Trust, but Verify
Two-Phase Typing for Dynamic Languages

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala

University of California, San Diego, La Jolla, CA 92093, USA

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. 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. First, we empirically demonstrate the ubiquity of value-based overloading. Next, we distill it into a core *source* language with union and intersection types. We formalize the trust phase as an elaboration to a simply typed *target* language without overloading, but with `DEAD`-casts and formalize the second phase that discharges the casts via classical refinement typing. Finally, we prove the equivalence of source and target to establish the end-to-end soundness of two-phase typing, thereby providing a new foundation for building static analyses for dynamic languages.

1 Introduction

Modern *dynamic scripting* languages – like JavaScript, Python, and Ruby – have popularized the use of higher-order constructs that were once solely in the *functional* realm. For example, consider the function:

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

```
function $reduce(a, f, x) {
  var res = x, i;
  for (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);
}
```

Fig. 1: Computing the minimum-valued index with Higher-Order Functions

which computes the index with 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 higher-order `$reduce` function (Figure 1) which frees programmers 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 bottom of Figure 1.

This virtuous trend towards abstraction and reuse poses a vexing problem for 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, those 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 [30] 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 use it to type `$reduce` as:

```
  forall 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 principal attraction of dynamic languages is *value-based overloading*, where syntactic entities (*e.g.* variables) may by 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 it is omitted, the first array element `a[0]` is used in its stead. Here, `reduce` really has *two* different types. In one avatar it is a function that takes 3 parameters, and in the other, it takes 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, ironically, 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 In this paper, we introduce the framework of *two-phased typing*, a new strategy for statically analyzing dynamic languages. The key insight in our approach 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 phase. 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* [11] 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 [29,17] extend classical type systems to account for a fixed set of `typeof`-style tests, and cannot reason about general value tests – *e.g.* the size of `arguments` – that often appear in idiomatic code. At the other end, systems like [7] embed the typing relation within an expressive program logic, thereby accommodating general value tests, but give up on basic type structure thereby sacrificing inference, leading to a significant annotation overhead. In contrast, our two-phased approach cleanly 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.
– ***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 [24], 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 [19] 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* [11] 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

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

Fig. 2: An Example TypeScript Program with Value-based Overloading

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.

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 [29,17]. 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. Indeed, 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 cannot verify those signatures, it can use the signatures as trusted interfaces for external JavaScript libraries and code completion.) The DEFINITELYTYPED 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* so-called 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 [13]. 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 DEFINITELYTYPED. 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`

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**

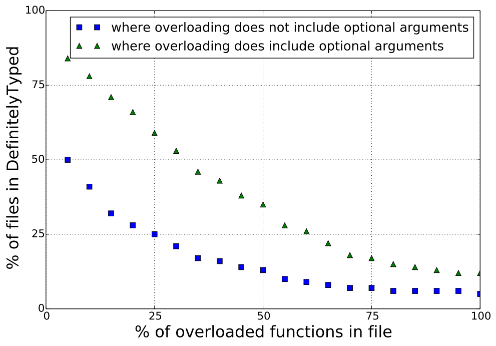

Fig. 3: The prevalence of value-based overloading. **(L)** Libraries from [13]: **#Funs** is the number of functions in the signature, **%Ovl** is %-functions with *multiple* signatures, **%Opt** is %-functions with *optional* arguments, and **%Any** is %-functions with either of these features. **(R)** Overloading across *all* files in DEFINITELYTYPED. A point (x, y) means $y\%$ of files have more than $x\%$ overloaded functions.

The signatures in DEFINITELYTYPED have not been soundly checked against[2] their implementations. Indeed, the overall goal of our work is to develop the first such checker. 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 [26]). For example, `{v:number| v != 0}` describes the *subset* of numbers that are non-zero. We write A to abbreviate the trivially refined type $\{v{:}A \mid \mathit{true}\}$, *e.g.* `number` is an abbreviation for $\{v{:}\mathtt{number} \mid \mathit{true}\}$.

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,

$$(x{:}0 \leqslant x) \Rightarrow \{v{:}\mathtt{number} \mid x < v\}$$

[2] [13] describes an effective but unsound inconsistency detector.

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 [30], the basic types are established *without* requiring refinements. A classical refinement system is thus a conservative extension of the corresponding non-refined language, *i.e.* if you strip out the refinements from a DML program, you get a valid, well-typed ML program. Unfortunately, value-based overloading snatches away this crucial property, and forces us to snap the circular dependency between types and refinements.

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 special DEAD-casts, which are assertions that state the program is well-typed *assuming* those expressions are dead code. In the second phase we can reuse classical refinement typing techniques to *verify* that the DEAD-casts are indeed unreachable, thereby discharging the assumptions made by the first phase.

2.3 Phase 1: Trust

The first phase snaps the circular dependency between types and refinements by *elaborating* 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.

```
neg :: (tt, number ) => number   neg#1 :: (tt, number) => number
    /\ (ff, boolean) => boolean  function neg#1(flag, x){
function neg(flag, x){             if (flag) return 0-x;
  if (flag) return 0-x;            return !DEAD(x);
  return !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 = neg(1,1);     //OK       var a = fst(neg)(1,1);     //OK
var b = neg(0,true);  //OK       var b = snd(neg)(0,true);  //OK
var c = neg(0,1);     //ERR      var c = fst(neg)(0,1);     //ERR
var d = neg(1,true);  //ERR      var d = snd(neg)(1,true);  //ERR
```

Fig. 4: RSC source (l) and target (r) resulting from first phase elaboration.

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 `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 [23,11], in our context these checks almost surely "fail". For example, `neg#1` *does not* type-check as the parameter `x:number` and so we cannot compute `!x`. Similarly, `neg#2` fails because `x: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. This program 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 [15,4] or refinement typing [30,19,24,2] or contracts & abstract interpretation [21] to check that the assertions never fail in the target, 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 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:

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

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:

$$\Gamma_1 \doteq \mathtt{flag}{:}\mathtt{tt},\ \mathtt{x}{:}\mathtt{number},\ g_1{:}\{\nu{:}\mathtt{boolean} \mid \mathtt{flag} = 0\} \tag{1}$$

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:

$$\Gamma_2 \doteq \mathtt{flag}{:}\mathtt{ff},\ \mathtt{x}{:}\mathtt{boolean},\ g_1{:}\{\nu{:}\mathtt{boolean} \mid \mathtt{flag} \neq 0\} \tag{2}$$

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

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

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

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

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:

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

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:

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

Verification Conditions To verify subtyping obligations, we convert them into logical verification conditions (VCs), whose validity determines whether the subtyping holds. A subtyping obligation $\Gamma \vdash \{\nu\!:\!\texttt{b} \mid p\} \sqsubseteq \{\nu\!:\!\texttt{b} \mid q\}$ 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

By decoupling the circular dependency between types and refinements, 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% [24].

Next, we illustrate how two-phase inference works in the presence of value-based overloading. Suppose that 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 data flow analysis [17,12]). 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 [14,24].

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:

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

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

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

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:

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

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 [14,24] 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:

$$\kappa_1 \ \doteq \ \nu = 0 \qquad\qquad \kappa_4 \ \doteq \ \nu \neq 0$$

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:

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

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, the system of [24] automatically infers:

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

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 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 [11] except that we have value-based overloading instead of an explicit “merge” operator [23], 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 $\lambda^{\wedge}_{\vee}$, whose expressions and values are shown in Figure 5, and include the usual variables, functions, applications, let-bindings, a ternary if-then-else construct, and primitive constants `c` which include numbers $0, 1, \ldots$, operators $+, -, \ldots$ *etc.*.

Source Language: Syntax

$$\begin{array}{rl} \textit{Expressions} & e ::= \mathtt{c} \mid x \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \lambda x.e \mid e\ ?\ e_1 : e_2 \mid e_1\ e_2 \\ \textit{Values} & v ::= \mathtt{c} \mid x \mid \lambda x.e \end{array}$$

$$\begin{array}{rl} \textit{Primitive Types} & \mathbb{B} ::= \mathtt{number} \mid \mathtt{boolean} \\ \textit{Types} & A, B ::= \mathbb{B} \mid A \to B \mid A \wedge B \mid A \vee B \end{array}$$

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

$$\textit{Eval. Context} \quad E ::= \langle\,\rangle \mid \mathtt{let}\ x = E\ \mathtt{in}\ e \mid E\ ?\ e_1 : e_2 \mid E\ e \mid v\ E$$

$$\textsc{E-ECtx}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \textsc{E-App-1}\ \ \mathtt{c}\ v \longrightarrow [\![\mathtt{c}]\!](v) \qquad \textsc{E-App-2}\ \ (\lambda x.e)\ v \longrightarrow [v/x]\ e$$

$$\textsc{E-Cond-True}\ \ \mathtt{true}\ ?\ e_1 : e_2 \longrightarrow e_1 \qquad \textsc{E-Cond-False}\ \ \mathtt{false}\ ?\ e_1 : e_2 \longrightarrow e_2 \qquad \textsc{E-Let}\ \ \mathtt{let}\ x = v\ \mathtt{in}\ e \longrightarrow [v/x]\ e$$

Fig. 5: Syntax and Operational Semantics of $\lambda^{\wedge}_{\vee}$

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

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 easily inspected with a type check (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 $\mathsf{TAG}(A)$, 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 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 spe-

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

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

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

Fig. 6: Basic Type Well-Formedness

Target Language: Syntax

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

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

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

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

$$\begin{array}{l}\textsc{TE-Cond-True} \\ \mathtt{true} \,?\, M_1 : M_2 \longrightarrow M_1\end{array} \qquad \begin{array}{l}\textsc{TE-Cond-False} \\ \mathtt{false} \,?\, M_1 : M_2 \longrightarrow M_2\end{array}$$

$$\begin{array}{l}\textsc{TE-Let} \\ \mathtt{let}\ x = W\ \mathtt{in}\ M \longrightarrow [W/x]\ M\end{array} \qquad \begin{array}{l}\textsc{TE-Proj} \\ \mathtt{proj}_k(M_1, M_2) \longrightarrow M_k\end{array}$$

$$\begin{array}{l}\textsc{TE-Case} \\ \mathtt{case}\ \mathtt{inj}_k\ W\ \mathtt{of}\ \mathtt{inj}_1\, x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\, x_2 \Rightarrow M_2 \longrightarrow [W/x_k]\ M_k\end{array}$$

Fig. 7: Syntax and Small Step Operational Semantics of $\lambda^{\times}_{+}$

cial constant term $\mathtt{DEAD}_{A \downarrow B}\langle M \rangle$ which denotes an erroneous computation. Intuitively, a $\mathtt{DEAD}_{A \downarrow B}\langle M \rangle$ is 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 $\mathcal{E}$ 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 $\mathtt{DEAD}_{A \downarrow B}\langle W \rangle$ *is* a value.

