Verifying Cake-Cutting, Faster

Noah Bertram, Tean Lai, Justin Hsu

Cornell University

Abstract. Envy-free cake-cutting protocols procedurally divide an infinitely divisible good among a set of agents so that no agent prefers another's allocation to their own. These protocols are highly complex and difficult to prove correct. Recently, Bertram, Levinson, and Hsu introduced a language called Slice for describing and verifying cake-cutting protocols. Slice programs can be translated to formulas encoding envy-freeness, which are solved by SMT. While Slice works well on smaller protocols, it has difficulty scaling to more complex cake-cutting protocols.

We improve Slice in two ways. First, we show any protocol execution in Slice can be replicated using piecewise uniform valuations. We then reduce Slice's constraint formulas to formulas within the theory of linear real arithmetic, showing that verifying envy-freeness is efficiently decidable. Second, we design and implement a linear type system which enforces that no two agents receive the same part of the good. We implement our methods and verify a range of challenging examples, including the first nontrivial four-agent protocol.

Keywords: Fair division · Automated verification · Type system

1 Introduction

How would you divide a piece of cake between two children? Classic wisdom would say to have one child cut the piece into two, and have the other take their preferred slice. Procedures that divide an infinitely divisible good amongst a set of agents are called *cake-cutting protocols*. If the protocol ensures no agent prefers what another received, it is called *envy-free*. While classic wisdom gives an envy-free protocol for two agents, a three-agent envy-free protocol was not discovered until 1960, and a four-agent envy-free protocol that does not dispose any cake was only proposed in 2015 by Aziz and Mackenzie [2]. Modern cake-cutting protocols are highly complex, and proving envy-freeness requires checking an enourmous number of cases.

Verifying envy-freeness. To make cake-cutting protocols easier to verify, Bertram et al. [3] introduced a language called Slice that can describe cake-cutting protocols, and encode envy-freeness as a logical formula that can be dispatched to an SMT

solver. While Slice can verify envy-freeness fully automatically, it has some drawbacks. First, it is not able to verify that agents receive non-overlapping pieces. This basic property, known as *disjointness*, is crucial for correctness.

Another drawback of Slice is the SMT instances encoding envy-freeness for complicated protocols are difficult to solve, and only scale to some three-agent protocols—non-trivial algorithms, but relatively simple compared to modern cake-cutting protocols. One reason the instances are difficult is they are *higher order*: they quantify over *valuations*, which are functions that describe the agents' preferences.

Our work: verifying disjointness and envy-freeness, faster. We address these weaknesses in Slice. To verify disjointness, we develop an affine type system for Slice which restricts usage of the cake and then prove that well-typed programs are disjoint. Typechecking is straightforward and syntax-directed, requiring no use of SMT.

To verify envy-freeness more efficiently, we reduce Slice constraints into linear real arithmetic formulas, removing the need to quantify over valuations. This reduction leverages a key observation: the behavior of a protocol on any valuation can be replicated by a piecewise uniform valuation, which enables envy-freeness to be encoded as a first-order formula in linear real arithmetic. As a side benefit, our work shows that verifying envy-freeness of Slice protocols is decidable.

Finally, we implement both our affine type system and formula reduction procedure on top of the Slice implementation and transcribe two significantly more complicated protocols into Slice, including the first nontrivial four agent cake-cutting protocol [15]. For all Slice protocols, our type system establishes disjointness and our constraints encoding envy-freeness can be verified in substantially less time than in the previous version of Slice.

Outline. After describing the cake-cutting model (Section 2), we present the Slice language and our new linear type system for verifying disjointness (Section 3). We then review Slice's constraints (Section 4) and describe our new constraint translation (Section 5). We discuss our implementation and evaluation (Section 6), and then conclude with related work and future directions (Section 7).

2 Cake-Cutting Preliminaries

In this section, we introduce the basics of cake-cutting protocols; the reader can consult a standard text for more background [13].

We begin by fixing a finite set of agents $\mathbb{A}$. The cake or good is modeled by the unit interval $[0, 1]$. A *piece* P is a finite union of intervals from the cake: $P = [r_1, r'_1] \cup \cdots \cup [r_n, r'_n]$ where $r_1 \leq r'_1 < r_2 \leq r'_2 < \cdots < r_n \leq r'_n$; the points r_i and r'_i are *boundary points* of P, and we write ∂P for the set of all boundary points. Two

pieces P_1 and P_2 are *disjoint* if $(P_1 \setminus \partial P_1) \cap (P_2 \setminus \partial P_2) = \emptyset$, that is, P_1 and P_2 only share possibly their boundary points.

Cake-cutting protocols produce an *allocation* of pieces to agents, i.e., an $\mathbb{A}$-tuple of pieces $(P_a \mid P_a \in \mathbb{P}, a \in \mathbb{A})$. Protocols produce allocations based on agent preferences, which are typically modelled by functions $V : \mathbb{P} \to [0, 1]$ called *valuations*. We assume that valuations satisfy five standard assumptions: (1) Additivity: $V(P \cup P') = V(P) + V(P')$ provided P and P' are disjoint; (2) Non-negativity: $V(P) \geq 0$; (3) Continuity: $V([r, r'])$ is continuous in both r and r'; (4) Monotonicity: $V(P) \geq V(P')$ if $P' \subseteq P$; and (5) Normalization: $V([0, 1]) = 1$. We will often write $V[r, r']$ for $V([r, r'])$. A *valuation set* $\overline{V}$ is an $\mathbb{A}$-tuple of valuations $(V_a \mid a \in \mathbb{A})$. We write $\overline{V}_a$ for agent a's valuation.

Cake-cutting protocols aim to produce fair allocations where no agent prefers another agent's piece. More precisely, if A is an allocation and $\overline{V}$ is a valuation set, we say A is *envy-free* (with respect to $\overline{V}$) if $\overline{V}_a(A_a) \geq \overline{V}_a(A_{a'})$ for all $a, a' \in \mathbb{A}$.

Protocols are assumed to have indirect access to agent valuations through specific kinds of *agent queries*. Slice implements the Robertson-Webb (RW) query model [14], which is the typical query model in the cake-cutting literature and captures most protocols. In the RW model, there are two kinds of queries. An *eval query* takes as input an agent and a piece and reports the agent's value of that piece:

$$\mathsf{eval}_a(P) \text{ reports } \overline{V}_a(P).$$

A *mark query*, when supplied an interval and a value, reports how much of the interval is needed to attain that value:

$$\mathsf{mark}_a([\ell, r], v) \text{ reports } r' \text{ where } \overline{V}_a[\ell, r'] = v, \text{ provided that } v \leq \overline{V}_a[\ell, r].$$

This query enables us to find intervals within the cake which have a specified value for a certain agent. For example, $\mathsf{mark}_a([0, 1], 1/2)$ will output a point r' such that $\overline{V}_a[0, r'] = 1/2 = \overline{V}_a[r', 1]$. The assumption $v \leq \overline{V}_a[\ell, r]$ is required since $\overline{V}_a$ is monotone: if $v > \overline{V}_a[\ell, r]$, no such point exists. Note that if multiple points r' are a valid answer to a mark query, then mark can report any of them.

3 Language and Type System

We review the language [3] before describing our novel affine type system. Full details for this section can be found in the appendix.

3.1 Syntax of Base Slice

The set of all basic Slice expressions $\mathcal{E}$ is given by the grammar shown in Figure 1. The expression v is a value and $\mathcal{X}$ is an infinite set of variables. We can form tuples

and, through the split expression, extract their components. We have standard if-then-else expression, and a set $\mathcal{O}$ consisting of primitive operations like $+$, $\geq$, etc.

The remaining expressions are cake-cutting specific. The expression cake represents the whole cake, divide takes an interval and a point, splitting the interval into two at the point, and piece takes in a list of intervals and forms a piece out of them. The expression eval_a implements the eval query by taking in an interval or piece, and producing its value according to agent a. The expression mark_a implements the mark query by taking in an interval and the target value, returning any point satisfying the query.

$$
\begin{aligned}
e ::= \;& v \mid x \in \mathcal{X} \mid (e_1, \ldots, e_n) \mid \mathsf{let}\ x_1, \ldots, x_n = \mathsf{split}\ e_1\ \mathsf{in}\ e_2 \\
& \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid o(e_1, \ldots, e_n) \qquad (o \in \mathcal{O}) \\
& \mid \mathsf{cake} \mid \mathsf{divide}(e_1, e_2) \mid \mathsf{piece}(e_1, \ldots, e_n) \\
& \mid \mathsf{mark}_a(e_1, e_2) \mid \mathsf{eval}_a(e) \qquad (a \in \mathbb{A}) \\
\\
v ::= \;& \mathsf{true} \mid \mathsf{false} \mid r\#\mathsf{Pt} \mid \left[r, r'\right] \quad (r \leq r') \\
& \mid (v_1, \ldots, v_n) \mid P\left[r_1, r_1'\right], \ldots, \left[r_n, r_n'\right] \quad (r_i \leq r_i') \\
& \mid r_1 \cdot V_{a_1}(P_1) + \cdots + r_n \cdot V_{a_n}(P_n)
\end{aligned}
$$

Fig. 1: The grammar for Slice expressions (top) and values (bottom).

The set of all values is denoted by $\mathcal{V}$. We have boolean constants, *points* $r\#\mathsf{Pt}$, and *intervals* $[r, r']$. Points represent positions within the cake. Intervals describe contiguous pieces of the cake. Tuple values enable us to describe allocations.

Values of the form $P[r_1, r_1'], \ldots, [r_n, r_n']$ and $r_1 \cdot V_{a_1}(P_1) + \cdots + r_n \cdot V_{a_n}(P_n)$ are referred to as *pieces* and *valuations*, respectively. Note that for piece values, we do not assume that $[r_i, r_i']$ is disjoint from $[r_j, r_j']$ if $i \neq j$. We sometimes write piece values as $P_{i=1}^n[r_i, r_i']$, or $P_i[r_i, r_i']$, where i ranges over a finite set. Within the valuation value, $P_1, \ldots, P_n$ are interval or piece values, $a_1, \ldots, a_n$ are agents, and $r_1, \ldots, r_n$ are real numbers. We sometimes write $\sum_{i=1}^n r_i \cdot V_{a_i}(P_i)$, or $\sum_i r_i \cdot V_{a_i}(P_i)$ for short.

Figure 2 shows the two agent protocol described in Section 1 implemented in Slice; for now, we can ignore the bars over variables. This protocol uses the eval and mark queries to divide the cake into two pieces equally preferred by agent 1, and then uses eval queries for agent 2's comparison.

3.2 A Linear Type System for Slice

In this section, we develop a new, affine type system for Slice. At a high level, our type system ensures that no two agents receive overlapping pieces in the allocation.

```
let p = split cake in
let p1, p2 = split divide(p, mark1(p̄, 1/2 · eval1(p̄))) in
if eval2(p̄1) ≥ eval2(p̄2) then
  (piece(p2), piece(p1))
else
  (piece(p1), piece(p2))
```

Fig. 2: Cut-choose in Slice.

In order to accomplish this, it suffices to ensure that any duplicated variable bound to an interval or piece cannot be used either make further cuts or form more pieces. After all, you can't have your cake and eat it too!

Types. Slice types include affine types τ and non-affine types $\hat{\tau}$:

$$\hat{\tau} ::= \mathsf{Bool} \mid \mathsf{Point} \mid \mathsf{Vltn} \mid \overline{\mathsf{Intvl}} \mid \overline{\mathsf{Piece}}$$
$$\tau ::= \mathsf{Intvl} \mid \mathsf{Piece} \mid \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_n$$

Any non-linear type can be viewed as a linear type (i.e., as a unary product).

We treat Intvl and Piece as affine types to prevent their values from being duplicated. However, restricting interval and piece types poses a problem: protocols often query an agent before using the same interval or piece for division or allocation. For example, in the second line in Figure 2, p needs to be used to mark itself appropriately before being divided. To address this issue, we include two new base non-affine types, $\overline{\mathsf{Intvl}}$ and $\overline{\mathsf{Piece}}$, called "read only" types. Since these types are non-affine, variables of these types can be freely used in queries. However, dividing or forming pieces from read-only types is not allowed. This restriction ensures we can only create disjoint pieces.

Values and expressions. We extend Slice with values of read-only type:

$$v ::= \cdots \mid \overline{[r, r']} \mid \overline{P\,[r_1, r'_1], \ldots, [r_n, r'_n]}$$

The overline syntax is also extended to notation on other values and types, e.g. $\overline{r} = r$, $\overline{(v_1, v_2)} = (\overline{v_1}, \overline{v_2})$, $\overline{\mathsf{Vltn}} = \mathsf{Vltn}$, and $\overline{\tau_1 \times \tau_2} = \overline{\tau_1} \times \overline{\tau_2}$. Next, we extend Slice expressions with two new classes of variables. Affine variables are drawn from $\mathcal{W}$, while read-only variables are drawn from $\overline{\mathcal{W}}$. Finally, we extend the syntax of the split expression to bind these variables:

$$\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2$$

This expression implicitly binds read-only variables $\overline{w_1}, \ldots, \overline{w_{n'}}$ corresponding to the affine variables $w_1, \ldots, w_n$. For example, in Figure 2, $\overline{p}$ is bound in the first line and both $\overline{p_1}$ and $\overline{p_2}$ are bound in the second line.

Affine typing rules. Our typing judgements are of the form $\Gamma; \Delta \vdash e : \tau$, for Γ a partial map from $\mathcal{X} \cup \overline{\mathcal{W}}$ to non-affine types, and Δ a partial map from $\mathcal{W}$ to linear types. We present a selection of rules in Figure 3. For affine type contexts Δ_1

$$\frac{}{\Gamma, x : \hat{\tau}; \Delta \vdash x : \hat{\tau}}\ \text{T-Var} \qquad \frac{}{\Gamma; w : \tau \vdash w : \tau}\ \text{T-AffVar}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathsf{Intvl} \quad \cdots \quad \Gamma; \Delta_n \vdash e_n : \mathsf{Intvl}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \mathsf{piece}(e_1, \ldots, e_n) : \mathsf{Piece}}\ \text{T-Piece}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathsf{Intvl} \quad \Gamma; \Delta_2 \vdash e_2 : \mathsf{Point}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{divide}(e_1, e_2) : \mathsf{Intvl} \times \mathsf{Intvl}}\ \text{T-Div}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \overline{\mathsf{Intvl}} \quad \Gamma; \Delta_2 \vdash e_2 : \mathsf{Vltn}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{mark}_a(e_1, e_2) : \mathsf{Point}}\ \text{T-Mark} \qquad \frac{\Gamma; \Delta \vdash e : \overline{\mathsf{Piece}}}{\Gamma; \Delta \vdash \mathsf{eval}_a(e) : \mathsf{Vltn}}\ \text{T-EvalPc}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'} \\ \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}}; \Delta_2, w_1 : \tau_1, \ldots, w_{n'} : \tau_{n'} \vdash e_2 : \tau\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2 : \tau}\ \text{T-Split}$$

Fig. 3: Select typing rules for Slice expressions.

through Δ_n, the concatenation $\Delta_1, \ldots, \Delta_n$ denotes the union of *disjoint* contexts: $\mathrm{dom}(\Delta_i) \cap \mathrm{dom}(\Delta_j) = \emptyset$ if $i \neq j$.

The variable rules [T-Var] and [T-AffVar] type the given variable based on its context. The rule [T-Piece] shows the role of affine variables. The premise has expressions e_i under linear type contexts Δ_i, while the conclusion has the combined affine type context $\Delta_1, \ldots, \Delta_n$. Since the Δ_i must have disjoint domain, the expressions e_i cannot share affine variables. In particular, it is not possible for the same interval variable to appear more than once in a piece.

The rules [T-Piece], [T-Div], [T-Mark], and [T-EvalPc] highlight the difference between affine types and their read-only variants. [T-Piece] requires its subexpressions have type Intvl as it is forming a piece. [T-Div] requires the first argument have type Intvl since it is forming new pieces. In contrast, [T-Mark] requires the first argument to have type $\overline{\mathsf{Intvl}}$ as it is querying a valuation, not forming a piece, and similarly for [T-EvalPc].

The most complicated rule is [T-Split], which binds multiple variables at once by pattern matching on tuples.

3.3 Semantics

We present a big-step style semantics, defined by a relation $\Downarrow_{\overline{V}} \subseteq \mathcal{E} \times \mathcal{V}$ indexed by a valuation set $\overline{V}$, so our judgments are of the form $e \Downarrow_{\overline{V}} v$. We omit $\overline{V}$ when clear from context. Our big-step rules are straightforward. We present a few rules in Figure 4 and discuss them here.

$$\frac{e_1 \Downarrow v_1 \qquad \cdots \qquad e_n \Downarrow v_n}{(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n)}\ \text{E-Tup}$$

$$\frac{e_1 \Downarrow \overline{[r_1, r_1']} \qquad e_2 \Downarrow \sum_i r_i V_{a_i}(P_i) \qquad V_a([r_1, r]) = \sum_i r_i \cdot V_{a_i}(P_i)}{\mathsf{mark}_a(e_1, e_2) \Downarrow r}\ \text{E-Mark}$$

$$\frac{e \Downarrow \overline{P\ [r_1, r_1'], \ldots, [r_n, r_n']}}{\mathsf{eval}_a(e) \Downarrow V_a(P[r_1, r_1'], \ldots, [r_n, r_n'])}\ \text{E-EvalPc}$$

$$\frac{e_1 \Downarrow [r_1, r_1'] \qquad e_2 \Downarrow r_2 \qquad r_1 \le r_2 \le r_1'}{\mathsf{divide}(e_1, e_2) \Downarrow ([r_1, r_2], [r_2, r_1'])}\ \text{E-Div}$$

$$\frac{\begin{array}{c} e_1 \Downarrow (v_1, \ldots, v_{n+n'}) \\ e_2\{x_i \mapsto v_i \mid 1 \le i \le n\}\{\overline{w_i} \mapsto \overline{v_{i+n}}, w_i \mapsto v_{i+n} \mid 1 \le i \le n'\} \Downarrow v \end{array}}{\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2 \Downarrow v}\ \text{E-Split}$$

Fig. 4: Select evaluation rules for Slice expressions.

The rule [E-Tup] forms a tuple out of a collection of values. [E-Mark] implements the mark query; since the equation $\overline{V}_a[r_1, r] = \sum_i r_i \overline{V}_{a_i}(P_i)$ can be satisfied by more than one r, the big-step semantics is non-deterministic. [E-EvalPc] implements the eval query. [E-Div] splits an interval into two, requiring the interval to contain the split point. Both this condition and the condition for [E-Mark] mean that some well-typed protocols may become stuck: they may not evaluate to values. Lastly, [E-Split] binds the variables $x_1, \ldots, x_n, w_1, \ldots, w_{n'}$ to the values $v_1, \ldots, v_n, \ldots, v_{n+n'}$, and binds $\overline{w_1}, \ldots, \overline{w_n}$ to read-only versions $\overline{v_{n+1}}, \ldots, \overline{v_{n+n'}}$ of the affine values. It is straightforward to show that evaluation preserves types:

Proposition 1 (Type soundness). *If* $\cdot \vdash e : \tau$ *and* $e \Downarrow v$, *then* $\cdot \vdash v : \tau$.

3.4 Disjointness

Our affine type system is designed to ensure that well-typed programs produce only disjoint allocations, i.e., tuples of pieces that do not overlap. To prove this claim, we generalize and define disjointness for values and general expressions, and then show that a well-typed disjoint program can only evaluate to a disjoint value.

Informally, an expression is disjoint if all interval values within it, excluding read-only versions, are disjoint from each other. Disjointness ignores read-only values since well-typed programs are allowed to duplicate them; this does not affect disjointness verification since we are only concerned with programs that return allocations, i.e., values of type $\mathsf{Piece}^{\mathbb{A}}$.

Since our type system prevents multiple uses of variables with type Intvl and Piece, they cannot be duplicated so programs cannot construct pieces and intervals with overlapping components. This invariant enables us to show that disjoint expressions only evaluate to disjoint values.

Proposition 2. *If $\cdot \vdash e : \tau$ and e is disjoint, then $e \Downarrow v$ implies v is disjoint.*

Checking that a well-typed protocol is disjoint is easily done syntactically, and in the protocols we are concerned with, amounts to ensuring cake is only used once.

Example 1. We illustrate our type system with the two-agent *Surplus* protocol. In brief, both agents are asked to mark the cake at half the value of the whole cake. The agent that marked furthest to the left is given all the cake to the left of their own mark. Symmetrically, the other agent is given everything to the right of their own mark, leaving the cake lying between the marks un-allocated. The Slice programs shown in Figure 5 both correctly implement the Surplus protocol, however, the left program is not well-typed, while the right one is. The left program does not type check because it divides the whole cake twice (highlighted in red), leaving either p_1 and p_4 or p_2 and p_3 to overlap. Disjointness cannot be verified in this instance since there are intermediate expressions that will not be in its evaluation. The right program avoids this issue by only dividing the cake once it is known where the marks lie in relation to each other. □

4 Constraints

Now that we've seen the Slice language, we review the original Slice constraint translation. For full details see the appendix.

Paths. As is standard, we consider each path through a program separately. Paths b are Slice expressions (Section 3) with an assert expression assert b_1 in b_2 in place of if-then-else.

```
let p = split cake in
let m_1 = mark_1(p̄, 1/2 · eval_1(p̄)) in
let m_2 = mark_2(p̄, 1/2 · eval_2(p̄)) in
let p_1, p_2 = split divide(p, m_1) in
let p_3, p_4 = split divide(p, m_2) in
if m_2 ≥ m_1 then
  (piece(p_1), piece(p_4))
else
  (piece(p_2), piece(p_3))
```

(a) Not well-typed.

```
let p = split cake in
let m_1 = mark_1(p̄, 1/2 · eval_1(p̄)) in
let m_2 = mark_2(p̄, 1/2 · eval_2(p̄)) in
if m_2 ≥ m_1 then
  let p_1, p_2 = split divide(p, m_1) in
  let p_2, p_3 = split divide(p_2, m_2)in
  (piece(p_1), piece(p_3))
else
  let p_1, p_2 = split divide(p, m_2) in
  let p_2, p_3 = split divide(p_2, m_1)in
  (piece(p_3), piece(p_1))
```

(b) Well-typed.

Fig. 5: The Surplus protocol written in two ways.

Logical syntax Protocol paths are translated into a multi-sorted first order logic. The logic is standard, so most details areomitted, though we make note of select function symbols:

- $[\_, \_]$, $\cup$ for forming intervals and pieces respectively
- $\mathtt{l}$, $\mathtt{r}$ for obtaining the left and right endpoints of an interval respectively
- $\mathtt{V}_a$ for each $a \in \mathbb{A}$ for representing agent valuations
- π_i for the ith component of a tuple
- $\mathtt{O}$ which contains logical counterparts to the primitive operations O (e.g. $+$, $\geq$)

Through the function symbols and constants, any program value v can be encoded as a logical term $\mathtt{v}$. Throughout, the typewriter font designates logical counterparts to program objects. We also include a special set of variables $\mathcal{Y}$, disjoint from $\mathcal{X}$, which will only be used to represent points in our formulas.

