rangle)$ if ℓ is an object location with prototype ℓ' — where y is the value stored at ℓ in the current heap environment Σ at the call-site. This singleton type encodes the fact that the location is unmodified by the function call, matching the intuition that the deep part of the heap is irrelevant to the function. (To facilitate our handling of sugared types, the inferred heap binding is actually $(\ell \mapsto \mathit{Selfify}(\Gamma(y), y))$, where Γ is the type environment at the call-site.) In our benchmarks, we are able to omit all heap arguments.

Location Invariants. If a function refers to a mutable variable from an outer scope, its heap type must explicitly list its location and type. As a result, function types can be verbose. For example, consider the following function and (one possible) verifiable annotation, assuming that the desugarer uses the predictable location a_{pi} for the mutable variable `pi` of type $\mathit{Ref}\ a_{pi}$:

```
var pi = 3.14, e = 2.718;

//: getPi :: () / (a_pi ↦ n:{x | x = 3.14}) → {x | x = 3.14} / same
var getPi = function () { return pi; };
```

Without the binding for a_{pi}, the function body would not be allowed to dereference `pi`; the location for `e` can be omitted from the type since it is not dereferenced. Clearly, writing annotations like this is quite unpleasant.

To ease this burden, we (transitively) collect the free variables x for each function definition and, for each location a_x that is missing from its type annotation, automatically add a heap binding for a_x, depending on which of the following three ways the variable is declared:

```
var a = ...;
var b /*: T */ = ...;
var c /*: Ref T */ = ...;
```

- If the variable x is declared without any annotation (as a is), the binding $(a_x \mapsto \{y \,|\, y = v\})$ is added to both the input and output heap types, where v is the current value of a_x in the heap environment at the location of the function definition. That is, the lack of annotation an annotation on x signifies that any inner function that refers to x (*i.e.* that ”closes over x”) can only be called when x stores v and must not modify x (or if it does, it must restore v before returning).

- If the variable x is declared with an annotation T (as b is), the binding $(a_x \mapsto y\!:\!T)$ is added to the input heap and $(a_x \mapsto y'\!:\!T)$ is added to the output heap. That is, any inner function that refers to x can only be called when x stores a value of type T and must maintain this invariant, although the particular value stored may be different.

- If the variable x is declared with an annotation *Ref* T (as c is), the binding $(\ell \mapsto \langle y\!:\!T, \ell' \rangle)$ is added to the input heap and $(\ell \mapsto \langle y'\!:\!T, \ell' \rangle)$ is added to the output heap, where ℓ is the location of the reference value (*i.e.* object or array) initialized on the right-hand side of the declaration and ℓ' is its prototype. This kind of annotated declaration is the analog to the previous one for variables that store reference values instead of scalar values.

There are several aspects worth noting. First, this is a purely syntactic strategy that simply adds heap bindings to the types of subsequent closures, and these bindings are not trusted; nothing is said about whether these annotations will actually be satisfied during type checking. Second, these annotations have no impact on what values can be stored in variables except at the boundaries of functions that refer to them. If desired, it would be simple to extend the type checker to, furthermore, require that variables always satisfy their location invariants. Finally, due to the presence of higher-order functions, the set of free variables transitively accessed by a function is not always syntactically manifest, in which case our simple heuristics fail to automatically insert corresponding heap bindings.

As an example of how location invariants for variable declarations reduce the size of function type annotations, consider the following:

```
var pi = 3.14, e = 2.718;

//: getPi :: () → {x | x = 3.14}
var getPi = function () { return pi; };
```

Because pi is unannotated, the function type for getPi is augmented to require that pi stores 3.14 at both function boundaries, as in the initial version above. If pi is annotated with *Num*, then the following less precise type can be verified:

```
var pi /*: Num */ = 3.14, e = 2.718;

//: getPi :: () → Num
var getPi = function () { return pi; };
```

In this case, the return type $\{x \mid x = 3.14\}$ *cannot* be verified, because the (augmented) input heap says only that a_{pi} stores a number. In the following, the location invariant allows the closure incN to modify obj, as long as it maintains the invariant that the "n" key stores an integer:

```
var obj /*: Ref {n:Int} */ = {"n": 0};

//: incN :: () → Any
var incN = function () { obj.n = 1 + obj.n; };
```

The three ways above to choose location invariants apply only to variable declarations. Because objects cross function boundaries, we also allow heap bindings to mention location invariants. For example, the heap binding $(\ell \mapsto \langle x\!:\!T, \ell' \rangle\ T')$ is like the usual one augmented with the invariant T' to be automatically inserted at subsequent function boundaries. This heap binding is analogous to the variable location invariant for reference values.

Because loops desugar to functions that close over variables, the location invariant mechanism significantly reduces the annotation burden for loop boundaries.

Location invariants are also useful for improving the performance of type checking if-expressions, which normally use the procedure *Join* to compute the "exact join" of two branches (§6.3.4). This strategy introduces disjunctions into the antecedents of validity queries, which can be slow to answer since the SMT solver must carry out expensive case-splits. When this precision is not needed, the user can choose a mode that computes "inexact joins" for if-expressions, where the value of each variable x on either branch is required to satisfy the declared location invariant for x, but the exact values are not propagated.

Thaw and Freeze. System DJS requires that all weak references be thawed before use. Often times, however, accesses to a single weak object do not need to be correlated, so having to insert unnecessary thaw and freeze operations can be onerous. To reduce this burden, we include several specialized rules that allow weak objects to be used directly. For example, the following rule supports key lookup on a weak object:

$$\frac{\begin{array}{c}\Gamma;\ \Sigma \vdash v \ \rhd\ \{y\!:\!(\mathit{Ref}\ \tilde{\ell}) \mid p\} \\ \mathit{QuickThaw}(\Gamma,\Sigma,v,\tilde{\ell}) = (x\!:\!T_1^*, y, \ell, \Sigma_1) \\ \Gamma_1,\ x\!:\!T_1^*,\ y\!:\!\mathit{Ref}\ \ell;\ \Sigma_1;\ \Omega \vdash \texttt{getElem}\ (y,\ \text{“}\texttt{f}\text{”}) \ \rhd\ T_2^* \,/\, \Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{getElem}\ (v,\ \text{“}\texttt{f}\text{”}) \ \rhd\ \exists x\!:\!T_1^*.\ \exists y\!:\!\mathit{Ref}\ \ell.\ T_2^* \,/\, \Sigma}\ \text{[TS-GetElem-Weak-1*]}$$

Notice that the object is a reference to a weak location $\tilde{\ell}$, but the type for `getElem` requires a strong reference. This rule uses *QuickThaw* (not shown) to “inline” the effect of a `thaw` expression by checking that $\tilde{\ell}$ is frozen in Σ, creating a new thawed strong location ℓ, pointed to by y, that stores a value x of the weak location type T_1^* in a new heap environment Σ_1. The rule then derives a type for key lookup on this temporarily-thawed object. We include several other rules that wrap object operations with implicit `thaw` and `freeze` operations in a similar manner.

Although useful, the approach above does not work when successive accesses to a weak object need to be related with the precision of refinement types (as in the `sumWeights` example from §6.1.5). Extending thaw and freeze inference to surround larger sequences of expressions would, in future work, help to further reduce the annotation burden in these situations.

Untampered Natives. We augment the input and output heap of every function type with the binding $(a_{ArrayProto} \mapsto \langle\{d \mid d = d_{ArrayProto}\}, a_{root}\rangle)$ to reduce the annotation burden given the restriction we impose on arrays, described earlier. Since many user programs will not need to modify `Object.prototype`, by default, we insert a similar binding for the initial location $a_{ObjectProto}$ and value $d_{ObjectProto}$ of `Object.prototype` from the prelude. The user can override the insertion of this binding by manually specifying a binding for $a_{ObjectProto}$ in the heap types of an arrow.

7.3 Experiments

To demonstrate the expressiveness of DJS, we have annotated and type checked several small examples from various sources: *JavaScript: The Good Parts* [23] a popular reference book for JavaScript programmers; the SunSpider benchmark [93]; the V8 [49] benchmark; the Google Closure Library [48]; Google Gadgets examples studied earlier in the literature [52]; and the IBEX project [50] for writing secure browser extensions with refinement types. These examples exercise a variety of invariants, besides those demonstrated by previous examples (*e.g.* `negate`, `sumWeights`, *etc.*). We also ported the `counter` and `dispatch` examples from System D (Chapter 2) to DJS to demonstrate the nesting of function types inside objects with dynamic keys.

Figure 7.2 summarizes our results. For each example: “Un” is the number of (non-whitespace, non-comment) lines of code in the unannotated benchmark; “Ann” is the lines of code in the annotated DJS version (including comments because they contain annotations); “Time” is the running time, rounded to the nearest second, on a 2.66GHz machine with 4GB of RAM running Ubuntu; and “Queries” is the number of validity queries issued to Z3 during type checking.

Adapted Benchmark	**Un**	**Ann**	**Queries**	**Time**
JavaScript: The Good Parts				
`prototypal`	18	23	39	0.1
`pseudoclassical`	15	21	47	0.3
`functional`	19	37	46	0.4
`parts`	11	18	28	0.2
SunSpider				
`string-fasta`	10	17	51	0.5
`access-binary-trees`	34	37	96	1.7
`access-nbody`	129	160	648	7.0
V8				
`splay`	17	25	19	0.1
Google Closure Library				
`typeOf`	15	20	52	0.2
Google Gadgets				
`morse`	403	412	4014	4.1
`resistor`	600	615	8551	15.1
IBEX				
`delicious`	17	18	6	0.1
`facepalm`	73	80	95	0.3
`gmail-plus`	20	25	61	0.2
`hover-magnifier`	24	31	64	0.8
`js-tools`	13	15	4	0.1
`print-new-yorker`	19	24	36	0.1
`nit-twit`	86	114	193	1.4
`typograf`	47	52	29	0.2
`untiny`	22	24	5	0.1
Other				
`negate`	9	9	6	0.1
`sumWeights`	9	15	45	0.1
`counter`	16	18	24	0.1
`dispatch`	4	8	7	0.1
`initArray`	7	13	51	0.2
Totals	1637	1831	14217	33.6

Un: LOC without annotations; Ann: LOC with annotations;
Queries: Number of Z3 queries; Time: Running time in seconds

Figure 7.2. DJS Benchmarks

In the rest of this section, we highlight some of the features of DJS that our benchmarks leverage; our discussion is organized by the source of benchmarks.

JavaScript: The Good Parts. Besides the prototypal pattern discussed in §6.1.2, Crockford [23] presents three additional inheritance patterns using JavaScript's construction mechanism. Each of these examples relies on the support for imperative, prototype-based objects in DJS. The following example, `parts.js`, demonstrates construction "by parts" where functions extend arbitrary object arguments with methods:

```
//: make_dog :: (x:Ref) / (x ↦ ⟨d:Dict, ẋ⟩)
//:   → Any / (x ↦ ⟨d′:Dict{EqMod(d′,d,{“bark”}) ∧ sel(d′,“bark”) :: (this:Any) → Str}, ẋ⟩)
var make_dog = function (x) {
  x.bark = function () /*: (this:Any) → Str */ { return "bark"; };
};

//: make_cat :: (x:Ref) / (x ↦ ⟨d:Dict, ẋ⟩)
//:   → Any / (x ↦ ⟨d′:Dict{EqMod(d′,d,{“purr”}) ∧ sel(d′,“purr”) :: (this:Any) → Str}, ẋ⟩)
var make_cat = function (x) {
  x.purr = function () /*: (this:Any) → Str */ { return "purr"; };
};

var hybrid = {};
make_dog(hybrid);
make_cat(hybrid);
var noise = hybrid.bark() + hybrid.purr();
assert (typeof noise == "string");
```

The annotated line count reported for `parts.js` in Figure 7.2 is higher than what appears in this listing, because the concrete syntax parsed by our tool is not as compact as the typesetting here. Notice that there are no annotations besides function types, as the bidirectional type checker is able to infer all polymorphic instantiations. In future work, certain patterns of code like this can be handled specially to remove the need to annotate the inner functions, as their type annotations appear elsewhere.

SunSpider. The `makeCumulative` function in `string-fasta.js` iterates over an object with an unknown number of keys that all store integers, and sums them in place within the object. While iterating over the keys of the object, the function uses a variable to store the key from the previous iteration, a subtle invariant that DJS is able to express by describing the heap before and after each iteration. Compared to the original version, we allow the bindings to store arbitrary values and use a tag-test to sum only the integer bindings. To specify the original version requires universally quantified formulas, which DJS avoids to retain decidable type checking.

Our largest example adapted from SunSpider is `access-nbody.js`, which defines a constructor function NBodySystem that creates a container object to store an array of Body objects. The prototypes of both constructors are augmented with methods, and the thaw/freeze mechanism is heavily used while iterating over the array of Body objects to read and write their fields.

V8. The splay.js benchmark defines the following interesting tree node constructor (annotations omitted). Rather than initializing each "own" object with null left and right subtrees, the constructor's prototype object stores the defaults.

```
function Node(k,v) { this.k = k; this.v = v; }
Node.prototype.left = null;
Node.prototype.right = null;
```

After construction, however, Nodes are often extended with explicit subtrees. Using the flexibility of refinements, we assign each Node x a type with the predicate $\mathit{has}(x,\text{“}\texttt{left}\text{”}) \Rightarrow \mathit{sel}(x,\text{“}\texttt{left}\text{”}) = \texttt{null} \vee \mathit{sel}(x,\text{“}\texttt{left}\text{”}) :: \mathit{Ref}\ \tilde{}L$, where $\tilde{}L$ is a weak location that describes Nodes, to ensure that retrieving the "left" key produces another (possibly-null) Node regardless of whether it is stored on the object or not (and similarly for "right").

Google Closure Library. The behavior of the typeOf function is like the typeof operator except that it returns the more informative result "null" for null and "array" for arrays; the operator returns "object" in both cases. The type specification for typeOf depends on the ability to express intersections of function types in DJS, and verifying it requires control-flow tracking in the presence of mutation as well as a precise specification for the native (ES5) function Array.isArray, which we model in prelude.js.

Google Gadgets. Strobe [52] is a non-dependent type system for λ_{JS} that supports untagged unions and strong updates for scalar values within (but not across) function boundaries, but does not reason about prototype inheritance or the pecularities of JavaScript arrays. Regardless, they are able to rule out useful classes of run-time errors from their benchmarks, including some from the Google Gadgets [47] collection.