Types The target language will be checked against a refinement type checker. Thus, we modify the type language to account for the new language terms and to accomodate 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, which does not appear in the program, but which can appear inside the refinement p. Function types are of the form $x : T \rightarrow S$, to express the fact that (refinement predicates within) the return type S may refer to the value of the argument x in its refinement. 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 the elaboration judgment of [11], 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 that are assigned to source terms should be thought of as 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 type-checking & elaboration 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 [11], unions and intersections are translated to simpler, standard 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 [23], we are concerned with the dynamic setting where overloading is value-based, leading to conventional type "errors".

Elaboration Modes: Strict and Lax 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 $\mathsf{DEAD}_{A \downarrow B}\langle M \rangle$. However, these error forms do not appear indiscriminately, but under certain conditions, which are specified by two elaboration modes: (1) a *lax* judgment ($\vdash_{L}$) that denotes rules that may yield $\mathsf{DEAD}_{A \downarrow B}\langle M \rangle$ 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 lax mode when checking calls to non-overloaded functions (with a *single* conjunct) and strict mode when checking calls to overloaded functions. In the

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

$$\textsc{T-TopLevel}\ \dfrac{\cdot \vdash_{L} e :: A \overset{F}{\hookrightarrow} M}{\cdot \vdash e :: A \hookrightarrow M} \qquad \textsc{T-Weaken}\ \dfrac{\Gamma \vdash_{S} e :: A \overset{\theta}{\hookrightarrow} M}{\Gamma \vdash_{L} e :: A \overset{\theta}{\hookrightarrow} M}$$

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

$$\textsc{T-Var}\ \dfrac{x{:}A \in \Gamma}{\Gamma \vdash_{\alpha} x :: A \overset{\theta}{\hookrightarrow} x} \qquad \textsc{T-If}\ \dfrac{\Gamma \vdash_{L} e :: \mathtt{boolean} \overset{F}{\hookrightarrow} M \qquad \forall i \in \{1,2\}\,.\ \Gamma \vdash_{\alpha} e_i :: A \overset{\theta}{\hookrightarrow} M_i}{\Gamma \vdash_{\alpha} e\ ?\ e_1\ :\ e_2 :: A \overset{\theta}{\hookrightarrow} M\ ?\ M_1\ :\ M_2}$$

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

$$\textsc{T-Lam}\ \dfrac{\vdash A \to B \qquad \Gamma, x{:}A \vdash_{\alpha} e :: B \overset{\bullet}{\hookrightarrow} M}{\Gamma \vdash_{\alpha} \lambda x.e :: A \to B \overset{F}{\hookrightarrow} \lambda x.M} \qquad \textsc{T-App}\ \dfrac{\Gamma \vdash_{\alpha} e_1 :: A \to B \overset{T/F}{\hookrightarrow} M_1 \qquad \Gamma \vdash_{S/L} e_2 :: A \overset{\bullet}{\hookrightarrow} M_2}{\Gamma \vdash_{\alpha} e_1\ e_2 :: B \overset{F}{\hookrightarrow} M_1\ M_2}$$

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

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

Fig. 8: Elaboration Typing rules

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, using the labels F and T, to annotate the elaboration hook-arrow. As with strictness, we parametrize over F and T with the variable θ, and use $\bullet$ 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 lax sense and assumes we are not in the context of intersection elimination (T-TopLevel). It also holds that an elaboration that succeeds in strict mode is also successful in lax mode (T-Weaken), so all strict rules can also be used as lax 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 an expression of the form `let` $x = e_1$ `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 have a pair form.

Unions Rule T-$\vee$I for union introduction is standard. The union elimination rule, taken from [11], 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 [10] 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 lax 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 lax mode, potentially producing `DEAD`-casts.

Trusting via **`DEAD`*-Casts*** The cornerstone of the "trust" phase lies in the presence of the T-$\bot$ rule. As we mentioned earlier, this rule can only be used in lax 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 `number` and `number` $\rightarrow$ `number`) 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 *lax* context, then for the application $f\ x$, where f has been assigned the type $f{:}\mathtt{I} \rightarrow \mathtt{I} \wedge \mathtt{B} \rightarrow \mathtt{B}$ and $x{:}\mathtt{B}$, the following derivation would be possible:

$$\text{T-App}\ \frac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_{\mathsf{L}} f :: \mathtt{I} \rightarrow \mathtt{I} \overset{\mathsf{T}}{\hookrightarrow} \mathsf{proj}_1 f} \qquad \text{T-}\bot\ \dfrac{\begin{array}{c}\ldots \vdash_{\mathsf{L}} x :: \mathtt{B} \overset{\mathsf{F}}{\hookrightarrow} x \\ \mathsf{TAG}(\mathtt{B}) \cap \mathsf{TAG}(\mathtt{I}) = \emptyset\end{array}}{\ldots \vdash_{\mathsf{L}} x :: \mathtt{I} \overset{\mathsf{F}}{\hookrightarrow} \mathsf{DEAD}_{\mathtt{B}\downarrow\mathtt{I}}\langle x\rangle}}{f{:}\mathtt{I} \rightarrow \mathtt{I} \wedge \mathtt{B} \rightarrow \mathtt{B}, x{:}\mathtt{B} \vdash_{\mathsf{L}} f\ x :: \mathtt{I} \overset{\mathsf{F}}{\hookrightarrow} (\mathsf{proj}_1 f)\ (\mathsf{DEAD}_{\mathtt{B}\downarrow\mathtt{I}}\langle x\rangle)}$$

But, clearly, the intended derivation here is:

$$\text{T-App}\ \frac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_{\mathsf{L}} f :: \mathtt{B} \rightarrow \mathtt{B} \overset{\mathsf{T}}{\hookrightarrow} \mathsf{proj}_2 f} \qquad \ldots \vdash_{\mathsf{S}} x :: \mathtt{B} \overset{\mathsf{F}}{\hookrightarrow} \mathtt{B}}{f{:}\mathtt{I} \rightarrow \mathtt{I} \wedge \mathtt{B} \rightarrow \mathtt{B}, x{:}\mathtt{B} \vdash_{\mathsf{L}} f\ x :: \mathtt{B} \overset{\mathsf{F}}{\hookrightarrow} (\mathsf{proj}_2 f)\ x}$$

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 ($\leqslant$) and a subsumption rule for our typing elaboration. The rules for this subtyping relation would include, among others, function subtyping:

$$\frac{A_1' \leqslant A_1 \qquad A_2 \leqslant A_2'}{A_1 \rightarrow A_2 \leqslant A_1' \rightarrow A_2'}$$

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 ($\triangleright$):

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

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 Consistency

Next, we present the theorems that precisely connect the semantics of source programs with their elaborated targets. The first result is analogous to one from [11] and 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.

Theorem 1 (Consistency). *If* $\cdot \vdash e :: A \hookrightarrow M$ *and* $M \longrightarrow M'$ *then there exists* e' *such that* $e \longrightarrow^* e'$ *and* $\cdot \vdash e' :: A \hookrightarrow M'$.

Theorem 1 states that the behaviors of the target *under-approximate* the behaviors of the source. 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. Hence, unlike [11], we establish a completeness result that states that if the source term steps, then the elaborated program will also eventually step to a corresponding (by elaboration) term.

Theorem 2 (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* $M \longrightarrow^+ M'$.

In other words, Theorem 2 states that the behaviors of the elaborated target *over-approximate* those of the source, and hence, in conjunction with Theorem 1, ensure that the source "goes wrong" iff the target does.

The main challenge towards establishing the above theorems is 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. For space reasons, we defer the full proofs to the appendix; but we describe the notable auxiliary lemmas used to establish the above theorems.

Value Monotonicity This result fills in the mismatch that emerges when (non-value) expressions in the source language elaborate to values in the target language. The lemma states that, 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 *s.t.:*

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

We also require the reverse of the above lemma, 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. 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:

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

This lemma establishes 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.

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

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 $\mathtt{c}$ 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 Assumption 1.

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

Substitution lemma The value-monotonicity lemma and primitive substitution assumption establish the substitution lemma, which connects values in the source and target.

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

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 1 and 2 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 [9]: as long as the target is safe, the consistency theorems guarantee that the source is safe. Thus, we have completely decoupled reasoning about values (phase 2) from reasoning about basic types (phase 1).

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

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

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

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

$$\begin{array}{l}\textsc{R-If}\\ \dfrac{G \vdash M :: \mathtt{boolean} \qquad G, M \vdash M_1 :: T \qquad G, \neg M \vdash M_2 :: T}{G \vdash M\ ?\ M_1\ :\ M_2 :: T}\end{array} \qquad \begin{array}{l}\textsc{R-Lam}\\ \dfrac{G, x : T_x; G \vdash M :: T}{G \vdash \lambda x.M :: T_x \to T}\end{array}$$

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

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

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

$$\sqsubseteq\textsc{-Base}\ \frac{\mathsf{Valid}([\![G]\!] \wedge [\![p]\!] \Rightarrow [\![p']\!])}{G \vdash \{v{:}\mathbb{B} \mid p\} \sqsubseteq \{v{:}\mathbb{B} \mid p'\}} \qquad \sqsubseteq\textsc{-Fun}\ \frac{G \vdash T'_x \sqsubseteq T_x \qquad G, x{:}T'_x \vdash T \sqsubseteq T'}{G \vdash (x : T_x) \to T \sqsubseteq (x : T'_x) \to T'}$$

Fig. 9: Refined Typechecking

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 [24]. 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 [19], 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 [24]. 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 [19] each primitive constant c has a refined type $\mathtt{ty\_c}$, and a variable x with type T is typed as $\mathsf{sngl}(T, x)$ which is $\{v{:}\mathbb{B} \mid v = x\}$ 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, $\mathsf{DEAD}_{A \downarrow B}\langle M \rangle$ is treated as call to:

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

The notation $[\cdot]$ denotes the elaboration of $\lambda^{\wedge}_{\vee}$ types to $\lambda^{\times}_{+}$ types [11]:

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

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

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

Returning to rule R-APP for `DEAD`-casts and inverting, the 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 refinement 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 [19]. 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. ben ♣ the lemma would look a lot nicer if the name weren't abbreviated and there was an "and" in the list of conditions - do we need to save a line that badly? ♣

Lemma 4 (Corresp.). *If* $\Gamma \vdash e :: A \hookrightarrow M$, $G \vdash M :: T$, $[\Gamma] = \|G\|$, *then* $[A] = \|T\|$.

The target language satisfies a progress and preservation theorem [19]:

Theorem 3 (Refinement Type Safety). *If* $\cdot \vdash M : T$ *then either* M *is a value or there exists* M' *such that* $M \longrightarrow M'$ *and* $\cdot \vdash M' : T$.

5.2 Two-Phase Type Safety

Well Two-Typed Terms 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 1 and 2 with refinement safety 3 yield end-to-end soundness: well two-typed terms do not get stuck, and step to well two-typed terms.