With our logic, we can express envy-freeness, where x represents allocation:

$$E(x) \triangleq \bigwedge_{a,a' \in \mathbb{A}} \mathtt{V}_a(\pi_a x) \geq \mathtt{V}_a(\pi_{a'} x). \tag{1}$$

Logical semantics. Formula semantics are given by an interpretation $\mathcal{A}$ and variable assignment μ. An interpretation associates sorts with sets and function symbols with functions on these sets. A variable assignment is a map from variables to elements of these sets. For our purposes, we fix a base interpretation that interprets everything but the symbols $\mathtt{V}_a$ for all a, and all full interpretations agree with the base. The base interprets objects as one would expect, e.g. $[\![\mathtt{l}]\!]([r, r']) = r$. For full interpretations, the symbols $\mathtt{V}_a$ are interpreted over all possible valuations. Thus, full interpretations

are uniquely determined by the choice of valuation set, and we write $\mathcal{A}_{\overline{V}}$ for the interpretation such that $[\![\mathtt{V}_a]\!]_{\mathcal{A}_{\overline{V}}} = \overline{V}_a$.

For a logical term t, we let $[\![t]\!]^{\mu}_{\mathcal{A}}$ denote the interpretation of t according to $\mathcal{A}$, with variable values determined by μ, defined in the usual way (e.g., $[\![\mathtt{V}_a([y, y'])]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = \overline{V}_a[\mu(y), \mu(y')]$). Likewise, for a formula φ, we write $\mathcal{A}, \mu \vDash \varphi$ if φ is true when interpreted through $\mathcal{A}$ with variable values determined by μ, also defined in the usual way. We write $\mathcal{A} \vDash \varphi$ if for all assignments μ we have $\mathcal{A}, \mu \vDash \varphi$. If t is a term containing no $\mathtt{V}_a$ symbols, then for a fixed assignment μ, the term t is always interpreted the same way and we write just $[\![t]\!]^{\mu}$.

If $\mathtt{v}$ is an allocation, $\mathcal{A}_{\overline{V}}, \mu \vDash E(\mathtt{v})$ states that $[\![\mathtt{v}]\!]^{\mu}_{\mathcal{A}_{\overline{V}}}$ is an envy-free allocation. If e is a expression, we say that e *satisfies* $E(x)$ and write $e \vDash E(x)$ if for all valuation sets $\overline{V}$, $e \Downarrow_{\overline{V}} v$ implies $\mathcal{A}_{\overline{V}} \vDash E(\mathtt{v})$. Thus $e \vDash E(x)$ means e is envy-free.

In order to verify envy-freeness, Slice translates programs e to logical formulas ensuring $e \vDash E(x)$. We review this constraint translation next, before describing our improved translation.

Constraints. To translate protocols to formulas, we translate each path in a protocol to a formula consisting of a logical term $\rho(b)$ and a formula $c(b)$. Intuitively, $\rho(b)$ is the logical term representation of the value that b evaluates to assuming that the formula $c(b)$ holds. We give some cases of the definition in Figure 6.

$$\rho(\overline{v}) \triangleq \mathtt{v} \qquad \rho(v) \triangleq \mathtt{v} \qquad \rho(x) \triangleq x \qquad \rho(w) \triangleq w \qquad \rho(\overline{w}) \triangleq \overline{w}$$

$$\rho(\mathsf{divide}(b_1, b_2)) \triangleq \left([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))]\right) \qquad \rho(\mathsf{mark}_a(b_1, b_2)) \triangleq y \in \mathcal{Y}$$

$$\rho(\mathsf{eval}_a(b)) \triangleq \mathtt{V}_a(\rho(b)) \qquad \rho(\mathsf{assert}\ b_1\ \mathsf{in}\ b_2) \triangleq \rho(b_2)$$

$$c(\mathsf{divide}(b_1, b_2)) \triangleq c(b_1) \wedge c(b_2) \wedge \mathtt{l}(\rho(b_1)) \leq \rho(b_2) \leq \mathtt{r}(\rho(b_1)) \qquad c(\mathsf{eval}_a(b)) \triangleq c(b)$$

$$c(\mathsf{mark}_a(b_1, b_2)) \triangleq c(b_1) \wedge c(b_2) \wedge (\mathtt{V}_a([\mathtt{l}(\rho(b_1)), \rho(\mathsf{mark}_a(b_1, b_2))]) = \rho(b_2))$$

$$c(\mathsf{assert}\ b_1\ \mathsf{in}\ b_2) \triangleq (\rho(b_1) = \mathsf{true}) \wedge c(b_1) \wedge c(b_2)$$

Fig. 6: $\rho(b)$ and $c(b)$ for select path expressions b.

It is informative to compare these definitions to the big-step semantics shown in Section 3. For instance, $\rho(\mathsf{divide}(b_1, b_2))$ is a logical encoding of the original interval being split into two, $\rho(\mathsf{mark}_a(b_1, b_2))$ is a variable that represents the mark, $\rho(\mathsf{eval}_a(b))$ is the value of the interval or piece provided.

The formula $c(b)$ is referred to as the *constraint of* b. Roughly, the formula corresponds to the side conditions shown in the big-step semantics. The most interesting cases of the definition are in Figure 6. All constraints conjoin the conditions from their subexpressions. The constraint for divide encodes that the point dividing the interval must be within. The constraint of eval has no additional conditions to satisfy, so it is just the constraint of its subexpression. The constraint of mark ensures that the new point has the required property and the constraint of assert asserts that the guard must hold.

Example 2. Consider the following path, denoted b, from Cut-Choose (Figure 2):

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{split}\ \mathsf{cake}\ \mathsf{in}\\ \mathsf{let}\ p_1, p_2 = \mathsf{split}\ \mathsf{divide}(p, \mathsf{mark}_1(\overline{p}, 1/2 \cdot \mathsf{eval}_1(\overline{p})))\ \mathsf{in}\\ \mathsf{assert}\ \mathsf{eval}_2(\overline{p_1}) \geq \mathsf{eval}_2(\overline{p_2})\ \mathsf{in}\ (\mathsf{piece}(p_2), \mathsf{piece}(p_1))\end{array}$$

The path b gives the following (simplified) constraint:

$$\begin{array}{l}c(b) = (\mathtt{V}_1([0,y]) = 1/2 \cdot \mathtt{V}_1([0,1])) \wedge (\mathtt{V}_2([0,y]) \geq \mathtt{V}_2([y,1]))\\ \rho(b) = (\cup[y,1], \cup[0,y])\end{array}$$

The first conjunct in $c(b)$ is from the expression $\mathsf{mark}_1(\overline{p}, 1/2 \cdot \mathsf{eval}_1(\overline{p}))$, while the second is from $\mathsf{eval}_2(\overline{p_1}) \geq \mathsf{eval}_2(\overline{p_2})$. The term $\rho(b)$ is a logical encoding of b's evaluation. □

The following result, akin to Corollary 4.8 for Slice [3], characterizes paths in terms of their constraints.

Theorem 1. *Suppose* $\cdot \vdash b : \tau$. *Then* $b \Downarrow_{\overline{V}} v$ *if and only if there is a variable assingment* μ *such that* $\mathcal{A}_{\overline{V}}, \mu \vDash c(b)$ *and* $[\![\rho(b)]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = |v|$.

With our constraint translation being sound and complete, we look to use constraints to verify envy-freeness only by checking the validity of certain formulas involving the constraint. For the following, let $\mathcal{Y}_b$ be the set of free variables contained in $c(b)$, and let $B(e)$ be the set of paths within e. The following theorem forms the basis for automated verification in Slice.

Theorem 2. *Suppose that* e *is a well-formed expression and* $\cdot \vdash e : \mathsf{Piece}^{\mathbb{A}}$. *Then*

$$\mathcal{A}_{\overline{V}} \vDash \bigwedge_{b \in B(e)} \forall \mathcal{Y}_b.(c(b) \Rightarrow E(\rho(b))) \tag{2}$$

for all $\overline{V}$ *if and only if* $e \vDash E(x)$.

The formal definition for a well-formed expression can be found within the appendix, though the imposed conditions are mild; any typical cake-cutting protocol

is well-formed. This theorem can be generalized from $E(x)$ to general formulas $F(x)$ satisfying mild conditions.

We stress that in order to apply this theorem to conclude $e \vDash E(x)$, Formula (2) needs to be valid *for all* valuation sets. Our logic is not rich enough to quantify over valuations and their axioms, so for verification, these formulas must be embedded in a richer theory (e.g., from a modern SMT solver).

5 Piecewise uniform reduction

Now that we have seen how the existing constraint translation works in Slice, we show how to produce a result similar to Theorem 2, but instead with a formula in the theory of linear real arithmetic. Formula (2) contains terms like $\mathtt{l}([t_1, t_2])$, $\pi_k(t_1, \ldots, t_n)$, and $\mathtt{V}_a(t)$, which all need to be reduced to linear sums of real variables. Most terms can be reduced via syntactic simplifications, but reducing valuation terms $\mathtt{V}_a(t)$ is much more challenging.

The broad approach is to show a protocol execution on any valuation set can be replicated with a *piecewise uniform valuation* set, then replace terms $\mathtt{V}_a(t)$ with sums of differences of real variables that represent $\mathtt{V}_a(t)$. We discuss conditions under which protocol executions can be replicated, then show there are always piecewise uniform valuations meeting these conditions. Then, we describe how to construct the formula reduction, prove that it preserves validity, and then apply it to obtain an analog to Theorem 2. Our approach is inspired by Theorem 1 from Kurokawa, Lai, and Procaccia [8].

For this section only, we will assume that the operations $\mathcal{O}$ consist only of boolean operators, comparisons, constant multiplication and addition. These operations are sufficient for describing cake-cutting protocols. A more detailed description of $\mathcal{O}$ is shown in the appendix.

5.1 Replicating protocol executions

In this subsection, we give a condition when the same evaluation judgement holds for two possibly different valuation sets. For this, we define the following relationship between valuation sets.

Definition 1. *Let $M \supseteq \{0, 1\}$ a finite set of points. We say that valuation sets $\overline{U}$ and $\overline{V}$* agree on *M if for any piece P with boundary points in M, $V_a(P) = U_a(P)$ for all $a \in \mathbb{A}$.*

The following theorem says that we can identically derive an evaluation judgement with a different valuation set, as long as the valuation set agrees with the original on all pieces formed from points in the derivation.

Theorem 3. *Let $\overline{U}$ and $\overline{V}$ be valuation sets, and suppose $e \Downarrow_{\overline{V}} v$. If $\overline{U}$ and $\overline{V}$ agree on all points considered in the derivation of $e \Downarrow_{\overline{V}} v$, then $e \Downarrow_{\overline{U}} v$.*

There is an analog for formulas.

Theorem 4. *If valuation sets $\overline{U}$ and $\overline{V}$ agree on the set of points considered in a formula φ under variable assignment μ, then $\mathcal{A}_{\overline{V}}, \mu \vDash \varphi \iff \mathcal{A}_{\overline{U}}, \mu \vDash \varphi$.*

5.2 Piecewise uniform valuations

Now that we have seen what is required for replication, we show that there is always a special piecewise uniform valuation set that meets the requirements.

We first formally define piecewise uniform valuations. It is easiest to define these valuations in terms of their *density*. For our purposes, a *density* is a function $w : [0,1] \to \mathbb{R}_{\geq 0}$, and a valuation W *has density* w if $W(P) = \int_P w \quad$ for all $P \in \mathbb{P}$.

Definition 2. *We say that a valuation U is* piecewise uniform *if U has density u for which there exists a piece $P \in \mathbb{P}$ and a constant c such that*

$$u(x) = \begin{cases} c & \text{if } x \in P \\ 0 & \text{if } x \notin P. \end{cases}$$

We let $P(U)$ denote P and $c(U)$ denote c.

Because valuations are normalized, the constant associated with a piecewise uniform valuation U is the reciprocal length of $P(U)$. Therefore, any piece P uniquely determines a piecewise uniform valuation U_P, where $P(U_P) = P$.

Much of the advantage of these valuations lies in how we can represent their values on specific pieces. For intervals built from right endpoints of $P(U)$, the valuation reduces to a simple sum of differences between real numbers. If we write out $P(U) = [l_1, r_1] \cup \cdots \cup [l_n, r_n]$ where $l_1 \leq r_1 < \cdots < l_n \leq r_n$, then

$$U[r_i, r_{i'}] = c(U) \cdot \sum_{i' \geq j > i} (r_j - l_j). \tag{3}$$

This formula is key for our reduction, as it enables us to convert valuations applied to intervals (left) to sums of differences of real numbers (right).

We call a valuation set a *piecewise uniform valuation* set if all valuations within it are piecewise uniform. Our formula reduction will benefit from the following key conditions on piecewise uniform valuation sets.

Definition 3. *Let $\overline{U}$ be a piecewise uniform valuation set and let $M \supseteq \{0,1\}$ be a finite set of points. We say that $\overline{U}$ is* easily replaceable on M *if*

1. *For each $a \in \mathbb{A}$, and for each $m \in M^{\setminus 0} \triangleq M \setminus \{0\}$, there exists $l_a(m)$ such that if $l_a(m) < m'$ for $m' \in M^{\setminus 0}$, then $m \leq m'$ and*

$$P(\overline{U}_a) = \bigcup_{m \in M^{\setminus 0}} [l_a(m), m].$$

2. *For each $a, a' \in \mathbb{A}$, $c(\overline{U}_a) = c(\overline{U}_{a'})$.*

The first part is valuable for the reduction as it removes the need to keep track of distinct right endpoints for each agent. The second part means that the coefficient in Equation (3) can be ignored when comparing these valuations with each other, which will be important later for the formulas to be in real linear arithmetic.

Theorem 5. *For any valuation set $\overline{V}$ and any finite set of points $M \supseteq \{0, 1\}$, there exists a piecewise uniform valuation set that both agrees with $\overline{V}$ on M and is easily replaceable on M.*

The proof constructs a specific piecewise uniform valuation set that satisfies these properties. The construction is a slightly more general version of the construction shown in Theorem 1 by Kurokawa, Lai, and Procaccia [8].

A consequence of the theorem is that any protocol execution can be replicated by the specific piecewise uniform valuation set, and any formulas that hold for the original valuation set that only consider points from the execution will also hold for this specific valuation set.

Example 3. We illustrate the construction for $\mathbb{A} = \{1, 2\}$ with valuation $\overline{V}_1$ being the uniform valuation over the cake, and $\overline{V}_2$ having density $x \mapsto 2x$, and the set of points $M = \{0, 1/2, 1\}$. Set $\overline{U}_1 = U_{P_1(d)}$ and $\overline{U}_2 = U_{P_2(d)}$ for pieces

$$\begin{aligned} P_1(d) &= [1/2 - 1/2 \cdot 1/d, 1/2] \cup [1 - 1/2 \cdot 1/d, 1] \\ P_2(d) &= [1/2 - 1/4 \cdot 1/d, 1/2] \cup [1 - 3/4 \cdot 1/d, 1] \end{aligned}$$

for $d \geq 3/2$. Clearly both pieces have interval right endpoints of $\{1/2, 1\} = M^{\setminus 0}$. Also, it is easily to calculate that $c(U_{P_1}) = c(U_{P_2}) = d$. Thus, $\overline{U}$ is easily replaceable on M. We additionally have

$$\begin{aligned} \overline{U}_1[0, 1/2] &= d \cdot (1/2 - (1/2 - 1/2 \cdot 1/d)) = 1/2 = \overline{V}_1[0, 1/2], \\ \overline{U}_1[1/2, 1] &= d \cdot (1 - (1 - 1/2 \cdot 1/d)) \quad\;\; = 1/2 = \overline{V}_1[1/2, 1], \\ \overline{U}_2[0, 1/2] &= d \cdot (1/2 - (1/2 - 1/4 \cdot 1/d)) = 1/4 = \overline{V}_2[0, 1/2], \\ \overline{U}_2[1/2, 1] &= d \cdot (1 - (1 - 3/4 \cdot 1/d)) \quad\;\; = 3/4 = \overline{V}_2[1/2, 1], \end{aligned}$$

so $\overline{U}$ replicates $\overline{V}$ on M. □

5.3 Piecewise uniform replacment

We leverage the above results to produce a reduction on protocol constraints. At a high level, for any formula having only variables for points, we can simplify it to be a disjunction of inequalities in terms of the form $\sum_i r_i \cdot \mathtt{V}_{a_i}(P_i)$ for real r_i and logical pieces and intervals P_i. We then replace terms of the form $\mathtt{V}_a(P)$ with sums of differences of real variables. Using Theorem 4 and Theorem 5 we can show that the original formula holds if and only if the replaced formula holds for a piecewise uniform valuation set.

Simplified terms are the terms within the following sets, indexed by sort:

$$R_{\mathtt{Point}} \triangleq \mathcal{Y} \cup \{r\#\mathsf{Pt} \mid r \in \mathbb{R}\} \qquad R_{\mathtt{Piece}} \triangleq \{\cup(t_1, \ldots, t_i) \mid t_i \in R_{\mathtt{Intvl}}\}$$
$$R_{\mathtt{Intvl}} \triangleq \{[t, t'] \mid t, t' \in R_{\mathtt{Point}}\} \qquad R_{\mathtt{Vltn}} \triangleq \{\textstyle\sum_i r_i \cdot \mathtt{V}_{a_i}(P_i) \mid P_i \in R_{\mathtt{Intvl}} \cup R_{\mathtt{Piece}}\}$$

For any well-sorted term t containing only variables in $\mathcal{Y}$, we can produce an equivalent simplified version of it, which we denote by $\mathtt{R}(t)$. The notion of simplified and the simplification operation $\mathtt{R}$ easily extends to whole formulas as well. For further details of this step, see the appendix. For further use, if t is a simplified term or formula, we let $\#\mathsf{Pt}(t)$ denote the subset of $R_{\mathtt{Point}}$ contained as subterms of t.

Proceeding with the reduction, we introduce a new set of logical variables, $\mathcal{Z}$ and we assume for each $y \in \mathcal{Y} \cup \{1\}$, and $a \in \mathbb{A}$, there is a unique $z_{a,y} \in \mathcal{Z}$. To understand the purpose of $\mathcal{Z}$, consider a piecewise unifrom valuation set $\overline{U}$ that is easily replaceable on M. Then $P(\overline{U}_a)$, the piece corresponding to agent a's valuation, is the union of intervals of the form $[l_a(m), m]$ for $m \in M^{\setminus 0}$. If the variable y represents the variable m, then the variable $z_{a,y}$ then represents the left endpoint $l_a(m)$.

Definition 4. *A* piecewise uniform replacement *is a finite totally ordered subset* $(S, >_S)$ *of* $\mathcal{Y} \cup \{0, 1\}$ *such that* $\{0, 1\} \subseteq S$ *and* $0 \leq_S y \leq_S 1$ *for all* $y \in S$*. For* $y, y' \in S$*, we define* $S|_y^{y'} \triangleq \{y'' \in S \mid y' \geq_S y'' >_S y\}$*. We let* $S|_t \triangleq S|_y^{y'}$ *if* $t = [y, y']$ *and* $S|_t \triangleq S|_{y_1}^{y_1'} \cup \cdots \cup S|_{y_n}^{y_n'}$ *if* $t = \cup[y_1, y_1'], \ldots, [y_n, y_n']$*.*

The piecewise uniform replacement packages neatly all the data needed to replace valuation symbols in formulas. The order on S represents the ordering of real numbers, since variables from $\mathcal{Y}$ represent points. A replacement is applied to terms:

Definition 5. *The application of a piecewise uniform substitution* S *on a term* $t \in R_{\mathtt{S}}$ *for which* $\#\mathsf{Pt}(t) \subseteq S$ *is as follows:*

$$S(\mathtt{V}_a(t)) \triangleq \sum_{y \in S|_t} (y - z_{a,y}) \qquad S(\sum_i r_i \cdot t_i) \triangleq \sum_i r_i \cdot S(t_i) \qquad S(t) \triangleq t \text{ otherwise.}$$

S can be applied to formulas by passing itself down to its terms. When $S|_t$ is empty, we replace the term with 0*.*

The piecewise uniform replacement syntactically applies Equation (3) (ignoring the constant) to valuation terms for valuations of the form shown in Definition 3. The following example illustrates this concretely.

Example 4. Returning to the path b shown in Example 2, consider the piecewise uniform replacement $S = \{0, y, 1\}$ where $0 <_S y <_S < 1$. Then $\mathtt{V}_a([0, y])$ and $\mathtt{V}_a([y, 1])$ are replaced with $y - z_{a,y}$ and $1 - z_{a,1}$ respectively. The formula $c(b)$ simplifies to

$$S(c(b)) = (y - z_{1,y} = 1/2 \cdot (y - z_{1,y} + 1 - z_{1,1})) \wedge (y - z_{2,y} \geq 1 - z_{2,1}),$$

and the encoding of envy-freeness becomes

$$S(E(\rho(b))) = (y - z_{2,y} \geq 1 - z_{2,1}) \wedge (y - z_{1,y} \geq 1 - z_{1,1}).$$

Both are clearly linear inequalities in real variables. □

Piecewise uniform replacements are used to reduce simplified formulas to linear real inequalities:

Proposition 3. *Let f be a simplified formula. Let S be a piecewise uniform replacement such that $\#\mathsf{Pt}(f) \subseteq S$. Then $S(f)$ consists only of conjunctions and disjunctions of linear inequalities of real variables.*

To apply piecewise uniform replacements in a sound way, the variable assignment must properly line it up with the valuation set; the precise conditions for this are given in the following definition.

Definition 6. *Let $\overline{U}$ be a piecewise uniform valuation set. Let S is a piecewise uniform replacement and μ a variable assignment. We write $S \xrightarrow{\mu} \overline{U}$ if*

1. *$\overline{U}$ is easily replaceable on $\mu(S)$ (by convention $\mu(0) = 0$ and $\mu(1) = 1$)*
2. *$\mu(z_{a,y}) = l_a(\mu(y))$ if $y = \min\{y' \in S \mid \mu(y') = \mu(y)\}$*
3. *$\mu(z_{a,y}) = \mu(y)$ if $y \neq \min\{y' \in S \mid \mu(y') = \mu(y)\}$*
4. *If $\mu(y) < \mu(y')$ then $y <_S y'$.*

This definition formalizes how we think of variables in a piecewise uniform replacement. Condition (1) says that $\mu(S)$ captures the right endpoints of $P(\overline{U}_a)$ correctly, (2) and (3) together ensure that we don't repeat values in our sums, and (4) ensures that the variable ordering is compatible with the real ordering given by μ.

Example 5. Let $S = \{0, y, 1\}$, and let $\overline{U}$ be the piecewise uniform valuation set described in Example 3. Set $\mu(y) = 1/2$, and

$$\begin{array}{ll} \mu(z_{1,y}) = 1/2 - 1/2 \cdot 1/d & \mu(z_{1,1}) = 1 - 1/2 \cdot 1/d \\ \mu(z_{2,y}) = 1/2 - 1/4 \cdot 1/d & \mu(z_{2,1}) = 1 - 3/4 \cdot 1/d. \end{array}$$

Then $\mu(S) = \{0, 1/2, 1\}$ and Example 3 illustrates that $\overline{U}$ is easily replaceable on $\mu(S)$. Also, $z_{a,y}$ is the left endpoint, $l_a(1/2)$, of the left interval for $P_a(d)$, and $z_{a,1}$ is the left endpoint, $l_a(1)$, of the right interval for $P_a(d)$, hence condition (2) is satisfied. Condition (3) is vacuous here. Clearly, $0 < 1/2 < 1$ and $0 <_S y <_S 1$ so condition (4) is satisfied. Thus we have that $S \xrightarrow{\mu} \overline{U}$. □

Piecewise uniform replacements preserve validity when the conditions in Definition 6 are met.