We adapted two of these examples, morse.js and resistor.js, to DJS. These two are our largest benchmarks and, compared to most others, the annotation overhead is very small. These examples consist of many function definitions that operate over a few kinds of objects defined early in the programs. As a result, using the location invariant mechanism for variable declarations and other syntactic sugar, the annotations are rather lightweight. Regarding performance, these examples make heavy use of nested loops and conditionals. Although our efforts to improve performance (*e.g.* inexact joins based on location invariants) have already reduced the time to type check these examples, there are additional syntactic patterns of programs which need to be optimized in future work.

IBEX. The authors of the IBEX project [50] advocate writing browser extensions in Fine [94], a refinement type system for ML, that compile to JavaScript after verification [40]. By using uninterpreted predicates to encode security concerns for the browser extension architecture, refinement type checking ensures the absence of certain security-relevant run-time errors. They develop several small example browser extensions as a proof of concept.

One hindrance to the potential adoption of their approach is that their system requires writing programs in the functional language ML rather than JavaScript, which is the foundation of web development. Since DJS is a refinement type system for JavaScript, which can use uninterpreted predicates to encode security and other domain-specific properties, we ported several examples from IBEX to DJS. We found it straightforward to port these examples, though admittedly, the functional programming style of those examples are likely easier to port than the imperative styles of full, real-world extensions. Regardless, the combination of refinement types for security and refinement types for JavaScript is a promising direction for future work.

Using nested refinements, we were able to cleanly encode an idiom we encountered while porting these examples. The IBEX browser extension platform provides a function of the form

$$\texttt{getEltsByTagName\_1} :: \forall L.\ (\cdots) \rightarrow \mathit{Ref}\ L \,/\, (L \mapsto \langle \mathit{Arr}(\{x \mid p \wedge q\}), a_{\mathit{ArrayProto}} \rangle)$$

that returns a new array on the heap. The particular argument and return types are not important for this discussion, just that the elements of the array satisfy predicates p and q. The trouble is that some clients of this function require only that the output array have type $\mathit{Arr}(\{x \mid p\})$ and wish to use the array in a contexts which expect this type. Unfortunately, because array subtyping is invariant (defined in Figure 6.6), $\mathit{Arr}(\{x \mid p \wedge q\})$ is not a subtype of $\mathit{Arr}(\{x \mid p\})$. As one option for recourse, the library writer could provide a second version of the function with the same run-time semantics but a different compile-time specification:

$$\texttt{getEltsByTagName\_2} :: \forall L.\ (\cdots) \rightarrow \mathit{Ref}\ L \,/\, (L \mapsto \langle \mathit{Arr}(\{x \mid p\}), a_{\mathit{ArrayProto}} \rangle)$$

The client can then choose which version of the function to call. Of course, it is undesirable to duplicate functionality for all combinations of choices. Instead of duplication, we can factor the choice of specification into a *single* type using nested refinements as follows:

$$\texttt{getEltsByTagName} :: \forall A, L.\ (\cdots) \rightarrow \mathit{Ref}\ L \,/\, (L \mapsto \langle \mathit{Arr}(\{x \mid p \wedge (0 :: A \Rightarrow q)\}), a_{\mathit{ArrayProto}} \rangle)$$

The type variable A acts as a “guard” to the predicate q that some clients wish to have but others do not. The has-type predicate $0 :: A$ allows the caller of the function to choose whether or not to include the predicate q by instantiating A with, for example, $\{x \mid \mathit{true}\}$ to “include” it or $\{x \mid \mathit{false}\}$ to “exclude” it. In particular, the following two subtyping relationships are satisfied:

$$Arr(\{x \mid p \wedge (true \Rightarrow q)\}) \;<:\; Arr(\{x \mid p \wedge q\}) \quad \text{because } p \wedge (true \Rightarrow q) \Leftrightarrow p \wedge q$$

$$Arr(\{x \mid p \wedge (false \Rightarrow q)\}) \;<:\; Arr(\{x \mid p\}) \quad \text{because } p \wedge (false \Rightarrow q) \Leftrightarrow p$$

As a result, the single specification for `getEltsByTagName` captures both of the specific choices earlier. Note that the particular has-type predicate used to guard q is irrelevant; all that is needed is some predicate that allows instantiation to result in *true* and *false*.

Other. The last example we discuss is `initArray.js`. Although we could verify a less precise type for the function below with far fewer annotations, we elect to illustrate the expressive power of DJS. Given an integer n, `initArray` returns a packed array of integers, allocated at the location L, of length n if n is non-negative; otherwise, the length is zero.

```
1  //: pred $Goal(y,m)$ = $y :: Arr(Int) \wedge packed(y) \wedge if\ m < 0\ then\ len(y) = 0\ else\ len(y) = m$
2  //: initArray :: $\forall L.\ (n{:}Int) \rightarrow Ref\ L\,/\,(L \mapsto \langle\{x \mid Goal(x,n)\}, a_{ArrayProto}\rangle)$
3  var initArray = function (n) {
4    var arr = /*: L */ [];
5
6    //: $(a_i \mapsto j{:}Int\{j \geq 0 \wedge (n \geq 0 \Rightarrow j \leq n)\}) \oplus (L \mapsto \langle\{x_j \mid Goal(x_j,j)\}, a_{ArrayProto}\rangle)$
7    //:   $\rightarrow$ $(a_i \mapsto sameType) \oplus (L \mapsto \langle\{x_{exit} \mid Goal(x_{exit},n)\}, a_{ArrayProto}\rangle)$
8    for (var i = 0; i < n; i++) {
9      arr[i] = i;
10   }
11   return arr;
12 };
```

The annotation on line 4 allocates an empty array at L, the variable quantified by the type of the function. Because `arr` is declared without a location invariant, the heap types in the subsequent loop annotation are augmented with bindings for the location a_{arr} (corresponding to the variable `arr`) which constrain the reference value of type $Ref\ L$ to be unmodified (*i.e.* points to this same array throughout the loop). The input heap annotation on line 6 describes two requirements that must hold before each iteration: (a) that the location a_i (corresponding to the variable `i`) stores a positive integer j that is no greater than n if n is non-negative; and (b) that the value x_j at location L is a packed array of integers of length j. The output heap annotation on line 7 declares two properties that should hold after the loop exits: (a) that a_i stores an integer j' (possibly different from j) that satisfies the same type as in the input heap — the syntactic sugar *sameType* allows the programmer to avoid duplicating the corresponding type from the input heap; and (b) that the value x_{exit} stored at L is a packed array of integers of length n. DJS verifies these two properties directly as a result of desugaring the annotated loop to an annotated recursive function. The output heap type of the loop ensures that L satisfies the output heap type of `initArray`.

Summary of Evaluation. We have demonstrated that our type system, System DJS, is expressive enough to support the invariants from a series of small but varied examples drawn from existing JavaScript benchmarks. We have found that the full range of features in DJS are indeed required, but that many examples fall into patterns that do not simultaneously exercise all features. The average annotation overhead for the entire benchmark suite is 12% (the annotated programs are approximately 1.12 times as large as the unannotated ones), but the overhead is quite large for many complicated examples. For example, the average annotation overhead for `morse.js` and `resistor.js` alone is only 2% but the average for all others is 27%. Regarding performance, the running time of our type checker is acceptable for small examples, but less so as the number of queries to the SMT solver increases. That said, our preliminary efforts to reduce the annotation burden and improve performance (described in §7.2) have already made a large impact on our benchmarks. Therefore, we believe that future work on desugaring and on type checking can treat common cases specially in order to further reduce the annotation burden and running time, and fall back to the full expressiveness of the system when necessary.

7.4 Limitations and Future Work

In the past three chapters, we have shown how to scale up the technique of nested refinements of System D — a type system for dynamic languages in a functional setting — to a more full-featured language like JavaScript — with imperative updates, prototype-based objects, and arrays — through a combination of strong updates, prototype chain unrolling, and other encodings. We believe that Dependent JavaScript is the most promising approach, to date, for supporting real-world dynamic languages like JavaScript. DJS already supports a large subset of JavaScript that can be used for projects where all the code is controlled (*e.g.* server-side applications), and future work on integrating with run-time environments could allow DJS code to run alongside full untyped JavaScript. In the rest of this section, we identify several current limitations of DJS.

Recursion Through Mutable Variables. To define a recursive function in DJS, we require that the function value be named (by providing an identifier just before the argument list, which is optional in JavaScript). For example, the following is the factorial function in DJS; we use the JavaScript ternary conditional expression e_1 ? e_2 : e_3 for concision:

```
//: fact :: (Int) → Int
var fact = function fact(n) { return (n <= 0) ? 1 : (n * fact(n-1)); };
```

In idiomatic JavaScript, on the other hand, the name `fact` can be omitted from the function value, because by the time the function body is called and evaluated, the variable `fact` will store the function value being defined, thus capturing the recursion. Regardless, it is a rather small burden that in DJS the function must be named in order to reason about the recursion.

The more significant issue is that our handling of recursive definitions prohibits common patterns of mutually recursive functions. For example, consider the following functions that test the parity of their arguments:

```
var isEven, isOdd;
isEven = function (n) {
  return (n == 0) ? true : ((n < 0) ? false : isOdd(n-1));
};
isOdd = function (n) {
  return (n == 1) ? true : ((n < 1) ? false : isEven(n-1));
};
```

Each variable initially stores the dummy `undefined` value and is then strongly updated with a function value that calls the other. By the time either function is called, both variables have been strongly updated, so the mutual recursion evaluates safely at run-time. Given the support for strong updates in DJS, we might expect to reason about these functions with the following types, where the location constants a_{isEven} and a_{isOdd} correspond to the mutable variables `isEven` and `isOdd`, respectively:

$$\texttt{isOdd} :: (\mathit{Int}) \,/\, (a_{\mathit{isEven}} \mapsto (\mathit{Int}) \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool} \,/\, \mathit{same}$$
$$\texttt{isEven} :: (\mathit{Int}) \,/\, (a_{\mathit{isOdd}} \mapsto (\mathit{Int}) \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool} \,/\, \mathit{same}$$

DJS can, indeed, verify that these specifications are satisfied, but it cannot type check a *call* to either function. For example, to verify a call to `isOdd`, the problem is that the type assigned to `isEven` is *not* a subtype of $(\mathit{Int}) \rightarrow \mathit{Bool}$ as required by the type of `isOdd`; the former requires that a_{isOdd} stores a function of a particular type, but the latter requires that the function make no assumptions about the input heap. That DJS cannot verify the safety of calls to these functions based on the above specifications is reassuring, because they do not capture the fact that either of the variables could be overwritten later, thus breaking the mutual recursion.

The lack of support for mutually recursive functions is a significant limitation. In future work, DJS can be extended with *open scoping* [16], where we would assign the following types:

$$\texttt{isOdd} :: (a_{\mathit{isEven}} \mapsto (\mathit{Int}) \rightarrow \mathit{Bool}) \Rrightarrow (\mathit{Int}) \rightarrow \mathit{Bool}$$
$$\texttt{isEven} :: (a_{\mathit{isOdd}} \mapsto (\mathit{Int}) \rightarrow \mathit{Bool}) \Rrightarrow (\mathit{Int}) \rightarrow \mathit{Bool}$$

Each function type declares its requirements on the heap (factored to the left of the $\Rrightarrow$ symbol), and the crucial mechanism in open scoping is that all of the requirements are checked for mutual consistency at every function *call* rather than once at each function *definition*.

Recursion Through the Heap. Support for recursion through the heap in DJS is currently limited. Consider the following, where the mutual recursion is resolved through the heap via `this`:

```
var obj = {};
obj.isEven = function (n) {
  return (n == 0) ? true : ((n < 0) ? false : this.isOdd(n-1));
};
obj.isOdd = function (n) {
  return (n == 1) ? true : ((n < 1) ? false : this.isEven(n-1));
};
```

Trying to assign types in the style for `isEven` and `isOdd` above, again, fails to suffice for checking calls to either function. However, because weak locations allow for self-reference, the following weak location can be used to describe the object:

$$\mathit{EvenOdd}[\tilde{L}] \;\doteq\; \{\text{``}\texttt{isOdd}\text{''}\!:\!(\mathit{this}\!:\!\mathit{Ref}\ \tilde{L},\ \mathit{Num}) \rightarrow \mathit{Bool};\ \text{``}\texttt{isEven}\text{''}\!:\!(\mathit{this}\!:\!\mathit{Ref}\ \tilde{L},\ \mathit{Num}) \rightarrow \mathit{Bool}\}$$

Although this workaround is sufficient for certain situations, the imprecision of weak locations is not always acceptable, so mutual recursion between strong objects on the heap is important to add in future work.

The open scoping approach can be extended to deal with recursion through "own" objects on the heap, but a primary challenge will be reasoning about open recursion through the heap in our flow-sensitive, dependent setting. A solution is likely to involve *refined heaps* where predicates constrain heap locations, analogous to how refinement types use predicates to constrain values.

Integration with Contracts. In Chapter 1, we described how the hybrid typing approach integrates run-time contract checking for properties that are beyond the reach of static reasoning and for code that is unavailable at compile-time (*e.g.* because of dynamic code loading with `eval`). Future work on contracts will need to integrate with the combination of nested refinements, flow-sensitive heaps, and prototype reasoning that we have introduced in DJS. Flow sensitivity provides a natural path for integrating with dynamically-loaded code, since this facility can be used to specify the pre- and post-conditions required at a particular call to `eval`.

Features Currently Unsupported. In addition to general use of `apply` and `call` as discussed in §7.1, there are several other JavaScript constructs that are currently unsupported in DJS:

- To allow mutation of prototype links via the non-standard "`__proto__`" property, we could add a `setproto` expression to the language and detect cycles during heap unrolling.
- The `eval` statement allows a string to be parsed and executed, which is useful but dangerous if misused. Since DJS is flow-sensitive, we can constraint `eval` with heap invariants before and after the statement, and then perform *staged type checking* in the style of [17] at run-time.

- ES5 introduces optional per-object and per-property attributes (for example, to prevent modifications or deletions) that can likely be incorporated into our encoding of dictionaries. One benefit of such an extension is that the type system could reason more precisely about which objects are in a prototype chain. For example, we could then allow non-array objects to bind unsafe strings as long as we prevent them from appearing in the prototype chain of arrays, thus weakening the distinction we impose between array and non-array objects (§6.1.3). A second benefit is that native objects could be marked as unmodifiable, statically enforcing the pattern they are usually "untampered" as discussed in §7.2.

- ES5 getters and setters interpose on object reads and writes. Since this is a deep change to the semantics of object operations (invoking arbitrary functions), adding general support for these will likely be heavyweight. Interestingly, one can think of our treatment of the special array "`length`" property (§6.1.3) as a built-in getter/setter.