Theorem 4 (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.*

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. [18] 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* [25] 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, [28,29] formalize the support for type tests as *occurrence typing* and extend it to an interprocedural, higher-order setting by introducing propositional *latent predicates* that reflect the result of tests in TypedRacket function signatures.

Types for Dynamic Imperative Languages [27] and [1] describe early attempts towards a static type system for JavaScript, and [16] present DRuby, a tool for type inference for Ruby scripts. However, these systems do not account for or check value-based overloading (like TypeScript, DRuby allows overloaded specifications for external functions). [17,20] account for tests using flow analysis to bring occurrence typing to the imperative JavaScript setting. However, unlike our two-phased approach, the above methods restrict themselves to a *fixed* set of type-testing idioms (*e.g.* `typeof`), thereby 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. [3] investigate semantic subtyping in a first order language with refinements and type-test expressions. [7] introduce *nested refinement types*, where the typing relation itself is a predicate in the refinement logic, and presents a feature-rich language of predicates that 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 and so on) 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, by breaking the dependency, 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 is the use of intersection and union types: [22] indicates the connection between unions and intersections with sums and products, that is the basis of the elaboration scheme from [11] on which we build. However, [11] studies *static* source languages that use *explicit* overloading via a merge operator [23]. 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. [6] describe a λ&-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 [30] describe an early refinement type system composing ML's types with a decidable constraint system. [19] present *hybrid* type checking for a language with arbitrary refinements over basic types. They use a static type system to verify basic specifications and defer more complex ones to dynamically checked contracts, since the logic of their specifications 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* [24] to infer refinements using predicate abstraction.

7 Conclusions and Future Work

In this paper we introduced 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 [29], we are not limited to a fixed set of tag- or type- tests, and unlike purely program logic-based approaches like [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 *e.g.* ownership types [8] or [31] in the *first phase*, and we intend to investigate this route in future work to obtain a practical verifier for TypeScript.

References

1. Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. Towards Type Inference for JavaScript. In *ECOOP*, 2005.
2. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 2011.
3. Gavin M Bierman, Andrew D Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an smt solver. In *ICFP*, 2010.
4. L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *ENTCS*, 80, 2003.
5. Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, 1991.
6. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *LFP*, 1992.
7. Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.
8. David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL*. ACM, 1977.
10. Joshua Dunfield. Untangling typechecking of intersections and unions. In *ITRS*, volume 45 of *EPTCS*, pages 59–70, 2011. `arXiv:1101.4428[cs.PL]`.
11. Joshua Dunfield. Elaborating intersection and union types. In *ICFP*, 2012.
12. Flow: A Static Type Checker for JavaScript. `http://flowtype.org`.
13. Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *OOPSLA*. ACM, 2014.
14. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001.
15. R.W. Floyd. Assigning meanings to programs. In *Math. Asp. of Comp. Sc.* 1967.
16. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *SAC*, 2009.
17. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *ESOP*, 2011.
18. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. In *FPCA*, 1995.
19. K.W. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 2010.
20. Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: Retrofitting type systems for javascript. In *DLS*, 2013.
21. Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *ICFP*, 2014.
22. Benjamin Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University, 1991.
23. John C. Reynolds. Algol-like languages, volume 1. chapter Design of the Programming Language FORSYTHE, pages 173–233. Birkhauser Boston Inc., 1997.
24. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid Types. In *PLDI*, 2008.
25. Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007.
26. The Satisfiability Modulo Theories Library. `http://smt-lib.org`.
27. Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.
28. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.

29. Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.
30. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.
31. Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE*, 2007.

Appendix

For the proofs of the following theorems we are going to use the plain version of the elaboration relation, *i.e.* without mode annotations:

$$\Gamma \vdash e :: A \hookrightarrow M$$

The annotations on the judgment merely determine which rules are available at type-checking. The majority of the proofs below involve induction over the elaboration derivation, which is fixed once type-checking is complete, so the annotations can be safely ignored.

In certain lemmas the reader is referred to [11]. The proofs there refer to a language similar but not exactly the same as ours. The main proof ideas, however, hold.

Numbering refers to the appendix itself unless otherwise stated.

A Assumptions

Assumption 1 (Primitive constant application) *If*

(1) $\cdot \vdash \mathsf{c} :: A \to B \hookrightarrow \mathsf{c}$,

(2) $\cdot \vdash v :: A \hookrightarrow W$,

(3) $W \not\equiv \mathsf{DEAD}_{\downarrow A}\langle\cdot\rangle$,

then

- $\mathsf{c}\, v \longrightarrow [\![\mathsf{c}]\!](v)$
- $\mathsf{c}\, W \longrightarrow [\![\mathsf{c}]\!](W)$
- $\cdot \vdash [\![\mathsf{c}]\!](v) :: B \hookrightarrow [\![\mathsf{c}]\!](W)$

Assumption 2 (Lambda Application) *If*

(1) $\cdot \vdash \lambda x.e :: A \to B \hookrightarrow \lambda x.M$,

(2) $\cdot \vdash v :: A \hookrightarrow W$,

(3) $W \not\equiv \mathsf{DEAD}_{\downarrow A}\langle\cdot\rangle$,

then

- $(\lambda x.e)\, v \longrightarrow [v/x]\, e$
- $(\lambda x.M)\, W \longrightarrow [W/x]\, M$

Assumption 3 (Canonical Forms)

(1) If $\Gamma \vdash \lambda x.e :: A \to B \hookrightarrow W$ *then*

- $W \equiv \lambda x.M$ *for some* M*, or*
- $W \equiv \mathsf{DEAD}_{\cdot \downarrow A \to B}\langle W' \rangle$ *for some* W'

(2) If $\Gamma \vdash c :: A \hookrightarrow W$ *then*

- $W \equiv c$*, or*
- $W \equiv \mathsf{DEAD}_{\cdot \downarrow A}\langle W' \rangle$ *for some* W'

B Auxiliary lemmas

Lemma 1 (Multi-Step Source Evaluation Context).
If $e \longrightarrow^* e'$ *then* $E[e] \longrightarrow^* E[e']$.

Proof. Based on Lemma 7 of [11].

Lemma 2 (Multi-Step Target Evaluation Context).

- *If* $M \longrightarrow^* M'$ *then* $\mathcal{E}[M] \longrightarrow^* \mathcal{E}[M']$
- *If* $M \longrightarrow^+ M'$ *then* $\mathcal{E}[M] \longrightarrow^+ \mathcal{E}[M']$

Proof. Similar to proof of lemma 1.

Lemma 3 (Unions/Injections). *If* $\Gamma \vdash e :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_k\ M$ *then* $\Gamma \vdash e :: A_k \hookrightarrow M$.

Proof. Based on Lemma 8 of [11].

Lemma 4 (Intersections/Pairs). *If* $\Gamma \vdash e :: A_1 \wedge A_2 \hookrightarrow (M_1, M_2)$ *then there exist* e_1' *and* e_2' *such that:*

(1) $e_1 \longrightarrow^* e_1'$ *and* $\Gamma \vdash e_1' :: A_1 \hookrightarrow M_1$

(2) $e_2 \longrightarrow^* e_2'$ *and* $\Gamma \vdash e_2' :: A_2 \hookrightarrow M_2$

Proof. Based on Lemma 9 of [11].

Lemma 5 (Beta Reduction Canonical Form). *If*

(1) $\cdot \vdash \lambda x.e :: A \to B \hookrightarrow W_1$,

(2) $\cdot \vdash v_2 :: A \hookrightarrow W_2$,

(3) $(\lambda x.e)\ v_2 \longrightarrow [v_2/x]\ e$

Then $W_1 \equiv \lambda x.M$ *for some* M.

Lemma 6 (Primitive Reduction Canonical Form). *If*

(1) $\cdot \vdash \mathtt{c} :: A \to B \hookrightarrow W_1$

(2) $\cdot \vdash v :: A \hookrightarrow W_2$,

(3) $\mathtt{c}\ v \longrightarrow [\![\mathtt{c}]\!](v)$

Then:

- $W_1 \equiv \mathtt{c}$
- $W_2 \not\equiv \mathtt{DEAD}_{\cdot \downarrow A}\langle\cdot\rangle$

Lemma 7 (Conditional Canonical Form). *If*

(1) $\cdot \vdash \mathtt{c} :: \mathtt{boolean} \hookrightarrow W$,

(2) $\cdot \vdash e_1 :: A \hookrightarrow M_1$ *and* $\cdot \vdash e_2 :: A \hookrightarrow M_2$,

(3) $\mathtt{c}\ ?\ e_1\ :\ e_2 \longrightarrow e_k$

Then:

- $k = 1 \Rightarrow \mathtt{c} \equiv W \equiv \mathtt{true}$.
- $k = 2 \Rightarrow \mathtt{c} \equiv W \equiv \mathtt{false}$.

Lemma 8 (Value Monotonicity). *If* $\Gamma \vdash e :: A \hookrightarrow W$*, then there exists* v *s.t.:*

(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. Parts (1) and (2) of the lemma has been proved by [11] for a similar language, so here we are just going to prove part (3).

We will show this by induction on the length i of the path: $e \longrightarrow^* e_i$.

- For the case $i = 0$: $e = e_i$, so it trivially holds.
- Suppose it holds for $i = k$, *i.e.* for $e \longrightarrow^* e_k$, it holds that:

$$\Gamma \vdash e_k :: A \hookrightarrow W \tag{2.1}$$

We will show that it holds for $i = k + 1$, *i.e.* for e_{k+1} such that:

$$e_k \longrightarrow e_{k+1} \tag{2.2}$$

We will do this by induction on the derivation (2.1), but limit ourselves to the terms e_k that elaborate to *values*:

- ▷ Cases T-CST, T-VAR, T-$\wedge$I, T-LAM: For these cases, term e_k is already a value, so doesn't step.

- ▷ Case T-$\vee$I (assume left injection – the case for right injection is similar):

$$\frac{\Gamma \vdash e_k :: A_1 \hookrightarrow W \qquad \vdash A_1 \vee A_2}{\Gamma \vdash e_k :: A_1 \vee A_2 \hookrightarrow \texttt{inj}_1\ W}$$

  By inversion:

$$\Gamma \vdash e_k :: A_1 \hookrightarrow W$$

  By i.h., using (2.2):

$$\Gamma \vdash e_{k+1} :: A_1 \hookrightarrow W$$

  Applying rule T-$\vee$I on the latter one:

$$\Gamma \vdash e_{k+1} :: A_1 \vee A_2 \hookrightarrow \texttt{inj}_1\ W$$

Lemma 9 (Reverse Value Monotonicity). *If* $\Gamma \vdash v :: A \hookrightarrow M$*, then exists* W *s.t.:* $M \longrightarrow^* W$ *and* $\Gamma \vdash v :: A \hookrightarrow W$*.*

Lemma 10 (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. *Based on Lemma 12 of [11].*

Corollary 1 (Target Multi-step Preservation). *If* $\cdot \vdash M :: T$ *and* $M \longrightarrow^* M'$ *then* $\cdot \vdash M' :: T$*.*

Proof. *Stems from Theorem 3 from main paper.*

Corollary 2 (DEAD-cast Invalid). $\cdot \nvdash \mathsf{DEAD}_{A \downarrow B}\langle M \rangle :: T$

C Theorems

Theorem 1 (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. By induction on the derivation $\cdot \vdash e :: A \hookrightarrow M$

- Cases T-CST, T-VAR, T-$\wedge$I, T-$\wedge$E and T-LAM:
  The respective target expression does not step.