Theorem 6. *Let f be a simplified formula and let S be a piecewise uniform replacement such that $\#\mathsf{Pt}(f) \subseteq S$. Let μ be an assignment and $\overline{U}$ a piecewise uniform valuation set. If $S \xrightarrow{\mu} \overline{U}$ then $\mathcal{A}_{\overline{U}}, \mu \vDash f \iff \mathcal{A}_{\overline{U}}, \mu \vDash S(f)$.*

Example 6. We illustrate the theorem by applying it with $S = \{0, y, 1\}$ for the formula $c(b)$ Example 2 reproduced here:

$$c(b) = (\mathtt{V}_1([0, y]) = 1/2 \cdot \mathtt{V}_1([0, 1])) \wedge (\mathtt{V}_2([0, y]) \geq \mathtt{V}_2([y, 1])).$$

Supposing $S \xrightarrow{\mu} \overline{U}$, this formula is equivalent to its reduced version from Example 4:

$$S(c(b)) = (y - z_{1,y} = 1/2 \cdot (y - z_{1,y} + 1 - z_{1,1})) \wedge (y - z_{2,y} \geq 1 - z_{2,1}).$$

One can verify this equivalence for the example $\overline{U}$ and μ shown in Example 5.

Associated with each piecewise uniform replacement S, is a formula $\psi(S)$.

Definition 7. *Let S be a piecewise uniform replacement, written $S = \{y_1, \ldots, y_n\}$ so that $y_1 <_S \cdots <_S y_n$. We let $\psi(S)$ denote the conjunction of the following formulas for all agents $a, a' \in \mathbb{A}$:*

$$0 \leq z_{a,y_1} \leq y_1 \leq \cdots \leq z_{a,y_n} \leq y_n \leq 1, \qquad \sum_{y \in S} y - z_{a,y} = \sum_{y \in S \setminus \{0\}} y - z_{a',y}.$$

Whenever the above formula holds for some variable assignment μ, a piecewise uniform valuation set $\overline{U}$ that is easily replaceable on $\mu(S)$ can be constructed:

$$P(\overline{U}_a) = \bigcup_{y \in S \setminus \{0\}} [\mu(z_{a,y}), \mu(y)], \qquad c(\overline{U}_a) = \sum_{y \in S \setminus \{0\}} \mu(y) - \mu(z_{a,y}).$$

This assists us in showing that our constraint reduction procedure is complete.

We now state our main theorem. For a path b, let S_b be the set of piecewise uniform replacements on $\mathcal{Y}_b \cup \{0, 1\}$—note that this set is *finite*.

Theorem 7. *Suppose e is well-formed and $\cdot \vdash e : \mathsf{Piece}^{\mathbb{A}}$. Then $e \vDash E(x)$ if and only if*

$$\vDash \bigwedge_{b \in B(e)} \bigwedge_{S \in S_b} \forall \mathcal{Y}_b . S(\mathtt{R}(c(b) \wedge \psi(S) \Rightarrow E(\rho(b)))). \tag{4}$$

In contrast to Theorem 2, we no longer need to quantify over valuations—a valuation set is baked into the formula through $\psi(S)$. This also gives a valuation set witness whenever Formula (4) does not hold.

Similar to Theorem 2, this theorem can be extended to more general formulas $F(x)$.

Proof (sketch). For the forward direction, we assume that $e \Downarrow_{\overline{V}} v$ and apply Theorems 3 and 5 to obtain a piecewise uniform valuation set $\overline{U}$ for which $e \Downarrow_{\overline{U}} v$. Then it is a matter of applying Theorem 6 and Theorem 4 to obtain that $E(\mathtt{v})$ is satisfied. For the backward direction, we suppose that Formula (4) doesn't hold for some b and S_b, and use $\psi(S)$ to construct a piecewise uniform valuation set that evaluates to v yet $E(\mathtt{v})$ is not satisfied.

According to Proposition 3, Formula (4) consists entirely of linear inequalities of real variables. Thus, we have the following corollary.

Corollary 1. *Let e be a well-formed and well-typed Slice protocol. Checking if e is envy-free is decidable.*

6 Implementation & Evaluation

We implemented our type system and formula reduction on top of the Slice implementation. Protocols are first type-checked following our linear typing rules, and then compiled to linear real arithmetic constraints encoding envy-freeness, which are dispatched to Z3 [11].

Benchmark protocols. In our benchmarks, we include all original protocols implemented in Slice [3]; we briefly describe them here. *Cut-choose* is the classic 2 agent protocol where one agent cuts and the other picks. *Surplus* is a two agent protocol which leaves a "surplus" piece of the cake in the center. *Selfridge-Conway-Full* is the classic three agent protocol [5]. *Selfridge-Conway-Surplus* is a variant of Selfridge-Conway-Full that disposes the trimming, and *Waste-Makes-Haste-3* [15] effectively is a minor variant on Selfridge-Conway-Surplus.

We also implement two new, more complicated protocols. The first, *Aziz-Mackenzie-3*, is the three-agent variant of the first bounded envy-free four agent protocol with no free disposal [2]. Briefly, this protocol obtains an envy-free allocation by first obtaining a partial allocation where one agent does not care how the rest is allocated

amongst the others. Cut-Choose is then applied. The second, *Waste-Makes-Haste-4*, is the four-agent connected variant of the Waste-Makes-Haste free disposal protocol [15, Section 6]. This protocol relies on *equalize* queries: $\text{Equalize}_a(n)$ has agent a divide the cake to produce n equally most preferred (according to a) pieces of the cake. It can be shown using Hall's marriage theorem that an envy-free allocation can be made from a set of pieces following some sequence of equalize queries among the agents (for 4 agents, $n \leq 5$), although the allocation must be found through exhaustive search. This protocol exhaustively tries certain sequences of equalize queries until an envy-free allocation is obtained. Notably, this is the first four agent envy-free cake-cutting protocol to be implemented and verified.

Evaluation. Table 1 presents some statistics from verifying envy-freeness for each of our benchmark protocols. Our experiments were conducted on an M1 MacBook Pro with 16 GB of RAM. We measured the time both to compile protocols to constraints, and the actual time Z3 took to solve. We also record here the number of paths in each protocol, as well as the number of lines for the protocol implementation and the constraint formula. Each path corresponds to a distinct disjunct in the constraint. We measure this against the solving time for the original Slice constraints (Old), which uses non linear real arithmetic formulas. Our results demonstrate a reduction in solving time compared with the old constraints. The four-agent protocol is significantly more complex than the others, though Z3 can still solve the constraints efficiently.

Table 1: Verifying envy-freeness (averaged over 5 runs).

		Size (lines)		Time (seconds)		
Protocol	#Paths	Program	Constraints	Compile	Z3	Z3 (Old)
Cut-Choose	2	6	35	1.31	0.00	0.02
Surplus	2	11	56	1.23	0.00	0.02
Waste-Makes-Haste-3	24	8	924	0.85	0.02	0.84
Selfridge-Conway-Surplus	216	19	7726	1.09	0.01	0.82
Selfridge-Conway-Full	1800	21	98292	9.20	0.46	19.38
Aziz-Mackenzie-3	93384	23	8086180	2m4	6.82	n/a
Waste-Makes-Haste-4	1953792	290	157553237	37m02	1m22	n/a

7 Related & Future Work

Cake Cutting Verification. In recent work, Lester [10] proposes a system called Crumbs to verify and disprove envy-freeness, using C bounded model checker (CBMC)

instead of SMT. While performance on correct (envy-free) protocols is similar to the prior version of Slice, Lester [10] shows that Crumbs is much more effective at finding counterexamples for incorrect protocols. By using our new constraint reduction, our current work significantly outperforms Slice and Crumbs for correct protocols, and we can efficiently construct counterexamples for incorrect protocols (details in the appendix). In terms of expressivity, Crumbs supports a more restrictive, higher-level query model, enabling constraint solving over bounded integer arithmetic. In contrast, our work supports all protocols written in the standard Robertson-Webb model. Our system also establishes disjointness, which Crumbs does not consider.

Substructural Type Systems. Our type system is an example of a *substructural type system*, which originate from substructural logics. In brief, substructural logics restrict the application of assumptions in proofs. Likewise, substructural type systems restrict the usage of variables, enabling computational resource usage to be restricted. A classic example is *linear logic*, due to Girard [7], which led to *linear type systems* [9, 1, 16, 17]. Walker [18] provides a resource for learning about substructural type systems. Our type system is designed to ensures *physical disjointness* of parts of the cake; we are not aware of prior work that uses substructural types for a single divisible good, though there are similar ideas in separation logic (e.g., [4]).

Formal Methods and Social Choice. Cake-cutting protocols belong to a broader literature on social choice theory, which has had many fruitful interactions with formal methods. In one direction, formal methods researchers have used interactive theorem provers to verify classical protocols and impossibility theorems in social choice theory (e.g., [12]). In the other direction, social choice researchers have used computer-aided solvers to prove novel theorems in social choice theory (e.g., [6]).

Conclusions and future work. Our work makes progress in cake-cutting protocol verification, through an affine type system for disjointness and a formula reduction that enables much more efficient envy-freeness checking. However, there are envy-free cake-cutting protocols even more complex than what we can verify here. The complexity of these protocols makes it difficult to even write them down in Slice, let alone the constraint compiling and solving time involved. New Slice language features may be needed to address transcription effort, while improvements to the Slice implementation and early pruning of unreachable paths could significantly descrease both compile and solving time. The most notable of these protocols is the four agent version of Aziz-Mackenzie-*3* [2], which does not discard any cake, unlike Waste-Makes-Haste-4. By making our affine type system instead linear, it could be used to verify no cake is discarded for that protocol, and others already implemented.

Acknowledgments. We thank Cornell's PL discussion group (PLDG) and Martin Lester for discussions about this work. We also thank the anonymous reviewers for their close reading and detailed feedback. This work is partially supported by NSF grant CCF-2319186.

Bibliography

[1] Abramsky, S.: Computational interpretations of linear logic. Theoretical Computer Science **111**(1), 3–57 (1993), https://doi.org/10.1016/0304-3975(93)90181-R

[2] Aziz, H., Mackenzie, S.: A discrete and bounded envy-free cake cutting protocol for four agents. In: ACM SIGACT Symposium on Theory of Computing (STOC), Cambridge, Massachusetts, pp. 454–464 (2016), https://doi.org/10.1145/2897518.2897522

[3] Bertram, N., Levinson, A., Hsu, J.: Cutting the cake: A language for fair division. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida (2023), https://doi.org/10.1145/3591293

[4] Boyland, J.: Fractional permissions. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850, pp. 270–288, Springer (2013), https://doi.org/10.1007/978-3-642-36946-9_10

[5] Brams, S.J., Jones, M.A., Klamler, C.: Better ways to cut a cake. Notices of the AMS **53**(11), 1314–1321 (2006), URL https://www.ams.org/cgi-bin/notices/

[6] Geist, C.: Generating Insights in Social Choice Theory via Computer-aided Methods. Ph.D. thesis, Technical University Munich, Germany (2016), URL https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20160906-1296898-1-8

[7] Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987), https://doi.org/10.1016/0304-3975(87)90045-4

[8] Kurokawa, D., Lai, J., Procaccia, A.: How to cut a cake before the party ends. In: AAAI Conference on Artificial Intelligence, Bellevue, Washington (2013), https://doi.org/10.1609/aaai.v27i1.8629

[9] Lafont, Y.: The linear abstract machine. Theoretical Computer Science **59**(1), 157–180 (1988), https://doi.org/10.1016/0304-3975(88)90100-4

[10] Lester, M.M.: Cutting the cake into crumbs: Verifying envy-free cake-cutting protocols using bounded integer arithmetic. In: Practical Aspects of Declarative Languages (PADL), London, England (2024), https://doi.org/10.1007/978-3-031-52038-9_7

[11] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary (2008), https://doi.org/10.1007/978-3-540-78800-3_24

[12] Nipkow, T.: Social choice theory in HOL. Journal of Automated Reasoning **43**(3), 289–304 (2009), https://doi.org/10.1007/S10817-009-9147-4

[13] Procaccia, A.D.: Cake cutting algorithms. In: Brandt, F., Conitzer, V., Endriss, U., Lang, J., Procaccia, A.D. (eds.) Handbook of Computational Social Choice, p. 311–330, Cambridge University Press (2016), https://doi.org/10.1017/CBO9781107446984.014

[14] Robertson, J., Webb, W.: Cake-Cutting Algorithms: Be Fair If You Can. A K Peters/CRC Press (1998), https://doi.org/10.1201/9781439863855

[15] Segal-Halevi, E., Hassidim, A., Aumann, Y.: Waste makes haste: Bounded time algorithms for envy-free cake cutting with free disposal. ACM Transactions on Algorithms **13**(1), 1–32 (2016), https://doi.org/10.1145/2988232

[16] Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods, Sea of Galilee, Israel (1990)

[17] Wadler, P.: Is there a use for linear logic? In: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut, USA (1991), https://doi.org/10.1145/115865.115894

[18] Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, The MIT Press (2004), https://doi.org/10.7551/mitpress/1104.003.0003

Appendix

7.1 Values

Recall the definition of values. Here we define the read and unread operations carefully.

Definition 8. *Define the "read" operation and "unread" operation as follows:*

$$\overline{v} = \begin{cases} \overline{[r, r']} & v = [r, r'] \\ \overline{P_i[r_i, r_i']} & v = P_i[r_i, r_i'] \\ (\overline{v_1}, \ldots, \overline{v_n}) & v = (v_1, \ldots, v_n) \\ v & \textit{otherwise.} \end{cases} \quad |v| = \begin{cases} [r, r'] & v = \overline{[r, r']} \\ P_i[r_i, r_i'] & v = \overline{P_i[r_i, r_i']} \\ (|v_1|, \ldots, |v_n|) & v = (v_1, \ldots, v_n) \\ v & \textit{otherwise.} \end{cases}$$

These extend to sets: $\overline{\mathbb{V}} = \{\overline{v} \mid v \in \mathbb{V}\}$, $|\mathbb{V}| = \{|v| \mid v \in \mathbb{V}\}$.

7.2 Types

$$\begin{aligned} \hat{\tau} &::= \mathsf{Bool} \mid \mathsf{Point} \mid \mathsf{Vltn} \mid \overline{\mathsf{Intvl}} \mid \overline{\mathsf{Piece}} \\ \tau &::= \mathsf{Intvl} \mid \mathsf{Piece} \mid \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_n \end{aligned}$$

Fig. 7: Types for Slice expressions.

Recall our types here, or in Figure 7. Here we give more details on our type setup. Given type contexts Γ and Γ', the concatenation of them, denoted Γ, Γ' is the type context given as follows:

$$\Gamma, \Gamma'(x) = \begin{cases} \Gamma'(x) & \text{if } x \in \mathrm{dom}(\Gamma') \\ \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma'). \end{cases}$$

When concatenating linear type contexts Δ and Δ', we assert that $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset$. For each $o \in \mathcal{O}$, there is a unique signature $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}$ (note that this means operations do not apply to the affine types). Our complete typing rules are shown in Figure 8.

Proposition 4. *For a value v, there is a unique type τ such that $\cdot \vdash v : \tau$.*

Proof. We proceed by induction on the structure of values. For all values but those of the form $(v_1, \ldots, v_n)$, examining the typing rules show that each form of the value has only one typing rule that can be applied. Now assume that $v = (v_1, \ldots, v_n)$. By induction, there are unique types $\tau_1, \ldots, \tau_n$ such that $\cdot \vdash \tau_i : v_i$ for $1 \le i \le n$. The only rule that can be applied here is [T-Tup], for which we obtain $\cdot \vdash (v_1, \ldots, v_n) : \tau_1 \times \cdots \times \tau_n$.

Let $[\![\tau]\!] \triangleq \{v \in \mathcal{V} \mid \cdot \vdash v : \tau\}$, that is, the set of values that have type τ. Clearly, if $\tau \neq \tau'$, then $[\![\tau]\!] \cap [\![\tau']\!] = \emptyset$. Because of Proposition 4, we can concretely describe $[\![\tau]\!]$ for each τ:

$$[\![\mathsf{Bool}]\!] = \{\mathsf{true}, \mathsf{false}\} \quad [\![\mathsf{Point}]\!] = \{r\#\mathsf{Pt} \mid r \in \mathbb{R}\}$$

$$\frac{}{\Gamma; \Delta \vdash b : \mathsf{Bool}}\ \text{T-Bool} \qquad \frac{}{\Gamma; \Delta \vdash r\#\mathsf{Pt} : \mathsf{Point}}\ \text{T-Point} \qquad \frac{}{\Gamma; \Delta \vdash [r, r'] : \mathsf{Intvl}}\ \text{T-Intvl}$$

$$\frac{}{\Gamma; \Delta \vdash \overline{[r, r']} : \overline{\mathsf{Intvl}}}\ \text{T-RdIntvl} \qquad \frac{}{\Gamma; \Delta \vdash P[r_1, r_1'], \ldots, [r_n, r_n'] : \mathsf{Piece}}\ \text{T-Piece}$$

$$\frac{}{\Gamma; \Delta \vdash \overline{P[r_1, r_1'], \ldots, [r_n, r_n']} : \overline{\mathsf{Piece}}}\ \text{T-RdPiece} \qquad \frac{}{\Gamma; \Delta \vdash r_1 \cdot V_{a_1}(P_1) + \cdots + r_n \cdot V_{a_n}(P_n) : \mathsf{Vltn}}\ \text{T-Vltn}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{cake} : \mathsf{Intvl}}\ \text{T-Cake} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \quad \cdots \quad \Gamma; \Delta_n \vdash e_n : \hat{\tau}_n \quad o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash o(e_1, \ldots, e_n) : \hat{\tau}}\ \text{T-Ops}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma; \Delta_n \vdash e_n : \tau_n}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}\ \text{T-Tup} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \mathsf{Bool} \quad \Gamma; \Delta \vdash e_2 : \tau \quad \Gamma; \Delta \vdash e_3 : \tau}{\Gamma; \Delta_1, \Delta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\ \text{T-Cond}$$

$$\frac{}{\Gamma, x : \hat{\tau}; \Delta \vdash x : \hat{\tau}}\ \text{T-Var} \qquad \frac{}{\Gamma; w : \tau \vdash w : \tau}\ \text{T-AffVar}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'} \\ \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1 : \tau_1}, \ldots, \overline{w_n : \tau_{n'}}; \Delta_2, w_1 : \tau_1, \ldots, w_n : \tau_{n'} \vdash e_2 : \tau\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_n = \mathsf{split}\ e_1\ \mathsf{in}\ e_2 : \tau}\ \text{T-Split}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathsf{Intvl} \quad \Gamma; \Delta_2 \vdash e_2 : \mathsf{Point}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{divide}(e_1, e_2) : \mathsf{Intvl} \times \mathsf{Intvl}}\ \text{T-Div} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \overline{\mathsf{Intvl}} \quad \Gamma; \Delta_2 \vdash e_2 : \mathsf{Vltn}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{mark}_a(e_1, e_2) : \mathsf{Point}}\ \text{T-Mark}$$

$$\frac{\Gamma; \Delta \vdash e : \overline{\mathsf{Intvl}}}{\Gamma; \Delta \vdash \mathsf{eval}_a(e) : \mathsf{Vltn}}\ \text{T-EvalIntvl} \qquad \frac{\Gamma; \Delta \vdash e : \overline{\mathsf{Piece}}}{\Gamma; \Delta \vdash \mathsf{eval}_a(e) : \mathsf{Vltn}}\ \text{T-EvalPc}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathsf{Intvl} \quad \cdots \quad \Gamma; \Delta_n \vdash e_n : \mathsf{Intvl}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash \mathsf{piece}\ e_1, \ldots, e_n : \mathsf{Piece}}\ \text{T-Piece}$$

Fig. 8: Typing rules

$$[\![\mathsf{Vltn}]\!] = \{\sum_i r_i \cdot V_{a_i}(P_i) \mid r_i \in \mathbb{R}, a_i \in \mathbb{A}, P_i \in [\![\mathsf{Piece}]\!]\}$$

$$[\![\overline{\mathsf{Intvl}}]\!] = \{\overline{v} \mid v \in [\![\mathsf{Intvl}]\!]\} \qquad [\![\overline{\mathsf{Piece}}]\!] = \{\overline{v} \mid v \in [\![\mathsf{Piece}]\!]\}$$

$$[\![\mathsf{Intvl}]\!] = \{[r, r'] \mid r, r' \in \mathbb{R}, r \leq r'\} \qquad [\![\mathsf{Piece}]\!] = \{P_i v_i \mid v_i \in [\![\mathsf{Intvl}]\!]\}$$

$$[\![\hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'}]\!] = [\![\hat{\tau}_1]\!] \times \cdots \times [\![\hat{\tau}_n]\!] \times [\![\tau_1]\!] \times \cdots \times [\![\tau_{n'}]\!]$$

7.3 Semantics

Before we can state the complete rules, we require a few definitions. For each $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau} \in O$, we assume there is a function $[\![o]\!] : \bigtimes_{i=1}^{n} |[\![\hat{\tau}_i]\!]| \to |[\![\hat{\tau}]\!]|$. In this way, operations are defined not on the read only versions, but on the underlying values, which assists in the compatibility with the logic. Recall that our semantics is given by a relation $\Downarrow_{\overline{V}} \subseteq \mathcal{E} \times \mathcal{V}$, indexed by a set of valuations $\overline{V}$ (which we omit when the set in question is clear). The big-step rules are given in Figure 9.

$$\frac{}{v \Downarrow v}\ \text{E-Val} \qquad \frac{}{\mathsf{cake} \Downarrow [0,1]}\ \text{E-Cake}$$

$$\frac{e_1 \Downarrow v_1 \quad \cdots \quad e_n \Downarrow v_n \quad [\![o]\!](|v_1|, \ldots, |v_n|) = v}{o(e_1, \ldots, e_n) \Downarrow \overline{v}}\ \text{E-Ops}$$

$$\frac{e_1 \Downarrow v_1 \quad \cdots \quad e_n \Downarrow v_n}{(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n)}\ \text{E-Tup} \qquad \frac{e_1 \Downarrow \mathsf{true} \quad e_2 \Downarrow v_2}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_2}\ \text{E-True} \qquad \frac{e_1 \Downarrow \mathsf{false} \quad e_3 \Downarrow v_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_3}\ \text{E-False}$$

$$\frac{e_1 \Downarrow (v_1, \ldots, v_{n+n'}) \quad e_2\{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{\overline{w_i} \mapsto \overline{v_{i+n}} \mid 1 \leq i \leq n'\}\{w_i \mapsto v_{i+n} \mid 1 \leq i \leq n'\} \Downarrow v}{\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2 \Downarrow v}\ \text{E-Split}$$

$$\frac{e_1 \Downarrow [r_1, r_1'] \quad e_2 \Downarrow r_2 \quad r_1 \leq r_2 \leq r_1'}{\mathsf{divide}(e_1, e_2) \Downarrow ([r_1, r_2], [r_2, r_1'])}\ \text{E-Div}$$

$$\frac{e_1 \Downarrow [r_1, r_1'] \quad e_2 \Downarrow \sum_i r_i V_{a_i}(P_i) \quad \overline{V}_a([r_1, r]) = \sum_i r_i V_{a_i}(P_i)}{\mathsf{mark}_a(e_1, e_2) \Downarrow r}\ \text{E-Mark} \qquad \frac{e \Downarrow [r, r']}{\mathsf{eval}_a(e) \Downarrow \overline{V}_a[r, r']}\ \text{E-EvalIntvl}$$

$$\frac{e \Downarrow P\ [r_1, r_1'], \ldots, [r_n, r_n']}{\mathsf{eval}_a(e) \Downarrow \overline{V}_a(P[r_1, r_1'], \ldots, [r_n, r_n'])}\ \text{E-EvalPc} \qquad \frac{e_1 \Downarrow [r_1, r_1'] \quad \cdots \quad e_n \Downarrow [r_n, r_n']}{\mathsf{piece}\ e_1, \ldots, e_n \Downarrow P\ [r_1, r_1'], \ldots, [r_n, r_n']}\ \text{E-Piece}$$

Fig. 9: Big-step semantics

Here we also introduce the following derivation notation, which is helpful later on. Let D be a derivation of an evaluation judgement. If the conclusion of D is $e \Downarrow_{\overline{V}} v$, we write $D : e \Downarrow_{\overline{V}} v$. Given a derivation $D : e \Downarrow_{\overline{V}} v$, if e has subexpressions e_i, we let D_i be the derivation of the premise of $e \Downarrow_{\overline{V}} v$ that contains e_i. Similarly, if e is an expression with one subexpression e', we let D' denote the derivation of the premise containing e'. For example, if $D : (e_1, \ldots, e_n) \Downarrow_{\overline{V}} (v_1, \ldots, v_n)$, then $D_i : e_i \Downarrow_{\overline{V}} v_i$, and if $D : \mathsf{eval}_a(e') \Downarrow V_a([r, r'])$, then $D' : e' \Downarrow_{\overline{V}} [r, r']$.