- Each function has an implicit `arguments` array that binds all parameters supplied by the caller, regardless of how many formals the function defines. Current ES6 proposals include a modified version, where an explicit parameter can bind a variable number of arguments beyond those named by formals, similar in style to Python.

- The `x instanceof Foo` operator checks whether or not `Foo.prototype` is somewhere along the prototype chain of `x`. We could add a primitive to match these semantics.

- Scalar values can be explicitly coerced by wrapper functions, such as `Boolean`, in addition to the implicit coercion we have discussed.

Undesirable Features. The last three features we discuss regularly compete for the title of worst among several "warts" in the language (*e.g.* [23]) that lead to confusing code and hard-to-detect bugs. Incidentally, the λ_{JS} translations of all three are straightforward and can be supported in DJS, but we see no reason to given their demerits.

- The `with` statement adds the fields of an object to the current scope of a block, allowing them to be accessed without qualification. There is hardly a good reason to use this feature, and it is banned in ES5 "strict" mode.

- All `var` declarations are implicitly lifted to the top of the enclosing function, resulting in "function scope" rather than lexical scope. Although simple to detect when `var`-lifting kicks in, we opt for the latter. ES6 will likely add an explicit `let` binding form that is not subject to lifting. In DJS, `var` is essentially the new `let` form, but we stick with the traditional syntax for familiarity.

- For a "method call" `x.f(y)`, the receiver `x` is supplied for the `this` argument to the function, but for a "direct call" `x(y)`, JavaScript implicitly supplies the global object for `this`, masking common errors. We choose to statically reject direct calls to functions that require a `this` parameter.

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.
- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript (Technical Appendix).** `arXiv:1112.4106v3 [cs.PL]`, August 2012.
- Ravi Chugh and Ranjit Jhala. **Dependent Types for JavaScript.** `arXiv:1112.4106v1 [cs.PL]`, December 2011.

Part IV

Epilogue

Chapter 8

Conclusion

The motivating challenge for this work was to develop decidable, static reasoning techniques to rule out run-time errors in the presence of programming idioms found in untyped languages. Throughout this dissertation, we have developed several refinement-type based techniques, summarized below, which can be applied to a variety of programming languages, both typed and untyped, that incorporate higher-order functions, objects, and references (*i.e.* the core language in Figure 1.1).

Key Technique 1: Nested Refinement Types. Our first contribution was to eliminate the syntactic distinction between base and function types found in prior refinement systems by nesting the typing relation inside the logic using uninterpreted functions. In particular, all types in System D are described as refinement types, and syntactic type constructors, like arrows, can appear inside formulas; these two changes to the language of refinement types are depicted below:

$$\begin{array}{lcllclcl} T & ::= & \{x\!:\!B \mid p\} & \rightsquigarrow & T & ::= & \{x \mid p\} \\ & \mid & \forall \overline{A}.\ x\!:\!T_1 \to T_2 & \rightsquigarrow & p & ::= & \cdots \mid y :: \forall \overline{A}.\ x\!:\!T_1 \to T_2 \end{array}$$

Nested refinements are well-suited to describe dictionary-based objects that store values of arbitrary, heterogeneous types. Many untyped languages, such as Python, Ruby, and JavaScript, conflate records (with string literal keys) and dictionaries (with arbitrary string keys), so nested refinements can be applied in these settings. Furthermore, nested refinements may prove useful in traditional statically typed languages, as well, to help reason about programming patterns involving dictionaries.

Key Technique 2: Flow-Sensitive Refinement Types. Our second contribution was to extend dependent function types with input and output heaps, depicted with the following syntax:

$$\forall \overline{A}.\ x\!:\!T_1 \to T_2 \qquad \rightsquigarrow \qquad \forall \overline{A}, \overline{M}, \overline{H}.\ \Psi \Rightarrow W_1 \to W_2$$

Our approach draws from and extends the work of Alias Types [91] — a flow-sensitive, syntactic type system for higher-order programs — and Low-Level Liquid Types [86] — a flow-sensitive, refinement type system for first-order programs. We formulated System !D with nested refinements, but the flow-sensitive, dependent function types we presented can be incorporated into traditional refinement type systems, as well.

Key Technique 3: Refinement Type Encoding for Prototypes. Our last main technique was to use refinement predicates to encode the semantics of prototype inheritance without using a fancier refinement logic than one with McCarthy's theory of arrays [71]. In System DJS, we formulated the notion of uninterpreted heap predicates that are unrolled (or expanded) when reasoning about function application. This notion can also be incorported into refinement type systems (with and without nesting) for functional languages, as well as languages with nominal (*i.e.* class-based) inheritance, because classes can be encoded with prototypes.

Beyond these three general mechanisms, we also provided JavaScript-specific encodings for arrays and primitive operations in System DJS in order to demonstrate the feasibility of applying our techniques to JavaScript.

8.1 Future Work

We have argued that our new refinement-type based techniques enable static reasoning for programming idioms found in dynamic languages. Along the way, we have identified several aspects to this work that warrant further consideration: the metatheory of flow-sensitive refinements, developed in System !D, and prototype unrolling, developed in System DJS, needs to be studied to provide a foundation of type safety; the expressiveness of System DJS needs to be extended to handle additional patterns, for example, mutually recursive functions; a larger evaluation of real-world examples is needed, which will require additional efforts to improve the usability (*i.e.* performance, annotation overhead, and error messages) of the system; the new techniques need to be integrated with contracts for situations where code is either unavailable or beyond the reach of static reasoning; and the new techniques should be studied in the context of different languages, both typed and untyped. Next, we identify several additional future directions that may build on our work.

Iterative Refinement Type Inference. Our techniques require explicit type annotations at function boundaries, and we perform local type inference for many intermediate expressions. Global type inference would be desirable in many situations, especially when function type annotations are relatively simple and uninformative, and when migrating existing, untyped programs. Liquid Types [85, 65, 86] is an effective technique for performing global refinement type inference with a very low annotation burden. The approach requires that simple syntactic types are inferred by a first phase, after which precise refinement predicates are inferred. Unfortunately, there is no simple syntactic type structure to start with in untyped languages — which is why we proposed nested refinements — so the Liquid Types approach cannot be applied directly.

One possibility for modular, global inference in this setting is to start by computing verification conditions for the arguments to each function, based on the refinement types of primitive operators, and then iteratively coarsen them until a weaker, but suitably informative, type is synthesized. It may be useful to interact with the user (*e.g.* by asking *abductive* questions [30]) to determine the level of precision required.

"Gradual Gradual Typing". Many JavaScript programmers use tools like Google Closure Compiler and Microsoft TypeScript that allow optional type annotations to be written. These annotations can provide helpful documentation, identify simple kinds of errors, and, perhaps most importantly, enable basic IDE support like auto-completion. However, to describe these tools as "optional type systems" is misleading, because the annotations are trusted, rather than verified, in certain cases. As a result, writing annotations in these systems can be thought of as "gradual gradual typing," moving untyped programs towards gradual typing (where annotations are verified), gradually.

Nevertheless, Closure Compiler and TypeScript provide value to programmers and are becoming more popular as a result. In future work, translations from the Closure Compiler and TypeScript annotation languages to DJS can help programmers migrate their applications from the former systems to DJS to obtain stronger guarantees.

Software Engineering Tools. Tools like Closure Compiler and TypeScript are attractive to programmers because they enable a degree of IDE support, which is crucial when developing large applications but noticeably lacking for dynamic languages. IDE tools can be built on the techniques in DJS to support JavaScript development. Some existing IDE support for JavaScript is based on pointer analysis [34, 35]; because of the aliasing restrictions enforced by DJS to enable strong updates, DJS-typable programs may be able to leverage precise pointer information and, thus, opportunities for sound refactoring.

Furthermore, it might be fruitful to incorporate the results of previous run-time traces into the information presented when viewing a source program, especially for expressions that have been left untyped. This information can be used to inform the programmer about how to modify the program to increase the degree of safety or, alternatively, design more test cases to increase the confidence that trusted assumptions in the code are not violated at run-time.

Browser Extension Security. Third-party JavaScript code is routinely downloaded and run in Web browsers as extensions that augment the functionality and appearance of the browser. To improve the security of third-party extensions, researchers and practitioners have proposed: (i) that programmers develop in higher-level languages that are more amenable to verifying fine-grained security policies which then compile to JavaScript for execution (*e.g.* IBEX [50]); and (ii) that sandboxes be built to limit the capabilities of third-party JavaScript and then verify that the sandboxes correctly enforce coarse-grained security policies (*e.g.* ADsafety [79]). Although both approaches have been fruitful, the former requires programming in higher level languages than JavaScript, and the the latter guarantees only relatively weak security properties. As discussed in §7.3, we believe a promising direction of future work is to use refinement types to encode security properties for programs written directly in (Dependent) JavaScript.

8.2 Related Work

We presented background on a variety of syntactic, refinement, and dependent type systems in Chapter 1 to set the stage for our contributions. To wrap up our presentation, we discuss additional related work with respect to the specific challenges we have addressed in System D and System DJS.

8.2.1 Verification for Functional Untyped Languages

In this section, we highlight related approaches to statically verifying features of functional dynamic languages. For a thorough introduction to contract-based and other hybrid approaches, see [36, 88, 67].

Dynamic Unions and Control Flow. Among the earliest attempts at mixing static and dynamic typing was adding the special type `dyn` to a statically-typed language like ML [2]. In this approach, an arbitrary value can be injected into `dyn`, and a typecase construct allows inspecting its precise type at run-time. However, one cannot guarantee that a particular `dyn` value is of one of a subset of types (cf. `negate` from §2.1). Several researchers have used union types and tag-test sensitive control-flow analyses to support such idioms. Most recently, Typed Racket [99] and Strobe [51] 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. Type checking 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 types for objects (*e.g.* [14, 1, 77]). Several early advances incorporate records into ML [81], but the use of records in these systems is 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. [6] 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 [98], which models dictionaries using row types labeled by singletons indexed by string constants, and depth subtyping. A recent proposal [105] 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 [43] 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. [68] 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. [64] presents an SMT-based system for statically inferring dependent types that verify the safety of ad-hoc unions in legacy C programs. [21] 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)` notion 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 [9] 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 [104] and further studied in [26, 31, 85, 67, 7, 94]. 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 mixing logical reasoning by SMT solvers and syntactic reasoning by subtyping is reminiscent of work on combining decision procedures [74, 87]. However, such techniques require the component theories 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.

8.2.2 Verification for Imperative Untyped Languages

In this section, we discuss topics related to types for imperative dynamic languages, hence, strong updates and inheritance.

Location Sensitive Types. The way we handle reference types draws from the approach of Alias Types [91], in which strong updates are enabled by describing reference types with abstract location names and by factoring reasoning into a flow-insensitive tying environment and a flow-sensitive heap. Low-Level Liquid Types [86] employs their approach in the setting of a first-order language with dependent types. In contrast, our setting includes higher-order functions, and our formulation of heap types gives variable names to unknown heaps to reason about prototypes and gives names to all heap values, which enables the specification of precise relationships between values of different heaps; the heap binders of [86] allow only relationships between values in a single heap to be described.

The original Alias Types work also includes support for weak references that point to zero or more values, for which strong updates are not sound. Several subsequent proposals [39, 4, 33, 3, 86, 92] allow strong updates to weak references under certain circumstances to support temporary invariant violations. We adapt the thaw and freeze mechanism from [3] and [86] with mostly cosmetic changes.

Prototype Inheritance. Unlike early class-based languages, such as Smalltalk and C++, the (untyped) language Self allows objects to be extended after creation and feature prototype, or delegation, inheritance. Static typing disciplines for class-based languages (*e.g.* [1]) explicitly preclude object extension to retain soundness in the presence of *width subtyping*, the ability to forget fields of an object. To mitigate the tension between object extension and subtyping, several proposals [10, 45] feature quite a different flavor: the fields of an object are split into a “reservation” part, which may be added to an object but cannot be forgotten, and a “sealed part” that can be manipulated with ordinary subtyping. Our approach provides additional precision in two important respects. First, we precisely track prototype hierarchies, whereas the above approaches flatten them into a single collection of fields. Second, we avoid the separation of reservation and sealed fields but still allow subtyping, since “width subtyping” in System !D is simply logical implication over refinement formulas; forgetting a field — discarding a $\mathit{has}(d,k)$ predicate — does not imply that $\neg \mathit{has}(d,k)$, which guards the traversal of the prototype chain.

Typed Subsets of JavaScript. Several (syntactic) type systems for various JavaScript subsets have been proposed. Among the earliest is [98], which identifies silent errors that result from implicit type coercion and the fact that JavaScript returns `undefined` when trying to look up a nonexistent key from an object. The approach in [6] distinguishes between *potential* and *definite* keys, similar to the reservation and sealed discussed above; this general approach has been extended with flow-sensitivity and polymorphism [105]. The notion of *recency types*, similar to Alias Types, was applied to JavaScript in [57], in which typing environments, in addition to heap types, are flow-sensitive. Prototype support in [57] is limited to the finite number of prototype links tracked by the type system, whereas the *heap symbols* in System !D enable reasoning about *entire* prototype hierarchies. Unlike System !D, all of the above systems provide global type inference; our system does not have principal types, so we can only provide local type inference [78]. ADsafety [79] is a type system for ADsafe, a JavaScript sandbox, that restricts access to some fields. Although expressive enough to check ADsafe, which heavily uses large object literals, they do not support strong update and so cannot reason about object extension. Unlike System !D, none of the above systems include dependent types, which are required to express truly dynamic object keys and precise control-flow based invariants.

Recent work on JavaScript verification uses separation logic [44] to track precise flow-sensitive invariants. They support only first-order programs, and the expressiveness of their logic takes them beyond automatic verification, thus requiring properties to be manually proved. JS* [95] is a verification system for JavaScript that, like DJS, translates programs to a variant of λ_{JS} for analysis. Their underlying refinement type system, F*, is higher-order and, hence, capable of reasoning about more complex properties than System DJS. As is usual for higher-order dependent systems, the tradeoff is that analysis time and the size of specifications can be large.

JavaScript Semantics. We chose the JavaScript “semantics-by-translation” of λ_{JS} [51] since it targets a conventional core language that has been convenient for our study. An alternate semantics [70] inherits unconventional aspects of the language specification [61] (*e.g.* “scope objects”), which complicates the formulation of static reasoning.

Endnotes

Acknowledgements. This chapter contains material adapted from the following publications:

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript.** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

- Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript (Technical Appendix).** `arXiv:1112.4106v3 [cs.PL]`, August 2012.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing.** In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2012.