- Case T-LET:

$$\frac{\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \qquad x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2}{\cdot \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: A_2 \hookrightarrow \texttt{let}\ x = M_1\ \texttt{in}\ M_2}$$

  By inversion:

$$\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \tag{2.1}$$

$$x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2 \tag{2.2}$$

  Cases on the form of $M \longrightarrow M'$:

▷ Subcase:

$$\frac{M_1 \longrightarrow M_1'}{\texttt{let}\ x = M_1\ \texttt{in}\ M_1 \longrightarrow \texttt{let}\ x = M_1'\ \texttt{in}\ M_2}$$

By inversion:

$$M_1 \longrightarrow M_1' \tag{2.3}$$

By i.h. on (2.1) and (2.3) there exists e_1' *s.t.*:

$$\begin{aligned} e_1 &\longrightarrow^* e_1' \\ \cdot \vdash e_1' &:: A_1 \hookrightarrow M_1' \end{aligned} \tag{2.4}$$

Applying rule T-LET on (2.2) and (2.4):

$$\cdot \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 :: A_2 \hookrightarrow \texttt{let}\ x = M_1'\ \texttt{in}\ M_2$$

▷ Subcase:

$$\texttt{let}\ x = W_1\ \texttt{in}\ M_2 \longrightarrow [W_1/x]\ M_2$$

Equation (2.1) becomes:

$$\cdot \vdash e_1 :: A_1 \hookrightarrow W_1$$

By Lemma 8 there exists v_1 such that:

$$e_1 \longrightarrow^* v_1 \tag{2.5}$$

$$\cdot \vdash v_1 :: A_1 \hookrightarrow W_1 \tag{2.6}$$

By Lemma 1 using (2.5) on $E \equiv \texttt{let}\ x = \langle\ \rangle\ \texttt{in}\ e_2$:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow^* \texttt{let}\ x = v_1\ \texttt{in}\ e_2$$

By Lemma 10 on (2.2) and (2.6), there exists $e' \equiv [v_1/x]\ e$ *s.t.*:

$$\begin{aligned} \texttt{let}\ x = v_1\ \texttt{in}\ e_2 &\longrightarrow [v_1/x]\ e_2 \\ \cdot \vdash [v_1/x]\ e_2 &:: A_2 \hookrightarrow [W_1/x]\ M_2 \end{aligned}$$

- Case T-IF:

$$\frac{\cdot \vdash e_c :: \texttt{boolean} \hookrightarrow M \qquad \forall i \in \{1,2\}\ .\ \cdot \vdash e_i :: A \hookrightarrow M_i}{\cdot \vdash e_c\ ?\ e_1\ :\ e_2 :: A \hookrightarrow M_c\ ?\ M_1\ :\ M_2}$$

By inversion:

$$\cdot \vdash e_c :: \texttt{boolean} \hookrightarrow M_c \tag{3.1}$$

$$\cdot \vdash e_1 :: A \hookrightarrow M_1 \tag{3.2}$$

$$\cdot \vdash e_2 :: A \hookrightarrow M_2 \tag{3.3}$$

Cases on the form of $M \longrightarrow M'$:

▷ Subcase:

$$\frac{M_c \longrightarrow M_c'}{M_c \mathrel{?} M_1 : M_2 \longrightarrow M_c' \mathrel{?} M_1 : M_2}$$

By inversion:

$$M_c \longrightarrow M_c' \tag{3.4}$$

By i.h. using (3.1) and (3.4) there exists e_c' such that

$$e_c \longrightarrow^* e_c'$$
$$\cdot \vdash e_c' :: \texttt{boolean} \hookrightarrow M_c' \tag{3.5}$$

Applying rule T-IF on (3.5), (3.2) and (3.3) we get:

$$\cdot \vdash e_c' \mathrel{?} e_1 : e_2 :: A \hookrightarrow M_c' \mathrel{?} M_1 : M_2$$

▷ Subcase:

$$\texttt{true} \mathrel{?} M_1 : M_2 \longrightarrow M_1$$

Equation 3.1 becomes:

$$\cdot \vdash e_c :: \texttt{boolean} \hookrightarrow \texttt{true}$$

By Lemma 8 there exists v_c such that:

$$e_c \longrightarrow^* v_c \tag{3.6}$$
$$\cdot \vdash v_c :: \texttt{boolean} \hookrightarrow \texttt{true} \tag{3.7}$$

The only possible case for (3.7) to hold is:

$$v_c \equiv \texttt{true}$$

By applying Lemma 1 using (3.6) on $E \equiv \langle\,\rangle \mathrel{?} e_1 : e_2$:

$$e_c \mathrel{?} e_1 : e_2 \longrightarrow^* \texttt{true} \mathrel{?} e_1 : e_2$$

By E-COND-TRUE:

$$\texttt{true} \mathrel{?} e_1 : e_2 \longrightarrow e_1$$

So there exists $e' \equiv e_1$, such that $e \longrightarrow^* e'$ and by (3.2) it holds that:

$$\cdot \vdash e' :: A \hookrightarrow M_1$$

▷ Subcase:

$$\texttt{false} \mathrel{?} M_1 : M_1 \longrightarrow M_2$$

This case is similar to the previous one.

- Case T-APP: *Similar to proof given by [11] in proof of theorem 13.*

- Case T-$\vee$I:
$$\frac{\cdot \vdash e :: A_k \hookrightarrow M_0 \qquad \vdash A_1 \vee A_2}{\cdot \vdash e :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_k\ M_0}$$
By inversion:
$$\cdot \vdash e :: A_k \hookrightarrow M_0 \tag{5.1}$$
$$\vdash A_1 \vee A_2 \tag{5.2}$$
The only possible case for $M \longrightarrow M'$ is:
$$\frac{M_0 \longrightarrow M_0'}{\mathtt{inj}_k\ M_0 \longrightarrow \mathtt{inj}_k\ M_0'}$$
By inversion:
$$M_0 \longrightarrow M_0' \tag{5.3}$$
By i.h. using (5.1) and (5.3) there exists an e' such that:
$$e \longrightarrow^* e' \tag{5.4}$$
$$\cdot \vdash e' :: A_k \hookrightarrow M_0' \tag{5.5}$$
By T-$\vee$I on (5.5) and (5.2):
$$\cdot \vdash e' :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_k\ M_0'$$
- Case T-$\vee$E:
$$\frac{\cdot \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{l} x_1 : A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \\ x_2 : A_2 \vdash E[x_2] :: B \hookrightarrow M_2 \end{array}}{\cdot \vdash E[e_0] :: B \hookrightarrow \mathtt{case}\ M_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow M_2}$$
By inversion:
$$\cdot \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \tag{6.1}$$
$$x_1 : A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \tag{6.2}$$
$$x_2 : A_2 \vdash E[x_2] :: B \hookrightarrow M_2 \tag{6.3}$$
Cases on the form of $M \longrightarrow M'$:

  ▷ Subcase:
$$\frac{M_0 \longrightarrow M_0'}{\begin{array}{c}\mathtt{case}\ M_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow \\ \mathtt{case}\ M_0'\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow M_2\end{array}}$$
  By inversion:
$$M_0 \longrightarrow M_0' \tag{6.4}$$

By i.h. using (6.1) and (6.4) there exists e_0' such that

$$e_0 \longrightarrow^* e_0' \tag{6.5}$$
$$\cdot \vdash e_0' :: A_1 \vee A_2 \hookrightarrow M_0' \tag{6.6}$$

Applying T-$\vee$E on (6.6), (6.2) and (6.3):

$$\cdot \vdash E[e_0'] :: A_1 \hookrightarrow \texttt{case}\ M_0'\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2$$

By applying Lemma 2 using 6.5:

$$E[e_0] \longrightarrow^* E[e_0']$$

▷ Subcase:

$$\texttt{case}\ \texttt{inj}_1\ W\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow [W/x_1]\ M_1$$

Equation (6.1) becomes:

$$\cdot \vdash e_0 :: A_1 \vee A_2 \hookrightarrow \texttt{inj}_1\ W \tag{6.7}$$

Applying Lemma 8 on (6.7), there exists v_0 such that:

$$e_0 \longrightarrow^* v_0 \tag{6.8}$$
$$\cdot \vdash v_0 :: A_1 \vee A_2 \hookrightarrow \texttt{inj}_1\ W \tag{6.9}$$

By applying Lemma 3 on (6.9):

$$\cdot \vdash v_0 :: A_1 \hookrightarrow W \tag{6.10}$$

Applying Lemma 10 on (6.2) and (6.10):

$$\cdot \vdash [v_0/x_1]\ E[x_1] :: A_1 \hookrightarrow [W/x_1]\ M_1$$

Or, after the substitutions[3]:

$$\cdot \vdash E[v_0] :: A_1 \hookrightarrow [W/x_1]\ M_1 \tag{6.11}$$

Applying Lemma 1 on (6.8):

$$E[e_0] \longrightarrow^* E[v_0]$$

▷ Subcase:

$$\texttt{case}\ \texttt{inj}_2\ W\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow [W/x_2]\ M_2$$

is similar to the previous one. □

[3] Variable x_1 is only referenced in the "hole" of the evaluation context $E[x_1]$.

- Case T-⊥:

$$\frac{\cdot \vdash e :: A \hookrightarrow M \qquad \mathtt{TAG}(A) \cap \mathtt{TAG}(B) = \emptyset}{\cdot \vdash e :: B \hookrightarrow \mathtt{DEAD}_{A\downarrow B}\langle M \rangle}$$

  By inversion:

$$\cdot \vdash e :: A \hookrightarrow M \tag{7.1}$$
$$\mathtt{TAG}(A) \cap \mathtt{TAG}(B) = \emptyset \tag{7.2}$$

  The only possible step here is:

$$\frac{M \longrightarrow M'}{\mathtt{DEAD}_{A\downarrow B}\langle M \rangle \longrightarrow \mathtt{DEAD}_{A\downarrow B}\langle M' \rangle}$$

  By inversion:

$$M \longrightarrow M' \tag{7.3}$$

  By i.h. using (7.1) and (7.3) there exists e' such that:

$$e \longrightarrow^* e'$$
$$\cdot \vdash e' :: A \hookrightarrow M' \tag{7.4}$$

  By applying T-⊥ on (7.4) and (7.2):

$$\cdot \vdash e' :: B \hookrightarrow \mathtt{DEAD}_{A\downarrow B}\langle M' \rangle$$

Theorem 2 (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. By induction on the derivation $\cdot \vdash e :: A \hookrightarrow M$

- Cases T-Cst, T-Var, T-∧I, T-Lam:
  The respective source expression does not step.

