Trust, but Verify: Two-Phase Typing for Dynamic Languages

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala

Univeristy of California, San Diego
La Jolla, CA, 92093, USA
{pvekris,blcosman,rjhala}@cs.ucsd.edu

Abstract

A key challenge when statically typing so-called dynamic languages is the ubiquity of *value-based overloading*, where a given function can dynamically reflect upon and behave according to the types of its arguments. Thus, to establish basic types, the analysis must reason precisely about values, but in the presence of higher-order functions and polymorphism, this reasoning itself can require basic types. In this paper we address this chicken-and-egg problem by introducing the framework of two-phased typing. The first "trust" phase performs classical, i.e. flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with DEAD-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase uses refinement typing, a flow- and path-sensitive analysis, that decorates the first phase's types with logical predicates to track value relationships and thereby verify the casts and establish other correctness properties for dynamically typed languages.

1998 ACM Subject Classification D.3.3 [Programming Languages]: Language Constructs and Features – Constraints, Polymorhpism; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs – Assertions, Pre- and post-conditions; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs – Type structure

Keywords and phrases Dynamic Languages, Type Systems, Refinement Types, Intersection Types, Overloading

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

Higher-order constructs are increasingly adopted in *dynamic scripting* languages, as they facilitate the production of clean, correct and maintainable code. Consider, for example, the following (first-order) JavaScript function

```
function minIndexFO(a) {
  if (a.length ≤ 0)
    return -1;
  var min = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] < a[min])
      min = i;
  }
  return min;
}
```

which computes the index of the minimum value in the array a by looping over the array, updating the min value with each index i whose value a[i] is smaller than the "current" a[min]. Modern dynamic languages let programmers factor the looping pattern into a

© Panagiotis Vekris, Benjamin Cosman, Ranjit Jhala;
licensed under Creative Commons License CC-BY

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}

function reduce(a, f, x) {
  if (arguments.length === 3)
    return $reduce(a, f, x);
  return $reduce(a, f, a[0]);
}
```

```
function minIndex(a) {
  if (a.length ≤ 0)
    return -1;
  function step(min, cur, i) {
    return cur < a[min] ? i:min;
  }
  return reduce(a, step, 0);
}
```

Figure 1 Computing the minimum-valued index with Higher-Order Functions

higher-order `$reduce` function (Figure 1), which frees them from manipulating indices and thereby prevents the attendant "off-by-one" mistakes. Instead, the programmer can compute the minimum index by supplying an appropriate `f` to `reduce` as in `minIndex` shown at the right of Figure 1.

This trend towards abstraction and reuse poses a challenge to static program analyses: *how to precisely trace value relationships across higher-order functions and containers?* A variety of dataflow- or abstract interpretation- based analyses could be used to verify the safety of array accesses in `minIndexFO` by inferring the loop invariant that `i` and `min` are between `0` and `a.length`. Alas, these analyses would fail on `minIndex`. The usual methods of procedure summarization apply to first-order functions, and it is not clear how to extend higher-order analyses like CFA to track the *relationships* between the values and closures that flow to `$reduce`.

An Approach: Refinement Types. Refinement types [31] hold the promise of a precise and compositional analysis for higher-order functions. Here, *basic* types are decorated with *refinement* predicates that constrain the values inhabiting the type. For example, we can define

```
type idx<x> = {v:number | 0 ≤ v && v < len(x) }
```

to denote the set of valid indices for an array `x` and can be used to type `$reduce` as

```
$reduce :: <A,B>(a: A[], f: (B,A,idx<a>) ⇒ B, x: B) ⇒ B
```

The above type is a precise *relational summary* of the behavior of `$reduce`: the higher-order `f` is only invoked with valid indices for `a`. Consequently, `step` is only called with valid indices for `a`, which ensures array safety.

Problem: Value-based Overloading. A main attraction of dynamic languages is *value-based overloading*, where syntactic entities (*e.g.* variables) may be bound to multiple types at run-time, and furthermore, computations may be customized to particular types, by reflecting on the values bound to variables. For example, it is common to simplify APIs by overloading the `reduce` function to make the initial value `x` optional; when omitted, the first array element `a[0]` is used instead (Figure 1). Here, `reduce` really has *two* different function types: one with 3 parameters and another one with 2. Furthermore, `reduce` *reflects* on the size of `arguments` to select the behavior appropriate to the calling context.

Value-based overloading conflicts with a crucial prerequisite for refinements, namely that the language possesses an *unrefined* static type system that provides basic invariants about values which can then be refined using logical predicates. Unfortunately, as shown by `reduce`, to soundly establish basic typing we must reason about the logical relationships between values, which is exactly the problem we wished to solve via refinement typing. In other words, value-based overloading creates a chicken-and-egg problem: refinements require us to first establish basic typing, but the latter itself requires reasoning about values (and hence, refinements!).

Solution: Trust but Verify. We introduce *two-phased typing*, a new strategy for statically analyzing dynamic languages. The key insight is that we can completely decouple reasoning about *basic* types and *refinements* into distinct phases by converting "type errors" from the first phase into "assertion failures" for the second. Two-phase typing starts with a source language where value-based overloading is specified using *intersections* and (untagged) *unions* of the different possible (run-time) types.

The first phase performs classical, *i.e.* flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with `DEAD`-casts which allow the first phase to proceed, *trusting* that the expressions have the casted types. In other words, the first phase *elaborates* [10] the source language with intersection and (untagged) union types, into a target ML-like language with classical products, (tagged) sums and `DEAD`-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase carries out *refinement*, *i.e.* flow- and path-sensitive inference, to decorate the basic types (from the first phase) with predicates that precisely track relationships about values, and uses the refinements to *verify* the casts and other properties, discharging the assumptions of the first phase.

For example, `reduce` is described as the intersection of two contexts, *i.e.* function types which take two and three parameters respectively. The trust-phase checks the body under both contexts (separately). In each context, one of the calls to `$reduce` is "ill-typed". In the context where the function takes two inputs, the call using `x` is undefined; when the function takes three inputs, there is a mismatch in the types of `f` and `a[0]`. Consequently, each ill-typed expression is wrapped with a *cast* which obliges the verify phase to prove that the call is dead code in that context, thereby verifying overloading in a cooperative manner.

Benefits. While it is possible to account for value-based overloading in a single phase, the currently known methods that do so are limited to the extremes of types and program logics. At one end, systems like Typed Racket [28] and Flow Typing [16] extend classical type systems to account for a fixed set of `typeof`-style tests, but cannot reason about general value tests (*e.g.* the size of `arguments`) that often appear in idiomatic code. At the other end, systems like System D [7] embed the typing relation in an expressive program logic, allowing general value tests, but give up on basic type structure, thereby sacrificing inference, causing a significant annotation overhead. In contrast, our approach separates the concerns of basic typing and reasoning about values, thereby yielding several concrete benefits by *modularizing* specification, verification and soundness.

- ***Specification:*** Instead of a fixed set of type-tests, two-phase typing handles complex value relationships which can be captured inside refinements in an expressive logic. Furthermore, the *expressiveness* of the basic type system and logics can be extended independently, *e.g.* to account for polymorphism, classes or new logical theories, directly yielding a more expressive specification mechanism.