7.4 Type soundness

Here we prove type soundness. As a reminder, the statement is the following:

Proposition 5 (Type soundness). *If $\cdot \vdash e : \tau$ then $e \Downarrow v$ implies $\cdot \vdash v : \tau$.*

We first provide some definitions. For each type τ, we let $R^{\vdash}_{\tau} \triangleq \{e \in \mathcal{E} \mid \cdot \vdash e : \tau, e \Downarrow v \Rightarrow v \in [\![\tau]\!]\}$, that is, the set of closed expressions that have type τ and evaluate only to values within $[\![\tau]\!]$. We now build up some lemmas in order to prove type soundness.

Lemma 1. $v \in [\![\tau]\!] \iff \cdot \vdash v : \tau \iff v \in R^{\vdash}_{\tau}$.

Proof. By inspection of definition of $[\![\tau]\!]$ and value typing rules.

Lemma 2. *Suppose $\cdot \vdash e : \tau$. If $e \Downarrow v \Rightarrow v \in R^{\vdash}_{\tau}$, then $e \in R^{\vdash}_{\tau}$.*

Proof. Follows immediately from Lemma 1.

We will aim to prove type soundness by induction on the typing derivation. For the proof to go through, we need to generalize the statement to open expressions. Let γ be an expression substitution replacing variables only with values. We say that $\gamma \mathrel{\widetilde{\vDash}} \Gamma$ if $\mathrm{dom}(\gamma) \subseteq \mathrm{dom}(\Gamma)$ and $\gamma(x) \in R^{\vdash}_{\Gamma(x)}$ for all $x \in \mathrm{dom}(\gamma)$. We say that $\gamma \vDash \Gamma$ and refer to γ as a *closing* substitution if $\gamma \mathrel{\widetilde{\vDash}} \Gamma$ and $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$. For linear type contexts, instead of using γ to denote a substitution, we use δ. So given a closing substitution γ and a linear substitution δ, we let $\gamma; \delta$ be the substitution that combines them together.

Proposition 6. *Suppose $\Gamma; \Delta \vdash e : \tau$. Let γ, δ be such that $\gamma \vDash \Gamma$ and $\delta \vDash \Delta$. Then $\gamma; \delta(e) \in R^{\vdash}_{\tau}$.*

Proof. This proceeds by induction on the typing derivation. Suppose that $\gamma \vDash \Gamma$, $\delta \vDash \Delta$, and $\Gamma; \Delta \vdash e : \tau$. We break up our argument in cases based on the last typing rule used.

[T-Ops] This is immediate since we assume $[\![o]\!] : |[\![\hat{\tau}_1]\!]| \times \cdots \times |[\![\hat{\tau}_n]\!]| \to |[\![\hat{\tau}]\!]|$, meaning if $\gamma; \delta(o(e_1, \ldots, e_n)) \Downarrow v$, then $v = \overline{[\![o]\!]}(|v_1|, \ldots, |v_n|) \in [\![\hat{\tau}]\!] = [\![\tau]\!]$, where $\gamma; \delta(e_i) \Downarrow v_i$.

[T-Tup] Then $\tau = \tau_1 \times \cdots \times \tau_n$, $\Delta = \Delta_1, \ldots, \Delta_n$, and $\Gamma; \Delta_i \vdash e_i : \tau_i$ for $1 \leq i \leq n$, where we would have $e = (e_1, \ldots, e_n)$. Suppose that $\gamma; \delta(e) \Downarrow v$. We need to show that $v \in [\![\tau_1 \times \cdots \times \tau_n]\!]$. By our typing rule, we know that δ can be partitioned as $\delta_1, \ldots, \delta_n$ where

$$\gamma; \delta(e) = (\gamma; \delta_1(e_1), \ldots, \gamma; \delta_n(e_n)) \tag{5}$$

and $\gamma; \delta(e_i) \Downarrow v_i$, $v = (v_1, \ldots, v_n)$. By induction $\gamma; \delta_i(e_i) \in R^{\vdash}_{\tau_i}$. By defintiion of $R^{\vdash}_{\tau_i}$, this means that $v_i \in [\![\tau_i]\!]$. This is enough to conclude that $v \in [\![\tau]\!]$ and therefore $\gamma; \delta(e) \in R^{\vdash}_{\tau}$.

[T-Piece] The argument for this case is nearly identical to that of [T-Tup].

[T-Split] Then $e = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = e_1\ \mathsf{in}\ e_2$. Set

$$\begin{aligned}
\tau_1 &= \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'} \\
\Gamma' &= \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}} \\
\Delta' &= \Delta_2, w_1 : \tau_1, \ldots, w_n : \tau_{n'}.
\end{aligned}$$

This means we have $\Gamma; \Delta_1 \vdash e_1 : \tau_1$ and $\Gamma'; \Delta' \vdash e_2 : \tau$. If $\gamma; \delta(e) \Downarrow v$, then we would have

$$\gamma; \delta(e_1) \Downarrow (v_1, \ldots, v_n, \ldots, v_{n+n'})$$

and

$$\gamma; \delta(e_2)\{x_i \mapsto v_i \mid 1 \le i \le n\}\{w_i \mapsto v_{i+n}, \overline{w_i} \mapsto \overline{v_{i+n}} \mid 1 \le i \le n'\} \Downarrow v$$

We can partition δ up into δ_1 and δ_2 such that $\delta_1 \vDash \Delta_1$ and $\delta_2 \vDash \Delta_2$ and such that $\gamma; \delta_1(e_1) = \gamma; \delta(e_1)$ and $\gamma; \delta_2(e_2) = \gamma; \delta(e_2)$. By induction, $\gamma; \delta_1(e_1) \in R^{\vdash}_{\tau_1}$. Then $(v_1, \ldots, v_{n+n'}) \in [\![\tau_1]\!]$, which means that $v_i \in [\![\tau_i]\!]$ if $1 \le i \le n$ and $v_{i+n} \in [\![\tau_i]\!]$, $\overline{v_{i+n}} \in [\![\overline{\tau_i}]\!]$ if $1 \le i \le n'$. So if we set

$$\gamma' = \gamma_2\{x_i \mapsto v_i \mid 1 \le i \le n\}\{\overline{w_i} \mapsto \overline{v_{i+n}} \mid 1 \le i \le n\}$$

and

$$\delta' = \delta_2\{w_i \mapsto v_{n+i} \mid 1 \le i \le n'\},$$

then $\gamma' \vDash \Gamma'$ and $\delta' \vDash \Delta'$. By induction, $\gamma'; \delta'(e_2) \in R^{\vdash}_{\tau}$. Moreover,

$$\gamma'; \delta'(e_2) = \gamma; \delta(e_2)\{x_i \mapsto v_i \mid 1 \le i \le n\}\{w_i \mapsto v_{n+i}, \overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \le i \le n'\}.$$

This gives us $\gamma'; \delta'(e_2) \Downarrow v$. Thus we have $v \in [\![\tau]\!]$, so by Lemma 1, $v \in R^{\vdash}_{\tau}$.

The above argument establishes for any v such that $\gamma; \delta(e) \Downarrow v$, we have $v \in R^{\vdash}_{\tau}$. By Lemma 2, $\gamma; \delta(e) \in R^{\vdash}_{\tau}$.

[T-Var] This follows easily as $\gamma \vDash \Gamma$.

[T-AffVar] This also follows easily as $\delta \vDash w : \tau$.

[T-Div] Then $e = \mathsf{divide}(e_1, e_2)$ and $\Delta = \Delta_1, \Delta_2$, with $\Gamma; \Delta_1 \vdash e_1 : \mathsf{Intvl}$ and $\Gamma; \Delta_2 \vdash e_2 : \mathsf{Point}$.

Suppose that $\gamma; \delta(e) \Downarrow v$.

Then $\gamma; \delta(e_1) \Downarrow [r_1, r_1']$, $\gamma; \delta(e_2) \Downarrow r_2\#\mathsf{Pt}$ with $r_1 \le r_2 \le r_1'$ and $v = ([r_1, r_2], [r_2, r_1'])$. It is immediate that $v \in R^{\vdash}_{\mathsf{Intvl} \times \mathsf{Intvl}}$.

[T-Mark] Then $e = \mathsf{mark}_a(e_1, e_2)$, $\tau = \mathsf{Point}$, $\Delta = \Delta_1, \Delta_2$, and $\Gamma; \Delta_1 \vdash e_1 : \overline{\mathsf{Intvl}}$, $\Gamma; \Delta_2 \vdash e_2 : \mathsf{Real}$. If $\gamma; \delta(e) \Downarrow v$, then $v = r'\#\mathsf{Pt}$ for some real r'. We are already done as $v \in \mathsf{PT}$.

[T-EvalPc] Then $e = \mathsf{eval}_a(e')$, $\tau = \mathsf{Vltn}$, and $\Gamma; \Delta \vdash e' : \mathsf{Piece}$. By induction, $\gamma; \delta(e') \in R^{\vdash}_{\mathsf{Piece}}$. This means that if $\gamma; \delta(e) \Downarrow v$, then $\gamma; \delta(e') \Downarrow P$ for some piece P, so $v = V_a(P)$. Thus $v \in \mathbb{V}$. This allows us to conclude $\gamma; \delta(e) \in R^{\vdash}_{\mathsf{Vltn}}$.

[T-EvalIntvl] Similar to [T-EvalPc].

As a corollary, we recieve type soundness.

7.5 Disjointness

We begin by defining disjointness for values and expressions. To do so, we introduce *interval lists*, which are just finite lists of intervals. For a value v, we let $I(v)$ denote the *interval list of* v, which is defined by induction on the structure of v:

$$I([r, r']) \triangleq [r, r'] \qquad I(P_i[r_i, r_i']) \triangleq [r_1, r_1'], \ldots, [r_n, r_n'] \qquad I((v_1, \ldots, v_n)) \triangleq I(v_1), \ldots, I(v_n) \qquad I(\overline{[r, r']}) \triangleq \emptyset$$

$$I(\overline{P_i[r_i, r_i']}) \triangleq \emptyset \qquad I(\overline{r\#\mathsf{Pt}}) \triangleq \emptyset \qquad I(r\#\mathsf{Pt}) \triangleq \emptyset \qquad I(\mathsf{true}) \triangleq \emptyset \qquad I(\mathsf{false}) \triangleq \emptyset$$

Let e be an expression. The *interval list of* e, denoted $I(e)$, is given as follows by induction on the structure of e:

$$I(x) \triangleq \emptyset \qquad I(w) \triangleq \emptyset \qquad I(\overline{w}) \triangleq \emptyset \qquad I((e_1, \ldots, e_n)) \triangleq I(e_1), \ldots, I(e_n) \qquad I(o(e_1, \ldots, e_n)) \triangleq I(e_1), \ldots, I(e_n)$$

$$I(\mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3) \triangleq I(e_1), I(e_2), I(e_3) \qquad I(\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2) \triangleq I(e_1), I(e_2)$$

$$I(\mathsf{cake}) \triangleq [0, 1] \qquad I(\mathsf{divide}(e_1, e_2)) \triangleq I(e_1), I(e_2) \qquad I(\mathsf{piece}(e_1, \ldots, e_n)) \triangleq I(e_1), \ldots, I(e_n) \qquad I(\mathsf{eval}_a(e)) \triangleq I(e)$$

$$I(\mathsf{mark}_a(e_1, e_2)) \triangleq I(e_1), I(e_2)$$

We also define interval lists for variable substitutions. Let S be a variable substitution. Then $I(S) \triangleq I(e_1), \ldots, I(e_n)$ if $S = \{w_i \mapsto e_i \mid 1 \leq i \leq n\}$. That is, $I(S)$ is the list of intervals in the range of S. We may also write $|L|$ to denote the set of intervals in L.

An interval list L is *disjoint* if L_i and L_j are disjoint for all $i \neq j$. Finally, a value v is *disjoint* if $I(v)$ is disjoint and an expression e is *disjoint* if $I(e)$ is disjoint.

We now argue for the following proposition:

Proposition 2. *If $\cdot \vdash e : \tau$ and e is disjoint, then $e \Downarrow v$ implies v is disjoint.*

Our technique will be similar to what was used for type soundness, though additional work is involved. First we extend $\delta \vDash \Delta$ to also reflect disjointness.

Definition 9. *Now let P be a piece and Δ a linear type context. We write $\delta \tilde{\vDash}^P \Delta$ if*

1. *$dom(\delta) \subseteq dom(\Delta)$*
2. *$\delta(w) \in R_{\Delta(w)}$ for all $w \in dom(\delta)$*
3. *$I(\delta)$ is disjoint*
4. *$\cup I(\delta)$ is disjoint with P (besides possibly at finitely many points)*

If $dom(\delta) = dom(\Delta)$, we write $\delta \vDash^P \Delta$.

We introduce some helpful set notation. Given a collection of sets C, we let $\cup C$ denote $\cup_{c \in C} c$. In particular, if I is a set of intervals, then $\cup I$ is the piece formed by all intervals within I.

Lemma 3. *If $e \Downarrow v$, then $\cup I(v) \subseteq \cup I(e)$.*

Proof. This goes by induction on the derivation of $e \Downarrow v$. We only show the interesting cases.

[E-Cake] This is clear.

[E-Ops] Then $v = \overline{[\![o]\!]}(|v_1|, \ldots, |v_n|)$, $e = o(e_1, \ldots, e_n)$, $e_i \Downarrow v_i$. We have $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}$ for some non-linear types $\hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau}$. We must have $v \in [\![\hat{\tau}]\!]$. All values contained in $[\![\hat{\tau}]\!]$ for any non-linear type $\hat{\tau}$ have their interval lists empty. Therefore, $I(v) = \emptyset$.

[E-Div] Then $v = ([r_1, r_2], [r_2, r_1'])$ where $e_1 \Downarrow [r_1, r_1']$ and $e_2 \Downarrow r_2$. Then $\cup I(v) \subseteq [r_1, r_1'] \subseteq \cup I(e_1) \subseteq \cup I(e)$, where the second containment follows by induction.

[E-Piece] Then $v = P[r_1, r_1'], \ldots, [r_n, r_n']$ and $b = \mathsf{piece}\ e_1, \ldots, e_n$ where $e_i \Downarrow [r_i, r_i']$. By induction, $[r_i, r_i'] \subseteq \cup I(e_i)$. Since $\cup I(e) = \cup I(e_1) \cup \cdots \cup I(e_n)$ and $\cup I(v) = \bigcup_i [r_i, r_i']$, we conclude $\cup I(v) \subseteq \cup I(e)$.

[E-Split] Then $e = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2$ and $e_1 \Downarrow (v_1, \ldots, v_{n+n'})$, and

$$e_2\{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{w_i \mapsto v_{n+i} \mid 1 \leq i \leq n'\} \Downarrow v.$$

Let e_2' denote the expression in the above judgement. By induction, $\cup I(v_1, \ldots, v_{n+n'}) \subseteq \cup I(e_1)$ and $\cup I(v) \subseteq \cup I(e_2')$. Note that $\cup I(e_2') = \cup(I(v_1, \ldots, v_{n+n'}) \cup I(e_2)) \subseteq \cup(I(e_1) \cup I(e_2)) = \cup I(e)$. This concludes the current case.

Lemma 4. *Let $v \in [\![\hat{\tau}]\!]$. Then $I(v) = \emptyset$.*

Proof. By inspection of definition of $I(v)$ for values in $[\![\hat{\tau}]\!]$ for some non-linear type $\hat{\tau}$. □

Lemma 5. *Suppose that $\gamma \vDash \Gamma$. Then $I(\gamma) = \emptyset$.*

Proof. Take $x \in \mathrm{dom}(\gamma)$. As $\gamma \vDash \Gamma$, $\gamma(x) \in R^{\vdash}_{\Gamma(x)}$. By Lemma 1, $\gamma(x) \in [\![\Gamma(x)]\!]$. By definition, $\Gamma(x) = \hat{\tau}$ for some $\hat{\tau}$. By Lemma 4, $I(\gamma(x)) = \emptyset$. Thus we can conclude $I(\gamma) = \emptyset$. □

Lemma 6. *For any expression e and substitution S, $\cup I(S(e)) = (\cup I(e)) \cup (\cup I(S))$.*

Proof. Straightforward induction on the structure of e. □

Lemma 7. *Let $\Delta = \Delta_1, \ldots, \Delta_n$ be a linear type context. Set $\delta_i = \delta|_{dom(\Delta_i)}$. The following three statements hold:*

(a) If $\delta \vDash \Delta$ then $\delta_i \vDash \Delta_i$
(b) If $\delta \tilde{\vDash}^P \Delta$, then $\delta_i \tilde{\vDash}^P \Delta_i$ for all $1 \leq i \leq n$.
(c) If $\delta \vDash^P \Delta$, then $\delta_i \vDash^P \Delta_i$ for all $1 \leq i \leq n$.

Proof. We first argue for (b), and then show how to use the arguments for (a) and (c). Suppose that $\delta \tilde{\vDash}^P \Delta$. We must show 1-4 for each δ_i.

1. As $\mathrm{dom}(\delta) \subseteq \mathrm{dom}(\Delta)$, we must have that $\mathrm{dom}(\delta_i) = \mathrm{dom}(\delta|_{\mathrm{dom}(\Delta_i)}) \subseteq \mathrm{dom}(\Delta_i)$ since $\mathrm{dom}(\Delta_i) \subseteq \mathrm{dom}(\Delta)$.
2. For each $w \in \mathrm{dom}(\delta_i)$, we have
$$\delta_i(w) = \delta(w) \in R_{\Delta(w)} = R_{\Delta_i(w)}$$
where the last equality uses that $\mathrm{dom}(\delta_i) \subseteq \mathrm{dom}(\Delta_i)$.
3. Since δ_i is a restriction of δ, we also have that $I(\delta_i)$ is disjoint as $I(\delta)$ is disjoint.
4. Since δ_i is a restriction of δ, $\cup I(\delta_i) \subseteq \cup I(\delta)$. As $I(\delta)$ is disjoint with P, it must be that $\cup I(\delta)$ is also disjoint with P.

These four parts establish that $\delta_i \tilde{\vDash}^P \Delta_i$. If $\mathrm{dom}(\delta) = \mathrm{dom}(\Delta)$, then $\mathrm{dom}(\delta_i) = \mathrm{dom}(\delta|_{\mathrm{dom}(\Delta_i)}) = \mathrm{dom}(\Delta|_{\mathrm{dom}(\Delta_i)}) = \mathrm{dom}(\Delta_i)$. This, along with 2 above establishes (a). The same fact, along with 2,3, and 4 establishes (c). □

Lemma 8. *Suppose that $I(e)$ is disjoint, $\Gamma; \Delta \vdash e : \tau$, $\gamma \tilde{\vDash} \Gamma$, $\delta \tilde{\vDash}^P \Delta$ and $\cup I(e) \subseteq P$. Then $\gamma; \delta(e)$ is disjoint.*

Proof. This proceeds by induction on the derivation of $\Gamma; \Delta \vdash e : \tau$.

[T-Ops] Then $e = o(e_1, \ldots, e_n)$, $\Gamma, \Delta_i \vdash e_i : \hat{\tau}_i$, $\tau = \hat{\tau}$ where $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}$, $\Delta = \Delta_1, \ldots, \Delta_n$. According to Lemma 7, we can partition δ into $\delta_1, \ldots, \delta_n$ such that $\delta_i \tilde{\vDash}^P \Delta_i$. Morover, we have $\gamma; \delta(e_i) = \gamma; \delta_i(e_i)$. Then by induction, $\gamma; \delta(e_i)$ is disjoint. By Lemma 5 and Lemma 6, we have that

$$\cup I(\gamma; \delta_i(e_i)) \subseteq (\cup I(\delta_i)) \cup (\cup I(e_i)).$$

As $I(\delta)$ is disjoint, we have that $I(\delta_i)$ is disjoint from $I(\delta_j)$ if $i \neq j$. As $I(e)$ is disjoint, we have that $I(e_i)$ is disjoint from $I(e_j)$ if $i \neq j$. Therefore $\gamma; \delta_i(e_i)$ is disjoint from $\gamma; \delta_j(e_j)$ if $i \neq j$. These facts allow us to conclude that $\gamma; \delta(e)$ is disjoint.

[T-Split] Then $e = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2$, $\Delta = \Delta_1, \Delta_2$ where

$$\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'}$$

and

$$\Gamma, \Gamma'; \Delta_2, \Delta' \vdash e_2 : \tau.$$

where

$$\Gamma' = x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}}$$
$$\Delta' = w_1 : \tau_1, \ldots, w_n : \tau_{n'}.$$

Also let γ_2 be γ but with $x_1, \ldots, x_n, \overline{w_1}, \ldots, \overline{w_{n'}}$ removed. According to Lemma 7, we can partition δ into δ_1 and δ_2 such that $\delta_1 \mathrel{\widetilde{\models}}^P \Delta_1$, $\delta_2 \mathrel{\widetilde{\models}}^P \Delta_2$ and $\gamma; \delta(e_1) = \gamma; \delta_1(e_1)$, $\gamma_2; \delta(e_2) = \gamma_2; \delta_2(e_2)$. As $I(e_1) \subseteq I(e)$, we have $\cup I(e_1) \subseteq P$, so we can apply induction to obtain that $\gamma; \delta_1(e_1)$ is disjoint.
As $I(e_2) \subseteq I(e)$, we have $\cup I(e_2) \subseteq P$. Therefore we can similarly apply induction to obtain $\gamma_2; \delta_2(e_2)$ is disjoint.
By Lemma 5 and Lemma 6, we have

$$\cup I(\gamma; \delta_1(e_1)) \subseteq (\cup I(\delta_1)) \cup (\cup I(e_1)) \quad \text{and} \quad \cup I(\gamma_2; \delta_2(e_2)) \subseteq (\cup I(\delta_2)) \cup (\cup I(e_2)).$$

As $I(\delta)$ is disjoint, $\cup I(\delta_1)$ is disjoint from $\cup I(\delta_2)$. As $I(e)$ is disjoint, $\cup I(e_1)$ is disjoint from $\cup I(e_2)$. This allows us to conclude that $I(\gamma; \delta_1(e_1))$ is disjoint from $I(\gamma_2; \delta_2(e_2))$. Note

$$\gamma; \delta(e) = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ \gamma; \delta_1(e_1)\ \mathsf{in}\ \gamma_2; \delta_2(e_2)$$

and because of this, we can conclude that $\gamma; \delta(e)$ is disjoint.