- Case T-Let:

$$\frac{\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \qquad x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2}{\cdot \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: A_2 \hookrightarrow \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2} \tag{2.1}$$

  By inversion:

$$\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \tag{2.2}$$
$$x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2 \tag{2.3}$$

  Cases on the form of $e \longrightarrow e'$:

▷ Subcase:

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow \texttt{let } x = e_1' \texttt{ in } e_2}$$

By inversion:

$$e_1 \longrightarrow e_1' \tag{2.4}$$

By i.h. using (2.2) and (2.4): There exists M_1' such that:

$$\cdot \vdash e_1' :: A_1 \hookrightarrow M_1' \tag{2.5}$$
$$M_1 \longrightarrow^+ M_1' \tag{2.6}$$

Applying rule T-LET on (2.3) and (2.5):

$$\cdot \vdash \texttt{let } x = e_1' \texttt{ in } e_2 :: A_2 \hookrightarrow \texttt{let } x = M_1' \texttt{ in } M_2$$

By Lemma 2 on (2.6) we get:

$$M \longrightarrow^+ M'$$

▷ Subcase:

$$\texttt{let } x = v_1 \texttt{ in } e_2 \longrightarrow [v_1/x]\ e_2$$

Equation (2.2) becomes:

$$\cdot \vdash v_1 :: A_1 \hookrightarrow M_1 \tag{2.7}$$

By Lemma 9 on (2.7) there exists W_1 such that:

$$M_1 \longrightarrow^* W_1 \tag{2.8}$$
$$\cdot \vdash v_1 :: A_1 \hookrightarrow W_1 \tag{2.9}$$

By Lemma 2 using (2.8) on $\mathcal{E} \equiv \texttt{let } x = \langle\,\rangle \texttt{ in } M_2$:

$$\texttt{let } x = M_1 \texttt{ in } M_2 \longrightarrow^* \texttt{let } x = W_1 \texttt{ in } M_2 \tag{2.10}$$

By TE-LET:

$$\texttt{let } x = W_1 \texttt{ in } M_2 \longrightarrow [W_1/x]\ M_2 \tag{2.11}$$

By (2.10) and (2.11):

$$\texttt{let } x = M_1 \texttt{ in } M_2 \longrightarrow^+ [W_1/x]\ M_2$$

And by Lemma 10 on (2.3) and (2.9):

$$\cdot \vdash [v_1/x]\ e_2 :: A_2 \hookrightarrow [W_1/x]\ M_2$$

- Case T-IF:

$$\frac{\cdot \vdash e_c :: \texttt{boolean} \hookrightarrow M \qquad \forall i \in \{1,2\} \,.\, \cdot \vdash e_i :: A \hookrightarrow M_i}{\cdot \vdash e_c \;?\; e_1 \;:\; e_2 :: A \hookrightarrow M_c \;?\; M_1 \;:\; M_2} \tag{3.1}$$

By inversion:

$$\cdot \vdash e_c :: \texttt{boolean} \hookrightarrow M_c \tag{3.2}$$
$$\cdot \vdash e_1 :: A \hookrightarrow M_1 \tag{3.3}$$
$$\cdot \vdash e_2 :: A \hookrightarrow M_2 \tag{3.4}$$

Cases on the form of $e \longrightarrow e'$:

▷ Subcase:

$$\frac{e_c \longrightarrow e_c'}{e_c \;?\; e_1 \;:\; e_2 \longrightarrow e_c' \;?\; e_1 \;:\; e_2}$$

By inversion:

$$e_c \longrightarrow e_c' \tag{3.5}$$

By i.h. using (3.2) and (3.5) there exists M_c' such that:

$$\cdot \vdash e_c' :: \texttt{boolean} \hookrightarrow M_c' \tag{3.6}$$
$$M_c \longrightarrow^+ M_c' \tag{3.7}$$

By Lemma 2 using (3.7):

$$M_c \;?\; M_1 \;:\; M_2 \longrightarrow^+ M_c' \;?\; M_1 \;:\; M_2$$

Applying rule T-IF on (3.6), (3.3) and (3.4) we get:

$$\cdot \vdash e_c' \;?\; e_1 \;:\; e_2 :: A \hookrightarrow M_c' \;?\; M_1 \;:\; M_2$$

▷ Subcase:

$$\texttt{true} \;?\; e_1 \;:\; e_2 \longrightarrow e_1 \tag{3.8}$$

Equation 3.2 becomes:

$$\cdot \vdash \texttt{true} :: \texttt{boolean} \hookrightarrow M_c \tag{3.9}$$

By Lemma 9 there exists W_c such that:

$$M_c \longrightarrow^* W_c \tag{3.10}$$
$$\cdot \vdash \texttt{true} :: \texttt{boolean} \hookrightarrow W_c \tag{3.11}$$

By Lemma 2 using (3.10):

$$M_c \;?\; M_1 \;:\; M_2 \longrightarrow^* W_c \;?\; M_1 \;:\; M_2 \tag{3.12}$$

By Lemma 7 on (3.11), (3.3), (3.4) and (3.8):

$$W_c \equiv \texttt{true} \tag{3.13}$$

By TE-COND-TRUE:

$$\texttt{true} \ ? \ M_1 \ : \ M_2 \longrightarrow M_1 \tag{3.14}$$

By (3.12) and (3.14):

$$M_c \ ? \ M_1 \ : \ M_2 \longrightarrow^+ M_1$$

Combining with (3.3) we get the wanted relation.

▷ Subcase:

$$\texttt{false} \ ? \ e_1 \ : \ e_1 \longrightarrow e_2$$

Similar to the previous case.

- Case T-∧E: *Similar to earlier cases.*
- Case T-APP:

$$\frac{\cdot \vdash e_1 :: A \to B \hookrightarrow M_1 \qquad \cdot \vdash e_2 :: A \hookrightarrow M_2}{\cdot \vdash e_1 \ e_2 :: B \hookrightarrow M_1 \ M_2} \tag{5.1}$$

By inversion:

$$\cdot \vdash e_1 :: A \to B \hookrightarrow M_1 \tag{5.2}$$
$$\cdot \vdash e_2 :: A \hookrightarrow M_2 \tag{5.3}$$

Cases on the form of $e \longrightarrow e'$:

▷ Subcase:

$$\frac{e_1 \longrightarrow e_1'}{e_1 \ e_2 \longrightarrow e_1' \ e_2}$$

Similar to eariler cases, e.g. $\texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow \texttt{let } x = e_1' \texttt{ in } e_2$

▷ Subcase:

$$\frac{e_2 \longrightarrow e_2'}{v_1 \ e_2 \longrightarrow v_1 \ e_2'}$$

By inversion:

$$e_2 \longrightarrow e_2' \tag{5.4}$$

By Lemma 9 on (5.2) there exists W_1 such that:

$$M_1 \longrightarrow^* W_1 \tag{5.5}$$
$$\cdot \vdash v_1 :: A_2 \hookrightarrow W_1 \tag{5.6}$$

By Lemma 2 using (5.5):

$$M_1\ M_2 \longrightarrow^* W_1\ M_2 \tag{5.7}$$

By i.h. using (5.3) and (5.4) there exists M_2' such that:

$$M_2 \longrightarrow^+ M_2' \tag{5.8}$$
$$\cdot \vdash e_2' :: A \hookrightarrow M_2' \tag{5.9}$$

By Lemma 2 using (5.8) on the target of (5.6):

$$W_1\ M_2 \longrightarrow^+ W_1\ M_2'$$

And combining with (5.7):

$$M_1\ M_2 \longrightarrow^+ W_1\ M_2'$$

By rule T-APP using (5.2) and (5.9):

$$\cdot \vdash v_1\ e_2' :: A \hookrightarrow W_1\ M_2'$$

▷ Subcase:

$$(\lambda x.e_0)\ v_2 \longrightarrow [v_2/x]\ e_0 \tag{5.10}$$

By Lemma 9 on (5.2) there exists W_1 such that:

$$\cdot \vdash \lambda x.e_0 :: A \to B \hookrightarrow W_1 \tag{5.11}$$
$$M_1 \longrightarrow^* W_1 \tag{5.12}$$

By applying Lemma 2 on $M \equiv M_1\ M_2$ given (5.15):

$$M_1\ M_2 \longrightarrow^* W_1\ M_2 \tag{5.13}$$

Equation (5.3) is:

$$\cdot \vdash v_2 :: A \hookrightarrow M_2 \tag{5.14}$$

By Lemma 9 on (5.14), there exists W_2 such that:

$$M_2 \longrightarrow^* W_2 \tag{5.15}$$
$$\cdot \vdash v_2 :: A \hookrightarrow W_2 \tag{5.16}$$

By Lemma 5 on (5.11), (5.16) and (5.10), there is a M_0 such that:

$$W_1 \equiv \lambda x.M_0$$

So (5.2) becomes:

$$\cdot \vdash \lambda x.e_0 :: A \to B \hookrightarrow \lambda x.M_0 \tag{5.17}$$

The only production of (5.17) is by T-LAM:

$$\frac{\vdash A \to B \qquad x{:}A \vdash e_0 :: B \hookrightarrow M_0}{\Gamma \vdash \lambda x.e_0 :: A \to B \hookrightarrow \lambda x.M_0}$$

By inversion:

$$x{:}A \vdash e_0 :: B \hookrightarrow M_0 \tag{5.18}$$

By applying Lemma 10 on (5.18) and (5.16) we get:

$$\cdot \vdash [v_2/x]\ e_0 :: B \hookrightarrow [W_2/x]\ M_0 \tag{5.19}$$

By applying Lemma 2 on $M \equiv (\lambda x.M_0)\ M_2$ given (5.15):

$$(\lambda x.M_0)\ M_2 \longrightarrow^* (\lambda x.M_0)\ W_2 \tag{5.20}$$

By rule TE-APP-2:

$$(\lambda x.M_0)\ W_2 \longrightarrow [W_2/x]\ M_0 \tag{5.21}$$

By (5.13), (5.20) and (5.21) we get:

$$M_1\ M_2 \longrightarrow^+ [W_2/x]\ M_0 \tag{5.22}$$

By (5.19) and (5.22) we get the wanted relation.