```
neg :: (number, number) ⇒ number              var a = neg(1,1);    // OK
    ∧  (number, boolean) ⇒ boolean            var b = neg(0,true); // OK
function neg(flag, x) {                       var c = neg(0,1);    // ERR
  if (flag) return 0-x;                       var d = neg(1,true); // ERR
  return !x;
}
```

Figure 2 An example program with value-based overloading

- ***Verification:*** Two-phase typing enables the straightforward composition of simple type checkers (uncomplicated by reasoning about values) with program logics (relying upon the basic invariants provided by typing – *e.g.* the parametric polymorphism needed to verify `minIndex`). Furthermore, two-phase typing allows us to compose basic typing with abstract interpretation [23], which drastically lowers the annotation burden for using refinement types.
- ***Soundness:*** Finally, our elaboration-based approach makes it straightforward to establish soundness for two-phased typing. The first phase ignores values and refinements, so we can use classical methods to prove the elaborated target is "equivalent to" the source. The second phase uses standard refinement typing techniques on the well-typed elaborated target, and hence lets us directly reuse the soundness theorems for such systems [18] to obtain end-to-end soundness for two-phased typing.

Contributions. Concretely, in this paper we make the following contributions. First, we informally illustrate (§ 2) how two-phase typing lets us statically analyze dynamic, value-based overloading patterns drawn from real-world code, where, we empirically demonstrate, value-based overloading is ubiquitous. Second, we formalize two-phase typing using a core calculus, RSC, whose syntax and semantics are detailed in § 3. Third, we formalize the first phase (§ 4), which *elaborates* [10] a source language with value-based overloading into a target language with `DEAD`-casts in lieu of overloading. We prove that the elaborated target preserves the semantics of the source, *i.e.* the `DEAD`-casts fail iff the source would hit a type error at run time. Finally, we demonstrate how standard refinement typing machinery can be applied to the elaborated well-typed target (§ 5) to statically verify the `DEAD`-casts, yielding end-to-end soundness for our system.

2 Overview

We begin with an overview illustrating how we soundly verify value-based overloading using our novel two-phased approach.

2.1 Value-based Overloading

Consider the code in Figure 2. The function `neg` behaves as follows. When a `number` is passed as input, indicated by passing in a *non-zero*, *i.e.* "truthy" `flag`, the function flips its sign by subtracting the input from `0`. Instead, when a `boolean` is passed in, indicated by a *zero*, *i.e.* "falsy" `flag`, the function returns the boolean negation. Hence, the calls made to assign `a` and `b` are legitimate and should be statically accepted. However, the calls made to assign `c` and `d` lead to run-time errors (assuming we eschew implicit coercions), and hence, should be rejected.

File	#Funs	%Ovl	%Opt	%Any
box2d	529	0	3	**3**
ace	484	1	5	**6**
pixi	123	0	12	**12**
fabricjs	371	5	9	**13**
threejs	1022	1	24	**24**
leaflet	414	12	38	**41**
underscore	344	25	34	**45**
sugar	446	29	37	**48**
d3	475	43	17	**52**
jquery	226	52	31	**67**

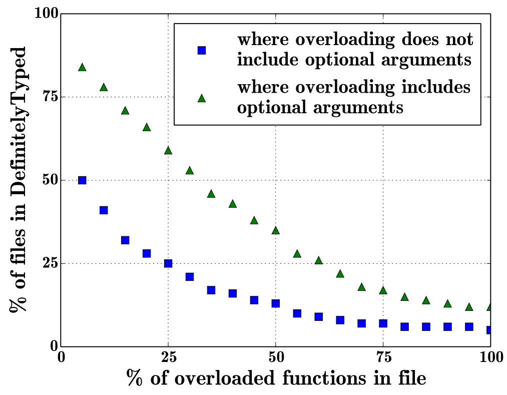

Figure 3 The prevalence of value-based overloading. **(L)** Libraries from the survey of Feldthaus *et al.* [12]: **#Funs** is the number of functions in the signature, **%Ovl** is %-functions with *multiple* signatures, **%Opt** is %-functions with *optional* arguments, and **%Any** is %-functions with either of these features. **(R)** Overloading across *all* files in Definitely Typed. A point (x, y) means y% of files have more than x% overloaded functions.

The function `neg` distils value-based overloading to its essence: a run-time test on one parameter's value is used to determine the type of, and hence the operation to be applied to, another value. Of course in JavaScript, one could use a single parameter and the `typeof` operator for this particular simple case, and design analyses targeted towards a fixed set of type tests, *e.g.* using variants of the `typeof` operator [28, 16]. However, arbitrary value tests – such as tests of the size of `arguments` shown in `reduce` in Figure 1 – can be and are used in practice. Thus, we illustrate the generality of the problem and our solution *without* using the `typeof` operator (which is a special case of our solution).

Prevalence of Value-based Overloading. The code from Figure 1 is not a pathological toy example. It is adapted from the widely used D3 visualization library. The advent of TypeScript makes it possible to establish the prevalence of value-based overloading in real-world libraries, as it allows developers to specify overloaded signatures for functions. (Even though TypeScript does not verify those signatures, it uses them as trusted interfaces for external JavaScript libraries and code completion.) The Definitely Typed repository [1] contains TypeScript interfaces for a large number of popular JavaScript libraries. We analyzed the TypeScript interfaces to determine the prevalence of value-based overloading. Intuitively, every function or method with multiple (overloaded) signatures or optional arguments has an implementation that uses value-based overloading.

Figure 3 summarizes the results of our study. On the left, we show the fraction of overloaded functions in the 10 benchmarks analyzed by Feldthaus *et al.* [12]. The data shows that over 25% of the functions in 4 of 10 libraries use value-based overloading, and an even larger fraction is overloaded in libraries like `jquery` and `d3`. On the right we summarize the occurrence of overloading across all the libraries in Definitely Typed. The data shows, for example, that in more than 25% of the libraries, *more than* 25% of the functions are overloaded with multiple types. The figure jumps to nearly 55% of functions if we also include optional arguments.

[1] `http://definitelytyped.org`

The signatures in Definitely Typed have not been soundly checked against[2] their implementations. Hence, it is possible that they mischaracterize the semantics of the actual code, but modulo this caveat, we believe the study demonstrates that value-based overloading is ubiquitous, and so to soundly and statically analyze dynamic languages, it is crucial that we develop techniques that can precisely and flexibly account for it.

2.2 Refinement Types

Types and Refinements. A basic refinement type T is a basic type, *e.g.* `number`, refined with a logical formula from an SMT decidable logic – for the purposes of this paper, the quantifier-free logic of uninterpreted functions and linear integer arithmetic (QF_UFLIA [25]). For example, `{v:number | v != 0}` describes the *subset* of numbers that are non-zero. We write A to abbreviate the trivially refined type $\{\nu : A \mid \mathit{true}\}$, *e.g.* `number` is an abbreviation for $\{\nu : \texttt{number} \mid \mathit{true}\}$.

Summaries: Function Types. We can specify the behavior of functions with refined function types, of the form

$$(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow T$$

where arguments are named x_i and have types T_i and the output is a T. In essence, the *input* types T_i specify the function's preconditions, and the *output* type T describes the postcondition. Furthermore, each input type and the output type can *refer to* the arguments x_i which yields precise function contracts. For example,

$$(x : 0 \leq x) \Rightarrow \{\nu : \texttt{number} \mid x < \nu\}$$

is a function type that describes functions that *require* a non-negative input, and *ensure* that the output is greater than the input.

Example. Returning to `neg` in Figure 2, we can define two refinements of `number`:

```
type tt = {v:number | v != 0}    // "truthy" numbers
type ff = {v:number | v  = 0}    // "falsy"  numbers
```

which are used to specify a refined type for `neg` shown on the left in Figure 4.

Problem: A Circular Dependency. While it is easy enough to specify a type signature, it is another matter to verify it, and yet another matter to ensure soundness. The challenge is that value-based overloading introduces a circular dependency between types and refinements. The soundness of basic types requires (*i.e.* is established by) the refinements, while the refinements themselves require (*i.e.* are attached to) basic types. In classical refinement systems like DML [31], basic types are established *without* requiring refinements. A classical refinement system is thus a conservative extension of the corresponding non-refined language, *i.e.* removing the refinements from a DML program, yields valid, well-typed ML. Unfortunately, value-based overloading removes this crucial property, posing a circular dependency between types and refinements.

[2] Feldthaus *et al.* [12] describe an effective but unsound inconsistency detector.

```
neg :: (tt, number) ⇒ number
     ∧ (ff, boolean) ⇒ boolean
function neg(flag, x) {
  if (flag) return 0-x;
  return !x;
}
```

```
var a = neg(1,1);     //OK
var b = neg(0,true);  //OK
var c = neg(0,1);     //ERR
var d = neg(1,true);  //ERR
```

```
neg#1 :: (tt, number) ⇒ number
function neg#1(flag, x) {
  if (flag) return 0-x;
  return !DEAD(x);
}
neg#2 :: (ff, boolean) ⇒ boolean
function neg#2(flag, x) {
  if (flag) return 0-DEAD(x);
  return !x;
}
var neg = (neg#1, neg#2);
```