- Ravi Chugh and Ranjit Jhala. **Dependent Types for JavaScript.** `arXiv:1112.4106v1 [cs.PL]`, December 2011.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing (Technical Appendix).** `arXiv:1103.5055v2 [cs.PL]`, September 2011.

- Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Dependent Dynamic Dictionaries.** `arXiv:1103.5055v1 [cs.PL]`, March 2011.

Appendix A

Soundness of System D

We prove the type soundness of System D, following the outline in §2.4, as well as the extensions to subtyping, discussed in Chapter 4.

Polarity. Before we attend to the metatheory, we define $\mathit{Polarity}(A,T) \;\circeq\; \mathit{Poles}(A,\texttt{+},T)$ in terms of the helper procedure *Poles* below, used to compute the polarity (either co-, contra-, or invariant) of type variables in datatype definitions, which we elided from the presentation of well-formed datatype definitions (§2.3.1). We define the inversion of polarities as $\neg(\texttt{+}) = \texttt{-}$ and $\neg(\texttt{-}) = \texttt{+}$. In the last case of the definition, we assume $\texttt{type}\ C[\overline{\theta B}] = \{\ \cdots\ \}$.

$$
\begin{array}{rcl}
\mathit{Poles}(A,\theta,\{x \mid p\}) & = & \mathit{Poles}(A,\theta,p) \\[1ex]
\mathit{Poles}(A,\theta,P(\overline{w})) & = & \emptyset \\
\mathit{Poles}(A,\theta,w :: U) & = & \mathit{Poles}(A,\theta,U) \\
\mathit{Poles}(A,\theta,p \wedge q) & = & \mathit{Poles}(A,\theta,p) \cup \mathit{Poles}(A,\theta,q) \\
\mathit{Poles}(A,\theta,p \vee q) & = & \mathit{Poles}(A,\theta,p) \cup \mathit{Poles}(A,\theta,q) \\
\mathit{Poles}(A,\theta,\neg p) & = & \mathit{Poles}(A,\neg\theta,p) \\[1ex]
\mathit{Poles}(A,\theta,A) & = & \{\theta\} \\
\mathit{Poles}(A,\theta,B) & = & \emptyset \\
\mathit{Poles}(A,\theta,\forall\overline{B}.\ x\!:\!T_1 \rightarrow T_2) & = & \mathit{Poles}(A,\neg\theta,T_1) \cup \mathit{Poles}(A,\theta,T_2) \\
\mathit{Poles}(A,\theta,C[\overline{T}]) & = & \cup_i \begin{cases} \mathit{Poles}(A,\theta,T_i) & \text{if } \theta_i = \texttt{+} \\ \mathit{Poles}(A,\neg\theta,T_i) & \text{if } \theta_i = \texttt{-} \\ \mathit{Poles}(A,\texttt{+},T_i) \cup \mathit{Poles}(A,\texttt{-},T_i) & \text{if } \theta_i = \texttt{=} \end{cases}
\end{array}
$$

$\mathit{Poles}(A,\texttt{+},T)$ computes a subset of $\{\texttt{+},\texttt{-}\}$ that includes + (resp. -) if A occurs in at least one positive (resp. negative) position inside T. For each type variable, these polarities are computed across all field types in the definition and then checked against its variance annotation, as described in

§2.3.1. By convention, we assume that all type variables in a program are distinct, so the equation for polymorphic function types above safely ignores the type variables $\overline{B}$.

A.1 Preliminaries

We specify definitions and assumptions about our refinement logic. Recall that we define the application of types to logical values using substitution as follows:

$$T(w) \;=\; \{x \mid p\}(w) \;\mathrel{\circeq}\; p[w/x]$$

Definition A.1.1 (Refinement Logic)**.**

The refinement logic underlying System D$_n$ starts with the quantifier-free fragment of first-order logic with equality extended with the theory of uninterpreted functions. As described in §2.3, the domain of values is drawn from a single datasort, and we use uninterpreted functions to classify values into more precise sets. Expressions, formulas, and type terms are encoded in the logic as uninterpreted constructed terms. Function terms are pairs of formal parameters and expression terms. To assign meaning to type predicates, we use a three-valued logic, where a third "don't-care" or "undefined" truth value augments the usual true and false values. In the interpretation $\mathcal{I}_n$ at level n, we evaluate type predicates $v :: U$ to true as follows (we refer to this definition later as "Type Predicate Interpretation"):

- $\mathcal{I}_n \models \lambda x.e :: \forall \overline{A}.\ x{:}T_{11} \to T_{12}$

  if $\vdash \forall \overline{A}.\ x{:}T_{11} \to T_{12}$ and $\overline{A}, x{:}T_{11} \vdash_{n-1} e \,::\, T_{12}$

- $\mathcal{I}_n \models c :: \forall \overline{A}.\ x{:}T_{11} \to T_{12}$

  if $\vdash \forall \overline{A}.\ x{:}T_{11} \to T_{12}$, $\mathit{ty}(c) = \{y \mid y = c \wedge y :: \forall \overline{A}.\ x{:}T_{01} \to T_{02}\}$, and

  $\vdash_{n-1} \forall \overline{A}.\ x{:}T_{01} \to T_{02} <: \forall \overline{A}.\ x{:}T_{11} \to T_{12}$.

- $\mathcal{I}_n \models C(\overline{v}) :: C[\overline{T}]$

  if $\vdash \overline{T}$, $\texttt{type}\ C[\overline{\theta A}] = \{\overline{f} : \overline{T'}\}$, and $\vdash_{n-1} v_j \,::\, \mathit{Inst}(T'_j, \overline{A}, \overline{T})$ for all j.

In all other cases, type predicates evaluate to the undefined truth value. We make the assumption that there are no negative occurrences of type predicates in programs and, hence, in type checking obligations. This restriction, together with the use of a three-valued logic, ensures that the interpretations at each level are monotonic with respect to the set of formulas which evaluate to true; this property is captured by the Lifting lemma, described later.

Since there are no typing judgements with index less than 0, no type predicates evaluate to true in the interpretation $\mathcal{I}_0$ at level 0. Thus, $\mathcal{I}_0$ consists of only the underlying logic with uninterpreted functions.

Fact A.1.2 (Refinement Logic Properties)**.**

- *(Validity) We write Valid(p) to mean that, as usual, p is satisfiable in $\mathcal{I}_0$ with arbitrary assignments to free variables. In the* I-VALID *rule, we appeal to a decision procedure to check whether Valid(p).*
- *(Free Variable Substitution) If x appears free in p and q, then $p \Rightarrow q$ implies $p[v/x] \Rightarrow q[v/x]$ for all v.*
- *(Uninterpreted Predicate Substitution) If P is an uninterpreted predicate symbol in p and q, then $p \Rightarrow q$ implies $p[P'/P] \Rightarrow q[P'/P]$ for all P'.*

Assumption A.1.3 (Constant Types)**.** *Recall that the δ function defines the semantics of constants. For every constant $c \in dom(ty)$, the following properties hold.*

- *(Well-formed) $\vdash ty(c)$.*
- *(Normal) $ty(c) = \{x \,|\, x = c \wedge p\}$ where either $p = true$ or $p = x :: \forall\overline{A}.\ y\!:\!T_1 \to T_2$.*
- *(App) If $ty(c) = \{x \,|\, x = c \wedge x :: \forall\overline{A}.\ y\!:\!T_{11} \to T_{12}\}$, then for all v, n, and $\overline{T}$*

  $\vdash_n v :: Inst(T_{11}, \overline{A}, \overline{T})$ implies that $\delta(c,v)$ is defined and $\vdash_n \delta(c,v) :: Inst(T_{12}(v), \overline{A}, \overline{T})$.
- *(Valid) Valid$(ty(c)(c))$. In other words, we add the fact $ty(c)(c)$ to the initial type environment.*

Assumption A.1.4 (Datatype Representation)**.** *This assumption requires that the implementation treats constructed data just like ordinary dictionaries, in particular, that the fields of constructed data can be read and written using the ordinary dictionary primitives. Let* `type` $\mathsf{C}[\overline{\theta A}] = \{\overline{f\!:\!T'}\}$ *be a member of the fixed, global list of datatype definitions Ψ.*

- *If $\mathcal{I}_n \models v :: \mathsf{C}[\overline{T}]$, then $\mathcal{I}_n \models tag(v) =$ "*`dictionary`*" $\wedge\ (\wedge_j\ Inst(T'_j, \overline{A}, \overline{T})(sel(v, f_j)))$.*

A.2 Proofs

To reduce clutter, we elide the well-formedness requirements of all expressions, formulas, types, type terms, typing environments, and type definitions mentioned in the lemmas and theorems that follow. We write "IH" in proofs to abbreviate "induction hypothesis." We often use the metavariable S, in addition to T, to range over refinement types $\{x \,|\, p\}$.

Lemma A.2.1 (Inversion)**.** *We list only properties that we will need to use.*

1. *If $\Gamma \vdash_n \forall\overline{A}.\ x\!:\!T_{11} \to T_{12} <: \forall\overline{A}.\ x\!:\!T_{21} \to T_{22}$,*

   then $\Gamma, \overline{A} \vdash_n T_{21} \sqsubseteq T_{11}$ and $\Gamma, \overline{A}, x\!:\!T_{21} \vdash_n T_{12} \sqsubseteq T_{22}$.
2. *If $\Gamma \vdash_n \lambda x.e :: T$, then $\Gamma \vdash_n \lambda x.e :: \{y \,|\, y = \lambda x.e \wedge y :: \forall\overline{A}.\ x\!:\!T_1 \to T_2\}$*

Proof. By induction. □

Lemma A.2.2 (Reflexive Subtyping)**.**

1. $\Gamma \vdash_n U <: U$

2. $\Gamma \vdash_n T \sqsubseteq T$

Proof. By mutual induction. □

Lemma A.2.3 (Transitive Subtyping)**.**

1. *If* $\Gamma \vdash_n U_1 <: U_2$ *and* $\Gamma \vdash_n U_2 <: U_3$*, then* $\Gamma \vdash_n U_1 <: U_3$*.*

2. *If* $\Gamma \vdash_n T_1 \sqsubseteq T_2$ *and* $\Gamma \vdash_n T_2 \sqsubseteq T_3$*, then* $\Gamma \vdash_n T_1 \sqsubseteq T_3$*.*

Proof. By mutual induction. □

Lemma A.2.4 (Narrowing)**.** *Suppose* $\Gamma \vdash_n T \sqsubseteq T'$*.*

1. *If* $\Gamma, x{:}T' \Rightarrow_n p$*, then* $\Gamma, x{:}T \Rightarrow_n p$*.*

2. *If* $\Gamma, x{:}T' \vdash_n U_1 <: U_2$*, then* $\Gamma, x{:}T \vdash_n U_1 <: U_2$*.*

3. *If* $\Gamma, x{:}T' \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma, x{:}T \vdash_n T_1 \sqsubseteq T_2$*.*

4. *If* $\Gamma, x{:}T' \vdash_n e :: T_1$*, then* $\Gamma, x{:}T \vdash_n e :: T_1$*.*

Proof. By mutual induction. □

Lemma A.2.5 (Weakening)**.** *Suppose* $\Gamma = \Gamma_1, \Gamma_2$ *and* Γ' *is such that* $\vdash \Gamma'$ *and either*

$$\Gamma' = \Gamma_1, x{:}T, \Gamma_2 \quad \textit{or} \quad \Gamma' = \Gamma_1, p, \Gamma_2 \quad \textit{or} \quad \Gamma' = \Gamma_1, A, \Gamma_2.$$

1. *If* $\Gamma \Rightarrow_n q$*, then* $\Gamma' \Rightarrow_n q$*.*

2. *If* $\Gamma \vdash_n U_1 <: U_2$*, then* $\Gamma' \vdash_n U_1 <: U_2$*.*

3. *If* $\Gamma \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma' \vdash_n T_1 \sqsubseteq T_2$*.*

4. *If* $\Gamma \vdash_n e :: T$*, then* $\Gamma' \vdash_n e :: T$*.*

Proof. By mutual induction. □

Lemma A.2.6 (Free Variables in Subtyping)**.** *Suppose* w *is a closed, well-formed value.*

1. *If* $\Gamma \Rightarrow_n p$ *and* x *appears free in* p*, then* $\Gamma \Rightarrow_n p[w/x]$*.*

2. *If* $\Gamma \vdash_n \{x \mid p\} \sqsubseteq \{x \mid q\}$*, then* $\Gamma \vdash_n \{x \mid p[w/x]\} \sqsubseteq \{x \mid q[w/x]\}$*.*

Proof. By mutual induction. The premise of the I-VALID-N case is $\mathcal{I}_n \models \mathit{Embed}(\Gamma) \Rightarrow p$. Since x appears free in the implication, by Free Variable Substitution, $\mathcal{I}_n \models \mathit{Embed}(\Gamma) \Rightarrow p[w/x]$. Thus, by I-VALID-N, $\Gamma \Rightarrow_n p[w/x]$. The rest of the proof is a straightforward induction. □

Lemma A.2.7 (Sound Variance)**.**

1. *Suppose* $\Gamma \vdash_n T_1 \sqsubseteq T_2$.

   (a) *If B appears only positively in T, then* $\Gamma \vdash_n \mathit{Inst}(T,B,T_1) \sqsubseteq \mathit{Inst}(T,B,T_2)$.

   (b) *If B appears only positively in p, then* $\Gamma \vdash_n \{x \mid \mathit{Inst}(p,B,T_1)\} \sqsubseteq \{x \mid \mathit{Inst}(p,B,T_2)\}$.

2. *Suppose* $\Gamma \vdash_n T_1 \sqsubseteq T_2$.

   (a) *If B appears only negatively in T, then* $\Gamma \vdash_n \mathit{Inst}(T,B,T_2) \sqsubseteq \mathit{Inst}(T,B,T_1)$.

   (b) *If B appears only negatively in p, then* $\Gamma \vdash_n \{x \mid \mathit{Inst}(p,B,T_2)\} \sqsubseteq \{x \mid \mathit{Inst}(p,B,T_1)\}$.

3. *Suppose* $\Gamma \vdash_n T_1 \sqsubseteq T_2$ *and* $\Gamma \vdash_n T_2 \sqsubseteq T_1$.

   (a) *Then* $\Gamma \vdash_n \mathit{Inst}(T,B,T_2) \sqsubseteq \mathit{Inst}(T,B,T_1)$ *and* $\Gamma \vdash_n \mathit{Inst}(T,B,T_1) \sqsubseteq \mathit{Inst}(T,B,T_2)$.