[T-AffVar] Then $\Gamma; w : \tau \vdash w : \tau$ and $\gamma; \delta(e) = \delta(w)$. By assumption, $I(\delta)$ is disjoint, so this case is concluded.

[T-Tup] Then $e = (e_1, \ldots, e_n)$, $\Delta = \Delta_1, \ldots, \Delta_n$, and we have $\Gamma; \Delta_i \vdash e_i : \tau_i$ for $1 \leq i \leq n$. According to Lemma 7, we can partition δ into $\delta_1, \ldots, \delta_n$ such that $\delta_i \mathrel{\widetilde{\models}}^P \Delta_i$. Morover, we have $\gamma; \delta(e_i) = \gamma; \delta_i(e_i)$. Then by induction, $\gamma; \delta(e_i)$ is disjoint. By Lemma 5 and Lemma 6, we have that

$$\cup I(\gamma; \delta_i(e_i)) \subseteq (\cup I(\delta_i)) \cup (\cup I(e_i)).$$

As $I(\delta)$ is disjoint, we have that $I(\delta_i)$ is disjoint from $I(\delta_j)$ if $i \neq j$. As $I(e)$ is disjoint, we have that $I(e_i)$ is disjoint from $I(e_j)$ if $i \neq j$. Therefore $\gamma; \delta_i(e_i)$ is disjoint from $\gamma; \delta_j(e_j)$ if $i \neq j$. These facts allow us to conclude that $\gamma; \delta(e)$ is disjoint.

All other cases are either trivial, or very similar to [Tup]. □

Proposition 7. *Suppose $\Gamma; \Delta \vdash e : \tau$. Let γ, δ be such that $\gamma \models \Gamma$ and $\delta \models^P \Delta$ and $\cup I(e) \subseteq P$. If $I(e)$ is disjoint, then $\gamma; \delta(e) \in R_\tau$.*

Proof. Suppose that $\Gamma; \Delta \vdash e : \tau$, $\gamma \models \Gamma$, $\delta \models^P \Delta$, $\cup I(e) \subseteq P$, and $I(e)$ disjoint. We already immediately have by Proposition 6 that $\gamma; \delta(e) \in R_\tau^{\vdash}$. This means we only have to argue for disjointness of the value evaluated here. So suppose that $\gamma; \delta(e) \Downarrow v$. This argument goes by induction on the derivation of $\Gamma; \Delta \vdash e : \tau$ and we break up our argument into cases based on the last typing rule applied.

[T-Tup] Then $\tau = \tau_1 \times \cdots \times \tau_n$, $\Delta = \Delta_1, \ldots, \Delta_n$, and $\Gamma; \Delta_i \vdash e_i : \tau_i$ for $1 \leq i \leq n$, where $e = (e_1, \ldots, e_n)$. Also, $v = (v_1, \ldots, v_n)$ and $\gamma; \delta(e_i) \Downarrow v_i$ for $1 \leq i \leq n$.
Since we write $\Delta = \Delta_1, \ldots, \Delta_n$, by Lemma 7, we can partition δ into $\delta_1, \ldots, \delta_n$ such that $\delta_i \models^P \Delta_i$. As $\Gamma; \Delta_i \models^P e_i : \tau_i$, it must be that $\gamma; \delta_i(e_i) = \gamma; \delta(e_i)$.
As $\cup I(e) \subseteq P$, $\cup I(e_i) \subseteq P$. By Lemma 8, $\gamma; \delta(e)$ is disjoint, so that $\gamma; \delta_i(e_i)$ is disjoint for all $1 \leq i \leq n$.
As $\gamma \models \Gamma$, $\delta_i \models^P \Delta_i$, $\cup I(e_i) \subseteq P$, and $I(e_i)$ is disjoint, we can apply induction to obtain $\gamma; \delta_i(e_i) \in R_{\tau_i}$. Therefore $I(v_i)$ is disjoint for all $1 \leq i \leq n$. By Lemma 3, $\cup I(v_i) \subseteq \cup I(\gamma; \delta_i(e_i))$. Thus if $i \neq j$ then $\cup I(v_i)$ is disjoint from $\cup I(v_j)$. This is sufficent to conclude that $I(v)$ is disjoint, completing this case.

[T-Ops] Then $e = o(e_1, \ldots, e_n)$ and $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}$. As $\gamma; \delta(e) \in R_{\hat{\tau}}^{\vdash}$, we have $v \in [\![\hat{\tau}]\!]$. By Lemma 4, $I(v) = \emptyset$. Therefore $I(v)$ is disjoint.

[T-Piece] The argument for this case is nearly identical to that of [T-Tup].

[T-Split] Then $e = \mathsf{let}\ x_1, \ldots, x_n = e_1\ \mathsf{in}\ e_2$. Set

$$\begin{aligned}\tau_1 &= \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'} \\ \Gamma' &= \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_n} : \overline{\tau_{n'}} \\ \Delta_1' &= w_1 : \tau_1, \ldots, w_n : \tau_{n'} \\ \Delta' &= \Delta_2, \Delta_1'\end{aligned}$$

This means $\Gamma; \Delta_1 \vdash e_1 : \tau_1$ and $\Gamma'; \Delta' \vdash e_2 : \tau$. Since $\delta \vDash^P \Delta$, according to Lemma 7 we can partition δ up into δ_1 and δ_2 such that $\delta_1 \vDash^P \Delta_1$ and $\delta_2 \vDash^P \Delta_2$. Moreover, we have that $\gamma; \delta_1(e_1) = \gamma; \delta(e_1)$ and $\gamma; \delta_2(e_2) = \gamma; \delta(e_2)$. We have $I(e_1) \cup I(e)$ so $\cup I(e_1) \subseteq \cup I(e) \subseteq P$, and $I(e_1)$ is disjoint as $I(e)$ is disjoint.
As $\gamma; \delta(e) \Downarrow v$, it must be that $\gamma; \delta_1(e_1) \Downarrow (v_1, \ldots, v_n, \ldots, v_{n+n'})$ for some values v_i for $1 \leq i \leq n + n'$, and that

$$\gamma; \delta(e_2)\{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{\overline{w_i} \mapsto \overline{v_i}, w_i \mapsto v_i \mid 1 \leq i \leq n'\} \Downarrow v.$$

Since $\gamma \vDash \Gamma$, $\delta_1 \vDash^P \Delta_1$, $\cup I(e_1) \subseteq P$, and $I(e_1)$ is disjoint, we can apply induction to obtain that $I((v_1, \ldots, v_{n+n'}))$ is disjoint.
Now set

$$\gamma' = \gamma\{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{\overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \leq i \leq n'\},$$

$$\delta_1' = \{w_i \mapsto v_{n+i} \mid 1 \leq i \leq n'\},$$

and

$$\delta' = \delta_2 \delta_1'.$$

Clearly

$$\gamma'; \delta'(e_2) = \gamma; \delta(e_2)\{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{\overline{w_i} \mapsto \overline{v_i}, w_i \mapsto v_i \mid 1 \leq i \leq n'\}$$

so we also have $\gamma'; \delta'(e_2) \Downarrow v$. By Lemma 3,

$$\cup I((v_1, \ldots, v_{n+n'})) \subseteq \cup I(e_1).$$

This means

$$\cup I(\delta_1') \subseteq \cup I(e_1) \subseteq P.$$

As $\delta_2 \vDash^P \Delta_2$, $I(\delta_2)$ is disjoint from P and this means $I(\delta_2)$ is disjoint from $I(\delta_1')$. So setting $P' = P \setminus \cup I(e_1)$, we have $\delta_1' \vDash^{P'} \Delta_1'$. This means $\delta' \vDash^{P'} \Delta'$ as $\delta' = \delta_2 \delta_1'$, and $\Delta' = \Delta_2, \Delta_1'$.
As $I(e)$ is disjoint, $I(e_2)$ is disjoint and $\cup I(e_1)$ and $\cup I(e_2)$ are disjoint from each other. Because $\cup I(e) \subseteq P$ and $\cup I(e_2)$ is disjoint from $\cup I(e_1)$, it must be that $\cup I(e_2) \subseteq P'$.
Together we have $\gamma' \vDash \Gamma'$, $\delta' \vDash^{P'} \Delta'$, $\cup I(e) \subseteq P'$, and $I(e_2)$ is disjoint, so by induction, $\gamma'; \delta'(e_2) \in R_\tau$. In particular, this means that $I(v)$ is disjoint. This establishes that $\gamma; \delta(e) \in R_\tau$.

[T-Var] This follows easily as $\gamma \vDash \Gamma$.

[T-AffVar] This also follows easily as $\delta \vDash^P w : \tau$.

[T-Div] It must be the case that $I(v) = [r_1, r_2], [r_2, r_1']$ where $r_1 \leq r_2 \leq r_1'$. This is already enough to conclude that $I(v)$ is disjoint.

The remaining cases are straightforward. □

As a corollary, we receive our desired result.

7.6 Paths

Recall the section on paths. Here are the path rules for the assert expression.

$$\frac{\Gamma; \Delta_1 \vdash b_1 : \mathsf{Bool} \qquad \Gamma; \Delta_2 \vdash b_2 : \tau}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{assert}\ b_1\ \mathsf{in}\ b_2 : \tau}\ [\text{T-Assert}] \qquad \frac{b_1 \Downarrow \mathsf{true} \qquad b_2 \Downarrow v}{\mathsf{assert}\ b_1\ \mathsf{in}\ b_2 \Downarrow v}\ [\text{E-Assert}]$$

We define the set of paths $B(e)$ by induction on the structure of e:

$$B(v) \triangleq \{v\} \quad B(x) \triangleq \{x\} \quad B(w) \triangleq \{w\} \quad B(\overline{w}) \triangleq \overline{w}$$

$$\begin{aligned} B(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) &\triangleq \{\mathsf{assert}\ b_1\ \mathsf{in}\ b_2 \mid b_1 \in B(e_1), b_2 \in B(e_2)\} \\ &\quad \cup \{\mathsf{assert}\ \neg b_1\ \mathsf{in}\ b_3 \mid b_1 \in B(e_1), b_3 \in B(e_3)\} \\ B(e(e_1, \ldots, e_n)) &\triangleq \{e(b_1, \ldots, b_n) \mid b_1 \in B(e_1), \ldots, b_n \in B(e_n)\} \text{ otherwise.} \end{aligned}$$

for all other expressions $e(e_1, \ldots, e_n)$. Here we prove the theorems from Section 4

Proposition 8. *$\Gamma; \Delta \vdash e : \tau$ if and only if $\Gamma; \Delta \vdash b : \tau$ for all $b \in B(e)$.*

Proof. ($\Rightarrow$)
We proceed by induction on the typing derivation. We break up our cases based on the last rule applied.

[T-Cond] Then $e = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$, $\Delta = \Delta_1, \Delta'$, $\Gamma; \Delta' \vdash e_1 : \mathsf{Bool}$, $\Gamma; \Delta' \vdash e_2 : \tau$, and $\Gamma; \Delta' \vdash e_3 : \tau$. By induction, for all $b_1 \in B(e_1)$, $\Gamma; \Delta_1 \vdash b_1 : \mathsf{Bool}$ and for all $b \in B(e_2) \cup B(e_3)$, $\Gamma; \Delta' \vdash b : \tau$. Then for any $b_1 \in B(e_1)$ and $b_2 \in B(e_2)$, [T-Assert] nets $\Gamma; \Delta_1, \Delta' \vdash \mathsf{assert}\ b_1\ \mathsf{in}\ b_2 : \tau$. Since $\Gamma; \Delta_1 \vdash \neg b_1 : \mathsf{Bool}$ when $\Gamma; \Delta_1 \vdash b_1 : \mathsf{Bool}$, [T-Assert] also nets $\Gamma; \Delta_1, \Delta' \vdash \mathsf{assert}\ \neg b_1\ \mathsf{in}\ b_3 : \tau$. This means that for all $b \in B(e)$, $\Gamma; \Delta_1, \Delta' \vdash b : \tau$, concluding this case.

The remaining cases follow quickly from applying the inductive hypothesis.

($\Leftarrow$)

Suppose $\Gamma; \Delta \vdash b : \tau$ for all $b \in B(e)$. We proceed by induction on the structure of e.

[$e = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$] Then $b = \mathsf{assert}\ b_1\ \mathsf{in}\ b_2$ where $b_1 \in B(e_1)$, $b_2 \in B(e_2)$ or $b = \mathsf{assert}\ \neg b_1\ \mathsf{in}\ b_3$ where $b_1 \in B(e_1)$, $b_3 \in B(e_3)$. It also must be that $\Delta = \Delta_1, \Delta'$, and $\Gamma; \Delta_1 \vdash b_1 : \mathsf{Bool}$, $\Gamma; \Delta' \vdash b_2 : \tau$, and $\Gamma; \Delta' \vdash b_3 : \tau$ for any $b_1 \in B(e_1)$,$b_2 \in B(e_2)$, and $b_3 \in B(e_3)$. By induction, $\Gamma; \Delta_1 \vdash e_1 : \mathsf{Bool}$, $\Gamma; \Delta' \vdash e_2 : \tau$, and $\Gamma; \Delta' \vdash e_3 : \tau$. This means we can apply [T-Cond] to obtain $\Gamma; \Delta_1, \Delta' \vdash e : \tau$.

All other cases are routine applications of induction. □

Proposition 9. *$e \Downarrow v$ if and only if $b \Downarrow v$ for some $b \in B(e)$.*

Proof. ($\Rightarrow$)
We proceed by induction on the derivation of $e \Downarrow v$. Cases are broken up based on the last rule applied.

[E-True] Then $e = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$, $e_1 \Downarrow \mathsf{true}$ and $e_2 \Downarrow v$. By induction, there exists $b_1 \in B(e_1)$, $b_2 \in B(e_2)$ such that $b_1 \Downarrow \mathsf{true}$ and $b_2 \Downarrow v$. Then [E-Assert] nets $\mathsf{assert}\ b_1\ \mathsf{in}\ b_2 \Downarrow v$. Since $\mathsf{assert}\ b_1\ \mathsf{in}\ b_2 \in B(e)$, which means we are done with this case.

[E-False] This case is similar to that of [E-True].

The remaining cases follow quickly from applying the inductive hypothesis.

($\Leftarrow$)

We proceed by induction on the derivation of $b \Downarrow v$. We break up cases by the last rule applied.

[E-Assert] Then $b = \mathsf{assert}\ b_1\ \mathsf{in}\ b_2$, $b_1 \Downarrow \mathsf{true}$, $b_2 \Downarrow v$ as well as $e = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. Since $b \in B(e)$, $b_1 \in B(e_1)$ and and $b_2 \in B(e_2)$ by definition of $B(e)$. By induction, $e_1 \Downarrow \mathsf{true}$ and $e_2 \Downarrow v$. Applying [E-True] nets $e \Downarrow v$, which is the desired result.

[E-False] This is similar to [E-True].

The remaining cases follow quickly from applying the inductive hypothesis. □

7.7 Logical syntax

We use a multi-sorted first order logic, given by the signature $(\mathtt{S}, \mathtt{F}, \mathtt{C}, \mathtt{Q})$, with:

- Sorts $\mathtt{S}$: $\mathtt{Real}, \mathtt{Bool}, \mathtt{Vltn}, \mathtt{Point}, \mathtt{Intvl}, \mathtt{s}_1 \times \cdots \times \mathtt{s}_n$
- Predicate symbols $\mathtt{Q}$: $\{\neg, \geq, =\}$
- Constants $\mathtt{C}$: $\{r\#\mathsf{Pt} \mid r \in \mathbb{R}\} \cup \{\mathtt{cake}\}$
- Function symbols $\mathtt{F}$: $\mathtt{O} \cup \{(\_, \ldots, \_)_k, \pi_k \mid k \in \mathbb{N}\} \cup \{\mathtt{l}, \mathtt{r}, [\_, \_], \cup\} \cup \{\mathtt{V}_a \mid a \in \mathcal{A}\} \cup \{+, \cdot, *\}$

For each type τ, there is an associated sort $\mathtt{s}_\tau$ given in Figure 10. Above, the set $\mathtt{O} = \{\mathtt{o} : \mathtt{s}_{\hat{\tau}_1}, \ldots, \mathtt{s}_{\hat{\tau}_n} \to \mathtt{s}_{\hat{\tau}} \mid o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau} \in \mathcal{O}\}$, consists of logical function symbols corresponding to program operations. The function symbols $\mathtt{l}$ and $\mathtt{r}$ give the left and right endpoints of intervals respectively, and $[\_, \_]$ is an interval constructor. The unary operator $*$ takes a point to its corresponding real, and the binary symbols $+$ and $\cdot$ give arithmetic on real numbers. Using the function symbols and constants, we can encode any program value v as a logical term $\mathtt{v}$.

$$\mathtt{s}_{\mathsf{Real}} = \mathtt{Real} \qquad \mathtt{s}_{\mathsf{Bool}} = \mathtt{Bool} \qquad \mathtt{s}_{\mathsf{Point}} = \mathtt{Point} \qquad \mathtt{s}_{\mathsf{Intvl}} = \mathtt{Intvl} \qquad \mathtt{s}_{\mathsf{Piece}} = \mathtt{Piece} \qquad \mathtt{s}_{\mathsf{Vltn}} = \mathtt{Vltn} \qquad \mathtt{s}_{\overline{\mathsf{Intvl}}} = \mathtt{Intvl}$$

$$\mathtt{s}_{\overline{\mathsf{Piece}}} = \mathtt{Piece} \qquad s_{\tau_1 \times \cdots \times \tau_n} = s_{\tau_1} \times \cdots \times s_{\tau_n}$$

Fig. 10: Sort associated with each type

All logical variables are drawn from $\mathcal{X}$, $\mathcal{W}$, $\overline{\mathcal{W}}$, and $\mathcal{Y}$. The first three sets are the same sets used for program variables, and $\mathcal{Y}$ is a new countably infinite set of logical variables.

We use a standard sorting judgment for well-formed terms ($\mathcal{E}; \Upsilon \vdash t : \mathtt{s}$) and well-formed formulas ($\mathcal{E}; \Upsilon \vdash \psi :$ Formula). In the sorting judgments, $\mathcal{E}$ is a partial map from $\mathcal{X} \cup \mathcal{W} \cup \overline{\mathcal{W}}$ to $\mathtt{S}$, and Υ is a finite subset of $\mathcal{Y}$. $\mathcal{E}$ handles variables that arise in intermediate stages of translating expressions to their constraints, and Υ handles variables created in the constraint generation procedure. Every variable in $\mathcal{Y}$ is assumed to be a point, so Υ is a set and not a partial map. The sorting rules for terms and predicate formulas are given in Section 7.7.

Proposition 11 is proven under the assumption of the following operations.

$$\frac{}{\mathcal{E}, x : \mathtt{s}; \mathcal{Y} \vdash x : \mathtt{s}} \qquad \frac{}{\mathcal{E}; \mathcal{Y}, y \vdash y : \mathtt{Point}} \qquad \frac{}{\mathcal{E}; \mathcal{Y} \vdash r : \mathtt{Real}} \qquad \frac{}{\mathcal{E}; \mathcal{Y} \vdash r\#\mathsf{Pt} : \mathtt{Point}} \qquad \frac{}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{cake} : \mathtt{Piece}}$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{s}_1 \ \cdots \ \mathcal{E}; \mathcal{Y} \vdash t_n : \mathtt{s}_n \qquad \mathtt{o} : \mathtt{s}_1, \ldots, \mathtt{s}_n \to \mathtt{s}}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{o}(t_1, \ldots, t_n) : \mathtt{s}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t : \mathtt{Intvl}}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{l}(t) : \mathtt{Point}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t : \mathtt{Intvl}}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{r}(t) : \mathtt{Point}}$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t : \mathtt{Intvl}}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{V}_a(t) : \mathtt{Vltn}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t : \mathtt{Piece}}{\mathcal{E}; \mathcal{Y} \vdash \mathtt{V}_a(t) : \mathtt{Vltn}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Point} \qquad \mathcal{E}; \mathcal{Y} \vdash t_2 : \mathtt{Point}}{\mathcal{E}; \mathcal{Y} \vdash [t_1, t_2] : \mathtt{Intvl}}$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Intvl} \qquad \cdots \qquad \mathcal{E}; \mathcal{Y} \vdash t_n : \mathtt{Intvl}}{\mathcal{E}; \mathcal{Y} \vdash \cup t_1, \ldots, t_n : \mathtt{Piece}}$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{s}_1 \qquad \cdots \qquad \mathcal{E}; \mathcal{Y} \vdash t_k : \mathtt{s}_k}{\mathcal{E}; \mathcal{Y} \vdash (t_1, \ldots, t_k) : \mathtt{s}_1 \times \cdots \times \mathtt{s}_k} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t : \mathtt{s}_1 \times \cdots \times \mathtt{s}_k}{\mathcal{E}; \mathcal{Y} \vdash \pi_n t : \mathtt{s}_n} \ (1 \leq n \leq k)$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Bool}}{\mathcal{E}; \mathcal{Y} \vdash t_1 = \mathtt{true} : \text{Formula}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Real} \qquad \mathcal{E}; \mathcal{Y} \vdash t_2 : \mathtt{Real}}{\mathcal{E}; \mathcal{Y} \vdash t_1 \geqq t_2 : \text{Formula}} \qquad \frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Point} \qquad \mathcal{E}; \mathcal{Y} \vdash t_2 : \mathtt{Point}}{\mathcal{E}; \mathcal{Y} \vdash t_1 \geqq t_2 : \text{Formula}}$$

$$\frac{\mathcal{E}; \mathcal{Y} \vdash t_1 : \mathtt{Vltn} \qquad \mathcal{E}; \mathcal{Y} \vdash t_2 : \mathtt{Vltn}}{\mathcal{E}; \mathcal{Y} \vdash t_1 \geqq t_2 : \text{Formula}}$$

Fig. 11: Rules for sorting terms and well-formed formulas.

$$\wedge : \mathsf{Bool}, \mathsf{Bool} \to \mathsf{Bool} \quad \vee : \mathsf{Bool}, \mathsf{Bool} \to \mathsf{Bool} \quad \neg : \mathsf{Bool} \to \mathsf{Bool} \qquad + : \mathsf{Vltn}, \mathsf{Vltn} \to \mathsf{Vltn}$$

$$\geq : \mathsf{Real}, \mathsf{Real} \to \mathsf{Bool} \quad \geq : \mathsf{Vltn}, \mathsf{Vltn} \to \mathsf{Bool} \quad = : \mathsf{Vltn}, \mathsf{Vltn} \to \mathsf{Bool} \quad \cdot_r : \mathsf{Vltn} \to \mathsf{Vltn}$$

Fig. 12: The operations contained in $\mathcal{O}$.