```
var a = fst(neg)(1,1);     //OK
var b = snd(neg)(0,true);  //OK
var c = fst(neg)(0,1);     //ERR
var d = snd(neg)(1,true);  //ERR
```

Figure 4 Source program (l) and target (r) resulting from first phase elaboration.

Solution: Two-Phase Checking. We break the cycle by typing programs in two phases. In the first, we *trust* the basic types are correct and use them (ignoring the refinements) to elaborate source programs into a target overloading-free language. Inevitably, value-based overloading leads to "errors" when typing certain sub-expressions in the wrong context, *e.g.* subtracting a `boolean`-valued `x` from `0`. Instead of rejecting the program, the elaboration wraps ill-typed expressions with `DEAD`-casts, which are assertions stating the program is well-typed *assuming* those expressions are dead code. In the second phase we reuse classical refinement typing techniques to *verify* that the `DEAD`-casts are indeed unreachable, thereby discharging the assumptions made in the first phase.

2.3 Phase 1: Trust

The first phase *elaborates* the source program into an equivalent typed target language with two key properties: First, the target program is simply typed – *i.e.* has *no* union or intersection types, but just classical ML-style sums and products. Second, source-level type errors are elaborated to target-level `DEAD`-casts. The right side of Figure 4 shows the elaboration of the source from the left side. While we formalize the elaboration declaratively using a single judgment form (§ 4), it comprises two different steps. Critically, each step, and hence the entire first phase, is *independent* of the refinements – they are simply carried along unchanged.

A. Clone. In the first step, we create separate clones of each overloaded function, where each clone is assigned a single conjunct of the original overloaded type. For example, we create two clones `neg#1` and `neg#2` respectively typed using the two conjuncts of the original `neg`. The binder `neg` is replaced with a *tuple* of its clones. Finally, each use of `neg` extracts the appropriate element from the tuple before issuing the call.

Since the trust phase must be independent of refinements, the overload resolution in this step uses *only* the basic types at the call-site to determine which of the two clones to invoke. For example, in the assignment to `a`, the source call `neg(1,1)` – which passes in two `number` values, and hence, matches the first overload (conjunct) – is elaborated to the target call `fst(neg)(1,1)`. In the assignment to `d`, the source call `neg(1,true)` – which passes in a `number` and a `boolean`, and hence matches the second overload – is elaborated to the target call `snd(neg)(1,true)`, even though `1` does *not* have the refined type `ff`.

B: Cast. In the second step we check – using classical, unrefined type checking – that each clone adheres to its specified type. Unlike under usual intersection typing [22, 10], in our context these checks almost surely "fail". For example, `neg#1` *does not* type-check as the parameter `x` has type `number` and so we cannot compute `!x`. Similarly, `neg#2` fails because `x` has type `boolean` and so `0-x` is erroneous. Rather than reject the program, we wrap such failures with `DEAD`-casts. For example, the above occurrences of `x` elaborate to `DEAD(x)` on the right in Figure 4.

Intuitively, the *value relationships* established at the call-sites and guards ensure that the failures will not happen at run-time. However, recall that the first phase's goal is to decouple reasoning about types from reasoning about values. Hence, we just *trust* all the types but use `DEAD`-casts to *explicate* the value-relationship obligations that are needed to establish typing: namely that the `DEAD`-casts are indeed dead code.

2.4 Phase 2: Verify

The second phase takes as input the elaborated program emitted by the first phase, which is essentially a classical *well-typed* ML program with assertions and without any value-overloading. Hence, the second phase can use any existing program logic [14, 4], refinement typing [31, 18, 23, 2], or contracts & abstract interpretation [20] to check that the target's assertions never fail, which, we prove, ensures that the source is type-safe.

To analyze programs with closures, collections and polymorphism, (*e.g.* `minIndex` from Figure 1) we perform the second phase using the refinement types that are carried over unchanged by the elaboration process of the first phase. Intuitively, refinement typing can be viewed as a generalization of classical program logics where *assertions* are generalized to type bindings, and the rule of *consequence* is generalized as subtyping. While refinement typing is a previously known technique, to make the paper self-contained, we illustrate how the second phase verifies the `DEAD`-casts in Figure 4.

Refinement Type Checking. A refinement type checker works by building up an *environment* of type bindings that describe the machine state at each program point, and by checking that at each call-site, the actual argument's type is a refined *subtype* of the expected type for the callee, under the context described by the environment at that site. The subtyping relation for basic types is converted to a logical *verification condition* whose validity is checked by an SMT solver. The subtyping relation for *compound* types (*e.g.* functions, collections) is decomposed, via co- and contra-variant subtyping rules, into subtyping constraints over *basic* types, which can be discharged as above.

Typing `DEAD`-Casts. To use a standard refinement type checker for the second phase of verification, we only need to treat `DEAD` as a primitive operation with the refined type:

$$\texttt{DEAD} :: \forall A, B.(\{\nu\!:\!A \mid \mathit{false}\}) \Rightarrow B$$

That is, we assign `DEAD` the *precondition false* which states there are *no* valid inputs for it, *i.e.* that it should never be called (akin to `assert(false)` in other settings).

Environments. To verify `DEAD`-casts, the refinement type checker builds up an environment of type binders describing *variables* and *branch conditions* that are in scope at each program point. For example, the `DEAD` call in `neg#1`, has the environment:

$$\Gamma_1 \doteq \texttt{flag}\!:\!\texttt{tt},\ \texttt{x}\!:\!\texttt{number},\ \texttt{g}_1\!:\!\{\nu\!:\!\texttt{boolean} \mid \texttt{flag} = 0\} \qquad (1)$$

where the first two bindings are the function parameters, whose types are the input types. The third binding is from the "else" branch of the `flag` test, asserting the branch condition `flag` is "falsy" *i.e.* equals 0. At the `DEAD` call in `neg`#2 the environment is:

$$\Gamma_2 \doteq \texttt{flag}{:}\texttt{ff},\ \texttt{x}{:}\texttt{boolean},\ \texttt{g}_1{:}\{\nu{:}\texttt{boolean} \mid \texttt{flag} \neq 0\} \tag{2}$$

At the assignments to `a`, `b` and `c` the environments are respectively:

$$\Gamma_a \doteq \texttt{neg}{:}T_{\texttt{neg}} \tag{3}$$
$$\Gamma_b \doteq \Gamma_a,\ \texttt{a}{:}\texttt{number} \tag{4}$$
$$\Gamma_c \doteq \Gamma_b,\ \texttt{b}{:}\texttt{boolean} \tag{5}$$

where $T_{\texttt{neg}}$ abbreviates the *product* type of the (elaborated) tuple `neg`.

$$T_{\texttt{neg}} \doteq ((\texttt{tt}, \texttt{number}) \Rightarrow \texttt{number}) \times ((\texttt{ff}, \texttt{boolean}) \Rightarrow \texttt{boolean}) \tag{6}$$

Subtyping. At each function call-site, the refinement type system checks that the *actual* argument is indeed a subtype of the *expected* one. For example, the `DEAD` calls inside `neg`#1 and `neg`#2 yield the respective subtyping obligation:

$$\Gamma_1 \vdash \{\nu{:}\texttt{number} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu{:}\texttt{number} \mid \mathit{false}\} \tag{7}$$
$$\Gamma_2 \vdash \{\nu{:}\texttt{boolean} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu{:}\texttt{boolean} \mid \mathit{false}\} \tag{8}$$

The obligation states that the type of the argument `x` should be a subtype of the input type of `DEAD`. Similarly, at the assignments to `a`, `b` and `c` the first arguments generate the respective subtyping obligations:

$$\Gamma_a \vdash \{\nu{:}\texttt{number} \mid \nu = 1\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu \neq 0\} \tag{9}$$
$$\Gamma_b \vdash \{\nu{:}\texttt{number} \mid \nu = 0\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu = 0\} \tag{10}$$
$$\Gamma_c \vdash \{\nu{:}\texttt{number} \mid \nu = 0\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu \neq 0\} \tag{11}$$

Verification Conditions. To verify subtyping obligations, we convert them into logical verification conditions (VCs), whose validity determines whether the subtyping holds. A subtyping obligation $\Gamma \vdash \{\nu{:}b \mid p\} \sqsubseteq \{\nu{:}b \mid q\}$ translates to the VC $[\![\Gamma]\!] \Rightarrow (p \Rightarrow q)$ where $[\![\Gamma]\!]$ is the conjunction of the refinements of the binders in Γ. For example, the subtyping obligations (7) and (8) yield the respective VCs:

$$(\texttt{flag} \neq 0 \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{12}$$
$$(\texttt{flag} = 0 \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{13}$$

Here, the conjunct *true* arises from the trivial refinements *e.g.* the binding for `x`. The above VCs are deemed valid by an SMT solver as the hypotheses are inconsistent, which proves the call is indeed dead code. Similarly, (9), (10) respectively yield VCs:

$$\mathit{true} \Rightarrow \nu = 1 \Rightarrow \nu \neq 0 \tag{14}$$
$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu = 0 \tag{15}$$

which are deemed valid by SMT, verifying the assignments to `a`, `b`. However, by (11):

$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu \neq 0 \tag{16}$$

which is invalid, ensuring that we *reject* the call that assigns to `c`.

2.5 Two-Phase Inference

Our two-phased approach readily lends itself to abstract interpretation based *refinement inference* which can drastically lower the programmer annotations required to verify various safety properties, *e.g.* reducing the annotations needed to verify array bounds safety in ML programs from 31% of code size to under 1% [23]. Here we illustrate how inference works in the presence of value-based overloading. Suppose we are *not* given the refinements for the signature of `neg` but only the unrefined signature (either given to us explicitly as in TypeScript, or inferred via dataflow analysis [16, 11]). As inference is difficult with incorrect code, we omit the erroneous statements that assign to `c` and `d`.