   (b) *Then* $\Gamma \vdash_n \{x \mid \mathit{Inst}(p,B,T_2)\} \sqsubseteq \{x \mid \mathit{Inst}(p,B,T_1)\}$
   and $\Gamma \vdash_n \{x \mid \mathit{Inst}(p,B,T_1)\} \sqsubseteq \{x \mid \mathit{Inst}(p,B,T_2)\}$.

Proof. The proofs of (1) and (2) are by mutual induction on types and formulas. The proof of (3) is a stand-alone induction on types and formulas.

Proof of (1a). Let $T = \{x \mid p\}$. The goal follows by IH (1b), because, by definition of *Inst* on refinement types, $\mathit{Inst}(\{x \mid p\},B,T_1) = \{x \mid \mathit{Inst}(p,B,T_1)\}$ and $\mathit{Inst}(\{x \mid p\},B,T_2) = \{x \mid \mathit{Inst}(p,B,T_2)\}$. □

Proof of (1b).

Case: $p = P(\overline{w})$. Trivial, since $\mathit{Inst}(p,B,T_1) = \mathit{Inst}(p,B,T_2) = p$.

Cases: $p = q_1 \wedge q_2$, $p = q_1 \vee q_2$. By IH (1b), I-VALID, I-CNF, and S-REFINE.

Case: $p = \neg q$. By IH (2b), I-VALID, I-CNF, and S-REFINE.

Case: $p = w :: U$.

Subcase: $U = B$.
By definition, $\mathit{Inst}(p,B,T_1) = T_1(w)$ and $\mathit{Inst}(p,B,T_2) = T_2(w)$.
By Free Variables in Subtyping, $\Gamma \vdash_n \{x \mid T_1(w)\} \sqsubseteq \{x \mid T_2(w)\}$.

Subcase: $U = x{:}S_1 \to S_2$.
Let $S_{11} = \mathit{Inst}(S_1,B,T_1)$ and $S_{12} = \mathit{Inst}(S_1,B,T_2)$.
Let $S_{21} = \mathit{Inst}(S_2,B,T_1)$ and $S_{22} = \mathit{Inst}(S_2,B,T_2)$.

Since B appears only positively in p, it appears only negatively in S_1,

by the well-formedness of the type definition and the definition of *Poles*.

By IH (2a), $\Gamma \vdash_n S_{12} \sqsubseteq S_{11}$.

Since B appears only positively in p, it appears only positively in S_2.

By IH (1a), $\Gamma \vdash_n S_{21} \sqsubseteq S_{22}$.

By Weakening, $\Gamma, x{:}S_{21} \vdash_n S_{21} \sqsubseteq S_{22}$.

By U-Arrow, $\Gamma \vdash_n x{:}S_{11} \to S_{21} <: x{:}S_{12} \to S_{22}$.

By I-Valid, $\Gamma, p :: x{:}S_{11} \to S_{21} \Rightarrow_n p :: x{:}S_{11} \to S_{21}$.

By I-HasType, $\Gamma, p :: x{:}S_{11} \to S_{21} \Rightarrow_n p :: x{:}S_{12} \to S_{22}$.

By I-Cnf and S-Refine, $\Gamma \vdash_n \{y \mid w :: x{:}S_{11} \to S_{21}\} \sqsubseteq \{y \mid w :: x{:}S_{12} \to S_{22}\}$.

That is, $\Gamma \vdash_n \{x \mid \mathit{Inst}(p,B,T_1)\} \sqsubseteq \{x \mid \mathit{Inst}(p,B,T_2)\}$.

Subcase: $U = C[S]$, where $\texttt{type}\ C[\theta A] = \{\ \cdots\ \}$.

Subsubcase: $\theta = \texttt{+}$ and $\theta = \texttt{=}$.

Let $S_1 = \mathit{Inst}(S,B,T_1)$ and $S_2 = \mathit{Inst}(S,B,T_2)$.

Since B appears only positively in p, it appears only positively in S.

By IH (1a), $\Gamma \vdash_n S_1 \sqsubseteq S_2$.

By U-Datatype, $\Gamma \vdash_n C[S_1] <: C[S_2]$.

By I-Valid, I-HasType, I-Cnf, and S-Refine, $\Gamma \vdash_n \{y \mid w :: C[S_1]\} \sqsubseteq \{y \mid w :: C[S_2]\}$.

Subsubcase: $\theta = \texttt{-}$.

Similar. □

Proof of (2a) *and* (2b). Similar. □

Proof of (3). Straightforward induction. □

Lemma A.2.8 (Lifting Derivations)**.**

1. *If* $\Gamma \Rightarrow_n p$*, then* $\Gamma \Rightarrow_{n+1} p$*.*

2. *If* $\Gamma \vdash_n U_1 <: U_2$*, then* $\Gamma \vdash_{n+1} U_1 <: U_2$*.*

3. *If* $\Gamma \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma \vdash_{n+1} T_1 \sqsubseteq T_2$*.*

4. *If* $\Gamma \vdash_n e :: T$*, then* $\Gamma \vdash_{n+1} e :: T$*.*

5. *If* $\mathcal{I}_n \models p$*, then* $\mathcal{I}_{n+1} \models p$*.*

Furthermore, for each of the first four properties, the size of the output derivation is the same size as the original.

Proof. By mutual induction. In the I-Valid-n case of (1), the conclusion follows by I-Valid-n after applying IH (5). Property (5) states that the stratified interpretations are monotonic in the statements which evaluate to true. The type predicate case for (5) follows from IH (4). Because we forbid negative occurrences of type predicates, monotonicity extends to arbitrary formulas. □

Lemma A.2.9 (Strengthening)**.** *Suppose $\mathcal{I}_n \models p$.*

1. *If $p, \Gamma \Rightarrow_n q$, then $\Gamma \Rightarrow_n q$.*
2. *If $p, \Gamma \vdash_n U_1 <: U_2$, then $\Gamma \vdash_n U_1 <: U_2$.*
3. *If $p, \Gamma \vdash_n T_1 \sqsubseteq T_2$, then $\Gamma \vdash_n T_1 \sqsubseteq T_2$.*
4. *If $p, \Gamma \vdash_n e :: T$, then $\Gamma \vdash_n e :: T$.*

Furthermore, for each property, the size of the output derivation is the same size as the original.

Proof. By mutual induction.

Proof of (1).

Case: I-Valid-n.
$$\frac{\mathcal{I}_n \models \mathit{Embed}(p, \Gamma) \Rightarrow q}{p, \Gamma \Rightarrow_n q}$$

By expanding the embedding, $\mathcal{I}_n \models p \wedge \mathit{Embed}(\Gamma) \Rightarrow q$.
Thus, $\mathcal{I}_n \models p \Rightarrow \mathit{Embed}(\Gamma) \Rightarrow q$.
Because of the assumption, $\mathcal{I}_n \models \mathit{Embed}(\Gamma) \Rightarrow q$.
By I-Valid-n, $\Gamma \Rightarrow_n q$.

Case: I-Valid.
$$\frac{\mathit{Valid}(\mathit{Embed}(p, \Gamma) \Rightarrow q)}{p, \Gamma \Rightarrow_n q}$$

By Validity, $\mathcal{I}_n \models \mathit{Embed}(p, \Gamma) \Rightarrow q$.
The rest of the reasoning in this case follows the previous case.

Case: I-HasType.
$$\frac{\mathit{Valid}(\mathit{Embed}(p, \Gamma) \Rightarrow w :: U') \qquad p, \Gamma \vdash_n U' <: U}{p, \Gamma \Rightarrow_n w :: U}$$

By I-Valid, $p, \Gamma \Rightarrow_n w :: U'$.
By the assumption and IH (1), $\Gamma \Rightarrow_n w :: U'$.
By IH (2), $\Gamma \vdash_n U' <: U$.
By I-HasType, $\Gamma \Rightarrow_n w :: U$.

Case: I-Cnf. Straightforward. □

Proof of (2), (3), *and* (4). Straightforward induction. □

The following lemma intuitively captures the relationship between the type system and the underlying refinement logic: if a closed value v can be given the type T with a derivation at level n, then the formula $\mathit{Embed}(T)(v)$ is true in the System D interpretation at level $n+1$. This property plays a crucial role in the proof of Value Substitution.

Because the following lemma works only with the empty environment, the Strengthening lemma is helpful for proving the I-Cnf case, which has a premise that uses a non-empty environment.

Main Lemma A.2.10 (Satisfiable Typing)**.**

1. *If* $\Rightarrow_n p$, *then* $\mathcal{I}_{n+1} \models p$.
2. *If* $\vdash_n U_1 <: U_2$, *then* $\mathcal{I}_{n+1} \models x :: U_1 \Rightarrow x :: U_2$.
3. *If* $\vdash_n \{x \mid p\} \sqsubseteq \{x \mid q\}$, *then* $\mathcal{I}_{n+1} \models p \Rightarrow q$.
4. *If* $\vdash_n v :: T$, *then* $\mathcal{I}_{n+1} \models T(v)$.

In the first three properties, the variable x appears free in the implication. Thus, they are implicitly quantified over *all* values.

Proof. By mutual induction on the size of derivations, not by structural induction. The reason for this induction principle is that in some cases, subderivations are manipulated by Lifting and Strengthening (which preserve derivation size) before appealing to the induction hypothesis.

Proof of (1).

Case: I-Valid. $$\frac{\mathit{Valid}(\mathit{true} \Rightarrow p)}{\Rightarrow_n p}$$

By Validity, p is true in all interpretations. Thus, $\mathcal{I}_{n+1} \models p$.

Case: I-Valid-n. $$\frac{\mathcal{I}_n \models \mathit{true} \Rightarrow p}{\Rightarrow_n p}$$

By Lifting, $\mathcal{I}_{n+1} \models \mathit{true} \Rightarrow p$. Thus, $\mathcal{I}_{n+1} \models p$.

Case: I-HasType. $$\frac{\mathit{Valid}(\mathit{true} \Rightarrow w :: U') \qquad \vdash_n U' <: U}{\Rightarrow_n w :: U}$$

By I-Valid, $\Rightarrow_n w :: U'$.
By IH (1), $\mathcal{I}_{n+1} \models w :: U'$.
By IH (2), $\mathcal{I}_{n+1} \models x :: U' \Rightarrow\ x :: U$.
By Free Variable Substitution, $\mathcal{I}_{n+1} \models w :: U$.

Case: I-Cnf. $$\frac{\mathit{CNF}(p) = \wedge_i\ (q_i \Rightarrow Q_i) \qquad \forall i.\ \exists q' \in Q_i.\ q_i \Rightarrow_n q'}{\Rightarrow_n p}$$

To establish $\mathcal{I}_{n+1} \models p$, we consider its equivalent CNF formula.

For each clause $\mathcal{I}_{n+1} \models q_i \Rightarrow q'$, we assume $\mathcal{I}_{n+1} \models q_i$ and then prove $\mathcal{I}_{n+1} \models q'$.

By Strengthening on each premise (with indices i and i_j), $\Rightarrow_n q'$.

These derivations have the same size, so by IH (1) on each, $\mathcal{I}_{n+1} \models q'$. □

Proof of (2).

Case: U-Arrow.
$$\frac{\overline{A} \vdash_n T_{21} \sqsubseteq T_{11} \qquad \overline{A}, x\!:\!T_{21} \vdash_n T_{12} \sqsubseteq T_{22}}{\vdash_n \forall \overline{A}.\ x\!:\!T_{11} \to T_{12} <: \forall \overline{A}.\ x\!:\!T_{21} \to T_{22}}$$

Let $U_1 = \forall \overline{A}.\ x\!:\!T_{11} \to T_{12}$ and $U_2 = \forall \overline{A}.\ x\!:\!T_{21} \to T_{22}$.

We assume $\mathcal{I}_{n+1} \models v :: U_1$ and will prove $\mathcal{I}_{n+1} \models v :: U_2$.

By Type Predicate Interpretation, there are two cases.

Subcase: $v = \lambda x.e$ and $\overline{A}, x\!:\!T_{11} \vdash_n e :: T_{12}$.

By Narrowing, $\overline{A}, x\!:\!T_{21} \vdash_n e :: T_{12}$.

By T-Sub, $\overline{A}, x\!:\!T_{21} \vdash_n e :: T_{22}$.

Thus, by Type Predicate Interpretation, $\mathcal{I}_{n+1} \models \lambda x.e :: U_2$.

Subcase:

$v = c$, $ty(c) = \{y \mid y = c \wedge y :: \forall \overline{A}.\ x\!:\!T_{01} \to T_{02}\}$, and $\vdash_n \forall \overline{A}.\ x\!:\!T_{01} \to T_{02} <: U_1$.

By Inversion, $\overline{A} \vdash_n T_{11} \sqsubseteq T_{01}$ and $\overline{A}, x\!:\!T_{11} \vdash_n T_{02} \sqsubseteq T_{12}$.

By Transitive Subtyping, $\overline{A} \vdash_n T_{21} \sqsubseteq T_{01}$.

By Narrowing, $\overline{A}, x\!:\!T_{21} \vdash_n T_{02} \sqsubseteq T_{12}$.

By Transitive Subtyping, $\overline{A}, x\!:\!T_{21} \vdash_n T_{02} \sqsubseteq T_{22}$.

By U-Arrow, $\vdash_n \forall \overline{A}.\ x\!:\!T_{01} \to T_{02} <: U_2$.

By Type Predicate Interpretation, $\mathcal{I}_{n+1} \models c :: U_2$.

Case: U-Var. Trivial.

Case: U-Datatype.
$$\frac{\begin{array}{c}\texttt{type}\ C[\overline{\theta A}] = \{\overline{f\!:\!T'}\} \\ \forall i.\ \text{if } \theta_i \in \{\texttt{+},\texttt{=}\} \text{ then } \vdash_n T_{1i} \sqsubseteq T_{2i} \qquad \forall i.\ \text{if } \theta_i \in \{\texttt{-},\texttt{=}\} \text{ then } \vdash_n T_{2i} \sqsubseteq T_{1i}\end{array}}{\vdash_n C[\overline{T_1}] <: C[\overline{T_2}]}$$

We consider the special case when there is exactly one type parameter A with variance annotation θ. The type actuals are, therefore, labeled T_{11} and T_{21}. The reasoning extends to an arbitrary number of type parameters by a strong induction on the length of the sequence.

Subcase: θ = +.

Consider an arbitrary v_0 such that $\mathcal{I}_{n+1} \models v_0 :: C[T_{11}]$.

By Type Predicate Interpretation, $v_0 = C(\overline{v})$ and for all j, $\vdash_n v_j :: \mathit{Inst}(T'_j, A, T_{11})$.