Basic simplification Here we discuss basic formula simplification. Given a term t, we define the simplification $\mathtt{R}(t)$ inductively as follows:

$$\mathtt{R}(\pi_k(t)) \triangleq \begin{cases} t_k & \text{if } \mathtt{R}(t) = (t_1, \ldots, t_k, \ldots, t_n) \\ \pi_k(\mathtt{R}(t)) & \text{otherwise} \end{cases}$$

$$\mathtt{R}(\mathtt{l}(t)) \triangleq \begin{cases} t_1 & \text{if } \mathtt{R}(t) = [t_1, t_2] \\ \mathtt{l}(\mathtt{R}(t)) & \text{otherwise.} \end{cases}$$

$$\mathtt{R}(\mathtt{r}(t)) \triangleq \begin{cases} t_2 & \text{if } \mathtt{R}(t) = [t_1, t_2] \\ \mathtt{r}(\mathtt{R}(t)) & \text{otherwise.} \end{cases}$$

$$\mathtt{R}(f(t_1, \ldots, t_n)) \triangleq f(\mathtt{R}(t_1), \ldots, \mathtt{R}(t_n)) \text{ for } t = f(t_1, \ldots, t_n) \text{ otherwise.}$$

The goal of this simplification is to reduce formulas to real linear inequalities and equalities in $\mathtt{V}_a$ evaluated on pieces or intervals. To argue that this is the case, we define the following logical predicate on terms indexed by sort:

$$\begin{aligned}
R_{\mathtt{Bool}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Bool}, t = t_1 \wedge t_2 \text{ and } t_1, t_2 \in \mathtt{Bool} \text{ or } t = t_1 \vee t_2 \text{ and } t_1, t_2 \in \mathtt{Bool} \text{ or} \\
&\quad t = t_1 \geqq t_2 \text{ and } t_1, t_2 \in R_\tau, \tau = \mathtt{Real}, \mathtt{Vltn}\} \\
R_{\mathtt{Real}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Real}, t = t_1 + t_2 \text{ and } t_1, t_2 \in R_{\mathtt{Real}} \text{ or} \\
&\quad t = r \cdot t' \text{ and } t' \in R_{\mathtt{Real}} \text{ or } t = t'^*, t' \in R_{\mathtt{Point}} \text{ or } t = r\} \\
R_{\mathtt{Point}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Point}, t = r\#\mathsf{Pt} \text{ or } t \in \mathcal{Y}\} \\
R_{\mathtt{Vltn}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Vltn}, t = \sum_i t_i \text{ and } t_i \in R_{\mathtt{Vltn}}, \text{ or } t = r \cdot t' \text{ and } t' \in R_{\mathtt{Vltn}}, \\
&\quad \text{or } t = \mathtt{V}_a(t') \text{ and } t' \in R_{\mathtt{Intvl}} \cup R_{\mathtt{Piece}}\} \\
R_{\mathtt{Intvl}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Intvl}, t = [t_1, t_2], t_1, t_2 \in R_{\mathtt{Real}}\} \\
R_{\mathtt{Piece}} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{Piece}, t = \cup t_1 \cdots t_n, t_i \in R_{\mathtt{Intvl}}\} \\
R_{\mathtt{s}_1 \times \cdots \times \mathtt{s}_k} &= \{t \mid \cdot; \mathcal{Y} \vdash t : \mathtt{s}_1 \times \cdots \times \mathtt{s}_k, t = (t_1, \ldots, t_k), t_i \in R_{\mathtt{s}_i}\}
\end{aligned}$$

Proposition 10. *If* $\cdot; \mathcal{Y} \vdash t : \mathtt{s}$ *then* $\mathtt{R}(t) \in R_{\mathtt{s}}$.

Proof. This proceeds by induction on the sorting derivation. We break up cases based on the last rule used to sort t. We expand out the ops rule now that we've fixed the possible operations. Here we write $\geqq$ as short-hand for both $\geq$ and $=$ symbols.

$$\frac{}{\cdot; \mathcal{Y} \vdash r : \mathtt{Real}} \qquad \frac{}{\cdot; \mathcal{Y} \vdash b : \mathtt{Bool}} \qquad \frac{}{\cdot; \mathcal{Y} \vdash r\#\mathsf{Pt} : \mathtt{Point}} \qquad \frac{}{\cdot; \mathcal{Y}, y : \mathtt{Point} \vdash y : \mathtt{Point}}$$

These are obvious.

$$\frac{\cdot; \mathcal{Y} \vdash t_1 : \mathtt{Real} \qquad \cdot; \mathcal{Y} \vdash t_2 : \mathtt{Real}}{t_1 \geqq t_2 : \mathtt{Bool}} \qquad \frac{\cdot; \mathcal{Y} \vdash t_1 : \mathtt{Vltn} \qquad \cdot; \mathcal{Y} \vdash t_2 : \mathtt{Vltn}}{\cdot; \mathcal{Y} \vdash t_1 \geqq t_2 : \mathtt{Bool}} \qquad \frac{\cdot; \mathcal{Y} \vdash t_1 : \mathtt{Bool} \qquad \cdot; \mathcal{Y} \vdash t_2 : \mathtt{Bool}}{\cdot; \mathcal{Y} \vdash t_1 \wedge t_2 : \mathtt{Bool}}$$

$$\frac{\cdot; \mathcal{Y} \vdash t_1 : \mathtt{Bool} \qquad \cdot; \mathcal{Y} \vdash t_2 : \mathtt{Bool}}{\cdot; \mathcal{Y} \vdash t_1 \vee t_2 : \mathtt{Bool}} \qquad \frac{\cdot; \mathcal{Y} \vdash t : \mathtt{Bool}}{\cdot; \mathcal{Y} \vdash \neg t : \mathtt{Bool}}$$

These all readily follow by induction.

$$\frac{\cdot;\Upsilon \vdash t_1 : \texttt{Point} \qquad \cdot;\Upsilon \vdash t_2 : \texttt{Point}}{\cdot;\Upsilon \vdash [t_1, t_2] : \texttt{Intvl}} \qquad \frac{\cdot;\Upsilon \vdash t_1 : \texttt{Intvl} \qquad \cdots \qquad \cdot;\Upsilon \vdash t_n : \texttt{Intvl}}{\cdot;\Upsilon \vdash \cup t_1, \ldots, t_n : \texttt{Piece}}$$

$$\frac{\cdot;\Upsilon \vdash t_1 : \texttt{Real} \qquad \cdot;\Upsilon \vdash t_2 : \texttt{Real}}{\cdot;\Upsilon \vdash t_1 + t_2 : \texttt{Real}} \qquad \frac{\cdot;\Upsilon \vdash t_1 : \texttt{Vltn} \qquad \cdot;\Upsilon \vdash t_2 : \texttt{Vltn}}{\cdot;\Upsilon \vdash t_1 + t_2 : \texttt{Vltn}} \qquad \frac{\cdot;\Upsilon \vdash t : \texttt{Real}}{\cdot;\Upsilon \vdash r \cdot t : \texttt{Real}}$$

$$\frac{\cdot;\Upsilon \vdash t : \texttt{Vltn}}{\cdot;\Upsilon \vdash r \cdot t : \texttt{Vltn}} \qquad \frac{\cdot;\Upsilon \vdash t : \texttt{Point}}{\cdot;\Upsilon \vdash t^* : \texttt{Real}}$$

These arguments are straightforward applications of the inductive hypothesis.

$$\frac{\cdot;\Upsilon \vdash t' : \texttt{Intvl}}{\cdot;\Upsilon \vdash \texttt{l}(t') : \texttt{Point}} \qquad \frac{\cdot;\Upsilon \vdash t' : \texttt{Intvl}}{\cdot;\Upsilon \vdash \texttt{r}(t') : \texttt{Point}}$$

By induction $\texttt{s}(t) \in R_{\texttt{Intvl}}$ so $\texttt{s}(t) = [t_1, t_2]$ where $t_1, t_2 \in R_{\texttt{Point}}$. This means $\texttt{s}(\ell(t)) = t_1$. Similarly, $\texttt{s}(r(t)) = t_2$. As $t_1, t_2 \in R_{\texttt{Point}}$, we are done with these cases.

$$\frac{\cdot;\Upsilon \vdash t' : \texttt{Intvl}}{\cdot;\Upsilon \vdash \texttt{V}_a(t') : \texttt{Vltn}}$$

Then $\texttt{R}(t) = \texttt{V}_a(\texttt{R}(t'))$. By induction, $\texttt{R}(t') \in R_{\texttt{Intvl}}$, completing this case.

$$\frac{\cdot;\Upsilon \vdash t' : \texttt{Piece}}{\cdot;\Upsilon \vdash \texttt{V}_a(t') : \texttt{Vltn}}$$

Almost identical to the previous case.

$$\frac{\cdot;\Upsilon \vdash t_1 : \texttt{s}_1 \qquad \cdots \qquad \cdot;\Upsilon \vdash t_n : \texttt{s}_n}{\cdot;\Upsilon \vdash (t_1, \ldots, t_n) : \texttt{s}_1 \times \cdots \times \texttt{s}_n}$$

By induction, $\texttt{R}(t_i) \in R_{\texttt{s}_i}$. Since $\texttt{R}(t_1, \ldots, t_n) = (\texttt{R}(t_1), \ldots, \texttt{R}(t_n))$, we are done with this case.

$$\frac{\cdot;\Upsilon \vdash t' : \texttt{s}_1 \times \cdots \times \texttt{s}_n}{\cdot;\Upsilon \vdash \pi_k t' : \texttt{s}_k}\ (1 \leq k \leq n)$$

By induction, $\texttt{R}(t') \in R_{\texttt{s}_1 \times \cdots \times \texttt{s}_k}$ which means $\texttt{R}(t') = (t_1, \ldots, t_k)$ where $t_i \in R_{\texttt{s}_i}$. Therefore $\texttt{R}(t) = t_k \in R_{\texttt{s}_k}$. □

If $t \in R_{\mathsf{Bool}}$, there is a clear formula it can be converted to, which we denote $F(t)$. From this, we can extend $\texttt{R}$ to formulas. Given a well-sorted formula f, we inductively define $\texttt{R}(f)$ as follows

$$\begin{aligned}
\texttt{R}(t = \mathsf{true}) &\triangleq F(\texttt{R}(t)) \\
\texttt{R}(t_1 \geq t_2) &\triangleq \texttt{R}(t_1) \geq \texttt{R}(t_2) \\
\texttt{R}(t_1 = t_2) &\triangleq \texttt{R}(t_1) = \texttt{R}(t_2) \text{ if } t_2 \neq \mathsf{true} \\
\texttt{R}(\neg f) &\triangleq \neg(\texttt{R}(f)) \\
\texttt{R}(f_1 \wedge f_2) &\triangleq \texttt{R}(f_1) \wedge \texttt{R}(f_2) \\
\texttt{R}(f_1 \vee f_2) &\triangleq \texttt{R}(f_1) \vee \texttt{R}(f_2) \\
\texttt{R}(f_1 \Rightarrow f_2) &\triangleq \texttt{R}(f_1) \Rightarrow \texttt{R}(f_2)
\end{aligned}$$

We say that a formula is *simplified* if for each term contained in it, it is contained within $R_{\mathtt{s}}$ for some sort $\mathtt{s}$. By Proposition 10, $\mathtt{R}(f)$ is simplified for any well-sorted formula f. Based on our assumptions on predicate symbols, we have the following corollary.

Corollary 2. *Suppose that* $\cdot; \Upsilon \vdash f : \mathit{Formula}$. *Then* $\mathtt{R}(f)$ *consists entirely of conjunctions, disjunctions, and negations of the forms:*

$$\sum_i r_i \mathtt{V}_a(P_i) \geqq \sum_j r_j \mathtt{V}_{a'}(P_j)$$

where $P_i, P_j \in R_{\mathtt{Intvl}} \cup R_{\mathtt{Point}}$ *and*

$$\sum_i r_i t_i \geqq \sum_j r_j t_j$$

where $t_i, t_j \in \{t \mid t = t'^*, t' \in R_{\mathtt{Point}}\} \cup \mathbb{R}$ *and* $\geqq$ *stands for* $\geq$ *and* $=$.

In particular, $c(b)$ for any branch $\cdot \vdash b : \tau$ satisfies the corollary's conditions.

7.8 Logical semantics

We give semantics to formulas through means of an interpretation, $\mathcal{A}$, and variable assignment μ. An interpretation associates with each sort $\mathtt{s}$, a set $[\![\mathtt{s}]\!]_{\mathcal{A}}$, and with each function symbol f a function $[\![f]\!]_{\mathcal{A}}$. All sorts are interpreted as shown in Figure 13 regardless of $\mathcal{A}$ and all symbols are interpreted as shown in Figure 14 *besides* $\mathtt{V}_a$ for each a– these are interpreted as valuations determined by the specific interpretation. As such, interpretations are completely determined by the valuations they use to interpret. Thus for a valuation set $\overline{V}$, we let $\mathcal{A}_{\overline{V}}$ be the interpretation such that $[\![\mathtt{V}_a]\!]_{\mathcal{A}_{\overline{V}}} = \overline{V}\, a$ for all $a \in \mathbb{A}$. Before moving on we describe the function symbol interpretations in words. The symbol `cake` is the whole cake, `l` and `r` provide the left and right endpoints of an interval respectively, $\cup$ forms a piece from intervals. Recall that for each $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau} \in O$, there is $[\![o]\!] : |[\![\hat{\tau}_1]\!]| \times \cdots \times |[\![\hat{\tau}_n]\!]| \to |[\![\hat{\tau}]\!]|$.

Now a variable assignment is a map from variables to elements of the interpreted sorts. Given a term t, we let $[\![t]\!]^{\mu}_{\mathcal{A}}$ denote the interpretation of t according to $\mathcal{A}$, with variable values determined by μ, defined in the usual way. Likewise, given a formula f, we write $\mathcal{A}, \mu \vDash f$ if f is true when interpreted through $\mathcal{A}$ with variable values determined by μ, also defined in the usual way. We write $\mathcal{A} \vDash f$ if for all assignments μ we have $\mathcal{A}, \mu \vDash f$. If t is a term containing no $\mathtt{V}_a$ symbols, then for a fixed assignment μ, the term t is always interpreted the same way and we write just $[\![t]\!]^{\mu}$.

$$[\![\mathtt{Bool}]\!]_{\mathcal{A}} \triangleq \{\mathtt{true}, \mathtt{false}\} \qquad [\![\mathtt{Num}]\!]_{\mathcal{A}} \triangleq \mathbb{N} \qquad [\![\mathtt{Real}]\!]_{\mathcal{A}} \triangleq \mathbb{R} \qquad [\![\mathtt{Vltn}]\!]_{\mathcal{A}} \triangleq \mathbb{V} \qquad [\![\mathtt{Point}]\!]_{\mathcal{A}} \triangleq \mathtt{PT} \qquad [\![\mathtt{Intvl}]\!]_{\mathcal{A}} \triangleq \mathbb{I}$$

$$[\![\mathtt{Piece}]\!]_{\mathcal{A}} \triangleq \mathbb{PC}$$

Fig. 13: Interpretations of each sort.

We close off this section with the soundess of the previous section's formula simplification with respect to the semantics given here.

Proposition 11. *Let* $\mathcal{A}$ *be an interpretation and* μ *be a variable assignment. Let* φ *be any formula. Then* $\mathcal{A}, \mu \vDash \varphi \iff \mathcal{A}, \mu \vDash \mathtt{R}(\varphi)$.

$$[\![\mathtt{o}]\!]_{\mathcal{A}} = [\![o]\!] \qquad [\![\pi_i]\!]_{\mathcal{A}}(v_1, \ldots, v_i, \ldots, v_n) = v_i \qquad [\![(\_, \ldots, \_)_k]\!]_{\mathcal{A}}(v_1, \ldots, v_k) = (v_1, \ldots, v_k) \qquad [\![\mathtt{l}]\!]_{\mathcal{A}}([r, r']) = r$$

$$[\![\mathtt{r}]\!]_{\mathcal{A}}([r, r']) = r' \qquad [\![\mathsf{cake}]\!]_{\mathcal{A}} = [0, 1] \qquad [\![\cup]\!]_{\mathcal{A}}([r_1, r_1'], \ldots, [r_n, r_n']) = P_{i=1}^{n}[r_i, r_i']$$

Fig. 14: Interpretations of each function symbol.

7.9 Constraints

This section argues for soundness and completeness of the constraints.

We require each mark to have a unique identifier, $\#s$, to prevent variable clashes when translating expressions to their constraints:

$$\mathsf{mark}_a(e_1, e_2)\#s.$$

For each identifier in the program, $\#s$, we assume there is a unique member of $\mathcal{Y}$, denoted $y_{\#s}$, corresponding to it. We also let $\mathrm{Id}(b)$ denote the set of indentifiers in b. The complete definition for $\rho(b)$ is given here:

$$\rho(\overline{v}) = \mathtt{v} \qquad \rho(v) = \mathtt{v} \qquad \rho(x) = x \qquad \rho(w) = w \qquad \rho(\overline{w}) = \overline{w} \qquad \rho(\mathsf{cake}) \triangleq [0, 1]$$

$$\rho(\mathsf{divide}(b_1, b_2)) = ([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))]) \qquad \rho(\mathsf{mark}_a(b_1, b_2)\#s) = y_{\#s} \qquad \rho(\mathsf{eval}_a(b)) = \mathtt{V}_a(\rho(b))$$

$$\rho(\mathsf{assert}\ b_1\ \mathsf{in}\ b_2) = \rho(b_2) \qquad \rho(o(b_1, \ldots, b_n)) = o(\rho(b_1), \ldots, \rho(b_n)) \qquad \rho(\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ b_1\ \mathsf{in}\ b_2) \triangleq$$

$$\rho(b_2)\{\pi_i\rho(b_1) \mapsto x_i \mid 1 \le i \le n\}\{\pi_i\rho(b_1) \mapsto w_i, \pi_i\rho(b_1) \mapsto \overline{w_i} \mid 1 \le i \le n'\}$$

$$\rho(\mathsf{piece}(b_1, \ldots, b_n)) = \cup\rho(b_1), \ldots, \rho(b_n)$$

$$\rho((b_1, \ldots, b_n)) \triangleq (\rho(b_1), \ldots, \rho(b_n))$$

Here is the complete definition for $c(b)$:

$$c(\mathsf{divide}(b_1, b_2)) \triangleq c(b_1) \wedge c(b_2) \wedge \mathtt{l}(\rho(b_1)) \le \rho(b_2) \le \mathtt{r}(\rho(b_1))$$

$$c(\mathsf{mark}_a(b_1, b_2)\#s) = c(b_1) \wedge c(b_2) \wedge (\mathtt{V}_a([\mathtt{l}(\rho(b_1)), \rho(\mathsf{mark}_a(b_1, b_2)\#s)]) = \rho(b_2)) \qquad c(\mathsf{eval}_a(b)) = c(b)$$

$$c(\mathsf{assert}\ b_1\ \mathsf{in}\ b_2) = (\rho(b_1) = \mathsf{true}) \wedge c(b_1) \wedge c(b_2) \qquad c(\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ b_1\ \mathsf{in}\ b_2) \triangleq c(b_1) \wedge$$

$$c(b_2)\{x_i \mapsto \pi_i\rho(b_1)) \mid 1 \le i \le n\}\{w_i \mapsto \pi_{i+n}\rho(b_1), \overline{w_i} \mapsto \pi_{i+n}\rho(b_1) \mid 1 \le i \le n'\}$$

$$c((b_1, \ldots, b_n)) \triangleq c(b_1) \wedge \cdots \wedge c(b_n) \qquad c(\mathsf{piece}(b_1, \ldots, b_n)) \triangleq c(b_1) \wedge \cdots \wedge c(b_n)$$

Given a substitution from program variables variables to values S, let $|S|$ be the logical substitution $\{x \mapsto \rho(v) \mid S(x) = v\}$. Recall that ρ converts values to their logical counterparts, and makes a read only value a regular value.

Lemma 9. *Let b be a branch and S a substitution of values for program variables. Then $\rho(S(b)) = |S|(\rho(b))$, and $c(S(b)) = |S|(c(b))$.*

Proof. Once the first statement is established the second follows trivially. The first is argued by induction on the structure of b. We exhibit the interesting cases, and the rest follow easily.

[$b = x$] If x is not a variable in S then this is trivial, so suppose it is. Then we have

$$\rho(S(b)) = \rho(S(x)) = \rho(v) = |S|(x) = |S|(\rho(x)) = |S|(\rho(b)).$$

$c(x) = c(v)$ for any variable and value so clearly $|S|(c(b)) = c(S(b))$.

[$b = w$] This is identical to the above case.

[$b = \overline{w}$] If w is not a variable in S, then this is trivial, so suppose it is. Then we have

$$\rho(S(b)) = \rho(S(\overline{w})) = \rho(\overline{v}) = v = |S|(\overline{w}) = |S|(\rho(\overline{w})) = |S|(\rho(b)).$$

$c(\overline{w}) = c(\overline{v})$ for any variable and value so clearly $S(c(b)) = c(S(b))$.

[$b = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ b_1\ \mathsf{in}\ b_2$] Let

$$S' = S \setminus \{x_1, \ldots, x_n, w_1, \ldots, w_{n'}, \overline{w_1}, \ldots, \overline{w_n}\}.$$

We have that

$$S(b) = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ S(b_1)\ \mathsf{in}\ S'(b_2),$$

and also have

$$\rho(b) = \rho(b_2)\{x_i \mapsto \pi_i\rho(b_1) \mid 1 \le i \le n\}\{w_i \mapsto \pi_{i+n}\rho(b_1), \overline{w_i} \mapsto \pi_{i+n}\rho(b_1) \mid 1 \le i \le n'\}.$$

By induction,

$$\rho(S(b_1)) = |S|(\rho(b_1)) \quad \text{and} \quad \rho(S'(b_2)) = |S'|(\rho(b_2)).$$

We can then calculate:

$$\begin{aligned}\rho(S(b)) &= \rho(S'(b_2))\{x_i \mapsto \pi_i\rho(S(b_1)) \mid 1 \le i \le n\}\{w_i \mapsto \pi_{i+n}\rho(S(b_1)), \overline{w_i} \mapsto \pi_{i+n}\rho(S(b_1)) \mid 1 \le i \le n'\} \\ &= |S'|(\rho(b_2))\{x_i \mapsto |S|(\pi_i\rho(b_1)) \mid 1 \le i \le n\} \\ &\quad \{w_i \mapsto |S|(\pi_{i+n}\rho(b_1)), \overline{w_i} \mapsto |S|(\pi_{i+n}\rho(b_1)) \mid 1 \le i \le n'\}\end{aligned}$$

and

$$\begin{aligned}|S|(\rho(b)) &= |S|(\rho(b_2)\{x_i \mapsto \pi_i\rho(b_1) \mid 1 \le i \le n\}\{w_i \mapsto \pi_{i+n}\rho(b_1), \overline{w_i} \mapsto \pi_{i+n}\rho(b_1) \mid 1 \le i \le n'\}) \\ &= |S'|(\rho(b_2))\{x_i \mapsto |S|(\pi_i\rho(b_1)) \mid 1 \le i \le n\} \\ &\quad \{w_i \mapsto |S|(\pi_{i+n}\rho(b_1)), \overline{w_i} \mapsto |S|(\pi_{i+n}\rho(b_1)) \mid 1 \le i \le n'\}.\end{aligned}$$

Therefore $\rho(S(b)) = |S|(\rho(b))$. The argument for $c(S(b)) = |S|(c(b))$ in this case is very similar. □

Given a term t, let $\mathtt{FV}_{\mathcal{Y}}(t)$ be the set of free variables contained in t from $\mathcal{Y}$. Analogously for a formula f, let $\mathtt{FV}_{\mathcal{Y}}(f)$ be the set of free variables contained in f from $\mathcal{Y}$. Also let $\mathcal{Y}_b = \mathtt{FV}_{\mathcal{Y}}(c(b)) \cup \mathtt{FV}_{\mathcal{Y}}(\rho(b))$. We write $\mathcal{E} \vdash \Gamma; \Delta$ if for all $x \in \mathrm{dom}(\Gamma)$, $\mathcal{E}(x) = \mathtt{s}_{\Gamma(x)}$, and for all $w \in \mathrm{dom}(\Delta)$, $\mathcal{E}(w) = \mathtt{s}_{\Delta(w)}$.