Refinement inference proceeds in three steps. First, we create *templates* which are the basic types decorated with *refinement variables* κ in place of the unknown refinements. Second, we perform the *trust* phase to elaborate the source program into a well-typed target free of overloading. Remember that this phase uses only the basic types and is oblivious to the (in this case unknown) refinements. Third, we perform the *verify* phase which now generates VCs over the refinement variables κ. These VCs – *logical implications* between the refinements and κ variables – correspond to so-called Horn constraints over the κ variables, and can be solved via abstract interpretation [13, 23].

0. Templates: Let us revisit the program from Figure 2, with the goal of inferring the refinements. Recall that the (unrefined) type of `neg` is:

$$\begin{aligned}\texttt{neg} :: {} & (\texttt{number}, \texttt{number}) \Rightarrow \texttt{number}\\ & \wedge\ (\texttt{number}, \texttt{boolean}) \Rightarrow \texttt{boolean}\end{aligned}$$

We create a *template* by refining each base type with a (distinct) refinement variable:

$$\begin{aligned}\texttt{neg} :: {} & (\{\nu{:}\texttt{number} \mid \kappa_1\}, \{\nu{:}\texttt{number} \mid \kappa_2\}) \Rightarrow \{\nu{:}\texttt{number} \mid \kappa_3\}\\ & \wedge\ (\{\nu{:}\texttt{number} \mid \kappa_4\}, \{\nu{:}\texttt{boolean} \mid \kappa_5\}) \Rightarrow \{\nu{:}\texttt{boolean} \mid \kappa_6\}\end{aligned}$$

1. Trust: The trust phase proceeds as before, propagating the refinements to the signatures of the elaborated target, yielding the code on the right in Figure 4 except that `neg#1` and `neg#2` have the respective templates:

$$\begin{aligned}\texttt{neg\#1} &:: (\{\nu{:}\texttt{number} \mid \kappa_1\}, \{\nu{:}\texttt{number} \mid \kappa_2\}) \Rightarrow \{\nu{:}\texttt{number} \mid \kappa_3\}\\ \texttt{neg\#2} &:: (\{\nu{:}\texttt{number} \mid \kappa_4\}, \{\nu{:}\texttt{boolean} \mid \kappa_5\}) \Rightarrow \{\nu{:}\texttt{boolean} \mid \kappa_6\}\end{aligned}$$

2. Verify: The verify phase proceeds as before, but using templates instead of the types. Hence, at the `DEAD`-cast in `neg#1` and `neg#2`, and the calls to `neg` that assign to `a` and `b`, instead of the VCs (12), (13), (14) and (15), we get the respective Horn constraints:

$$(\kappa_1\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow\ \nu = \texttt{x}\ \Rightarrow\ \mathit{false} \tag{17}$$
$$(\kappa_4\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow\ \nu = \texttt{x}\ \Rightarrow\ \mathit{false} \tag{18}$$
$$\mathit{true} \Rightarrow\ \nu = 1\ \Rightarrow\ \kappa_1 \tag{19}$$
$$\mathit{true} \Rightarrow\ \nu = 0\ \Rightarrow\ \kappa_4 \tag{20}$$

These constraints are identical to the corresponding VCs except that κ variables appear in place of the unknown refinements for the corresponding binders. We can solve these constraints using fixpoint computations over a variety of abstract domains such as monomial predicate abstraction [13, 23] over a set of ground predicates which are arithmetic (in)equalities between program variables and constants, to obtain a solution mapping each κ to a concrete refinement:

$$\kappa_1 \doteq \nu = 0 \qquad \kappa_4 \doteq \nu \neq 0 \qquad \kappa_2, \kappa_3, \kappa_5, \kappa_6 \doteq \mathit{true}$$

which, when plugged back into the templates, allow us to infer types for `neg`.

Higher-Order Verification. Our two-phased approach generalizes directly to offer precise analysis for *polymorphic, higher-order* functions. Returning to the code in Figure 1, our two-phased inference algorithm infers the refinement types:

$$\begin{aligned}
\texttt{\$reduce} &:: \forall A, B.(a\!:\!A[\,], f\!:\!(B, A, \texttt{idx}\langle a\rangle) \Rightarrow B, x\!:\!B) \Rightarrow B\\
\texttt{reduce} &:: \forall A.(a\!:\!A[\,]^+, f\!:\!(A, A, \texttt{idx}\langle a\rangle) \Rightarrow A) \Rightarrow A\\
&\quad \wedge \forall A, B.(a\!:\!A[\,], f\!:\!(B, A, \texttt{idx}\langle a\rangle) \Rightarrow B, x\!:\!B) \Rightarrow B
\end{aligned}$$

where $\texttt{idx}\langle a\rangle$ describes *valid indices* for array a, and $A[\,]^+$ describes non-empty arrays:

$$\begin{aligned}
\texttt{idx}\langle a\rangle &\doteq \{\nu\!:\!\texttt{number} \mid 0 \leq \nu < \texttt{len}(a)\}\\
A[\,]^+ &\doteq \{\nu\!:\!A[\,] \mid 0 < \texttt{len}(\nu)\}
\end{aligned}$$

The above type is a precise *summary* for the higher-order behavior of `$reduce`: it describes the relationship between the input array a, the step ("callback") function f, and the initial value of the accumulator, and stipulates that the output satisfies the same *properties* B as the input x. Furthermore, it captures the fact that the callback f is only invoked on inputs that are valid indices for the array a that is being reduced. Consequently, Liquid Types [23], for example, would automatically infer:

$$\begin{aligned}
\texttt{step} &:: \forall A.(\texttt{idx}\langle a\rangle, A, \texttt{idx}\langle a\rangle) \Rightarrow \texttt{idx}\langle a\rangle\\
\texttt{minIndex} &:: \forall A.(A[\,]) \Rightarrow \texttt{number}
\end{aligned}$$

thereby verifying the safety of array accesses in the presence of higher order functions, collections, and value-based overloading.

3 Syntax and Operational Semantics of Rsc

Next, we formalize two-phase typing via a core calculus Rsc comprising a *source* language $\lambda^{\wedge}_{\vee}$ *with* overloading via union and intersection types, and a simply typed *target* language $\lambda^{\times}_{+}$ *without* overloading, where the assumptions for safe overloading are explicated via `DEAD`-casts. In § 4, we describe the first phase that elaborates source programs into target programs, and finally, in § 5 we describe how the second phase verifies the `DEAD`-casts on the target to establish the safety of the source. Our elaboration follows the overall compilation strategy of Dunfield [10] except that we have value-based overloading instead of an explicit "merge" operator [22], and consequently, our elaboration and proofs must account for source level "errors" via `DEAD`-casts.

3.1 Source Language ($\lambda^{\wedge}_{\vee}$)

Terms. We define a source language $\lambda^{\wedge}_{\vee}$, with syntax shown in Figure 5. Expressions include variables, functions, applications, let-bindings, a ternary conditional construct, and primitive constants `c` which include numbers $0, 1, \ldots$, operators $+, -, \ldots$, *etc.*

Source Language: Syntax

Values	v	$::=$	$\mathsf{c} \mid x \mid \lambda x.e$
Expressions	e	$::=$	$v \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid e\ ?\ e_1 : e_2 \mid e_1\ e_2$
Primitive Types	$\mathbb{B}$	$::=$	$\mathsf{Num} \mid \mathsf{Bool}$
Types	A, B	$::=$	$\mathbb{B} \mid A \to B \mid A \wedge B \mid A \vee B$

Source Language: Operational Semantics $\boxed{e \longrightarrow e'}$

E-ECtx
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

E-App-1
$$\mathsf{c}\ v \longrightarrow [\![\mathsf{c}]\!](v)$$

E-App-2
$$(\lambda x.e)\ v \longrightarrow [v/x]\ e$$

E-Cond-True
$$\mathtt{true}\ ?\ e_1 : e_2 \longrightarrow e_1$$

E-Cond-False
$$\mathtt{false}\ ?\ e_1 : e_2 \longrightarrow e_2$$

E-Let
$$\mathtt{let}\ x = v\ \mathtt{in}\ e \longrightarrow [v/x]\ e$$

Figure 5 Syntax and Operational Semantics of $\lambda^{\wedge}_{\vee}$

Operational Semantics. In figure 5 we also define a standard small-step operational semantics for $\lambda^{\wedge}_{\vee}$ with a left-to-right order of evaluation, based on evaluation contexts

$$E ::= \langle\,\rangle \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid E\ ?\ e_1 : e_2 \mid E\ e \mid v\ E$$

Types. Figure 5 shows the types A in the source language. These include primitive types $\mathbb{B}$, arrow types $A \to B$ and, most notably, intersections $A \wedge B$ and (untagged) unions $A \vee B$ (hence the name $\lambda^{\wedge}_{\vee}$). Note that the source level types are *not* refined, as crucially, the first phase *ignores* the refinements when carrying out the elaboration.

Tags. As is common in dynamically typed languages, runtime values are associated with *type tags*, which can be inspected with a type test (cf. JavaScript's `typeof` operator). We model this notion to our static types, by associating each type with a set of possible tags. The multiplicity arises from unions. The meta-function $\mathtt{TAG}(A)$, defined in Figure 6, returns the possible tags that values of type A may have at runtime.

Well-Formedness. In order to resolve overloads statically, we apply certain restrictions on the form of union and intersection types, shown by the judgment $\vdash A$ formalized in Figure 6. For convenience of exposition, the parts of an untagged union need to have distinct runtime tags, and intersection types require all conjuncts to have the same tag.

3.2 Target Language ($\lambda^{\times}_{+}$)

The target language $\lambda^{\times}_{+}$ eliminates (value-based) overloading and thereby provides a basic, well-typed skeleton that can be further refined with logical predicates. Towards this end, unions and intersections are replaced with classical *tagged unions*, *products* and $\mathtt{DEAD}$-casts, that encode the requirements for basic typing.

Terms. Figure 7 shows the terms M of $\lambda^{\times}_{+}$, which extend the source language with the introduction of pairs, projections, injections, a case-splitting construct and a special constant term $\mathtt{DEAD}_{A\downharpoonleft B}\langle M\rangle$ which denotes an erroneous computation. Intuitively, a $\mathtt{DEAD}_{A\downharpoonleft B}\langle M\rangle$ is

Well-Formed Types $\boxed{\vdash A}$

$$\vdash \mathbb{B} \qquad \frac{\vdash A \quad \vdash B}{\vdash A \to B} \qquad \frac{\vdash A \quad \vdash B \quad \mathtt{TAG}(A) = \mathtt{TAG}(B)}{\vdash A \wedge B} \qquad \frac{\vdash A \quad \vdash B \quad \mathtt{TAG}(A) \cap \mathtt{TAG}(B) = \emptyset}{\vdash A \vee B}$$