By well-formedness of the type definition, A appears only positively in every T'_j.

By Sound Variance (1), $\vdash_n \mathit{Inst}(T'_j, A, T_{11}) \sqsubseteq \mathit{Inst}(T'_j, A, T_{21})$.
By T-Sub, $\vdash_n v_j :: \mathit{Inst}(T'_j, A, T_{21})$.
By Type Predicate Interpretation, $\mathcal{I}_{n+1} \models C(\overline{v}) :: C[T_{21}]$.

Subcase: $\theta = -$. Similar, using Sound Variance (2).

Subcase: $\theta = =$. Similar, using Sound Variance (3). □

Proof of (3).

Case: S-Refine.

$$\frac{y \text{ fresh} \qquad \Rightarrow_n (p[y/x] \Rightarrow q[y/x])}{\vdash_n \{x \mid p\} \sqsubseteq \{x \mid q\}}$$

By IH (1), since alpha-renaming does not affect validity. □

Proof of (4). We only need to consider the rules that can derive a type T for a value v in the empty environment.

Case: T-Const. By Constant Types (Valid).

Case: T-Extend. Trivially, since $v = v_1[v_2 \mapsto v_3]$, $T = \{x \mid x = v\}$, and $T(v) = (v = v)$.

Case: T-Fun.

$$\frac{U = x{:}T_1 \to T_2 \qquad x{:}T_1 \vdash_n e :: T_2}{\vdash_n \lambda x.e :: \{y \mid y = \lambda x.e \wedge y :: U\}}$$

By Type Predicate Interpretation, $\mathcal{I}_{n+1} \models \lambda x.e :: U$.
Furthermore, by Validity, $\mathcal{I}_{n+1} \models \lambda x.e = \lambda x.e$.

Case: T-Fold.

$$\frac{\texttt{type } C[\overline{\theta A}] = \{\overline{f : T'}\} \qquad \forall j.\ \vdash_n v_j :: \mathit{Inst}(T_j, \overline{A}, \overline{T})}{\vdash_n C(\overline{v}) :: \{x \mid \mathit{Fold}(x, C, \overline{T}, \overline{v})\}}$$

We consider each of the components of the formula from *Fold*, defined in §2.3.2.
By Datatype Representation, $\mathcal{I}_{n+1} \models \mathit{tag}(C(\overline{v})) = \text{“}\texttt{dictionary}\text{”} \wedge (\wedge_j\ \mathit{sel}(C(\overline{v}), f_j) = v_j)$.
By Type Predicate Interpretation, $\mathcal{I}_{n+1} \models C(\overline{v}) :: C[\overline{T}]$.

Case: T-Unfold.

$$\frac{\vdash_n v :: \{x \mid x :: C[\overline{T}]\}}{\vdash_n v :: \{x \mid \mathit{Unfold}(x, C, \overline{T})\}}\ [\textsc{T-Unfold}]$$

By IH (3), $\mathcal{I}_{n+1} \models v :: C[\overline{T}]$.
The goal follows by Type Predicate Interpretation and Datatype Representation.

Case: T-Sub.

$$\frac{\vdash_n v :: T' \qquad \vdash_n T' \sqsubseteq T}{\vdash_n v :: T}$$

By IH (3), $\mathcal{I}_{n+1} \models \mathit{Embed}(T')(v) \Rightarrow \mathit{Embed}(T)(v)$.
By IH (4), $\mathcal{I}_{n+1} \models \mathit{Embed}(T')(v)$.
So, $\mathcal{I}_{n+1} \models \mathit{Embed}(T')(v)$. □

In the following lemma we lift substitution to judgements in the obvious way. For example, we write $(\Gamma \vdash_n e :: T)[v/x]$ to mean $\Gamma[v/x] \vdash_n e[v/x] :: T[v/x]$.

Main Lemma A.2.11 (Stratified Value Substitution)**.** *Let* $\vdash_n v :: T$.

1. *If* $x{:}T, \Gamma \Rightarrow_n p$*, then* $(\Gamma \Rightarrow_{n+1} p)[v/x]$.
2. *If* $x{:}T, \Gamma \vdash_n U_1 <: U_2$*, then* $(\Gamma \vdash_{n+1} U_1 <: U_2)[v/x]$.
3. *If* $x{:}T, \Gamma \vdash_n T_1 \sqsubseteq T_2$*, then* $(\Gamma \vdash_{n+1} T_1 \sqsubseteq T_2)[v/x]$.
4. *If* $x{:}T, \Gamma \vdash_n e :: T'$*, then* $(\Gamma \vdash_{n+1} e :: T')[v/x]$.

Proof. By mutual induction. The T-Var case is interesting because singleton types must be preserved after substitution.

Proof of (1). Case: I-Valid-n.

$$\frac{\mathcal{I}_n \models \mathit{Embed}(x{:}T, \mathit{\Gamma}) \Rightarrow p}{x{:}T, \Gamma \Rightarrow_n p}$$

Thus, $\mathcal{I}_n \models T(x) \wedge \mathit{Embed}(\mathit{\Gamma}) \Rightarrow p$.
That is, $\mathcal{I}_n \models T(x) \Rightarrow \mathit{Embed}(\mathit{\Gamma}) \Rightarrow p$.
By Lifting, $\mathcal{I}_{n+1} \models T(x) \Rightarrow \mathit{Embed}(\mathit{\Gamma}) \Rightarrow p$.
By Satisfiable Typing, $\mathcal{I}_{n+1} \models T(v)$.
Thus, $\mathcal{I}_{n+1} \models \mathit{Embed}(\mathit{\Gamma})[v/x] \Rightarrow p[v/x]$ by Free Variable Substitution.
So by I-Valid-n, $\Gamma[v/x] \Rightarrow_{n+1} p[v/x]$.

Case: I-Valid.

$$\frac{\mathit{Valid}(\mathit{Embed}(x{:}T, \mathit{\Gamma}) \Rightarrow p)}{x{:}T, \Gamma \Rightarrow_n p}$$

Thus, $\mathit{Valid}(T(x) \wedge \mathit{Embed}(\mathit{\Gamma}) \Rightarrow p)$.
So $\mathcal{I}_n \models T(x) \wedge \mathit{Embed}(\mathit{\Gamma}) \Rightarrow p$ by Validity.
The rest of the reasoning follows the I-Valid-n case.

Case: I-HasType.

$$\frac{\mathit{Valid}(\mathit{Embed}(x{:}T, \mathit{\Gamma}) \wedge p \Rightarrow w :: U') \qquad x{:}T, \Gamma \vdash_n U' <: U}{x{:}T, \Gamma \Rightarrow_n w :: U}$$

By I-Valid, $x{:}T, \Gamma \Rightarrow_{n+1} w :: U'$.
By IH (1), $\Gamma[v/x] \Rightarrow_{n+1} w[v/x] :: U'[v/x]$.
By IH (2), $\Gamma[v/x] \vdash_{n+1} U'[v/x] <: U[v/x]$.
By I-HasType, $\Gamma[v/x] \Rightarrow_{n+1} w :: U)[v/x]$.

Case: I-Cnf. By induction and equisatisfiability of CNF formulas. □

Proof of (2). Straightforward induction. □

Proof of (3). Straightforward induction. □

Proof of (4).

Case: T-Const.

By T-Const, $\Gamma[v/x] \vdash_{n+1} c :: \mathit{ty}(c)$.

By Constant Types (Well-formed), $\vdash \mathit{ty}(c)$, so $\mathit{ty}(c)$ has no free variables.

Thus, $\mathit{ty}(c)[v/x] = \mathit{ty}(c)$.

Also, $c[v/x] = c$, which concludes the case.

Case: T-Var. $$\frac{y \in \mathit{dom}(x{:}T, \Gamma)}{x{:}T, \Gamma \vdash_n y :: \{z \mid z = y\}}$$

Subcase: $x \neq y$.

By T-Var, $\Gamma[v/x] \vdash_{n+1} y :: \{z \mid z = y\}$.

This concludes the subcase since $y[v/x] = y$ and $\{z \mid z = y\}[v/x] = \{z \mid z = y\}$.

Subcase: $x = y$. Note that $x[v/x] = v$ and $\{z \mid z = x\}[v/x] = \{z \mid z = v\}$.

Subsubcase: $v = y'$. Impossible, since the typing environment is empty.

Subsubcase: $v = v_1[v_2 \mapsto v_3]$. Trivial, by T-Extend.

Subsubcase: $v = c$.

By T-Const, $\Gamma[v/x] \vdash_{n+1} c :: \mathit{ty}(c)$.

By Constant Types (Normal), $\mathit{ty}(c) = \{z \mid z = c \wedge p\}$.

By I-Valid, I-Cnf, and S-Refine, $\Gamma[v/x] \vdash_{n+1} \{z \mid z = c \wedge p\} \sqsubseteq \{z \mid z = c\}$.

By T-Sub, $\Gamma[v/x] \vdash_{n+1} c :: \{z \mid z = c\}$.

Subsubcase: $v = \lambda x'.e_0$.

By Inversion, $\vdash_{n+1} v :: \{z \mid z = v \wedge z :: U\}$.

By I-Valid, I-Cnf, and S-Refine, $\vdash_{n+1} \{z \mid z = v \wedge z :: U\} \sqsubseteq \{z \mid z = v\}$.

By T-Sub, $\vdash_{n+1} v :: \{z \mid z = v\}$.

By Weakening, $\Gamma[v/x] \vdash_{n+1} v :: \{z \mid z = v\}$.

Case: T-Fun. $$\frac{e = \lambda y.e_0 \qquad U = \forall \overline{A}.\ y{:}T_1 \to T_2 \qquad x{:}T, \Gamma, \overline{A}, y{:}T_1 \vdash_n e_0 :: T_2}{x{:}T, \Gamma \vdash_n e :: \{z \mid z = e \wedge z :: U\}}$$

By IH (4), $\Gamma[v/x], y{:}T_1[v/x] \vdash_{n+1} e_0[v/x] :: T_2[v/x]$.

By T-Fun, $\Gamma[v/x] \vdash_{n+1} e[v/x] :: \{z \mid z = e[v/x] \wedge z :: U[v/x]\}$.

Thus, $\Gamma[v/x] \vdash_{n+1} e[v/x] :: \{z \mid z = e \wedge z :: U\}[v/x]$.

Case: T-App. $$\frac{x{:}T,\Gamma \vdash_n v_1 :: \{z \,|\, z :: \forall\overline{A}.\ x{:}T_{11} \to T_{12}\} \qquad x{:}T,\Gamma \vdash_n v_2 :: \mathit{Inst}(T_{11},\overline{A},\overline{S})}{x{:}T,\Gamma \vdash_n [\overline{S}]\ v_1\ v_2 :: \mathit{Inst}(T_{12},\overline{A},\overline{S})[v_2/y]}$$

Let $\Gamma' = \Gamma[v/x]$, $v_1' = v_1[v/x]$, $v_2' = v_2[v/x]$, and $T_{11}' = T_{11}[v/x]$.
Let $T_{12}' = \mathit{Inst}(T_{12},\overline{A},\overline{S})[v/x]$.
By IH (4), $\Gamma' \vdash_{n+1} v_1' :: \{z \,|\, z :: \forall\overline{A}.\ y{:}T_{11} \to T_{12}\}[v/x]$.
Thus, $\Gamma' \vdash_{n+1} v_1' :: \{z \,|\, z :: \forall\overline{A}.\ y{:}T_{11}' \to T_{12}'\}$.
By IH (4), $\Gamma' \vdash_{n+1} v_2' :: \mathit{Inst}(T_{11}',\overline{A},\overline{S})$.
By T-App, $\Gamma' \vdash_{n+1} [\overline{S}]\ v_1'\ v_2' :: T_{12}'[v_2'/y]$.
Now we expand $T_{12}'[v_2'/y]$ to $\mathit{Inst}(T_{12},\overline{A},\overline{S})[v/x][v_2/y][v/x]$.
Since v and v_2 are closed values, and x and y are distinct, this is the same as $\mathit{Inst}(T_{12},\overline{A},\overline{S})[v_2/y][v/x][v/x]$.
Furthermore, this is $(\mathit{Inst}(T_{12},\overline{A},\overline{S})[v_2/y])[v/x]$.
Finally, we note that $v_1'\ v_2' = (v_1\ v_2)[v/x]$.
Thus, the derivation from T-App does indeed satisfy the goal.

Case: T-Sub. $$\frac{x{:}T,\Gamma \vdash_n e :: T'' \qquad x{:}T,\Gamma \vdash_n T'' \sqsubseteq T'}{x{:}T,\Gamma \vdash_n e :: T'}$$

By IH (4), $\Gamma[v/x] \vdash_{n+1} e[v/x] :: T''[v/x]$.
By IH (3), $\Gamma[v/x] \vdash_{n+1} T''[v/x] :: T'[v/x]$.
By T-Sub, $\Gamma[v/x] \vdash_{n+1} e[v/x] :: T'[v/x]$.

Cases: T-Let, T-If, T-Extend.

By IH on the premises and original rule to conclude.

Cases: T-Fold, T-Unfold.

By IH on the premises and original rule to conclude. □

In the following lemma we lift instantiation to judgements in the obvious way. For example, we write $\mathit{Inst}((\Gamma \vdash_n e :: T'),A,T)$ to mean $\mathit{Inst}(\Gamma,A,T) \vdash_n e :: \mathit{Inst}(T',A,T)$.

Lemma A.2.12 (Type Substitution)**.** *Let* $\vdash T$.

1. *If* $A,\Gamma \Rightarrow_n p$, *then* $\mathit{Inst}((\Gamma \Rightarrow_n p),A,T)$.

2. *If* $A,\Gamma \vdash_n U_1 <: U_2$, *then* $\mathit{Inst}((\Gamma \vdash_n U_1 <: U_2),A,T)$.

3. *If* $A,\Gamma \vdash_n T_1 \sqsubseteq T_2$, *then* $\mathit{Inst}((\Gamma \vdash_n T_1 \sqsubseteq T_2),A,T)$.

4. *If* $A,\Gamma \vdash_n e :: T'$, *then* $\mathit{Inst}((\Gamma \vdash_n e :: T'),A,T)$.

Proof. By mutual induction. Even (1) is straightforward, since type variables in the environment play no role in the embedding of formulas into the logic (they are embedded as *true*). □

Lemma A.2.13 (Canonical Forms)**.** *Suppose* $\vdash_n v :: S$.