Proposition 12. *Suppose $\Gamma; \Delta \vdash b : \tau$. Then for any $\mathcal{E}$ such that $\mathcal{E} \vdash \Gamma; \Delta$ and any Υ such that $\mathtt{FV}_{\mathcal{Y}}(b) \subseteq \Upsilon$,*

$$\mathcal{E}; \Upsilon \vdash \rho(b) : \tau \quad \textit{and} \quad \mathcal{E}; \Upsilon \vdash c(b) : \textit{Formula}.$$

Proof. This proceeds by induction on the typing derivation. Let $\mathcal{E}$ be such that $\mathcal{E} \vdash \Gamma; \Delta$ and let Υ be such that $\mathtt{FV}_{\mathcal{Y}}(b) \subseteq \Upsilon$.

[T-Vltn] This is immediate.

[T-Split] Then $b = (\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ b_1\ \mathsf{in}\ b_2)$, and necessarily that $\Gamma; \Delta \vdash b_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'}$, and

$$\Gamma, x_1 : \tau_1, \ldots, x_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}}; \Delta, w_1 : \tau_1, \ldots, w_n : \tau_{n'} \vdash b_2 : \tau.$$

We also have

$$\rho(b) = \rho(b_2)\{x_i \mapsto \pi_i \rho(b_1) \mid 1 \leq i \leq n\}\{w_i \mapsto \pi_{n+i}\rho(b_1), \overline{w_i} \mapsto \pi_{n+i}\rho(b_1) \mid 1 \leq i \leq n'\}.$$

Set

$$\begin{aligned} S_{\mathcal{X}} &= \{x_i \mapsto \pi_i \rho(b_1) \mid 1 \leq i \leq n\} \\ S_{\mathcal{W}} &= \{w_i \mapsto \pi_{n+i}\rho(b_1) \mid 1 \leq i \leq n'\} \\ S_{\overline{\mathcal{W}}} &= \{\overline{w_i} \mapsto \pi_{n+i}\rho(b_1) \mid 1 \leq i \leq n'\}. \end{aligned}$$

In this case, $\rho(b) = S_{\mathcal{X}} S_{\mathcal{W}} S_{\overline{\mathcal{W}}}(\rho(b_2))$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b_1) : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'}$. By the projection sorting rule, $\mathcal{E}; \Upsilon \vdash \pi_i \rho(b_1) : \hat{\tau}_i$ for $1 \leq i \leq n$ and $\mathcal{E}; \Upsilon \vdash \pi_{n+i}\rho(b_1) : \tau_i$ for $1 \leq i \leq n'$.
Let

$$\mathcal{E}' = \mathcal{E}, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \tau_1, \ldots, \overline{w_{n'}} : \tau_{n'}, w_1 : \tau_1, \ldots, w_n : \tau_{n'}.$$

Then

$$\mathcal{E}' \vdash \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}}; \Delta, w_1 : \tau_1, \ldots, w_n : \tau_{n'}.$$

As $\mathrm{dom}(S_x) \cup \mathrm{dom}(S_w)$ does not contain any variables from $\mathcal{Y}$, we also have $\mathtt{FV}_{\mathcal{Y}}(b_2)$. By induction, $\mathcal{E}'; \Upsilon \vdash \rho(b_2) : \tau$. Because

$$\begin{aligned} \mathrm{dom}(S_{\mathcal{X}}) &= \{x_1, \ldots x_n\} \\ \mathrm{dom}(S_{\mathcal{W}}) &= \{w_1, \ldots, w_{n'}\} \\ \mathrm{dom}(S_{\overline{\mathcal{W}}}) &= \{\overline{w_1}, \ldots, \overline{w_{n'}}\}, \end{aligned}$$

we have $\mathcal{E}; \Upsilon \vdash S_{\mathcal{X}} S_{\mathcal{W}} S_{\overline{\mathcal{W}}}(\rho(b_2)) : \tau$. This means $\mathcal{E}; \Upsilon \vdash \rho(b) : \tau$. Using this fact and in a similar manner, we can argue $\mathcal{E}; \Upsilon \vdash c(b) : \mathrm{Formula}$.

[T-Ops] Then $b = o(b_1, \ldots, b_n)$, and necessarily that $\Gamma; \Delta \vdash b_i : \hat{\tau}_i$ where $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \hat{\tau}$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b_i) : \hat{\tau}_i$, and $\mathcal{E}; \Upsilon \vdash c(b_i) : \mathrm{Formula}$. Then by the ops sorting rule, $\mathcal{E}; \Upsilon \vdash \rho(b) : \tau$ and by the conjunction sorting rule, $\mathcal{E}; \Upsilon \vdash c(b) : \mathrm{Formula}$.

[T-Assert] Then $b = \mathsf{assert}\ b_1\ \mathsf{in}\ b_2$, and necessarily $\Gamma; \Delta \vdash b_1 : \mathsf{Bool}$, $\Gamma; \Delta \vdash b_2 : \tau$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b_1) : \mathsf{Bool}$, $\mathcal{E}; \Upsilon \vdash \rho(b_2) : \tau$, and $\mathcal{E}; \Upsilon \vdash c(b_1)$, $\mathcal{E}; \Upsilon \vdash c(b_2) : \mathrm{Formula}$. From the first statement, we have $\mathcal{E}; \Upsilon \vdash \rho(b_1) = \mathsf{true} : \mathrm{Formula}$. As $\rho(b) = \rho(b_2)$, we have $\mathcal{E}; \Upsilon \vdash \rho(b) : \tau$. From the conjuction sorting rule, $\mathcal{E}; \Upsilon \vdash c(b) : \mathrm{Formula}$.

[T-Mark] Then $b = \mathsf{mark}_a(b_1, b_2)\#s$, and necessarily that $\Gamma; \Delta \vdash b_1 : \mathsf{Intvl}$, $\Gamma; \Delta \vdash b_2 : \mathsf{Vltn}$. As $\rho(b) \in \mathtt{FV}_{\mathcal{Y}}(b)$, and we assume that $\mathtt{FV}_{\mathcal{Y}}(b) \subseteq \Upsilon$, then we have $\mathcal{E}; \Upsilon \vdash \rho(b) : \mathsf{Point}$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b_1) : \mathsf{Intvl}$, $\mathcal{E}; \Upsilon \vdash \rho(b_2) : \mathsf{Real}$, and $\mathcal{E}; \Upsilon \vdash c(b_1) : \mathrm{Formula}$, $\mathcal{E}; \Upsilon \vdash c(b_2) : \mathrm{Formula}$. By the former two statements we can obtain through a straightforward series of sorting rules

$$\mathcal{E}; \Upsilon \vdash \mathtt{V}_a([\mathtt{1}(\rho(b_1)), \rho(b)]) = \rho(b_2) : \mathrm{Formula}.$$

We can apply the sorting conjuction rule with this and the latter two statements to net $\mathcal{E}; \Upsilon \vdash c(b) : \mathrm{Formula}$.

[T-EvalPc] Then $b = \mathsf{eval}_a(b')$, and necessarily that $\Gamma; \Delta \vdash b' : \mathsf{Piece}$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b') : \mathsf{Piece}$ and $\mathcal{E}; \Upsilon \vdash c(b') : \text{Formula}$, which means by the Val sorting rule $\mathcal{E}; \Upsilon \vdash \rho(b) : \mathsf{Vltn}$ and since $c(b') = c(b)$, $\mathcal{E}; \Upsilon \vdash c(b) : \text{Formula}$.

[T-EvalIntvl] Almost identical to the argument for [T-EvalPc].

[T-Div] Then $b = \mathsf{divide}(b_1, b_2)$, and necessarily that $\Gamma; \Delta \vdash b_1 : \mathsf{Intvl}$, $\Gamma; \Delta \vdash b_2 : \mathsf{Point}$. By induction, $\mathcal{E}; \Upsilon \vdash \rho(b_1) : \mathsf{Intvl}$, $\mathcal{E}; \Upsilon \vdash \rho(b_2) : \mathsf{Point}$, and $\mathcal{E}; \Upsilon \vdash\vdash c(b_1)$, $\mathcal{E}; \Upsilon \vdash c(b_2) : \text{Formula}$. The former statements, through a straightforward set of sorting rules can be used to show both

$$\mathcal{E}; \Upsilon \vdash ([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))]) : \mathsf{Intvl} \times \mathsf{Intvl}$$

and

$$\mathcal{E}; \Upsilon \vdash \mathtt{l}(\rho(b_1)) \leq \rho(b_2) \leq \mathtt{r}(\rho(b_1)) : \text{Formula}.$$

We can apply the sorting conjuction rule with this and the latter two statements from the induction hypothesis to net

$$\mathcal{E}; \Upsilon \vdash c(b) : \text{Formula}.$$

As $\rho(b) = ([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))])$, we have $\mathcal{E}; \Upsilon \vdash \rho(b) : \mathsf{Intvl} \times \mathsf{Intvl}$.

[T-Piece] This is very similar to that of [T-Ops]. □

We now move toward semantic results about constraints. Fix an interpretation and variable assignment. We say that two substitutions S_1 and S_2 are interpreted the same $\text{dom}(S_1) = \text{dom}(S_2)$ and for all $x \in \text{dom}(S_1)$, $\mathcal{A}, \mu \vDash S_1(x) = S_2(x)$. We have the following related lemma.

Lemma 10. *Let S_1 and and S_2 be two logical substitutions. If S_1 and S_2 are interpreted the same, then $\mathcal{A}, \mu \vDash S_1(t) = S_2(t)$ for any term t, and $\mathcal{A}, \mu \vDash S_1(f) \iff S_2(f)$.*

Since our constraints contain variables intended to represent points marked within the protocol, it will be important to ensure that our variable assignment maps the variables to the proper value for a given derivation.

Definition 10. *Let μ be a variable assignment and D a derivation. We say that μ* agrees with *D if for any $\#s$, occuring such that $\mathsf{mark}_a(e_1, e_2)\#s \Downarrow r\#\mathsf{Pt}$ occurs in D, we have that $\mu(y_{\#s}) = r\#\mathsf{Pt}$.*

If we assume that each mark within a protocol has a unique identifier, any derivation involving that protocol will always have an assignment that agrees with it.

Proposition 13 (Soundness). *Let $\overline{V}$ be a valuation set and suppose b is disjoint and $\cdot \vdash b : \tau$. If $D : b \Downarrow_{\overline{V}} v$, then $\mathcal{A}_{\overline{V}}, \mu \vDash c(b)$ and $[\![\rho(b)]\!]^{\mu}_{\mathcal{A}} = |v|$ for any variable assignment μ that agrees with D.*

Proof. This proceeds by induction on D. Suppose that $D : b \Downarrow_{\overline{V}} v$ and suppose μ agrees with D. We omit $\overline{V}$ as a subscript from here on. We break up our cases based on the last rule applied.

[Split] Then $b = (\mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ b_1\ \mathsf{in}\ b_2)$. It must be that $b_1 \Downarrow (v_1, \ldots, v_{n+n'})$ and $S(b_2) \Downarrow v$ where we set

$$S = \{x_i \mapsto v_i \mid 1 \leq i \leq n\}\{w_i \mapsto v_{n+i}, \overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \leq i \leq n'\}.$$

Immediately by induction, $\mathcal{A}, \mu \vDash c(b_1)$ and $[\![\rho(b_1)]\!]^{\mu}_{\mathcal{A}} = (|v_1|, \ldots, |v_{n+n'}|)$. Also immediately by induction, $\mathcal{A}, \mu \vDash c(S(b_2))$ and $[\![\rho(S(b))]\!]^{\mu}_{\mathcal{A}} = |v|$. By Lemma 9, $c(S(b_2)) = |S|(c(b_2))$ and $\rho(S(b_2)) = |S|(\rho(b_2))$. Therefore $\mathcal{A}, \mu \vDash |S|(c(b_2))$ and $[\![|S|(\rho(b_2))]\!]^{\mu}_{\mathcal{A}} = |v|$.

Set

$$S_\rho = \{x_i \mapsto \pi_i \rho(b_1) \mid 1 \leq i \leq n\}\{w_i \mapsto \pi_{n+i}\rho(b_1), \overline{w_i} \mapsto \pi_{n+i}\rho(b_1) \mid 1 \leq i \leq n'\}.$$

As $[\![\rho(b_1)]\!]^{\mu}_{\mathcal{A}} = (|v_1|, \ldots, |v_{n+n'}|)$, we have $[\![\pi_i\rho(b_1)]\!]^{\mu}_{\mathcal{A}} = |v_i|$. This means S_ρ and $|S|$ are interpreted the same. Then by Lemma 10, $\mathcal{A}, \mu \vDash_{\overline{V}} S_\rho(c(b_2))$, and $[\![S_\rho(\rho(b_2))]\!]^{\mu}_{\mathcal{A}} = |v|$. Because $c(b) = c(b_1) \wedge S_\rho(c(b_2))$ and $\rho(b) = S_\rho(\rho(b_2))$, we have established this case.

[Ops] Then $b = o(b_1, \ldots, b_n)$, $b_i \Downarrow v_i$, $\overline{[\![o]\!](|v_1|, \ldots, |v_n|)} = v$, $\rho(b) = \mathsf{o}(\rho(b_1), \ldots, \rho(b_n))$, and $c(b) = c(b_1) \wedge \cdots \wedge c(b_n)$. By induction, $[\![\rho(b_i)]\!]^\mu_{\mathcal{A}} = |v_i|$ and $\mathcal{A}, \mu \vDash c(b_i)$. Therefore $\mathcal{A}, \mu \vDash c(b)$. Also,

$$[\![\rho(b)]\!]^\mu_{\mathcal{A}} = [\![o]\!]([\![\rho(b_1)]\!]^\mu_{\mathcal{A}}, \ldots, [\![\rho(b_n)]\!]^\mu_{\mathcal{A}}) = [\![o]\!](|v_1|, \ldots, |v_n|) = |v|.$$

[Assert] Then $b = \mathsf{assert}\ b_1\ \mathsf{in}\ b_2$, and $b_1 \Downarrow \mathsf{true}$, $b_2 \Downarrow v$. By induction, $[\![\rho(b_1)]\!]^\mu_{\mathcal{A}} = \mathsf{true}$, $\mathcal{A}, \mu \vDash c(b_1)$, $[\![\rho(b_2)]\!]^\mu_{\mathcal{A}} = |v|$, and $\mathcal{A}, \mu \vDash c(b_2)$. This immediately gives us $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = |v|$ and $\mathcal{A}, \mu \vDash c(b)$.

[Mark] Then $b = \mathsf{mark}_a(b_1, b_2)$, and $b_1 \Downarrow [r_1, r_1']$ for some $r_1, r_1' \in [0, 1]$, $b_2 \Downarrow v_{a'}(P)$, with $V_a[r_1, r] = V_{a'}(P)$. Let $y = \rho(b)$. Since μ agrees with D, $\mu(y) = r\#\mathsf{Pt}$. This already establishes $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = |v|$.
By induction, $[\![\rho(b_1)]\!]^\mu_{\mathcal{A}} = [r_1, r_1']$, $[\![\rho(b_2)]\!]^\mu_{\mathcal{A}} = V_{a'}(P)$, and both $\mathcal{A}, \mu \vDash c(b_1)$ and $\mathcal{A}, \mu \vDash c(b_2)$. As $V_a[r_1, r] = V_{a'}(P)$ and $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = r\#\mathsf{Pt}$, we have $\mathcal{A}, \mu \vDash (\mathtt{V}_a[\mathtt{l}(\rho(b_1)), \rho(b)] = \rho(b_2))$. In culmination, $\mathcal{A}, \mu \vDash c(b)$.

[EvalPc] Then $b = \mathsf{eval}_a(b')$, and $b' \Downarrow P_i[r_i, r_i']$ and $v = V_a(P_i[r_i, r_i'])$. By induction, $[\![\rho(b')]\!]^\mu_{\mathcal{A}} = P_i[r_i, r_i']$ and $\mathcal{A}, \mu \vDash c(b')$. The former gives us $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = |v|$. The latter gives us $\mathcal{A}, \mu \vDash c(b)$ since $c(b) = c(b')$.

[EvalIntvl] The argument here is just a more simple verion of the argument for [EvalPc].

[Div] Then $b = \mathsf{divide}(b_1, b_2)$, and $b_1 \Downarrow [r_1, r_1']$, $b_2 \Downarrow r_2$ with $r_1 \leq r_2 \leq r_1'$. By induction, $[\![\rho(b_1)]\!]^\mu_{\mathcal{A}} = [r_1, r_1']$, $[\![\rho(b_2)]\!]^\mu_{\mathcal{A}} = r_2\#\mathsf{Pt}$, and both $\mathcal{A}, \mu \vDash c(b_1)$ and $\mathcal{A}, \mu \vDash c(b_2)$. We can conclude with the former inductive statements that $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = ([r_1, r_2], [r_2, r_1']) = |v|$. Since $r_1 \leq r_2 \leq r_1'$, we also have $\mathcal{A}, \mu \vDash \mathtt{l}(\rho(b_1)) \leq \rho(b_2) \leq \mathtt{r}(\rho(b_1))$ which we can use to conclude $\mathcal{A}, \mu \vDash c(b)$.

[Tup] Then $b = (b_1, \ldots, b_n)$, $b_i \Downarrow v_i$, $v = (v_1, \ldots, v_n)$, $\rho(b) = (\rho(b_1), \ldots, \rho(b_n))$, and $c(b) = c(b_1) \wedge \cdots \wedge c(b_n)$. By induction, $[\![\rho(b_i)]\!]^\mu_{\mathcal{A}} = |v_i|$, and $\mathcal{A}, \mu \vDash c(b_i)$. Clearly $\mathcal{A}, \mu \vDash c(b)$. Also,

$$[\![\rho(b)]\!]^\mu_{\mathcal{A}} = ([\![\rho(b_1)]\!]^\mu_{\mathcal{A}}, [\![\rho(b_n)]\!]^\mu_{\mathcal{A}}) = (|v_1|, \ldots, |v_n|) = |v|.$$

[Piece] This is very similar to the case for [Tup].

□

Proposition 14 (Completeness). *Let $\overline{V}$ be a valuation set, and $\Gamma; \Delta \vdash b : \tau$. Let $\gamma \vDash \Gamma$ and $\delta \vDash \Delta$. Suppose $\mathcal{A}_{\overline{V}}, \mu \vDash c(\gamma; \delta(b))$. Then there is a derivation D such that μ agrees with D and $D : b \Downarrow_{\overline{V}} v$ and $|v| = [\![\rho(\gamma; \delta(b))]\!]^\mu_{\mathcal{A}_{\overline{V}}}$.*

Proof. We prove for any μ and any $\gamma \vDash \Gamma$, $\delta \vDash \Delta$ such that $\mathcal{A}_{\overline{V}}, \mu \vDash c(\gamma; \delta(b))$, there is $D : b \Downarrow v$, $|v| = [\![\rho(\gamma; \delta b))]\!]^\mu_{\mathcal{A}}$ and μ agrees with D. This proceeds by induction on $\Gamma; \Delta \vdash b : \tau$. Suppose that $\mathcal{A}, \mu \vDash_{\overline{V}} c(\gamma; \delta(b))$. We break up our cases based on the structure of b. The cases for [Bool], [Real], [Point], [Piece], and [Cake] are obvious.

[Var] This is also obvious. Then $b = x$ and $x \in \mathcal{X}$, or $b = \overline{w}$ and $w \in \mathcal{W}$. The former is obvious so suppose the latter. Then $\Gamma(\overline{w}) = \overline{v}$ and $\gamma; \delta(\overline{w}) \Downarrow \overline{v}$. Also, $[\![\gamma; \delta(\overline{w})]\!]^\mu_{\mathcal{A}} = [\![\overline{v}]\!]^\mu_{\mathcal{A}} = v = |\overline{v}|$, which completes this case.

[AffVar] This is obvious.

[Split] Then

$$c(\gamma; \delta(b)) = c(\gamma; \delta(b_1)) \wedge S_b(c(\gamma; \delta(b_2))) \quad \text{and} \quad \rho(\gamma; \delta(b)) = S_b(\rho(\gamma; \delta(b_2)))$$

where we set

$$S_\rho = \{x_i \mapsto \pi_i \rho(b_1) \mid 1 \leq i \leq n\}\{w_i \mapsto \pi_{n+i}\rho(b_1), \overline{w_i} \mapsto \pi_{n+i}\rho(b_1) \mid 1 \leq i \leq n'\}.$$

We also have

$$\Gamma; \Delta_1 \vdash b_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_{n'}$$
$$\Gamma; x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}}; \Delta_2, w_1 : \tau_1, \ldots, w_{n'} : \tau_{n'} \vdash b_2 : \tau.$$

According to Lemma 7, we can partition δ into δ_1 and δ_2 such that $\gamma;\delta_1(b_1) = \gamma;\delta(b_1)$, $\gamma;\delta_2(b_2) = \gamma;\delta(b_2)$ and $\delta_1 \vDash \Delta_1$, $\delta_2 \vDash \Delta_2$. By induction, there is D_1 agreeing with μ for which

$$D_1 : \gamma;\delta_1(b_1) \Downarrow v',$$

and $|v'| = [\![\rho(\gamma;\delta_1(b_1))]\!]^\mu_{\mathcal{A}}$. As $\Gamma;\Delta_1 \vdash b_1 : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_n$, we have $\cdot \vdash \gamma;\delta_1(b_1) : \hat{\tau}_1 \times \cdots \times \hat{\tau}_n \times \tau_1 \times \cdots \times \tau_n$. by Proposition 6 we have $v' = (v_1, \ldots, v_{n+n'})$, where $v_i \in [\![\hat{\tau}_i]\!]$ for $1 \le i \le n$ and $v_{n+i} \in [\![\tau_{n'}]\!]$ for $1 \le i \le n'$. This means $[\![\pi_i \rho(\gamma;\delta_1(b_1))]\!]^{\mathcal{A}}_{\mu} = |v_i|$ for all i. By Lemma 10,

$$\mathcal{A}, \mu \vDash S_{b'}(c(\gamma;\delta_2(b_2))) \quad \text{and} \quad [\![S_b(\rho(\gamma;\delta_2(b_2)))]\!]^\mu_{\mathcal{A}} = [\![S_{b'}(\rho(\gamma;\delta_2(b_2)))]\!]^\mu_{\mathcal{A}} \tag{6}$$

where $S_{b'} = \{x_i \mapsto v_i \mid 1 \le i \le n + n'\}$. If we set

$$S = \{x_i \mapsto v_i \mid 1 \le i \le n\}\{w_i \mapsto v_{n+i} \mid 1 \le i \le n'\}\{\overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \le i \le n'\}$$

by Lemma 9,

$$\mathcal{A}, \mu \vDash c(S(\gamma;\delta_2(b_2))) \quad \text{and} \quad [\![\rho(S(\gamma;\delta_2(b_2)))]\!]^\mu_{\mathcal{A}} = [\![S_{b'}(\rho(\gamma;\delta_2(b_2)))]\!]^\mu_{\mathcal{A}}. \tag{7}$$

If we let $\gamma' = \gamma\{x_i \mapsto v_i \mid 1 \le i \le n\}\{\overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \le i \le n'\}$ and $\delta' = \delta_2\{w_i \mapsto v_{n+i} \mid 1 \le i \le n'\}$, then

$$S(\gamma;\delta_2(b_2)) = \gamma';\delta'(b_2). \tag{8}$$

Evidently $\gamma' \vDash \Gamma, x_1 : \hat{\tau}_1, \ldots, x_n : \hat{\tau}_n, \overline{w_1} : \overline{\tau_1}, \ldots, \overline{w_{n'}} : \overline{\tau_{n'}}$ and $\delta' \vDash \Delta_2, w_1 : \tau_1, \ldots, w_{n'} : \tau_{n'}$.
By induction, there is D_2 agreeing with μ such that $D_2 : \gamma';\delta'(b_2) \Downarrow v$ and $|v| = [\![\rho(\gamma';\delta'(b_2))]\!]^\mu_{\mathcal{A}}$. By Equation (8), we can conclude $D : \gamma;\delta(b) \Downarrow v$, with D_1 and D_2 being the premises of D. As μ agrees with both D_1 and D_2, μ also agrees with D.
By combining Equation (6), Equation (7), and Equation (8), we obtain that $[\![\rho(\gamma;\delta(b))]\!]^\mu_{\mathcal{A}} = [\![\rho(\gamma';\delta'(b_2))]\!]^\mu_{\mathcal{A}}$. This means $|v| = [\![\rho(\gamma;\delta(b))]\!]^\mu_{\mathcal{A}}$, concluding this case.