$$\begin{aligned} \mathtt{TAG}(\mathsf{Num}) &= \{\texttt{"number"}\} & \mathtt{TAG}(A \wedge A') &= \mathtt{TAG}(A) \\ \mathtt{TAG}(\mathsf{Bool}) &= \{\texttt{"boolean"}\} & \mathtt{TAG}(A \vee A') &= \mathtt{TAG}(A) \cup \mathtt{TAG}(A') \\ \mathtt{TAG}(A \to A') &= \{\texttt{"function"}\} \end{aligned}$$

Figure 6 Basic Type Well-Formedness

Target Language: Syntax

$$\begin{array}{llcl} \textit{Expressions} & M, N & ::= & \mathtt{c} \mid x \mid \lambda x.M \mid M \mathbin{?} M_1 : M_2 \mid M_1\, M_2 \\ & & \mid & (M_1, M_2) \mid \mathtt{proj}_k M \mid \mathtt{inj}_1\, M \mid \mathtt{inj}_2\, M \\ & & \mid & \mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1\, x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\, x_2 \Rightarrow M_2 \mid \mathtt{DEAD}_{A \downharpoonright B}\langle M \rangle \\ \textit{Values} & W & ::= & \mathtt{c} \mid x \mid \lambda x.M \mid \mathtt{inj}_1\, W \mid \mathtt{inj}_2\, W \mid (M, M) \mid \mathtt{DEAD}_{A \downharpoonright B}\langle W \rangle \\ \textit{Ref. Types} & T, S & ::= & \{\nu{:}\mathbb{B} \mid p\} \mid x{:}T \to S \mid T{+}S \mid T \times S \end{array}$$

Target Language: Operational Semantics $\boxed{M \longrightarrow M'}$

$$\textsc{TE-ECtx}\ \frac{M \longrightarrow M'}{\mathcal{E}[M] \longrightarrow \mathcal{E}[M']} \qquad \textsc{TE-App-1}\ \frac{W \not\equiv \mathtt{DEAD}_{A \downharpoonright B}\langle W' \rangle}{\mathtt{c}\ W \longrightarrow [\![\mathtt{c}]\!](W)} \qquad \textsc{TE-App-2}\ (\lambda x.M)\ W \longrightarrow [W/x]\ M$$

$$\textsc{TE-Cond-True}\ \mathtt{true} \mathbin{?} M_1 : M_2 \longrightarrow M_1 \qquad \textsc{TE-Cond-False}\ \mathtt{false} \mathbin{?} M_1 : M_2 \longrightarrow M_2 \qquad \textsc{TE-Let}\ \mathtt{let}\ x = W\ \mathtt{in}\ M \longrightarrow [W/x]\ M$$

$$\textsc{TE-Proj}\ \mathtt{proj}_k(M_1, M_2) \longrightarrow M_k \qquad \textsc{TE-Case}\ \mathtt{case}\ \mathtt{inj}_k\ W\ \mathtt{of}\ \mathtt{inj}_1\, x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\, x_2 \Rightarrow M_2 \longrightarrow [W/x_k]\ M_k$$

Figure 7 Syntax and Operational Semantics of $\lambda^{\times}_{+}$

produced in the elaboration phase whenever the actual type A for a term M is incompatible with an expected type B.

Operational Semantics. As in the source language we define evaluation contexts

$$\begin{array}{rl} \mathcal{E} ::= & \langle\,\rangle \mid \mathtt{let}\ x = \mathcal{E}\ \mathtt{in}\ M \mid \mathcal{E} \mathbin{?} M_1 : M_2 \mid \mathcal{E}\ M \mid v\ \mathcal{E} \mid \mathtt{inj}_k\ \mathcal{E} \\ \mid & \mathtt{proj}_k \mathcal{E} \mid \mathtt{DEAD}_{A \downharpoonright B}\langle \mathcal{E} \rangle \mid \mathtt{case}\ \mathcal{E}\ \mathtt{of}\ \mathtt{inj}_1\, x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\, x_2 \Rightarrow M_2 \end{array}$$

and use them to define a small-step operational semantics for the target in Figure 7. Note how evaluation is allowed in DEAD-casts and $\mathtt{DEAD}_{A \downharpoonright B}\langle W \rangle$ *is* a value.

Types. The target language is checked against a refinement type checker. Thus, we modify the type language to account for the new language terms and refinements. *Basic Refinement Types* are of the form $\{\nu{:}\mathbb{B} \mid p\}$, consisting of the same basic types $\mathbb{B}$ as source types, and a logical predicate p (over some decidable logic), which describes the properties that values of the type must satisfy. Here, ν is a special *value variable* that describes the inhabitants of the type, that does not appear in the program, but can appear inside the refinement p. Function types are of the form $x{:}T \to S$, to express the fact that the refinement predicate of the return type S may refer to the value of the argument x. Sum and product types have the usual structure found in ML-like languages.

4 Phase 1: Trust

Terms of $\lambda^{\wedge}_{\vee}$ are elaborated to terms of $\lambda^{\times}_{+}$ by a judgment: $\Gamma \vdash e :: A \hookrightarrow M$. This is read: under the typing assumptions in Γ, term e of the source language is assigned a type A and elaborates to a term M of the target language. This judgment follows closely Dunfield's elaboration judgment [10], but with crucial differences that arise due to dynamic, value-based overloading, which we outline below.

Elaboration Ignores Refinements. A key aspect of the first phase is that elaboration is based solely on the basic types, *i.e.* does *not* take type refinements into account. Hence, the types assigned to source terms are transparent with respect to refinements; or more precisely, they work just as placeholders for refinements that can be provided as user specifications. These specifications are propagated *as is* during the first phase along with the respective basic types they are attached to. Due to this transparency of refinements we have decided to omit them entirely from our description of the elaboration phase.

4.1 Source Language Type-checking and Elaboration

Figure 8 shows the rules that formalize the elaboration process. At a high-level, following Dunfield [10], unions and intersections are translated to simpler typing constructs like sums and products (and the attendant injections, pattern-matches, and projections). Unlike the above work, which focuses on the classical intersection setting where overloading is explicit via a "merge" construct [22], we are concerned with the dynamic setting where overloading is value-based, leading to conventional type "errors".

Elaboration Modes: Strict and Flexible. Thus, one of the distinguishing features of our type system is its ability to not fail in cases where conventional static type system would raise type incompatibility errors, but instead elaborate the offending terms to the special error form $\mathtt{DEAD}_{A\downharpoonleft B}\langle M\rangle$. However, these error forms do not appear indiscriminately, but under certain conditions, specified by two elaboration modes: (1) a *flexible* judgment ($\vdash_{F}$) for rules that may yield $\mathtt{DEAD}_{A\downharpoonleft B}\langle M\rangle$ terms, and (2) a *strict* judgment ($\vdash_{S}$) for those that don't. Most elaboration rules come in both flavors, depending on the surrounding rules in a typing derivation. We write α to parameterize over the two modes.

Intuitively, we use flexible mode when checking calls to non-overloaded functions (with a *single* conjunct) and strict mode when checking calls to overloaded ones. In the former case, a type incompatibility truly signals a (potential) run-time error, but in the latter case, incompatibility may indicate the wrong choice of overload. Consequently, the elaboration judgment also states whether the intersection rule has been used, or not, by annotating the hook-arrow with the label y or n, respectively. As with strictness, we parametrize over n and y with the variable θ, and use $\star$ to denote that the outcome is not important.

Top-level Elaboration. Our top-level judgment is agnostic of either of the aforementioned modes. Elaborating programs in an empty context ($\vdash$) is essentially elaborating in the flexible sense and assumes we are not in the context of intersection elimination (T-TopLevel).

Elaboration Typing $\boxed{\Gamma \vdash e :: A \hookrightarrow M}$

$$\text{T-TopLevel}\ \frac{\cdot \vdash_{F}\ e :: A \overset{n}{\hookrightarrow} M}{\cdot \vdash e :: A \hookrightarrow M} \qquad \text{T-Weaken}\ \frac{\Gamma \vdash_{S}\ e :: A \overset{\theta}{\hookrightarrow} M}{\Gamma \vdash_{F}\ e :: A \overset{\theta}{\hookrightarrow} M}$$

$$\text{T-Cst}\quad \Gamma \vdash_{\alpha} \mathtt{c} :: \mathtt{ty\_c} \overset{\theta}{\hookrightarrow} \mathtt{c} \qquad \text{T-Let}\ \frac{\Gamma \vdash_{\alpha} e_1 :: A_1 \overset{\star}{\hookrightarrow} M_1 \qquad \Gamma, x : A_1 \vdash_{\alpha} e_2 :: A_2 \overset{\theta}{\hookrightarrow} M_2}{\Gamma \vdash_{\alpha} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: A_2 \overset{\theta}{\hookrightarrow} \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2}$$

$$\text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash_{\alpha} x :: A \overset{\theta}{\hookrightarrow} x} \qquad \text{T-If}\ \frac{\Gamma \vdash_{F}\ e :: \mathsf{Bool} \overset{n}{\hookrightarrow} M \qquad \forall i \in \{1,2\}\ .\ \Gamma \vdash_{\alpha} e_i :: A \overset{\theta}{\hookrightarrow} M_i}{\Gamma \vdash_{\alpha} e\ ?\ e_1\ :\ e_2 :: A \overset{\theta}{\hookrightarrow} M\ ?\ M_1\ :\ M_2}$$

$$\text{T-}\wedge\text{I}\ \frac{\forall k \in \{1,2\}\ .\ \Gamma \vdash_{\alpha} v :: A_k \overset{\theta}{\hookrightarrow} M_k \qquad \vdash A_1 \wedge A_2}{\Gamma \vdash_{\alpha} v :: A_1 \wedge A_2 \overset{\theta}{\hookrightarrow} (M_1, M_2)} \qquad \text{T-}\wedge\text{E}\ \frac{\Gamma \vdash_{\alpha} e :: A_1 \wedge A_2 \overset{\star}{\hookrightarrow} M}{\Gamma \vdash_{\alpha} e :: A_k \overset{y}{\hookrightarrow} \mathtt{proj}_k M}$$

$$\text{T-Lam}\ \frac{\vdash A \to B \qquad \Gamma, x : A \vdash_{\alpha} e :: B \overset{\star}{\hookrightarrow} M}{\Gamma \vdash_{\alpha} \lambda x.e :: A \to B \overset{n}{\hookrightarrow} \lambda x.M} \qquad \text{T-App}\ \frac{\begin{array}{c}\Gamma \vdash_{\alpha} e_1 :: A \to B \overset{y/n}{\hookrightarrow} M_1 \\ \Gamma \vdash_{S/F} e_2 :: A \overset{\star}{\hookrightarrow} M_2\end{array}}{\Gamma \vdash_{\alpha} e_1\ e_2 :: B \overset{n}{\hookrightarrow} M_1\ M_2}$$

$$\text{T-}\bot\ \frac{\Gamma \vdash_{F}\ e :: A \overset{\theta}{\hookrightarrow} M \qquad \mathsf{TAG}(A) \cap \mathsf{TAG}(B) = \emptyset}{\Gamma \vdash_{F}\ e :: B \overset{\theta}{\hookrightarrow} \mathsf{DEAD}_{A \downarrow B}\langle M \rangle} \qquad \text{T-}\vee\text{I}\ \frac{\Gamma \vdash_{F}\ e :: A_k \overset{\theta}{\hookrightarrow} M \qquad \vdash A_1 \vee A_2}{\Gamma \vdash_{F}\ e :: A_1 \vee A_2 \overset{\theta}{\hookrightarrow} \mathtt{inj}_k\ M}$$

$$\text{T-}\vee\text{E}\ \frac{\Gamma \vdash_{\alpha} e_0 :: A_1 \vee A_2 \overset{\theta}{\hookrightarrow} M_0 \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash_{\alpha} E[x_1] :: B \overset{\theta}{\hookrightarrow} M_1 \\ \Gamma, x_2 : A_2 \vdash_{\alpha} E[x_2] :: B \overset{\theta}{\hookrightarrow} M_2\end{array}}{\Gamma \vdash_{\alpha} E[e_0] :: B \overset{\theta}{\hookrightarrow} \mathtt{case}\ M_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1\ \mathtt{|}\ \mathtt{inj}_2\ x_2 \Rightarrow M_2}$$

Figure 8 Elaboration Typing rules