1. *If* $S = \mathit{Bool}$, *then* $v = \texttt{true}$ *or* $v = \texttt{false}$.
2. *If* $S = \{y \,|\, y :: \forall\overline{A}.\ x{:}T_1 \to T_2\}$, *then either*
   (a) $v = \lambda x.e$ *and* $\overline{A}, x{:}T_1 \vdash_n e :: T_2$, *or*
   (b) $v = c$ *and for all* v', n, *and* $\overline{T}$ *s.t.* $\vdash_n v' :: \mathit{Inst}(T_1, \overline{A}, \overline{T})$, $\delta(c, v')$ *is defined and* $\vdash_n \delta(c, v') :: \mathit{Inst}(T_2, \overline{A}, \overline{T})[v'/x]$.

Proof of (1). By Satisfiable Typing, $\mathcal{I}_{n+1} \models \mathit{tag}(v) =$ "$\texttt{boolean}$". Thus, v is either $\texttt{true}$ or $\texttt{false}$. □

Proof of (2). By Satisfiable Typing, $\mathcal{I}_{n+1} \models v :: \forall\overline{A}.\ x{:}T_1 \to T_2$. The goal follows by Type Predicate Interpretation and Constant Types (App). □

We are now ready to prove type soundness of System D$_n$. Soundness of the basic type system, which is the system at level zero and is used for type checking source programs, follows as a corollary. Compared to the proof outline in §2.4, we prove progress and preservation together.

Theorem A.2.14 (Stratified System D$_n$ Type Soundness)**.**
If $\vdash_n e :: T$, *then either* e *is a value or* $e \hookrightarrow e'$ *and* $\vdash_{n+1} e' :: T$.

Proof. By induction on the typing derivation.

Case: T-Var.

Impossible, since the typing environment is empty.

Cases: T-Const, T-Extend, T-Fun, T-Fold.

Immediate, since e is a value.

Cases: T-Unfold.

By Satisfiable Typing and Type Predicate Interpretation, e is a value.

Case: T-If.
$$\frac{\vdash_n v :: \mathit{Bool} \qquad v = \texttt{true} \vdash_n e_1 :: S \qquad v = \texttt{false} \vdash_n e_2 :: S}{\vdash_n \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: S}$$

By Canonical Forms, there are two cases.

Subcase: $v = \texttt{true}$.

By E-IfTrue, $e' = e_1$.
$\mathit{Valid}(\texttt{true} = \texttt{true})$, so by Strengthening, $\vdash_n e_1 :: S$.
By Lifting, $\vdash_{n+1} e_1 :: S$.

Subcase: $v = \texttt{false}$.

By E-IfFalse, $e' = e_2$.
$\mathit{Valid}(\texttt{false} = \texttt{false})$, so by Strengthening, $\vdash_n e_2 :: S$.
By Lifting, $\vdash_{n+1} e_2 :: S$.

Case: T-App. $$\frac{\vdash_n v_1 :: \{y \,|\, y :: \forall\overline{A}.\ x\!:\!T_{11} \to T_{12}\} \qquad \vdash_n v_2 :: \mathit{Inst}(T_{11},\overline{A},\overline{T})}{\vdash_n [\overline{T}]\ v_1\ v_2 :: \mathit{Inst}(T_{12},\overline{A},\overline{T})[v_2/x]}$$

By Canonical Forms, there are two cases.

Subcase: $v_1 = \lambda x.e_0$ and $\overline{A}, x\!:\!T_{11} \vdash_n e_0 :: T_{12}$. .

By Type Substitution and Value Substitution, $\vdash_{n+1} e_0[v_2/x] :: \mathit{Inst}(T_{12},\overline{A},\overline{T})[v_2/x]$.
This concludes the subcase, since by E-App, $e' = e_0[v_2/x]$.

Subcase: $v_1 = c$.

Since $\vdash_n v_2 :: T_{11}$, we also have that $\delta(c,v_2)$ is defined and $\vdash_n \delta(c,v_2) :: T_{12}[v_2/x]$.
By Lifting, $\vdash_{n+1} \delta(c,v_2) :: T_{12}[v_2/x]$.
This concludes the subcase, since by E-Delta, $e' = \delta(c,v_2)$.

Case: T-Let. $$\frac{\vdash T_1 \qquad \vdash_n e_1 :: T_1 \qquad \vdash T_2 \qquad x\!:\!T_1 \vdash_n e_2 :: T_2}{\vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: T_2}$$

By the IH, there are two cases.

Subcase: e_1 is a value v.

By E-Let, $e' = e_2[v/x]$.
By Value Substitution, $\vdash_{n+1} e_2[v/x] :: T_2[v/x]$.
Since $\vdash T_2$, x does not appear free in T_2, so $T_2[v/x] = T_2$.

Subcase: $e_1 \hookrightarrow e_1'$ and $\vdash_{n+1} e_1' :: T$.

By E-Compat, $e' = \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$.
By Lifting, $x\!:\!T \vdash_{n+1} e_2 :: T_2$.
By T-Let, $\vdash_{n+1} \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 :: T_2$.

Case: T-Sub. $$\frac{\vdash_n v :: T' \qquad \vdash_n T' \sqsubseteq T \qquad \vdash T}{\vdash_n v :: T}$$

By IH, Lifting, and T-Sub. □

We write $e \hookrightarrow^* e'$ to denote the multi-step evaluation relation (*i.e.* the reflexive, transitive closure of the single-step evaluation relation $e_1 \hookrightarrow e_2$).

Corollary A.2.15 (System D$_n$ Type Soundness)**.**

If $\vdash_* e :: T$*, then either* e *diverges or* $e \hookrightarrow^* v$ *and* $\vdash_* v :: e$*.*

Corollary A.2.16 (System D Type Soundness)**.**

If $\vdash_0 e :: T$*, then either* T *diverges or* $e \hookrightarrow^* v$ *and* $\vdash_* v :: T$*.*

A.3 Soundness of Extensions

We now prove the soundness of the subtyping extensions presented in Chapter 4, starting with several lemmas that help manipulate the structure of formulas. For clarity, we omit polymorphic type variables on all arrows in the rest of this section.

Lemma A.3.1 (Guarded Derivations)**.** *Let* $\Gamma_1 = y\!:\!T_1, \Gamma$ *and* $\Gamma_2 = y\!:\!T_2, \Gamma$*.*

1. *If* $\Gamma_1 \Rightarrow_n q$*, then* $\Gamma_2 \Rightarrow_n T_1(y) \Rightarrow q$*.*
2. *If* $\Gamma_1 \vdash_n \{x \,|\, p_1\} \sqsubseteq \{x \,|\, p_2\}$*, then* $\Gamma_2 \vdash_n \{x \,|\, p_1\} \sqsubseteq \{x \,|\, T_1(y) \Rightarrow p_2\}$*.*
3. *If* $\Gamma_1 \vdash_n e :: \{x \,|\, q\}$*, then* $\Gamma_2 \vdash_n e :: \{x \,|\, T_1(y) \Rightarrow q\}$*.*

Proof. By induction. For (1), from the hypothesis $\mathit{Embed}(y\!:\!T_1, \Gamma_1) = T_1(y) \wedge \mathit{Embed}(\Gamma_1)$ is sufficient to discharge the goal q, so $\mathit{Embed}(\Gamma_2) \wedge T_1(y)$ is also. □

Lemma A.3.2 (Shuffling)**.** *Let* $\Gamma_1 = y\!:\!\{x \,|\, p_1 \vee p_2\}, \Gamma$ *and* $\Gamma_2 = y\!:\!\{x \,|\, p_2 \vee p_1\}, \Gamma$*.*

1. *If* $\Gamma_1 \Rightarrow_n p$*, then* $\Gamma_2 \Rightarrow_n p$*.*
2. *If* $\Gamma_1 \vdash_n U_1 <: U_2$*, then* $\Gamma_2 \vdash_n U_1 <: U_2$*.*
3. *If* $\Gamma_1 \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma_2 \vdash_n T_1 \sqsubseteq T_2$*.*
4. *If* $\Gamma_1 \vdash_n e :: T$*, then* $\Gamma_2 \vdash_n e :: T$*.*

Proof. By induction. □

Lemma A.3.3 (And Introduction)**.**

1. *If* $\Gamma \vdash_n T \sqsubseteq \{x \,|\, p\}$ *and* $\Gamma \vdash_n T \sqsubseteq \{x \,|\, q\}$*, then* $\Gamma \vdash_n T \sqsubseteq \{x \,|\, p \wedge q\}$*.*
2. *If* $\Gamma \vdash_n e :: \{x \,|\, p\}$ *and* $\Gamma \vdash_n e :: \{x \,|\, q\}$*, then* $\Gamma \vdash_n e :: \{x \,|\, p \wedge q\}$*.*

Proof. By induction. □

Lemma A.3.4 (Conjunction Weakening)**.** *Let* $\Gamma_1 = y\!:\!\{x \,|\, p_1\}, \Gamma$ *and* $\Gamma_2 = y\!:\!\{x \,|\, p_1 \wedge p_2\}, \Gamma$*.*

1. *If* $\Gamma_1 \Rightarrow_n q$*, then* $\Gamma_2 \Rightarrow_n q$*.*
2. *If* $\Gamma_1 \vdash_n U_1 <: U_2$*, then* $\Gamma_2 \vdash_n U_1 <: U_2$*.*

3. *If* $\Gamma_1 \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma_2 \vdash_n T_1 \sqsubseteq T_2$.

4. *If* $\Gamma_1 \vdash_n e :: T$*, then* $\Gamma_2 \vdash_n e :: T$.

Proof. By induction. □

Lemma A.3.5 (Reverse Or Elimination)**.** *Let* $\Gamma_1 = y\!:\!\{x \mid p_1\}, \Gamma$ *and* $\Gamma_2 = y\!:\!\{x \mid p_2\}, \Gamma$ *and* $\Gamma_3 = y\!:\!\{x \mid p_1 \vee p_2\}, \Gamma$.

1. *If* $\Gamma_1 \Rightarrow_n q$ *and* $\Gamma_2 \Rightarrow_n q$*, then* $\Gamma_3 \Rightarrow_n q$.

2. *If* $\Gamma_1 \vdash_n U_1 <: U_2$ *and* $\Gamma_2 \vdash_n U_1 <: U_2$*, then* $\Gamma_3 \vdash_n U_1 <: U_2$.

3. *If* $\Gamma_1 \vdash_n T_1 \sqsubseteq T_2$ *and* $\Gamma_2 \vdash_n T_1 \sqsubseteq T_2$*, then* $\Gamma_3 \vdash_n T_1 \sqsubseteq T_2$.

4. *If* $\Gamma_1 \vdash_n e :: T$ *and* $\Gamma_2 \vdash_n e :: T$*, then* $\Gamma_3 \vdash_n e :: T$.

Proof. By induction, assuming the presence of the I-Dnf rule. □

We now have all the formula-manipulating lemmas we need to prove soundness of the meet and join operators. Note that of the lemmas above, only Reverse Or Elimination requires that I-Dnf be included in the subtyping rules.

The following lemma states three properties of the meet operator. Only the first is required for soundness (in particular, for the Satisfiable Typing lemma). The remaining two are similar to the usual properties one would expect of a meet operator: that it computes a lower bound and that it computes the greatest lower bound. Proving the last property depends on Reverse Or Elimination, so the meet can only be considered maximal when I-Dnf is included in the system.

Theorem A.3.6 (Sound Meet). *Suppose* $U_1 \sqcap U_2 = U$.

1. *If* $\mathcal{I}_n \models w :: U_1$ *and* $\mathcal{I}_n \models w :: U_2$ *then* $\mathcal{I}_n \models w :: U$.

2. $\vdash_n U <: U_1$ *and* $\vdash_n U <: U_2$.

3. *If* $\vdash_n U' <: U_1$ *and* $\vdash_n U' <: U_2$*, then* $\vdash_n U' <: U$.

Proof. We consider only the case for arrows; the other cases are straightforward.

$$\begin{aligned} U_1 &= x\!:\!\{y \mid p_1\} \to \{y \mid q_1\} \\ U_2 &= x\!:\!\{y \mid p_2\} \to \{y \mid q_2\} \\ U &= x\!:\!\{y \mid p_1 \vee p_2\} \to \{y \mid p_1(x) \Rightarrow q_1 \wedge p_2(x) \Rightarrow q_2\} \end{aligned}$$

Proof of (1). By Type Predicate Interpretation, there are two cases.

Case: $w = \lambda x.e$, $x\!:\!\{y \,|\, p_1\} \vdash_{n-1} e :: \{y \,|\, q_1\}$, and $x\!:\!\{y \,|\, p_2\} \vdash_{n-1} e :: \{y \,|\, q_2\}$.

By Guarded Derivations, $x\!:\!\{y \,|\, p_1 \vee p_2\} \vdash_{n-1} e :: \{y \,|\, p_1(x) \Rightarrow q_1\}$.
By Guarded Derivations, $x\!:\!\{y \,|\, p_2 \vee p_1\} \vdash_{n-1} e :: \{y \,|\, p_2(x) \Rightarrow q_2\}$.
By Shuffling, $x\!:\!\{y \,|\, p_1 \vee p_2\} \vdash_{n-1} e :: \{y \,|\, p_2(x) \Rightarrow q_2\}$.
By And Introduction, $x\!:\!\{y \,|\, p_1 \vee p_2\} \vdash_{n-1} e :: \{y \,|\, p_1(x) \Rightarrow q_1(x) \wedge p_2(x) \Rightarrow q_2\}$.
By Type Predicate Interpretation, $\mathcal{I}_n \models w :: U$.

Case: $w = c$. Omitted. □

Proof of (2).

We will prove $\vdash_n U <: U_1$; proving $\vdash_n U <: U_2$ is similar.
By U-Arrow, we must show:
i. $\vdash_n \{y \,|\, p_1\} \sqsubseteq \{y \,|\, p_1 \vee p_2\}$, and
ii. $x\!:\!\{y \,|\, p_1\} \vdash_n \{y \,|\, p_1(x) \Rightarrow q_1 \wedge p_2(x) \Rightarrow q_2\} \sqsubseteq \{y \,|\, q_1\}$.
The former follows from S-Refine and I-Valid.
For the latter, consider an arbitrary clause of q_1, called Q.
First, we note that $\mathit{Valid}(p_1(x) \wedge (p_1(x) \Rightarrow q_1 \wedge \cdots) \Rightarrow q_1)$.
Because q_1 implies all of its clauses, $\mathit{Valid}(p_1(x) \wedge (p_1(x) \Rightarrow q_1 \wedge \cdots) \Rightarrow Q)$.
By I-Valid, $x\!:\!\{y \,|\, p_1\}, (p_1(x) \Rightarrow q_1 \wedge p_2(x) \Rightarrow q_2) \Rightarrow_n Q$.
Every clause Q of q_1 is discharged.
Therefore, $x\!:\!\{y \,|\, p_1\} \vdash_n \{y \,|\, p_1(x) \Rightarrow q_1 \wedge p_2(x) \Rightarrow q_2\} \sqsubseteq \{y \,|\, q_1\}$. □

Proof of (3).