▷ Subcase:

$$\mathsf{c}\ v \longrightarrow [\![\mathsf{c}]\!](v) \tag{5.23}$$

Equations (5.2) and (5.3) become:

$$\cdot \vdash \mathsf{c} :: A \to B \hookrightarrow M_1 \tag{5.24}$$
$$\cdot \vdash v :: A \hookrightarrow M_2 \tag{5.25}$$

By Lemma 9 on (5.24) there exists W_1 such that:

$$\cdot \vdash \mathsf{c} :: A \to B \hookrightarrow W_1 \tag{5.26}$$
$$M_1 \longrightarrow^* W_1 \tag{5.27}$$

By Lemma 2 on $M \equiv M_1\ M_2$ given (5.27):

$$M_1\ M_2 \longrightarrow^* W_1\ M_2 \tag{5.28}$$

By Lemma 9 on (5.25) there exists W_2 such that:

$$M_2 \longrightarrow^* W_2 \tag{5.29}$$
$$\cdot \vdash v :: A \hookrightarrow W_2 \tag{5.30}$$

By Lemma 2 on (5.29):

$$\mathtt{c}\ M_2 \longrightarrow^* \mathtt{c}\ W_2 \tag{5.31}$$

By Lemma 6 on (5.26), (5.30) and (5.23):

$$W_1 \equiv \mathtt{c} \tag{5.32}$$
$$W_2 \not\equiv \mathtt{DEAD}_{\downarrow A}\langle\cdot\rangle \tag{5.33}$$

So (5.26) becomes:

$$\cdot \vdash \mathtt{c} :: A \to B \hookrightarrow \mathtt{c} \tag{5.34}$$

So we can apply TE-App-1:

$$\mathtt{c}\ W_2 \longrightarrow [\![\mathtt{c}]\!](W_2) \tag{5.35}$$

By (5.34), (5.31) and (5.35):

$$M_1\ M_2 \longrightarrow^+ [\![\mathtt{c}]\!](W_2)$$

By assumption 1 using (5.34), (5.33) and (5.30):

$$\cdot \vdash [\![\mathtt{c}]\!](v) :: A \hookrightarrow [\![\mathtt{c}]\!](W_2)$$

- Case T-$\vee$I:

$$\frac{\cdot \vdash e :: A_k \hookrightarrow M \qquad \vdash A_1 \vee A_2}{\cdot \vdash e :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_k\ M}$$

By inversion:

$$\cdot \vdash e :: A_k \hookrightarrow M \tag{6.1}$$
$$\vdash A_1 \vee A_2 \tag{6.2}$$

By i.h. using (6.1) with $e \longrightarrow e'$, there exists M' such that:

$$\cdot \vdash e' :: A_k \hookrightarrow M' \tag{6.3}$$
$$M \longrightarrow^+ M' \tag{6.4}$$

By Lemma 2 using (6.4):

$$\mathtt{inj}_k\ M \longrightarrow^+ \mathtt{inj}_k\ M'$$

Applying T-$\vee$I with premises (6.3) and (6.2):

$$\cdot \vdash e' :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_k\ M'$$

- Case T-$\vee$E:

$$\frac{\cdot \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{l} x_1 : A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \\ x_2 : A_2 \vdash E[x_2] :: B \hookrightarrow M_2 \end{array}}{\cdot \vdash E[e_0] :: B \hookrightarrow \texttt{case}\ M_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2}$$

By inversion:

$$\cdot \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \tag{7.1}$$
$$x_1 : A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \tag{7.2}$$
$$x_2 : A_2 \vdash E[x_2] :: B \hookrightarrow M_2 \tag{7.3}$$

Cases on the form of $e \longrightarrow e'$:

▷ Subcase:

$$\frac{e_0 \longrightarrow e_0'}{E[e_0] \longrightarrow E[e_0']}$$

By inversion:

$$e_0 \longrightarrow e_0' \tag{7.4}$$

By i.h. using (7.1) and (7.4):

$$\cdot \vdash e_0' :: A_1 \vee A_2 \hookrightarrow M_0' \tag{7.5}$$
$$M_0 \longrightarrow^+ M_0' \tag{7.6}$$

Using rule T-$\vee$E on (7.5), (7.2) and (7.3):

$$\cdot \vdash E[e_0'] :: B \hookrightarrow \texttt{case}\ M_0'\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2$$

Also, applying Lemma 2 on $\mathcal{E} \equiv \texttt{case}\ \langle\ \rangle\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2$ using (7.6):

$$\begin{aligned} &\texttt{case}\ M_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow^+ \\ &\qquad \texttt{case}\ M_0'\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \end{aligned}$$

▷ Subcase:

$$e_0 \equiv v_0 \tag{7.7}$$
$$E[v_0] \longrightarrow e' \tag{7.8}$$

Because v_0 is a value, we can split cases for its type. Without loss of generality we can assume that its type is A_1 (the same exact holds for A_2). This is depicted on the form of M_0 in equation (7.1), which now becomes (for some M_{01}):

$$\cdot \vdash v_0 :: A_1 \vee A_2 \hookrightarrow \texttt{inj}_1\ M_{01} \tag{7.9}$$

By Lemma 3 on (7.9):

$$\cdot \vdash v_0 :: A_1 \hookrightarrow M_{01} \tag{7.10}$$

By Lemma 9 on (7.10) there exists W_{01} such that:

$$M_{01} \longrightarrow^* W_{01} \tag{7.11}$$
$$\cdot \vdash v_0 :: A_1 \hookrightarrow W_{01} \tag{7.12}$$

By Lemma 9 on (7.9) there exists W_{01}[4] such that:

$$\mathtt{inj}_1\ M_{01} \longrightarrow^* \mathtt{inj}_1\ W_{01} \tag{7.13}$$
$$\cdot \vdash v_0 :: A_1 \vee A_2 \hookrightarrow \mathtt{inj}_1\ W_{01} \tag{7.14}$$

Cases for the form of $E[v_0]$:

– $E[v_0] \equiv \mathtt{let}\ x = v_0\ \mathtt{in}\ e_1$
  The original elaboration judgment becomes:

$$\frac{\cdot \vdash v_0 :: A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{c} x_1 : A_1 \vdash \mathtt{let}\ x = x_1\ \mathtt{in}\ e_1 :: B \hookrightarrow M_1 \\ x_2 : A_2 \vdash \mathtt{let}\ x = x_2\ \mathtt{in}\ e_1 :: B \hookrightarrow M_2 \end{array}}{\cdot \vdash \mathtt{let}\ x = v_0\ \mathtt{in}\ e_1 :: B \hookrightarrow \mathtt{case}\ M_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow M_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow M_2}$$

  By inversion:

$$\begin{array}{c} \cdot \vdash v_0 :: A_1 \vee A_2 \hookrightarrow M_0 \\ x_1 : A_1 \vdash \mathtt{let}\ x = x_1\ \mathtt{in}\ e_1 :: B \hookrightarrow M_1 \\ x_2 : A_2 \vdash \mathtt{let}\ x = x_2\ \mathtt{in}\ e_1 :: B \hookrightarrow M_2 \end{array} \tag{7.15}$$

  The derivation for (7.15) is of the form:

$$\frac{x_1 : A_1 \vdash x_1 :: A_1 \hookrightarrow x_1 \qquad x_1 : A_1, x : A_1 \vdash e_1 :: B \hookrightarrow N_1}{x_1 : A_1 \vdash \mathtt{let}\ x = x_1\ \mathtt{in}\ e_1 :: B \hookrightarrow \underbrace{\mathtt{let}\ x = x_1\ \mathtt{in}\ N_1}_{M_1}}$$

  By inversion:

$$x_1 : A_1, x : A_1 \vdash e_1 :: B \hookrightarrow N_1 \tag{7.16}$$

  Variable x_1 does not appear in e_1, so the above is equivalent to:

$$x : A_1 \vdash e_1 :: B \hookrightarrow N_1 \tag{7.17}$$

  Applying Lemma 10 on (7.12) and (7.17) we get:

$$\cdot \vdash [v_0/x]\ e_1 :: B \hookrightarrow [W_{01}/x]\ N_1 \tag{7.18}$$

[4] This is the same that we got right before, due to uniqueness of normal forms.

By E-LET:

$$\texttt{let } x = v_0 \texttt{ in } e_1 \longrightarrow [v_0/x]\ e_1 \tag{7.19}$$

Also, by Lemma 2 using (7.13):

$$\begin{aligned}&\texttt{case inj}_1\ M_{01} \texttt{ of inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow^* \\ &\qquad \texttt{case inj}_1\ W_{01} \texttt{ of inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2\end{aligned}$$

By TE-CASE:

$$\texttt{case inj}_1\ W_{01} \texttt{ of inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow [W_{01}/x_1]\ M_1$$

From the equality: $M_1 \equiv \texttt{let } x = x_1 \texttt{ in } N_1$, and because x_1 does not appear in M_1:

$$[W_{01}/x_1]\ (\texttt{let } x = x_1 \texttt{ in } N_1) \equiv \texttt{let } x = W_{01} \texttt{ in } N_1 \longrightarrow [W_{01}/x]\ N_1$$

From the last relation along with (7.19) and (7.18) stems the wanted result.

– $E[v_0] \equiv v_0\ ?\ e_1\ :\ e_2$: *Similar to case* $e_c\ ?\ e_1\ :\ e_2$

– $E[v_0] \equiv v_0\ e$: *Similar to earlier cases.*

– $E[v_0] \equiv (\lambda x.e_0)\ v_0$: *Similar to earlier cases.*

• Case T-⊥:

$$\frac{\cdot \vdash e :: A \hookrightarrow M \qquad \mathsf{TAG}(A) \cap \mathsf{TAG}(B) = \emptyset}{\cdot \vdash e :: B \hookrightarrow \mathsf{DEAD}_{A\downarrow B}\langle M \rangle}$$

By inversion:

$$\cdot \vdash e :: A \hookrightarrow M \tag{8.1}$$
$$\mathsf{TAG}(A) \cap \mathsf{TAG}(B) = \emptyset \tag{8.2}$$

There also exists e' such that:

$$e \longrightarrow e' \tag{8.3}$$

By i.h. on (8.1) and (8.3) there exists M', such that:

$$M \longrightarrow^+ M' \tag{8.4}$$
$$\cdot \vdash e' :: A \hookrightarrow M' \tag{8.5}$$

By Lemma 2 on (8.4):

$$\mathsf{DEAD}_{A\downarrow B}\langle M \rangle \longrightarrow^+ \mathsf{DEAD}_{A\downarrow B}\langle M' \rangle$$

Applying rule T-⊥ on (8.5) and (8.2):

$$\cdot \vdash e' :: B \hookrightarrow \mathsf{DEAD}_{A\downarrow B}\langle M' \rangle$$

□

Lemma 11 (Corresp.).