Furthermore, an elaboration that succeeds in strict mode also succeeds in flexible mode (T-Weaken), so all strict rules can be used as flexible ones.

Standard Rules. Rules T-Cst, T-Var are standard and preserve the structure of the source program. Rule T-If expects the condition e of a conditional expression $e\ ?\ e_1\ :\ e_2$ to be of boolean type, and assigns the same type A to each branch of the conditional. Rule T-Let checks expressions of the form $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$. It assigns a type A_1 to expression e_1 and checks e_2 in an environment extended with the binding of A_1 for x.

Intersections. In rule T-$\wedge$I the choice of the type we assign to a value v causes different elaborated terms W_k, as different typing requirements cause the addition of DEAD-casts at different places. This rule is intended to be used primarily for abstractions, so it's limited to accept values as input. Rule T-$\wedge$E for eliminating intersections replaces a term e that is originally typed as an intersection with a projection of that part of the pair that has a matching type. By T-$\wedge$I values typed at an intersection get a pair form.

Unions. Rule T-$\vee$I for union introduction is standard. The union elimination rule, taken from Dunfield's elaboration scheme [10], states that an expression e_0 can be assigned a union type $A_1 \vee A_2$ when placed at the "hole" of an evaluation context E, so long as the evaluation context can be typed with the same type B, when the hole is replaced with a variable typed as A_1 on the one hand and as A_2 on the other. While the rule is inherently non-deterministic, it suffices for a declarative description of the elaboration process; see Dunfield's subsequent work on untangling type-checking of intersections and unions [9] for an algorithmic variant via a let-normal conversion.

Abstraction and Application. Rule T-LAM assumes the arrow type $A \to B$ is given as annotation and is required to conform to the well-formedness constraints. At the crux of our type system is the rule T-APP. Expression e_1 can be typed in flexible mode. Depending on whether intersection elimination was used for e_1 we toggle on the mode of checking e_2. To only allow sensible derivations, we disallow the use of the DEAD-cast insertion when choosing among the cases of an intersection type. Below, we justify this choice using an example. If on the other hand, the type for e_1 is assigned without choosing among the parts of an intersection, then expression e_2 can be typed in flexible mode, potentially producing DEAD-casts.

Trusting via DEAD-Casts. The cornerstone of the "trust" phase lies in the presence of the T-$\perp$ rule. As we mentioned earlier, this rule can only be used in flexible mode. The main idea here is to allow cases that are obviously wrong, as far as the simple first phase type system is concerned; but, at the same time, include a DEAD-cast annotation and defer sound type-checking for the second phase. The premises of this rule specify that a DEAD-cast annotation will only be used if the inferred and the expected type have different tags. One of the consequences of this decision is that it does not allow DEAD-casts induced by a mismatch between higher-order types, as the tags for both types would be the same (most likely `"function"`). Thus, such mismatches are ill-typed and rejected in the first phase. This limitation is due to the limited information that can be encoded using the tag mechanism. A more expressive tag mechanism could eliminate this restriction but we omit this for simplicity of exposition.

Semantics of DEAD-Casts. To prove that elaboration preserves source level behaviors, our design of DEAD-casts preserves the property that the target gets stuck *iff* the source gets stuck. That is, source level type "errors" *do not lead to early* failures (*e.g.* at function call boundaries). Instead, DEAD-casts correspond to *markers* for all source terms that can potentially cause execution to get stuck. Hence, the target execution itself gets stuck at the same places as the source – *i.e.* when applying to a non-function, branching on a non-boolean or primitive application over the wrong base value, except that in the target, the stuckness can only occur when the value in question carries a DEAD marker. Consider the source program $(\lambda x.x\ 1)\ 0$ which gets stuck *after* the top-level application, when applying 1 to 0. It could be elaborated to $(\lambda x.x\ 1)\ \mathsf{DEAD}_{A \Downarrow B}\langle 0 \rangle$ (where A and B are respectively Num and $\mathsf{Num} \to \mathsf{Num}$) which also has a top-level application and gets stuck at the second, inner application.

Necessity of elaboration modes. If we allowed the argument of an *overloaded* call-site to be checked in *flexible* context, then for the application $f\ x$, where f has been assigned the type $f:\mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}$ and $x:\mathtt{B}$, the following derivation would be possible:

$$\text{T-App}\ \dfrac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_F\ f :: \mathtt{I} \to \mathtt{I} \overset{y}{\hookrightarrow} \mathtt{proj}_1 f} \qquad \text{T-}\bot\ \dfrac{\begin{array}{c}\ldots \vdash_F\ x :: \mathtt{B} \overset{n}{\hookrightarrow} x \\ \mathtt{TAG(B)} \cap \mathtt{TAG(I)} = \emptyset\end{array}}{\ldots \vdash_F\ x :: \mathtt{I} \overset{n}{\hookrightarrow} \mathtt{DEAD}_{\mathtt{B}\downharpoonleft\mathtt{I}}\langle x \rangle}}{f:\mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}, x:\mathtt{B} \vdash_F\ f\ x :: \mathtt{I} \overset{n}{\hookrightarrow} (\mathtt{proj}_1 f)\ (\mathtt{DEAD}_{\mathtt{B}\downharpoonleft\mathtt{I}}\langle x \rangle)}$$

But, clearly, the intended derivation here is:

$$\text{T-App}\ \dfrac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_F\ f :: \mathtt{B} \to \mathtt{B} \overset{y}{\hookrightarrow} \mathtt{proj}_2 f} \qquad \ldots \vdash_S\ x :: \mathtt{B} \overset{n}{\hookrightarrow} x}{f:\mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}, x:\mathtt{B} \vdash_F\ f\ x :: \mathtt{B} \overset{n}{\hookrightarrow} (\mathtt{proj}_2 f)\ x}$$

Subtyping. This formulation has been kept simple with respect to subtyping. The only notion of subtyping appears in the T-$\vee$I rule, where a type A_1 is widened to $A_1 \vee A_2$. We could have employed a more elaborate notion of subtyping, by introducing a subtyping relation ($\leq$) and a subsumption rule for our typing elaboration. The rules for this subtyping relation would include, among others, function subtyping:

$$\dfrac{A_1' \leq A_1 \qquad A_2 \leq A_2'}{A_1 \to A_2 \leq A_1' \to A_2'}$$

However, supporting subtyping in higher-order constructs would only be possible with the introduction of wrappers around functions to accommodate checks on the arguments and results of functions. So, assuming that a cast c represents a dynamic check the above rule would correspond to a cast producing relation ($\triangleright$):

$$\dfrac{A_1' \triangleright A_1 \rightsquigarrow c_1 \qquad A_2 \triangleright A_2' \rightsquigarrow c_2}{A_1 \to A_2 \triangleright A_1' \to A_2' \rightsquigarrow \lambda f.\lambda x.(c_2\ (f\ (c_1\ x)))}$$

This formulation would just complicate the translation without giving any more insight in the main idea of our technique, and hence we forgo it.

4.2 Source and Target Language Consistency

In this section, we present the theorems that precisely connect the semantics of source programs with their elaborated targets. The main challenges towards establishing those are that: (1) the source and target do not proceed in lock-step, a single step of the one may be matched by several steps of the other (for example evaluating a projection in the target language does not correspond to any step in the source language), and (2) we must design the semantics of the DEAD-casts in the target to ensure that DEAD-casts cause evaluation to get stuck iff some primitive operation in the source gets stuck. We address these, next, with a number of lemmas and state our assumptions.

Value Monotonicity. This lemma fills in the mismatch that emerges when (non-value) expressions in the source language elaborate to values in the target language. Informally, if a source expression e elaborates to a target value W, then e evaluates (after potentially multiple steps) to a value v that is related to the target value W with an elaboration relation under the same type. Furthermore, all expressions on the path to the target value v elaborate to the same value and get assigned the same type.

▶ **Lemma 1** (Value Monotonicity). *If* $\Gamma \vdash e :: A \hookrightarrow W$*, then there exists* $v \cdot$

(1) $e \longrightarrow^* v$
(2) $\Gamma \vdash v :: A \hookrightarrow W$
(3) $\forall i$ *s.t.* $e \longrightarrow^* e_i \,.\, \Gamma \vdash e_i :: A \hookrightarrow W$

Proof. The first two parts are handled similarly to Dunfield's [10] Lemma 11. The last part is proved by induction on the length of the path $e \longrightarrow^* e_i$. Details of this proof can be found in the extended version of this paper [30]. ◀

The reverse of the above lemma also comes in handy. Namely, given a value v that elaborates to an expression M and gets assigned the type A, there exists a value in the target language W, such that v elaborates to W and get assigned the *same* type A.

▶ **Lemma 2** (Reverse Value Monotonicity). *If* $\Gamma \vdash v :: A \hookrightarrow M$*, then exists* $W \cdot M \longrightarrow^* W$ *and* $\Gamma \vdash v :: A \hookrightarrow W$*.*

Proof. Similar to proof of Lemma 1. ◀

This is an interesting result as it establishes that different derivations may assign the same type to a term and still elaborate it to different target terms. For example, one can assume derivations that consecutively apply the intersection introduction and elimination rules. It's easy to see that the same value v can be used in the following elaborations:

$$\cdot \vdash v :: A_1 \wedge A_2 \hookrightarrow (W_1, W_2)$$
$$\cdot \vdash v :: A_1 \wedge A_2 \hookrightarrow \underbrace{(\mathtt{proj}_1(W_1, W_2), \mathtt{proj}_2(W_1, W_2))}_{M}$$

Lemma 2 guarantees it will always be the case that $M \longrightarrow^* (W_1, W_2)$. It is up to the implementation of the type-checking algorithm to produce an efficient target term.