By inversion on syntactic subtyping, $U' = x\!:\!\{y \,|\, p_3\} \rightarrow \{y \,|\, q_3\}$.
Furthermore, by inversion on the hypotheses, we have:
a. $\vdash_n \{y \,|\, p_1\} \sqsubseteq \{y \,|\, p_3\}$
b. $\vdash_n \{y \,|\, p_2\} \sqsubseteq \{y \,|\, p_3\}$
c. $x\!:\!\{y \,|\, p_1\} \vdash_n \{y \,|\, q_3\} \sqsubseteq \{y \,|\, q_1\}$
d. $x\!:\!\{y \,|\, p_2\} \vdash_n \{y \,|\, q_3\} \sqsubseteq \{y \,|\, q_2\}$
To prove $\vdash_n U' <: U$, by U-Arrow we must show:
i. $\vdash_n \{y \,|\, p_1 \vee p_2\} \sqsubseteq \{y \,|\, p_3\}$, and
ii. $x\!:\!\{y \,|\, p_1 \vee p_2\} \vdash_n \{y \,|\, q_3\} \sqsubseteq \{y \,|\, p_1(x) \Rightarrow q_1 \wedge p_2(x) \Rightarrow q_2\}$.
The former follows from applying Reverse Or Elimination to (a) and (b).
The latter follows from Guarded Derivations, Shuffling, and And Introduction on (c) and (d). □

We prove three properties of the join operator analogous to those for the meet operator. The proof that the join computes the least upper bound depends on the presence of I-Dnf in the system.

Theorem A.3.7 (Sound Join). *Suppose $U_1 \sqcup U_2 = U$.*

1. *If $\mathcal{I}_n \models w :: U_1$ or $\mathcal{I}_n \models w :: U_2$, then $\mathcal{I}_n \models w :: U$.*
2. *$\vdash_n U_1 <: U$ and $\vdash_n U_2 <: U$.*
3. *If $\vdash_n U_1 <: U'$ and $\vdash_n U_2 <: U'$, then $\vdash_n U <: U'$.*

Proof. We consider only the case for arrows; the other cases are straightforward.

$$\begin{aligned} U_1 &= x\!:\!\{y \mid p_1\} \to \{y \mid q_1\} \\ U_2 &= x\!:\!\{y \mid p_2\} \to \{y \mid q_2\} \\ U &= x\!:\!\{y \mid p_1 \wedge p_2\} \to \{y \mid q_1 \vee q_2\} \end{aligned}$$

Proof of (1). By Type Predicate Interpretation, there are two cases.

Case: $w = \lambda x.e$, $x\!:\!\{y \mid p_1\} \vdash_{n-1} e :: \{y \mid q_1\}$, and $x\!:\!\{y \mid p_2\} \vdash_{n-1} e :: \{y \mid q_2\}$.

We consider the former case; the latter case is similar.
By Conjunction Weakening, $x\!:\!\{y \mid p_1 \wedge p_2\} \vdash_{n-1} e :: q_1$.
By T-Sub, $x\!:\!\{y \mid p_1 \wedge p_2\} \vdash_{n-1} e :: q_1 \vee q_2$.
By Type Predicate Interpretation, $\mathcal{I}_n \models w :: U$.

Case: $w = c$. Omitted. □

Proof of (2).

By S-Refine and I-Valid, $\vdash_n \{y \mid p_1 \wedge p_2\} \sqsubseteq \{y \mid p_1\}$.
By S-Refine and I-Valid, $x\!:\!\{y \mid p_1 \wedge p_2\} \vdash_n \{y \mid q_1\} \sqsubseteq \{y \mid q_1 \vee q_2\}$.
By U-Arrow, $\vdash_n U_1 <: U$.
The proof for $\vdash_n U_2 <: U$ is similar. □

Proof of (3).

By inversion on syntactic subtyping, $U' = x\!:\!\{y \mid p_3\} \to \{y \mid q_3\}$.
Furthermore, by inversion on the hypotheses, we have:

a. $\vdash_n \{y \mid p_3\} \sqsubseteq \{y \mid p_1\}$
b. $\vdash_n \{y \mid p_3\} \sqsubseteq \{y \mid p_2\}$
c. $x\!:\!\{y \mid p_3\} \vdash_n \{y \mid q_1\} \sqsubseteq \{y \mid q_3\}$
d. $x\!:\!\{y \mid p_3\} \vdash_n \{y \mid q_2\} \sqsubseteq \{y \mid q_3\}$

To prove $\vdash_n U <: U'$, by U-Arrow we must show:

i. $\vdash_n \{y \mid p_3\} \sqsubseteq \{y \mid p_1 \vee p_2\}$, and

ii. $x\!:\!\{y \mid p_3\} \vdash_n \{y \mid q_1 \vee q_2\} \sqsubseteq \{y \mid p_3\}$.

The former goal follows from And Introduction on (a) and (b).

The latter one from Reverse Or Elimination on (c) and (d). □

Lemma (Satisfiable Typing)**.**

Proof. Updating this proof for the new subtyping rules is simple. The case for I-HasType-Meet (resp. I-HasType-Join) is similar to I-HasType, except that the reasoning must also appeal to Sound Meet (1) (resp. Sound Join (1)) for the type term resulting from the meet (resp. join) operation used to combine type terms. The I-Dnf case is straightforward, since converting to DNF does not affect the satisfiability of a formula. □

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon Plotkin. Dynamic Typing in a Statically-Typed Language. In *Principles of Programming Languages (POPL)*, 1989.

[3] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A Linear Language with Locations. *Fundamenta Informaticae*, 2007.

[4] Alex Aiken, John Kodumal, Jeffrey S. Foster, and Tachio Terauchi. Checking and Inferring Local Non-Aliasing. In *Programming Language Design and Implementation (PLDI)*, 2003.

[5] Alexander Aiken, Edward Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Principles of Programming Languages (POPL)*, 1994.

[6] Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. Towards Type Inference for Javascript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. In *Computer Security Foundations (CSF)*, 2008.

[8] Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, 2004.

[9] Gavin Bierman, Andrew Gordon, Catalin Hritcu, and David Langworthy. Semantic Subtyping with an SMT Solver. In *International Conference on Functional Programming (ICFP)*, 2010.

[10] Viviana Bono and Kathleen Fisher. An Imperative, First-Order Calculus with Object Extension. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's Decidable About Arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.

[12] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded Polymorphism for Object-oriented Programming. In *Functional Programming Languages and Architecture (FPCA)*, 1989.

[13] Luca Cardelli. On Understanding Types, Data Abstraction, and Polymorphism. In *Computing Surveys*, 1985.

[14] Luca Cardelli and John C. Mitchell. *Operations On Records*. MIT Press, 1994.

[15] Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI)*, 1991.

[16] Ravi Chugh. A Fix for Dynamic Scope. In *Workshop on ML*, 2013.

[17] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged Information Flow for JavaScript. In *Programming Language Design and Implementation (PLDI)*, 2009.

[18] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 1936.

[19] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 1940.

[20] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[21] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying Type Checking and Property Checking for Low-Level Code. In *Principles of Programming Languages (POPL)*, 2009.

[22] Patrick Cousot. Abstract Interpretation. *ACM Computing Surveys*, 1996.

[23] Douglas Crockford. *JavaScript: The Good Parts*. Yahoo! Press, 2008.

[24] Haskell Curry and Robert Feys. *Combinatory Logic, Volume 1*. North Holland, 1958.

[25] Luis Damas and Robin Milner. Principal Type Schemes for Functional Programs. In *Principles of Programming Languages (POPL)*, 1982.

[26] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005.

[27] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[28] Leonardo de Moura and Nikolaj Bjørner. Generalized, Efficient Array Decision Procedures. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2009.

[29] R. DeLine and M.A. Fähndrich. Enforcing High-level Protocols in Low-level Software. In *Programming Language Design and Implementation (PLDI)*, 2001.

[30] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated Error Diagnosis Using Abductive Inference. In *Programming Language Design and Implementation (PLDI)*, 2012.

[31] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007.

[32] ECMA. TC-39 Committee. www.ecmascript.org/community.php.

[33] Manuel Fahndrich and Rob DeLine. Adoption and Focus: Practical Linear Types for

Imperative Programming. In *Programming Language Design and Implementation (PLDI)*, 2002.

[34] Asger Felthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-Supported Refactoring for JavaScript. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[35] Asger Felthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *International Conference of Software Engineering (ICSE)*, 2013.

[36] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[37] Cormac Flanagan. Hybrid Type Checking. In *Principles of Programming Languages (POPL)*, 2006.

[38] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Programming Language Design and Implementation (PLDI)*, 1993.

[39] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive Type Qualifiers. In *Programming Language Design and Implementation (PLDI)*, 2002.

[40] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully Abstract Compilation to JavaScript. In *Principles of Programming Languages (POPL)*, 2013.

[41] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Programming Language Design and Implementation (PLDI)*, 1991.

[42] Michael Furr, Jong-hoon (David) An, Jeffrey Foster, and Michael Hicks. Static Type Inference for Ruby. In *Symposium on Applied Computing (SAC)*, 2009.

[43] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static Type Inference for Ruby. In *Symposium on Applied Computing (SAC)*, 2009.

[44] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Principles of Programming Languages (POPL)*, 2012.

[45] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A Lambda Calculus of Objects with Self-Inflicted Extension. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1998.

[46] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de L'arithmétique D'order Supérieur*. PhD thesis, Université Paris VII, 1972.

[47] Google. Closure Gadgets. http://www.google.com/ig/directory.

[48] Google. Closure Library. https://developers.google.com/closure/library.

[49] Google. V8 Benchmark. http://v8.googlecode.com/svn/data/benchmarks/.

[50] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified Security for Browser Extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[51] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[52] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*, 2011.

[53] Shu-yu Guo and Brian Hackett. Fast and Precise Hybrid Type Inference for JavaScript. In *Programming Language Design and Implementation (PLDI)*, 2012.

[54] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[55] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 1992.

[56] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral Interface Specification Languages. *ACM Computing Surveys*, 2012.

[57] Phillip Heidegger and Peter Thiemann. Recency Types for Analyzing Scripting Languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[58] Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Functional Programming Languages and Computer Architecture*, 1995.

[59] Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 1969.

[60] Atsushi Igarashi. Union Types for Object-Oriented Programming. In *Symposium on Applied Computing (SAC)*, 2004.

[61] ECMA International. *ECMAScript Language Specification, ECMA-262, 3rd ed.* 1999.

[62] Samin Ishtiag and Peter O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Principles of Programming Languages (POPL)*, 2001.

[63] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 2009.

[64] Ranjit Jhala, Rupak Majumdar, and Ru-Gang Xu. State of the Union: Type Inference via Craig Interpolation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[65] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based Data Structure Verification. In *Programming Language Design and Implementation (PLDI)*, 2009.

[66] Kenneth Knowles and Cormac Flanagan. Compositional Reasoning and Decidable Checking for Dependent Contract Types. In *Programming Languages Meets Program Verification (PLPV)*, 2009.

[67] Kenneth Knowles and Cormac Flanagan. Hybrid Type Checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.

[68] Raghavan Komondoor, Ganesan Ramalingam, Satish Chandra, and John Field. Dependent Types for Program Understanding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[69] Xavier Leroy. Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant. In *Principles of Programming Languages (POPL)*, 2006.

[70] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[71] John McCarthy. Towards a Mathematical Science of Computation. In *International Federation for Information Processing (IFIP) Congress*, 1962.

[72] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 1978.

[73] Aleksander Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism, and Separation. *Journal of Functional Programming*, 2008.

[74] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1979.

[75] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic Typing with Dependent Types. In *IFIP TCS*, 2004.

[76] Benjamin C. Pierce. Programming with Intersection Types, Union Types, and Polymorphism. Technical report, Carnegie Mellon University, 1991.

[77] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[78] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Principles of Programming Languages (POPL)*, 1998.

[79] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based Verification of JavaScript Sandboxing. In *USENIX Security*, 2011.

[80] Ravi Chugh. Dependent JavaScript. http://github.com/ravichugh/djs.

[81] Didier Rémy. Typechecking Records and Variants in a Natural Extension of ML. In *Principles of Programming Languages (POPL)*, 1989.

[82] Didier Rémy. Programming Objects with ML-ART: An Extension to ML with Abstract and Record Types. In *Theoretical Aspects of Computer Software (TACS)*, 1994.

[83] John C. Reynolds. Towards a Theory of Type Structure. In *Colloque sur la Programmation*, 1974.

[84] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, 2002.

[85] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid Types. In *Programming Language Design and Implementation (PLDI)*, 2008.

[86] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-Level Liquid Types. In *Principles of Programming Languages (POPL)*, 2010.

[87] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 1984.

[88] Jeremy Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (SFP)*, 2006.

[89] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[90] Jeremy Siek and Philip Wadler. Threesomes, With and Without Blame. In *Principles of Programming Languages (POPL)*, 2010.

[91] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *European Symposium on Programming (ESOP)*, 2000.

[92] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class State Change in Plaid. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

[93] SunSpider. JavaScript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[94] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *European Symposium on Programming (ESOP)*, 2010.

[95] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Ben Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *Programming Language Design and Implementation (PLDI)*, 2013.

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

[97] The Python Software Foundation. Python 3.2 Standard Library. http://python.org/.

[98] Peter Thiemann. Towards a Type System for Analyzing Javascript Programs. In *European Symposium on Programming (ESOP)*, 2005.

[99] Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *International Conference on Functional Programming (ICFP)*, 2010.

[100] Mads Torgersen, Christian Plesner Hansen, and Erik Ernst. Adding Wildcards to the Java Programming Language. In *Journal of Object Technology*, 2004.

[101] Philip Wadler. Linear Types Can Change the World. In *Programming Concepts and Methods*, 1990.

[102] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In *Types in Compilation (TIC)*. 2000.

[103] Mitchell Wand. Complete Type Inference for Simple Objects. In *Logic in Computer Science (LICS)*, 1987.

[104] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Principles of Programming Languages (POPL)*, 1999.

[105] Tian Zhao. Polymorphic Type Inference for Scripting Languages with Object Extensions. In *Dynamic Languages Symposium (DLS)*, 2011.