$$\begin{aligned} \forall \Gamma, e, A, M, G, T \quad . \quad & \Gamma \vdash e :: A \hookrightarrow M && (0.1)\\ \wedge \quad & G \vdash M :: T && (0.2)\\ \wedge \quad & [\Gamma] = \|G\| && (0.3)\\ \Rightarrow \quad & [A] = \|T\| \end{aligned}$$

Proof. We prove this by induction on pairs T-*Rule*/R-*Rule* of derivations:

$$\begin{aligned} \Gamma \vdash e :: A \hookrightarrow M \\ G \vdash M :: T \end{aligned}$$

- Case T-CST/R-CST:

$$\begin{aligned} \Gamma \vdash \mathsf{c} :: \mathtt{ty\_c} \hookrightarrow \mathsf{c} \\ G \vdash \mathsf{c} :: \{\nu : \mathtt{ty\_c} \mid \nu = \mathsf{Const}(\mathsf{c})\} \end{aligned}$$

  Meta-function sngl operates entirely on the refinement so it holds that:

$$\|\{\nu : \mathtt{ty\_c} \mid \nu = \mathsf{Const}(\mathsf{c})\}\| = \|\mathtt{ty\_c}\|$$

  Also, it holds that:

$$[\mathtt{ty\_c}] = \|\mathtt{ty\_c}\|$$

- Case T-VAR/R-VAR:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x :: A \hookrightarrow x} \qquad \frac{x : T \in G}{G \vdash x :: \mathsf{sngl}(T, x)}$$

  By inversion:

$$\begin{aligned} x : A \in \Gamma && (2.1)\\ x : T \in G && (2.2) \end{aligned}$$

  If x is bound multiple times in Γ and G, we assume the we have picked the correct instances from each environment.
  By (0.3) we have that:

$$[\Gamma(x)] = \|G(x)\|$$

  Also meta-function sngl operates entirely on the refinement so it holds that:

$$\|T\| = \|\mathsf{sngl}(T, x)\| \qquad (2.3)$$

  By (2.1), (2.2) and (2.3) it holds that:

$$[A] = \|\mathsf{sngl}(T, x)\|$$

- Case T-Let/R-Let:
  From the first premise of the implication:

$$\frac{\Gamma \vdash e_1 :: A_1 \hookrightarrow M_1 \qquad \Gamma, x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: A_2 \hookrightarrow \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2}$$

  By inversion:

$$\Gamma \vdash e_1 :: A_1 \hookrightarrow M_1 \tag{3.1}$$
$$\Gamma, x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2 \tag{3.2}$$

  From the second premise of the implication:

$$\frac{G \vdash M_1 :: T_1 \qquad G, x{:}T_1 \vdash M_2 :: T_2}{G \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 :: T_2}$$

  By inversion:

$$G \vdash M_1 :: T_1 \tag{3.3}$$
$$G, x{:}T_1 \vdash M_2 :: T_2 \tag{3.4}$$

  By i.h. on (0.3), (3.1) and (3.3):

$$[A_1] = \|T_1\| \tag{3.5}$$

  By (0.3) and (3.5):

$$[\Gamma, x{:}A_1] = \|G, x : T_1\| \tag{3.6}$$

  By i.h. on (3.2), (3.4) and (3.6):

$$[A_2] = \|T_2\|$$

- Case T-If/R-If: *Similar to previous case.*

- Case T-$\wedge$I/R-Pair:
  From the first premise of the implication:

$$\frac{\forall k \in \{1,2\} \,.\, \Gamma \vdash v :: A_k \hookrightarrow W_k}{\Gamma \vdash v :: A_1 \wedge A_2 \hookrightarrow (W_1, W_2)}$$

  By inversion:

$$\forall k \in \{1,2\} \,.\, \Gamma \vdash v :: A_k \hookrightarrow W_k \tag{5.1}$$

  From the second premise of the implication:

$$\frac{\forall k \in \{1,2\} \,.\, G \vdash W_k :: T_k}{G \vdash (W_1, W_2) :: T_1 \times T_2}$$

By inversion:

$$\forall k \in \{1,2\} \,.\, G \vdash W_k :: T_k \tag{5.2}$$

By i.h. on (0.3), (5.1) and (5.2):

$$\forall k \in \{1,2\} \,.\, [A_k] = \|T_k\| \tag{5.3}$$

Using properties of $[\cdot]$ and $\|\cdot\|$:

$$[A_1 \wedge A_2] = [A_1] \times [A_2] = \|T_1\| \times \|T_2\| = \|T_1 \times T_2\|$$

- Case T-$\wedge$E/R-Proj: *Straightforward based on earlier cases.*
- Case T-Lam/R-Lam: *Straightforward based on earlier cases.*
- Case T-App/R-App: *Straightforward based on earlier cases.*
- Case T-$\vee$I/R-Inj: *Straightforward based on earlier cases.*
- Case T-$\vee$E/R-Case:
  From the first premise of the implication:

$$\frac{\Gamma \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{c}\Gamma, x_1 {:} A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \\ \Gamma, x_2 {:} A_2 \vdash E[x_2] :: B \hookrightarrow M_2\end{array}}{\Gamma \vdash E[e_0] :: B \hookrightarrow \texttt{case}\ M_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2}$$

By inversion:

$$\Gamma \vdash e_0 :: A_1 \vee A_2 \hookrightarrow M_0 \tag{10.1}$$
$$\Gamma, x_1 {:} A_1 \vdash E[x_1] :: B \hookrightarrow M_1 \tag{10.2}$$
$$\Gamma, x_2 {:} A_2 \vdash E[x_2] :: B \hookrightarrow M_2 \tag{10.3}$$

From the second premise of the implication:

$$\frac{G \vdash M_0 :: T_1 + T_2 \qquad G, x_1 : T_1 \vdash M_1 :: T \qquad G, x_2 : T_2 \vdash M_2 :: T}{G \vdash \texttt{case}\ M_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 :: T}$$

By inversion:

$$G \vdash M_0 :: T_1 + T_2 \tag{10.4}$$
$$G, x_1 : T_1 \vdash M_1 :: T \tag{10.5}$$
$$G, x_2 : T_2 \vdash M_2 :: T \tag{10.6}$$

By i.h. on (0.3), (10.1) and (10.4):

$$[A_1 \vee A_2] = \|T_1 + T_2\|$$

From properties of type elaboration and refinement types:

$$[A_1 \vee A_2] = [A_1]{+}[A_2]$$
$$\|T_1{+}T_2\| = \|T_1\|{+}\|T_2\|$$

The right-hand side of the last two equations are tagged unions, so it is possible to match the consituent parts by structure:

$$[A_1] = \|T_1\| \quad \text{and} \quad [A_2] = \|T_2\|$$

Combining the last equation with (0.3):

$$[\Gamma, x{:}A_1] = \|G, x : T_1\| \tag{10.7}$$
$$[\Gamma, x{:}A_2] = \|G, x : T_2\| \tag{10.8}$$

By i.h. on (10.2), (10.5) and (10.7) (or (10.3), (10.6) and (10.8)):

$$[B] = \|T\|$$

- Case T-$\perp$/R-App:
  From the first premise of the implication:

$$\frac{\Gamma \vdash e :: A \hookrightarrow M \qquad \mathtt{TAG}(A) \cap \mathtt{TAG}(B) = \emptyset}{\Gamma \vdash e :: B \hookrightarrow \mathtt{DEAD}_{A\downarrow B}\langle M\rangle}$$

  From the second premise of the implication:

$$\frac{G \vdash \mathtt{DEAD}_{A\downarrow B} :: \mathsf{Bot}([A]) \to \mathsf{Bot}([B]) \qquad G \vdash M :: S}{G \vdash \mathtt{DEAD}_{A\downarrow B}\langle M\rangle :: [M/x]\ \mathsf{Bot}([B])}$$

  The result type of the last derivation can also be written as:

$$[M/x]\ \mathsf{Bot}([B]) = \mathsf{Bot}([B])$$

  Because after the application of $\mathsf{Bot}(\cdot)$ all original refinement get erased. Also, after removing the refinements:

$$\|\mathsf{Bot}([B])\| = [B]$$

$\square$

Theorem 3 (Two-Phase Safety). $\forall\ e, A, M, T$ *s.t.:*

(1) $\cdot \vdash e :: A \hookrightarrow M$*,*

(2) $\cdot \vdash M :: T$*,*

Then, either e *is a value, or there exists* e' *such that:* $e \longrightarrow e'$ *and* $\cdot \vdash e' :: A \hookrightarrow M'$ *for* M'*, such that* $M \longrightarrow^{+} M'$ *and* $\cdot \vdash M' :: T$.

Proof. By induction on pairs T-*Rule*/R-*Rule* of derivations:

$$\begin{gathered}\Gamma \vdash e :: A \hookrightarrow M \\ G \vdash M :: T\end{gathered}$$

- Cases T-Cst/R-Cst, T-Var/R-Var, T-$\wedge$I/R-Pair, T-Lam/R-Lam:
  The term e is a value.