Primitive Semantics. To connect the failure of the DEAD-casts with source programs getting stuck, we assume that the primitive constants are well defined for all the values of their input domain *but not* for DEAD-cast values. This lets us establish that primitive operations c are invariant to elaboration. Hence, a source primitive application gets stuck iff the elaborated argument is a DEAD-cast. The forward version of this statement is the following assumption.

▶ Assumption 1 (Primitive constant application). If (1) $\cdot \vdash \mathsf{c} :: A \rightarrow B \hookrightarrow \mathsf{c}$, (2) $\cdot \vdash v :: A \hookrightarrow W$, and (3) $W \not\equiv \mathtt{DEAD}_{\downarrow A}\langle\cdot\rangle$, then (i) $\mathsf{c}\ v \longrightarrow [\![\mathsf{c}]\!](v)$, (ii) $\mathsf{c}\ W \longrightarrow [\![\mathsf{c}]\!](W)$, and (iii) $\cdot \vdash [\![\mathsf{c}]\!](v) :: B \hookrightarrow [\![\mathsf{c}]\!](W)$.

Substitution lemma. The proof of soundness relies upon the following substitution lemma.

▶ **Lemma 3** (Substitution). *If* $\Gamma, x\!:\!A \vdash e :: A' \hookrightarrow M$ *and* $\Gamma \vdash v :: A \hookrightarrow W$ *then* $\Gamma \vdash [v/x]\ e :: A' \hookrightarrow [W/x]\ M$*.*

Proof. Similar to Dunfield's substitution proof [10] (Lemma 12). ◀

We use the above lemmas and assumptions to obtain a consistency result, analagous to Dunfield's Consistency Theorem [10], which states that the elaboration produces terms that are *consistent* with the source in that each step of the target is matched by a corresponding step of the source, *i.e.* the behaviors of the target *under-approximate* the behaviors of the source.

▶ **Theorem 4** (Consistency). *If* $\cdot \vdash e :: A \hookrightarrow M$ *and* $M \longrightarrow M'$ *then there exists* e' *such that* $e \longrightarrow^* e'$ *and* $\cdot \vdash e' :: A \hookrightarrow M'$*.*

Proof. The proof of this theorem is by induction on the derivation $\cdot \vdash e :: A \hookrightarrow M$, adapting the proof scheme given by Dunfield [10], and using Lemma 1. Details of this proof can be found in the extended version of this paper [30]. ◀

While this suffices to prove *soundness* – intuitively if the target does not "go wrong" then the source cannot "go wrong" either – it is not wholly satisfactory as a trivial translation that converts every source program to an ill-typed target also satisfies the above requirement. So, unlike Dunfield [10], we also establish a completeness result stating that if the source term steps, then the elaborated program will also eventually step to a corresponding (by elaboration) term. Theorem 5 declares that behaviors of the elaborated target *over-approximate* those of the source, and hence, in conjunction with Theorem 4, ensure that the source "goes wrong" iff the target does.

▶ **Theorem 5** (Reverse Consistency). *If* $\cdot \vdash e :: A \hookrightarrow M$ *and* $e \longrightarrow e'$ *then there exists* M' *such that* $\cdot \vdash e' :: A \hookrightarrow M'$*, and* $M \longrightarrow^+ M'$*.*

Proof. Similar to the proof of Theorem 4, using adapted versions of the lemmas used by Dunfield [10] and Lemma 2. Again, details can be found in the accompanying report [30]. ◀

5 Phase 2: Verify

At the end of the first phase, we have elaborated the source with value based overloading into a classically well-typed target with conventional typing features and DEAD-casts which are really assertions that explicate the *trust assumptions* made to type the source. Thanks to Theorems 4 and 5 we know the semantics of the target are equivalent to the source. Thus, to verify the source, all that remains is to prove that the target will not "go wrong", that is to prove that the DEAD-casts are indeed never executed at run-time.

One advantage of our elaboration scheme is that at this point *any* program analysis for ML-like languages (*i.e.* supporting products, sums, and first class functions) can be applied to discharge the DEAD-cast [8]: as long as the target is safe, the consistency theorems guarantee that the source is safe. In our case, we choose to instantiate the second phase with *refinement types* as they: (1) are especially well suited to handle higher-order polymorphic functions, like `minIndex` from Figure 1, (2) can easily express other correctness requirements, *e.g.* array bounds safety, thereby allowing us to establish not just type safety but richer correctness properties, and, (3) are automatically inferred via the abstract interpretation framework of Liquid Typing [23]. Next, we recall how refinement typing works to show how DEAD-cast checking can be carried out, and then present the end-to-end soundness guarantees established by composing the two phases.

5.1 Refinement Type-checking

We present a brief overview of refinement typing as the target language falls under the scope of existing refinement type systems [18], which can, after accounting for DEAD-casts, be reused *as is* for the second phase. Similarly, we limit the presentation to *checking*; *inference* follows directly from Liquid Type inference [23]. Figure 9 summarizes the refinement system. The type-checking judgment is $G \vdash M :: T$, where *type environment* G is a sequence of bindings of variables x to refinement types T and *guard predicates*, which encode control flow information gathered by conditional checks. As is standard [18] each primitive constant c has a refined

Refined Typechecking $\boxed{G \vdash M :: T}$

$$\text{R-Sub}\ \frac{G \vdash M :: T_1 \qquad G \vdash T_1 \sqsubseteq T_2}{G \vdash M :: T_2} \qquad \begin{array}{l}\text{R-Cst}\\ G \vdash \mathtt{c} :: \mathtt{ty\_c}\end{array}$$

$$\text{R-Var}\ \frac{x{:}T \in G}{G \vdash x :: \mathsf{sngl}(T, x)} \qquad \text{R-Let}\ \frac{G \vdash M_1 :: T_1 \qquad G, x{:}T_1 \vdash M_2 :: T_2}{G \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 :: T_2}$$

$$\text{R-If}\ \frac{G \vdash M :: \mathsf{Bool} \qquad G, M \vdash M_1 :: T \qquad G, \neg M \vdash M_2 :: T}{G \vdash M \ ?\ M_1 \ :\ M_2 :: T} \qquad \text{R-Lam}\ \frac{G, x : T_x; G \vdash M :: T}{G \vdash \lambda x.M :: T_x \to T}$$

$$\text{R-App}\ \frac{G \vdash M_1 :: T_x \to T \qquad G \vdash M_2 :: T_x}{G \vdash M_1\ M_2 :: [M_2/x]\,T} \qquad \text{R-Pair}\ \frac{\forall k \in \{1,2\}\ .\ G \vdash M_k :: T_k}{G \vdash (M_1, M_2) :: T_1 \times T_2} \qquad \text{R-Proj}\ \frac{G \vdash M :: T_1 \times T_2}{G \vdash \mathtt{proj}_k M :: T_k}$$

$$\text{R-Inj}\ \frac{G \vdash M :: T_k}{G \vdash \mathtt{inj}_k\ M :: T_1{+}T_2} \qquad \text{R-Case}\ \frac{G \vdash M :: T_1{+}T_2 \qquad G, x_1 : T_1 \vdash M_1 :: T \qquad G, x_2 : T_2 \vdash M_2 :: T}{G \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow M_2 :: T}$$

Refinement Subtyping $\boxed{G \vdash T_1 \sqsubseteq T_2}$

$$\sqsubseteq\text{-Base}\ \frac{\mathsf{Valid}(\llbracket G \rrbracket \wedge \llbracket p \rrbracket \Rightarrow \llbracket p' \rrbracket)}{G \vdash \{\nu{:}\mathbb{B} \mid p\} \sqsubseteq \{\nu{:}\mathbb{B} \mid p'\}} \qquad \sqsubseteq\text{-Fun}\ \frac{G \vdash T'_x \sqsubseteq T_x \qquad G, x{:}T'_x \vdash T \sqsubseteq T'}{G \vdash (x : T_x) \to T \sqsubseteq (x : T'_x) \to T'}$$

Figure 9 Refined Type-checking

type `ty_c`, and a variable x with type T is typed as $\mathsf{sngl}(T, x)$ which is $\{\nu{:}\mathbb{B} \mid \nu = x\}$ if T is a basic type $\mathbb{B}$ and T otherwise.

Checking DEAD-casts. The refinement system verifies DEAD-casts by treating them as special function calls, *i.e.* discharging them via the application rule R-App. Formally, $\mathtt{DEAD}_{A \Downarrow B}\langle M \rangle$ is treated as call to:

$$\mathtt{DEAD}_{A \Downarrow B} :: \mathsf{Bot}([A]) \to \mathsf{Bot}([B])$$

The notation $[\cdot]$ denotes the elaboration of $\lambda^{\wedge}_{\vee}$ types to $\lambda^{\times}_{+}$ types [10]:

$$[\mathbb{B}] \doteq \mathbb{B} \quad [A \wedge B] \doteq [A] \times [B] \quad [A \vee B] \doteq [A]{+}[B] \quad [A \to B] \doteq [A] \to [B]$$

The meta-function $\mathsf{Bot}(T) \doteq \mathsf{Tx}(T,\ \mathit{false})$ where:

$$\begin{array}{lll@{\qquad}lll}
\mathsf{Tx}(\mathbb{B},\ r) & \doteq & \{\nu{:}\mathbb{B} \mid r\} & \mathsf{Tx}(S{+}T,\ r) & \doteq & \mathsf{Tx}(S,\ r){+}\mathsf{Tx}(T,\ r) \\
\mathsf{Tx}(S \to T,\ r) & \doteq & \mathsf{Tx}(S,\ \neg r) \to \mathsf{Tx}(T,\ r) & \mathsf{Tx}(S \times T,\ r) & \doteq & \mathsf{Tx}(S,\ r) \times \mathsf{Tx}(T,\ r)
\end{array}$$

Returning to rule R-App for DEAD-casts and inverting, expression M gets assigned a refinement type T. For simplicity we assume this is a base type $\mathbb{B}$. Due to R-Sub we get the subtyping constraint: $G \vdash \{\nu{:}\mathbb{B} \mid p\} \sqsubseteq \{\nu{:}\mathbb{B} \mid \mathit{false}\}$, which generates the VC: $\mathsf{Valid}(\llbracket G \rrbracket \wedge \llbracket p \rrbracket \Rightarrow \llbracket \mathit{false} \rrbracket)$. This holds if the environment combined with the refinement in the left-hand side is inconsistent, which means that the gathered flow conditions are infeasible, hence dead-code [18]. Thus, the refinements statically ensure that the specially marked `DEAD` values are *never created at run-time*. As only `DEAD` terms cause execution to get stuck, the refinement verification phase ensures that the source is indeed type safe.

Conditional Checking. R-IF and R-CASE check each branch of a conditional or case splitting statement, by enhancing the environment with a guard (M or $\neg M$) or the right binding ($x : T_1$ or $x : T_2$), that encode the boolean test performed at the condition, or the structural check at the pattern matching, respectively. Crucially, this allows the use of "tests" inside the code to statically verify `DEAD`-casts and other correctness properties. The other rules are standard and are described in the refinement type literature.

Correspondence of Elaboration and Refinement Typing. The following result establishes the fact that the type A assigned to a source expression e by elaboration and the type T assigned by refinement type-checking to the elaborated expression M are connected with the relation: $[A] = \|T\|$, where $\|T\|$ is merely a (recursive) elimination of all refinements appearing in T. The notation $[\Gamma] = \|G\|$ means that for each binding $x : A \in \Gamma$ there exists $x : T \in G$, such that $[A] = \|T\|$, and vice versa.