- Case T-Let/R-Let:
  From (1) we have:

  $$\frac{\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \qquad x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2}{\cdot \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: A_2 \hookrightarrow \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2}$$

  By inversion:

  $$\cdot \vdash e_1 :: A_1 \hookrightarrow M_1 \tag{2.1}$$
  $$x{:}A_1 \vdash e_2 :: A_2 \hookrightarrow M_2 \tag{2.2}$$

  From (2) we have:

  $$\frac{\cdot \vdash M_1 :: T_1 \qquad x{:}T_1 \vdash M_2 :: T_2}{\cdot \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 :: T_2}$$

  By inversion:

  $$\cdot \vdash M_1 :: T_1 \tag{2.3}$$
  $$x{:}T_1 \vdash M_2 :: T_2 \tag{2.4}$$

  By i.h. using (2.1) and (2.3) we have two cases on the form of e_1:

  ▷ Expression e_1 is a value:

  $$e_1 \equiv v_1$$

  By source language operational semantics:

  $$e \equiv \mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 \longrightarrow [v_1/x]\ e_2 \tag{2.5}$$

  ▷ There exists e_1' such that:

  $$e_1 \longrightarrow e_1' \tag{2.6}$$

  Hence, by E-ECtx:

  $$e \equiv \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \longrightarrow \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2 \tag{2.7}$$

  In either case, there exists e' such that:

  $$e \longrightarrow e' \tag{2.8}$$

By Theorem 2 on (1) and (2.8), there exists M' such that:

$$\begin{gathered} M \longrightarrow^{+} M' \\ \cdot \vdash e :: A_2 \hookrightarrow M' \end{gathered}$$

And by Corollary 1:

$$\cdot \vdash M' :: T$$

- Case T-IF/R-IF:
  From (1) we have:

$$\frac{\cdot \vdash e_c :: \mathtt{boolean} \hookrightarrow M \qquad \forall i \in \{1,2\} \,.\, \cdot \vdash e_i :: A \hookrightarrow M_i}{\cdot \vdash e_c \;?\; e_1 \;:\; e_2 :: A \hookrightarrow M_c \;?\; M_1 \;:\; M_2}$$

  By inversion:

$$\begin{aligned} \cdot \vdash e_c :: \mathtt{boolean} \hookrightarrow M_c && (3.1) \\ \cdot \vdash e_1 :: A \hookrightarrow M_1 && (3.2) \\ \cdot \vdash e_2 :: A \hookrightarrow M_2 && \end{aligned}$$

  From (2):

$$\frac{\cdot \vdash M_c :: \mathtt{boolean} \qquad M_c \vdash M_1 :: T \qquad \neg M_c \vdash M_2 :: T}{\cdot \vdash M_c \;?\; M_1 \;:\; M_2 :: T}$$

  By inversion:

$$\begin{aligned} \cdot \vdash M_c :: \mathtt{boolean} && (3.3) \\ M_c \vdash M_1 :: T && (3.4) \\ \neg M_c \vdash M_2 :: T && (3.5) \end{aligned}$$

  By i.h. using (3.1) and (3.3) we have two case on the form of e_c:

  ▷ Expression e_c is a value:

$$e_c \equiv v_c$$

  By a standard canonical forms lemma v_c is either `true` or `false`. Assume the first case (the latter case is identical but involving the "else" branch of the conditional): By E-COND-TRUE:

$$\mathtt{true} \;?\; e_1 \;:\; e_2 \longrightarrow e_1$$

  ▷ There exists e_c' such that:

$$e_c \longrightarrow e_c' \qquad (3.6)$$

  Hence, by E-ECTX:

$$e_c \;?\; e_1 \;:\; e_2 \longrightarrow e_c' \;?\; e_1 \;:\; e_2$$

In either case, there exists e' such that:

$$e \longrightarrow e' \tag{3.7}$$

By Theorem 2 on (1) and (3.7), there exists M' such that:

$$M \longrightarrow^{+} M'$$
$$\cdot \vdash e :: A \hookrightarrow M'$$

And by Corollary 1:

$$\cdot \vdash M' :: T$$

- Case T-$\wedge$E/R-Proj:
  Without loss of generality we're going to assume first projection (the same holds for the second projection).
  From (1):

  $$\frac{\cdot \vdash e :: A_1 \wedge A_2 \hookrightarrow M_0}{\cdot \vdash e :: A_1 \hookrightarrow \mathtt{proj}_1 M_0}$$

  By inversion:

  $$\cdot \vdash e :: A_1 \wedge A_2 \hookrightarrow M_0 \tag{4.1}$$

  From (2):

  $$\frac{\cdot \vdash M_0 :: T_1 \times T_2}{\cdot \vdash \mathtt{proj}_1 M_0 :: T_1}$$

  By inversion:

  $$\cdot \vdash M_0 :: T_1 \times T_2 \tag{4.2}$$

  By i.h. using (4.1) and (4.2) we have two case on the form of e:

  ▷ Expression e is a value:

  $$e \equiv v$$

  So the source term does not step.

  ▷ There exists e' such that:

  $$e \longrightarrow e' \tag{4.3}$$

  By Theorem 2 on (1) and (4.3), there exists M' such that:

  $$M \longrightarrow^{+} M'$$
  $$\cdot \vdash e :: A \hookrightarrow M'$$

  And by Corollary 1:

  $$\cdot \vdash M' :: T$$

- Case T-App/R-App:
  From (1):

$$\frac{\cdot \vdash e_1 :: A \to B \hookrightarrow M_1 \qquad \cdot \vdash e_2 :: A \hookrightarrow M_2}{\cdot \vdash e_1\ e_2 :: B \hookrightarrow M_1\ M_2}$$

  By inversion:

$$\cdot \vdash e_1 :: A \to B \hookrightarrow M_1 \tag{5.1}$$
$$\cdot \vdash e_2 :: A \hookrightarrow M_2 \tag{5.2}$$

  From (2):

$$\frac{\cdot \vdash M_1 :: T_x \to T \qquad \cdot \vdash M_2 :: T_x}{\cdot \vdash M_1\ M_2 :: [M_2/x]\ T}$$

  By inversion:

$$\cdot \vdash M_1 :: T_x \to T \tag{5.3}$$
$$\cdot \vdash M_2 :: T_x \tag{5.4}$$

  By i.h. using (5.1) and (5.3) we have three cases on the form of e_1:

  ▷ Expression e_1 is a *primitive* value:

$$e_1 \equiv \mathsf{c}$$

  Elaboration (5.1) becomes:

$$\cdot \vdash \mathsf{c} :: A \to B \hookrightarrow M_1 \tag{5.5}$$

  By applying lemma 9 on (5.5) there exists W_1, such that:

$$M_1 \longrightarrow^* W_1 \tag{5.6}$$
$$\cdot \vdash \mathsf{c} :: A \to B \hookrightarrow W_1 \tag{5.7}$$

  By Corollary 1 using (5.3) and (5.22):

$$\cdot \vdash W_1 :: T_x \to T \tag{5.8}$$

  By Assumption 3 on (5.7):

$$W_1 \equiv \mathsf{c}$$

  or

$$W_1 \equiv \mathsf{DEAD}_{\downarrow A} \langle W_1' \rangle$$

  The latter case combined with (5.8) contradicts Corollary 2, so we end up with:

$$W_1 \equiv \mathsf{c} \tag{5.9}$$

  By lemma 2 using (5.6):

$$M_1\ M_2 \longrightarrow^* \mathsf{c}\ M_2 \tag{5.10}$$

  By i.h. using (5.2) and (5.4) we have two cases on the form of e_2:

- Expression e_2 is a value:

  $$e_2 \equiv v_2$$

  Elaboration (5.2) becomes:

  $$\cdot \vdash v_2 :: A \hookrightarrow M_2 \tag{5.11}$$

  By lemma 9 on (5.11) there exists W_2, such that:

  $$M_2 \longrightarrow^* W_2 \tag{5.12}$$
  $$\cdot \vdash v_2 :: A \hookrightarrow W_2 \tag{5.13}$$

  By lemma 2 using (5.12):

  $$\mathsf{c}\ M_2 \longrightarrow^* \mathsf{c}\ W_2 \tag{5.14}$$

  For the sake of contradiction assume:

  $$W_2 \equiv \mathsf{DEAD}_{B \downarrow A} \langle W_2' \rangle \tag{5.15}$$

  for some W_2'. By (5.4):

  $$\cdot \vdash W_2 :: T_x \tag{5.16}$$

  So, by (5.16) and Corollary 2 we have a contradiction. So:

  $$W_2 \not\equiv \mathsf{DEAD}_{B \downarrow A} \langle W_2' \rangle \tag{5.17}$$

  By Assumption 1 on (5.7), (5.9) (5.13) and (5.17):

  $$\mathsf{c}\ v_2 \longrightarrow [\![\mathsf{c}]\!](v_2) \tag{5.18}$$

- There exists e_2' such that:

  $$e_2 \longrightarrow e_2' \tag{5.19}$$

  By E-ECTX:

  $$\mathsf{c}\ e_2 \longrightarrow \mathsf{c}\ e_2'$$

▷ Expression e_1 is an abstraction:

$$e_1 \equiv \lambda x.e_0 \tag{5.20}$$

Elaboration (5.1) becomes:

$$\cdot \vdash \lambda x.e_0 :: A \to B \hookrightarrow M_1 \tag{5.21}$$

By applying lemma 9 on (5.5) there exists W_1, such that:

$$M_1 \longrightarrow^* W_1 \tag{5.22}$$
$$\cdot \vdash \lambda x.e_0 :: A \to B \hookrightarrow W_1 \tag{5.23}$$

By Corollary 1 using (5.3) and (5.22):

$$\cdot \vdash W_1 :: T_x \to T \tag{5.24}$$

By Assumption 3 on (5.23):

$$W_1 \equiv \lambda x.M_0$$

or

$$W_1 \equiv \mathsf{DEAD}_{\cdot\downarrow A \to B} \langle W_1' \rangle$$

The latter case combined with (5.24) contradicts Corollary 2, so we end up with:

$$W_1 \equiv \lambda x.M_0 \tag{5.25}$$

So (5.1) becomes:

$$\cdot \vdash \lambda x.e_0 :: A \to B \hookrightarrow \lambda x.M_0 \tag{5.26}$$

By i.h. using (5.2) and (5.4) we have two cases on the form of e_2:

– Expression e_2 is a value:

$$e_2 \equiv v_2$$

Equation (5.2) becomes (for some M_2):

$$\cdot \vdash v_2 :: A \hookrightarrow M_2$$

By lemma 9, there exists W_2 such that:

$$\cdot \vdash v_2 :: A \hookrightarrow W_2 \tag{5.27}$$

For the sake of contradiction assume:

$$W_2 \equiv \mathsf{DEAD}_{B\downarrow A} \langle W_2' \rangle \tag{5.28}$$

for some W_2'. By (5.4):

$$\cdot \vdash W_2 :: T_x \tag{5.29}$$

So, by (5.29) and Corollary 2 we have a contradiction. So:

$$W_2 \not\equiv \mathsf{DEAD}_{B\downarrow A} \langle W_2' \rangle \tag{5.30}$$

By Assumption 2 on (5.26), (5.27) and (5.30):

$$(\lambda x.e_0)\ v_2 \longrightarrow [v_2/x]\ e_0$$

– There exists e_2' such that:

$$e_2 \longrightarrow e_2' \tag{5.31}$$

By E-ECtx:

$$(\lambda x.e_0)\ e_2 \longrightarrow (\lambda x.e_0)\ e_2'$$

▷ There exists e_1' such that:

$$e_1 \longrightarrow e_1' \tag{5.32}$$

By E-ECtx:

$$e_1\ e_2 \longrightarrow e_1'\ e_2$$

In all cases, there exists e' such that:

$$e \longrightarrow e' \tag{5.33}$$

By Theorem 2 on (1) and (5.33), there exists M' such that:

$$M \longrightarrow^{+} M'$$
$$\cdot \vdash e :: B \hookrightarrow M'$$

And by Corollary 1:

$$\cdot \vdash M' :: T$$

- Case T-$\vee$I/R-Inj: *Following similar methodology as before.*
- Case T-$\vee$E/R-Case: *Following similar methodology as before.*
- Case T-$\perp$/R-App:
  Corollary 2 contradicts the second premise (2), so the theorem does not apply here.

□