▶ **Lemma 6** (Correspondence). *If* $\Gamma \vdash e :: A \hookrightarrow M$, $G \vdash M :: T$ *and* $[\Gamma] = \|G\|$, *then* $[A] = \|T\|$.

Proof. By induction on pairs of derivations: $\Gamma \vdash e :: A \hookrightarrow M$ and $G \vdash M :: T$. Details of this proof can be found in the extended version of this paper [30]. ◀

The target language satisfies a progress and preservation theorem [18]:

▶ **Theorem 7** (Refinement Type Safety). *If* $\cdot \vdash M : T$ *then either* M *is a value or there exists* M' *such that* $M \longrightarrow M'$ *and* $\cdot \vdash M' : T$.

Proof. Given by Vazou *et al.* [29] for a similar language. ◀

5.2 Two-Phase Type Safety

We say that a source term e is *well two-typed* if there exists a source type A, target term M and target (refinement) type T such that: (1) $\cdot \vdash e :: A \hookrightarrow M$, and, (2) $\cdot \vdash M :: T$. That is, e is well two-typed if it elaborates to a refinement typed target. The Consistency Theorems 4 and 5, along with the Safety Theorem 7, yield end-to-end soundness: well two-typed terms do not get stuck, and step to well two-typed terms.

▶ **Theorem 8** (Two-Phase Soundness). *If e is well two-typed then, either e is a value, or there exists e' such that:*

*(1) **(Progress)** $e \longrightarrow e'$*
*(2) **(Preservation)** e' is well two-typed.*

Proof. By induction on pairs of derivations: $\Gamma \vdash e :: A \hookrightarrow M$ and $G \vdash M :: T$. Details are to be found in the extended version of this paper [30]. ◀

6 Related Work

We focus on the highlights of prior work relevant to the key points of our technique: static types for dynamic languages, intersections and union types, and refinement types.

Types for Dynamic Functional Languages. *Soft typing* [5] incorporates static analysis to statically type dynamic languages: whenever a program cannot be proven safe statically, it is not rejected, but instead runtime checks are inserted. Henglein and Rehof [17] build up on this work by extending soft typing's monomorphic typing to polymorphic coercions and providing a translation of Scheme programs to ML. These works foreshadow the notion of *gradual typing* [24] that allows the programmer to control the boundary between static and dynamic checking depending on the trade-off between the need for static guarantees and deployability. Returning to purely static enforcement, Tobin-Hochstadt *et al.* [27, 28] formalize the support for type tests as *occurrence typing* and extend it to an inter-procedural, higher-order setting by introducing propositional *latent predicates* that reflect the result of tests in Typed Racket function signatures.

Types for Dynamic Imperative Languages. Thiemann [26] and Anderson *et al.* [1] describe early attempts towards static type systems for JavaScript, and Furr *et al.* [15] present DRuby, a tool for type inference for Ruby scripts. However, these systems do not handle value-based overloading (like TypeScript, DRuby allows overloaded specifications for external functions). *Flow typing* [16] and TeJaS [19] account for tests using flow analysis, bringing occurrence typing to the imperative JavaScript setting, but, unlike our approach, they restrict themselves to a *fixed* set of type-testing idioms (*e.g.* `typeof`), precluding *general* value-based overloading *e.g.* as in `reduce` from Figure 1.

Logics for Dynamic Languages. The intuition of expressing subtyping relations as logical implication constraints and using SMT solvers to discharge these constraints allows for a more extensive variety of typing idioms. Bierman *et al.* [3] investigate semantic subtyping in a first order language with refinements and type-test expressions. In *nested refinement types* [7], the typing relation itself is a predicate in the refinement logic and a feature-rich language of predicates accounts for heavily dynamic idioms, like run-time type tests, value-indexed dictionaries, polymorphism and higher order functions. While program logics allow the use of arbitrary tests to establish typing, the circular dependency between values and basic types leads to two significant problems in theory and practice. First, the circular dependency complicates the *metatheory* which makes it hard to add extra (basic) typing features (*e.g.* polymorphism, classes) to the language. Second, the circular dependency complicates the *inference* of types and refinements, leading to significant annotation overheads which make the system difficult to use in practice. In contrast, two-phase typing allows arbitrary type tests while enabling the trivial composition of soundness proofs and inference algorithms.

Intersection and Union Types. Central to our elaboration phase are intersection and union types: Pierce [21] indicates the connection between unions and intersections with sums and products, that is the basis of Dunfield's elaboration scheme [10] on which we build. However, Dunfield studies *static* source languages that use *explicit* overloading via a merge operator [22]. In contrast, we target *dynamic* source languages with implicit value based overloading, and hence must account for "ill-typed" terms via `DEAD`-casts discharged via the second phase refinement check. Castagna *et al.* [6] describe a $\lambda\&$-calculus, where functions are overloaded by combining several different branches of code. The branch to be executed is determined at run-time by using the arguments' typing information. This technique resembles the code duplication that happens in our approach, but overload resolution (*i.e.* deciding which branch is executed) is determined at runtime whereas we do so statically.

Refinement Types. DML [31] is an early refinement type system composing ML's types with a decidable constraint system. *Hybrid type checking* [18] uses arbitrary refinements over basic types. A static type system verifies basic specifications and more complex ones are defered to dynamically checked contracts, since the specification logic is statically undecidable. In these cases, the source language is well typed (ignoring refinements), and lacks intersections and unions. Our second phase can use Liquid Types [23] to infer refinements using predicate abstraction.

7 Conclusions and Future Work

In this paper, we introduce two-phased typing, a novel framework for analyzing dynamic languages where value-based overloading is ubiquitous. The advantage of our approach over previous methods is that, unlike purely type-based approaches [28], we are not limited to a fixed set of tag- or type- tests, and unlike purely program logic-based approach [7], we can decouple reasoning about basic typing from values, thereby enabling inference.

Hence, we believe two-phased typing provides an ideal foundation for building expressive and automatic analyses for imperative scripting languages like JavaScript. However, this is just the first step; much remains to achieve this goal. In particular we must account for the imperative features of the language. We believe that decoupling makes it possible to address this problem by applying various methods for tracking mutation and aliasing [32] in the *first phase*, and we intend to investigate this route in future work to obtain a practical verifier for TypeScript.

Acknowledgements We would like to thank our anonymous reviewers for their feedback. This work was supported by NSF Grants CNS-1223850, CNS-0964702 and gifts from Microsoft Research.

References

1 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

2 Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), February 2011.

3 Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

4 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3), June 2005.

5 Robert Cartwright and Mike Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

6 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, 1992.

7 Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

8 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

9 Joshua Dunfield. Untangling Typechecking of Intersections and Unions. In *Proceedings of the 5th Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010.*, 2010.

10 Joshua Dunfield. Elaborating Intersection and Union Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.

11 Flow: A Static Type Checker for JavaScript. http://flowtype.org.

12 Asger Feldthaus and Anders Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Language and Applications*, 2014.

13 Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation Inference for Modular Checkers. *Inf. Process. Lett.*, 77(2-4), February 2001.

14 Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.

15 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, 2009.

16 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, 2011.

17 Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 1995.

18 Kenneth Knowles and Cormac Flanagan. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.*, 32(2), February 2010.

19 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, 2013.

20 Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.

21 Benjamin C. Pierce. *Programming With Intersection Types, Union Types, and Polymorphism.* PhD thesis, Carnegie Mellon University, 1991.

22 John C. Reynolds. ALGOL-like Languages, Volume 1. In Peter W. O'Hearn and Robert D. Tennent, editors, *Algol-like Languages*, chapter Design of the Programming Language FORSYTHE. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

23 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

24 Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

25 The Satisfiability Modulo Theories Library. http://smt-lib.org.

26 Peter Thiemann. Towards a Type System for Analyzing Javascript Programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, 2005.

27 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

28 Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

29 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.

30 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust but Verify: Two-Phase Typing for Dynamic Languages — Extended. http://arxiv.org/abs/1504.08039.

31 Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

32 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and Reference Immutability Using Java Generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.