[Ops] Then $\rho(b) = o(\rho(b_1), \ldots, \rho(b_n))$, $c(b) = c(b_1) \wedge \cdots \wedge c(b_n)$ and $o : \hat{\tau}_1, \ldots, \hat{\tau}_n \to \tau$ so $\Gamma;\Delta_i \vdash b_i : \hat{\tau}_i$, where $\Delta = \Delta_1, \ldots, \Delta_n$. According to *Lemma* 7, $\gamma;\delta_i(b_i) = \gamma;\delta(b_i)$ and $\delta_i \vDash \Delta$. Then by induction, $D_i : b_i \Downarrow v_i$ and $|v_i| = [\![\gamma;\delta_i(b_i)]\!]^\mu_{\mathcal{A}}$ and μ agrees with D_i. By Proposition 6, $v_i \in [\![\hat{\tau}_i]\!]$. This enables us to apply [Ops] so $\gamma;\delta(b) \Downarrow [\![o]\!](v_1, \ldots, v_n)$. We also have $[\![\gamma;\delta(\rho(b))]\!]^\mu_{\mathcal{A}} = [\![o]\!]([\![\gamma;\delta_1(b_1)]\!]^\mu_{\mathcal{A}}, \ldots, [\![\gamma;\delta_n(b_n)]\!]^\mu_{\mathcal{A}})$.

[Mark] Then $\rho(b) = y$ for some $y \in \mathcal{Y}$, $c(b) = c(b_1) \wedge c(b_2) \wedge (\mathtt{V}_a([\mathtt{1}(\rho(b_1)), \rho(b)]) = \rho(b_2))$, and $\tau = \mathsf{Point}$, $\Gamma;\Delta_1 \vdash b_1 : \overline{\mathsf{Intvl}}$, $\Gamma;\Delta_2 \vdash b_2 : \mathsf{Vltn}$, where $\Delta = \Delta_1, \Delta_2$. According to Lemma 7, $\gamma;\delta_1(b_1) = \gamma;\delta(b_1)$, $\gamma;\delta_2(b_2) = \gamma;\delta(b_2)$, and $\delta_1 \vDash \Delta_1$, $\delta_2 \vDash \Delta_2$. By induction, $D_1 : \gamma;\delta_1(b_1) \Downarrow v_1$, where $|v_1| = [\![\rho(\gamma;\delta_1(b_1))]\!]^\mu_{\mathcal{A}}$ and μ agrees with D_1. By Proposition 6 $v_1 = \overline{[r, r']}$ for some $r, r' \in \mathbb{R}$. Also by induction, $D_2 : \gamma;\delta_2(b_2) \Downarrow v_2$, where $|v_2| = [\![\gamma;\delta_2(\rho(b_2))]\!]^\mu_{\mathcal{A}}$ and μ agrees with D_2. By Proposition 6, $v_2 = \mathtt{V}_{a'}(P)$ for some piece or interval P and some agent a'.
Now $[\![\mathtt{1}(\rho(\gamma;\delta_1(b)))]\!]^\mu_{\mathcal{A}} = r\#\mathsf{Pt}$. As $\mathcal{A}, \mu \vDash c(b)$, we have $V_a[r, \mu(y)] = V_{a'}(P)$. So using D_1, D_2, and $V_a[r, \mu(y)] = V_{a'}(P)$ as premises for [Mark], we can apply it to obtain $D : \gamma;\delta(b) \Downarrow \mu(y)$ for which μ evidently agrees with D. Now $|\mu(y)| = \mu(y) = [\![y]\!]^\mu_{\mathcal{A}}$, completing this case.

[EvalPc] Then $\Gamma;\Delta \vdash b : \mathsf{Vltn}$, and $\Gamma;\Delta \vdash b' : \mathsf{Piece}$. By induction, there is $D' : b' \Downarrow v'$ and $|v'| = [\![\rho(\gamma;\delta(b'))]\!]^\mu_{\mathcal{A}}$, and μ agrees with D'. By Proposition 6, $v = \overline{P}$ for some piece P. We can apply [EvalPc] to obtain $\gamma;\delta(b) \Downarrow V_a(P)$. Now

$$[\![\rho(\gamma;\delta(b))]\!]^\mu_{\mathcal{A}} = [\![\mathtt{V}_a(\rho(\gamma;\delta(b')))]\!]^\mu_{\mathcal{A}} = V_a([\![\rho(\gamma;\delta(b'))]\!]^\mu_{\mathcal{A}}) = V_a(P) = |V_a(P)|.$$

The argument for when $\Gamma;\Delta \vdash b' : \mathsf{Intvl}$ is nearly identical.

[EvalIntvl] The argument here is nearly identical to the argument for [EvalPc].

[Div] Then $\rho(b) = ([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))])$, $c(b) = c(b_1) \wedge c(b_2) \wedge \mathtt{l}(\rho(b_1)) \leq \rho(b_2) \leq \mathtt{r}(\rho(b_1))$, $\Gamma; \Delta_1 \vdash b_1 : \mathsf{Intvl}$, $\Gamma; \Delta_2 \vdash b_2 : \mathsf{Point}$, and $\Delta = \Delta_1, \Delta_2$. By Lemma 7, $\gamma; \delta_1(b_1) = \gamma; \delta(b_1)$ and $\gamma; \delta_2(b_2) = \gamma; \delta(b_2)$, and $\delta_1 \vDash \Delta_1$, $\delta_2 \vDash \Delta_2$. So by induction, $D_1 : \gamma; \delta_1(b_1) \Downarrow [r, r']$, $[r, r'] = [\![\rho(\gamma; \delta_1(b_1))]\!]^{\mu}_{\mathcal{A}}$ and μ agrees with D_1. Also by induction, $D_2 : \gamma; \delta_2(b_2) \Downarrow r_2 \# \mathsf{Pt}$, $r_2 \# \mathsf{Pt} = [\![\rho(\gamma; \delta_2(b_2))]\!]^{\mu}_{\mathcal{A}}$, and μ agrees with D_2. Thus, $[\![\mathtt{l}(\rho(\gamma; \delta_1(b_1)))]\!]^{\mu}_{\mathcal{A}} = r_1$ and $[\![\mathtt{r}(\rho(\gamma; \delta_1(b_1)))]\!]^{\mu}_{\mathcal{A}} = r_1'$. Since $\mathcal{A}, \mu \vDash c(b)$, we have $r_1 \leq r_2 \leq r_1'$. Therefore we can apply [Div] with D_1 and D_2 as premises to obtain $D : \gamma; \delta(b) \Downarrow ([r_1, r_2], [r_2, r_1'])$, and μ agrees with D as it agreed with D_1 and D_2. We are done as $[\![([\mathtt{l}(\rho(b_1)), \rho(b_2)], [\rho(b_2), \mathtt{r}(\rho(b_1))])]\!]^{\mu}_{\mathcal{A}} = ([r_1, r_2], [r_2, r_1']) = |([r_1, r_2], [r_2, r_1'])|$.

[Assert] Then $\rho(b) = \rho(b_2)$, $c(b) = (\rho(b_1) = \mathsf{true}) \wedge c(b_1) \wedge c(b_2)$, and $\Gamma; \Delta_1 \vdash b_1 : \mathsf{Bool}$, $\Gamma; \Delta_2 \vdash b_2 : \tau$, where $\Delta = \Delta_1, \Delta_2$. According to Lemma 7, $\gamma; \delta_1(b_1) = \gamma; \delta(b_1)$, $\gamma; \delta_2(b_2) = \gamma; \delta(b_2)$, and $\delta_1 \vDash \Delta_1$, $\delta_2 \vDash \Delta_2$. By induction, $D_1 : \gamma; \delta_1(b_1) \Downarrow \mathsf{true}$, where μ agrees with D_1. Also by induction, $D_2 : \gamma; \delta_2(b_2) \Downarrow v$ where $|v| = [\![\rho(\gamma; \delta_2(b_2))]\!]^{\mu}_{\mathcal{A}}$, and μ agrees with D_2. This allows us to use [Assert] to obtain $D : \gamma; \delta(b) \Downarrow v$, with premises D_1 and D_2. As D is composed in this way, μ agrees with D. Since $\rho(b) = \rho(b_2)$, we are done with this case.

[Tup] Then $b = (b_1, \ldots, b_{n+n'})$, $\rho(b) = (\rho(b_1), \ldots, \rho(b_{n+n'}))$, $c(b) = c(b_1) \wedge \cdots \wedge c(b_{n+n'})$, and $\Gamma; \Delta_i \vdash b : \hat{\tau}_i$ for $1 \leq i \leq n$ and $\Gamma; \Delta_{n+i} \vdash b_{n+i} : \tau_i$ for $1 \leq i \leq n'$, and $\Delta = \Delta_1, \ldots, \Delta_{n+n'}$. According to Lemma 7, $\gamma; \delta_i(b_i) = \gamma; \delta(b_i)$, and $\delta_i \vDash \Delta_i$ for $1 \leq i \leq n + n'$. By induction, $D_i : \gamma; \delta_i(b_i) \Downarrow v_i$, where $|v_i| = [\![\rho(\gamma; \delta_i(b_i))]\!]^{\mu}_{\mathcal{A}}$, and μ agrees with D_i for $1 \leq i \leq n + n'$. Using $D_1, \ldots, D_{n+n'}$ as premises, we can apply [Tup] to obtain $D : \gamma; \delta(b) \Downarrow (v_1, \ldots, v_{n+n'})$ where μ agrees with D. We have

$$\begin{aligned} |(v_1, \ldots, v_{n+n'})| = (|v_1|, \ldots, |v_{n+n'}|) &= ([\![\rho(\gamma; \delta_1(b_1))]\!]^{\mu}_{\mathcal{A}}, \ldots, [\![\rho(\gamma; \delta_{n+n'}(b_{n+n'}))]\!]^{\mu}_{\mathcal{A}}) \\ &= [\![(\rho(\gamma; \delta_1(b_1)), \ldots, \rho(\gamma; \delta_{n+n'}(b_{n+n'})))]\!]^{\mu}_{\mathcal{A}} \\ &= [\![\rho(\gamma; \delta((b_1, \ldots, b_{n+n'})))]\!]^{\mu}_{\mathcal{A}}, \end{aligned}$$

concluding this case.

[Piece] This is nearly identical to the tuple case.

Soundness and completeness results lead to the following corollary.

Theorem 1. *Suppose $\cdot \vdash b : \tau$. Then $b \Downarrow_{\overline{V}} v$ if and only if there is a variable assingment μ such that $\mathcal{A}_{\overline{V}}, \mu \vDash c(b)$ and $[\![\rho(b)]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = |v|$.*

We now argue for the following theorem:

Theorem 2. *Suppose that e is a well-formed expression and $\cdot \vdash e : \mathsf{Piece}^{\mathbb{A}}$. Then*

$$\mathcal{A}_{\overline{V}} \vDash \bigwedge_{b \in B(e)} \forall \mathcal{Y}_b.(c(b) \Rightarrow E(\rho(b))) \tag{2}$$

for all $\overline{V}$ if and only if $e \vDash E(x)$.

We first generalize to the envy-freeness formula to formulae that have mild restrictions.

Definition 11. *We say that a formula F is a* well-formed property *if $x : \mathsf{S}_\tau \vdash F :$ Formula, and F contains no point values.*

Here, x stands in for what a protocol might possibly output. The point value restriction is not overly restrive, as most interesting properties do not reference constant points, however, our theory could be modified slightly to relax this restriction. Equation (1) is easily seen to be a well-formed property. We also require some mild assumptions on our expressions.

Definition 12. *We say that an expression e is* well-formed *if e is closed, well-typed, disjoint, e does not contain any point values (besides 0 and 1), and each mark subexpression contains a unique identifier.*

Similar to protocol properties, very few interesting protocols consider constant points (besides 0 and 1) within the cake.

Here we also generalise $e \vDash E(x)$ to arbitrary formulae. Given an expression $\cdot \vdash e : \tau$ and a formula F such that $x : \mathtt{s}_\tau \vdash F : \text{Formula}$, we say that $e \vDash F$ if for all valuation sets $\overline{V}$, $e \Downarrow_{\overline{V}} v$ implies $\mathcal{A}_{\overline{V}} \vDash F\{\mathtt{v}/x\}$. Now we state and prove our generalized theorem.

Proposition 15. *Suppose that e is a well-formed expression and $\cdot \vdash e : \tau$. Let F be a formula such that $x : \mathtt{S}_\tau \vdash F : \text{Formula}$. Then*

$$\mathcal{A}_{\overline{V}} \vDash \bigwedge_{b \in B(e)} \forall \mathcal{Y}_b.(c(b) \Rightarrow F\{\rho(b)/x\}) \tag{9}$$

for all $\overline{V}$ if and only if $e \vDash F$.

Proof. ($\Rightarrow$) Let $\mathcal{A} = \mathcal{A}_{\overline{V}}$. Suppose that $D : e \Downarrow_{\overline{V}} v$. By Proposition 8 and Proposition 9, there is $b \in B(e)$ such that $\vdash b : \tau$ and $D : b \Downarrow_{\overline{V}} v$. By Proposition 13, for any μ that agrees with D, $\mathcal{A}, \mu \vDash c(b)$ and $[\![\rho(b)]\!]^\mu_{\mathcal{A}} = |v|$. By Equation (9), $\mathcal{A}_{\overline{V}}, \mu \vDash F\{\rho(b)/x\}$. By Lemma 10, $\mathcal{A}_{\overline{V}}, \mu \vDash F\{\mathtt{v}/x\}$. $F\{v/x\}$ is a closed formula so $\mathcal{A}_{\overline{V}} \vDash F\{\mathtt{v}/x\}$.

($\Leftarrow$) Suppose Equation (9) does not hold. Then there is some interpretation $\mathcal{A}$ and variable assignment μ such that $\mathcal{A}, \mu \vDash c(b)$ yet $\mathcal{A}, \mu \nvDash F\{\rho(b)/x\}$. Then by Proposition 14, $b \Downarrow_{\overline{V}} v$, where $|v| = [\![\rho(b)]\!]^\mu_{\mathcal{A}}$ for some $b \in B(e)$. By Proposition 9, $e \Downarrow_{\overline{V}} v$. By Lemma 10, we have that $\mathcal{A}, \mu \nvDash F\{\mathtt{v}/x\}$, and since $F\{\mathtt{v}/x\}$ is closed, $\mathcal{A} \nvDash F\{\mathtt{v}/x\}$, which completes the proof.

7.10 Protocol execution replication

The goal of this subsection is to prove the following theorems:

Theorem 3. *Let $\overline{U}$ and $\overline{V}$ be valuation sets, and suppose $e \Downarrow_{\overline{V}} v$. If $\overline{U}$ and $\overline{V}$ agree on all points considered in the derivation of $e \Downarrow_{\overline{V}} v$, then $e \Downarrow_{\overline{U}} v$.*

Theorem 4. *If valuation sets $\overline{U}$ and $\overline{V}$ agree on the set of points considered in a formula φ under variable assignment μ, then $\mathcal{A}_{\overline{V}}, \mu \vDash \varphi \iff \mathcal{A}_{\overline{U}}, \mu \vDash \varphi$.*

We first become more precise about derivations and points considered within them. First recall our derivation notation. Briefly, given a derivation of an evaluation judgement D, we write $D : e \Downarrow_{\overline{V}} v$ if the conclusion of D is $e \Downarrow_{\overline{V}} v$. Define

$$M(D) \triangleq \{r \mid r\#\mathsf{Pt} \text{ is a subexpression of } v, v \text{ appears in } D\}.$$

Theorem 3 can now be expressed as:

Theorem 8. *Let $\overline{U}$ and $\overline{V}$ be a valuation set, and suppose $D : e \Downarrow_{\overline{V}} v$. If $\overline{U}$ and $\overline{V}$ agree on $M(D)$, then $D : e \Downarrow_{\overline{U}} v$.*

Proof. This goes by induction on D. We break it up into cases based on the last rule applied, of which there are 4 interesting cases.

[E-EvalPc] Here $e = \mathsf{eval}(e')$ for some e', $D' : e' \Downarrow_{\overline{V}} P^n_{i=1}[r_i, r'_i]$, and $v = \overline{V}_a(P^n_{i=1}[r_i, r'_i])$. Now $r_i, r'_i \in M(D)$ for all i so $\cup^n_{i=1}[r_i, r'_i]$ has all boundary points in $M(D)$. Therefore, since $\overline{U}$ and $\overline{V}$ agree on $M(D)$, $\overline{U}_a(\cup^n_{i=1}[r_i, r'_i]) = \overline{V}_a(\cup^n_{i=1}[r_i, r'_i])$. By induction $D' : e' \Downarrow_{\overline{U}} P^n_{i=1}[r_i, r'_i]$, so we can apply [E-EvalPc] to obtain $D : e \Downarrow_{U_{\overline{V}}} v$.

[E-EvalIntvl] This argument is nearly identical to the argument for [E-EvalPc].

[E-Mark] Here $e = \mathsf{mark}_a(e_1, e_2)$ for some e_1 and e_2, $D_1 : e_1 \Downarrow_{\overline{V}} [r_1, r_2]$, $D_2 : e_2 \Downarrow_{\overline{V}} v_2$, $\overline{V}_a[r_1, r] = v_2$ with $v = r\#\mathsf{Pt}$. Now $r_1, r \in M(D)$. Since $\overline{U}$ and $\overline{V}$ agree on $M(D)$, $\overline{U}_a[r_1, r] = v_2$. By induction, $D_1 : e_1 \Downarrow_{\overline{U}} [r_1, r_2]$ and $D_2 : e_2 \Downarrow_{\overline{U}} v_2$, so applying [E-Mark], we obtain $D : e \Downarrow_{\overline{U}} v$. □

We restate Theorem 4 in the following way:

Theorem 9. *Suppose that $\Upsilon \vdash \varphi$: Formula. Let $\overline{V}$ and $\overline{U}$ be valuation sets and μ be a variable assignment. Suppose $\overline{U}$ and $\overline{V}$ agree on $\{[\![t]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} \mid \Upsilon \vdash t : \texttt{Point}, t \text{ occurs in } \varphi\}$. Then $\mathcal{A}_{\overline{V}}, \mu \vDash \varphi \iff \mathcal{A}_{\overline{U}}, \mu \vDash \varphi$.*

Proof. By Proposition 11, it suffices to check this on simplified formulas. So assume that φ is simplified. This argument proceeds by induction on the structure of φ. Set $M = \{[\![t]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} \mid \Upsilon \vdash t : \texttt{Point}, t \text{ occurs in } \varphi\}$. The whole argument boils down to showing $[\![\texttt{V}_a(t)]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = [\![\texttt{V}_a(t)]\!]^{\mu}_{\mathcal{A}_{U_{\overline{V}}}}$.

Let $\texttt{V}_a(t)$ be a term contained in φ. Then either $t = [t_1, t_1']$ for some $t_1, t_1' \in R_{\texttt{Point}}$ or $t = \cup([t_1, t_1'], \ldots, [t_n, t_n'])$ for some $t_i, t_i' \in R_{\texttt{Point}}$. We note that $[\![t_i]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = [\![t_i]\!]^{\mu} = [\![t_i]\!]^{\mu}_{\mathcal{A}_{\overline{U}}}$ for all i, which also gives $[\![t]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = [\![t]\!]^{\mu} = [\![t]\!]^{\mu}_{\mathcal{A}_{\overline{U}}}$. Since $\overline{U}$ and $\overline{V}$ agree on M, we have that $\overline{U}_a([\![t]\!]^{\mu}) = \overline{V}_a([\![t]\!]^{\mu})$ as $[\![t]\!]^{\mu}$ has boundary points in M. This means $[\![\texttt{V}_a(t)]\!]^{\mu}_{\mathcal{A}_{\overline{V}}} = [\![\texttt{V}_a(t)]\!]^{\mu}_{\mathcal{A}_{U_{\overline{V}}}}$. □

7.11 Piecewise uniform valuations

Here we discuss the arguments for piecewise uniform valuations. In this section we prove the following theorem.

Theorem 5. *For any valuation set $\overline{V}$ and any finite set of points $M \supseteq \{0, 1\}$, there exists a piecewise uniform valuation set that both agrees with $\overline{V}$ on M and is easily replaceable on M.*

We first start by introducing a form of a valuation and argue some facts about it. Following this, we consider a whole valuation set of this form.

Given an arbitrary valuation V and a finite set of points $\{0, 1\} \subseteq M$, we can have a piecewise uniform valuation replicate V on pieces whose endpoints are in M.

Let $M = \{m_1, \ldots, m_k\}$ be given such that $0 = m_0 \leq m_1 < \cdots < m_k = 1$. Now let V be any valuation on $[l, r]$, and define

$$\max V/M = \max_{0 \leq i < k} V[m_i, m_{i+1}]/(m_{i+1} - m_i).$$

This quantity is referred to as the *max density of V on M*. It represents the largest density of V on adjacent points in M. Recall that a piece uniquely determines a piecewise uniform valuation on that piece.

Definition 13. *Let $d \geq \max V/M$. We let $U_V(M, d)$ be the piecewise uniform valuation determined by the piece*

$$P = \bigcup_{i=1}^{k} [m_i - V[m_{i-1}, m_i]/d, m_i] = [m_1 - V[0, m_1]/d, m_1] \cup \cdots \cup [m_k - V[m_{k-1}, m_k]/d, m_k].$$

That is, $U_V(M, d) = U_P$, in the notation below Definition 2.

$U_V(M, d)$ is not well-defined for $d < \max V/M$. For if this is the case, some intervals in the piece above would overlap further than their endpoints.

We first aim to show this valuation agrees with V on M. To do so, we state a basic fact about $U_V(M, d)$.

Lemma 11. *$c(U_V(M, d)) = d$.*

Proof. Let c be the constant associated with $U_V(M,d)$. Then

$$\begin{aligned} V[l,r] &= U_V(M,d)[l,r] \\ &= U_V(M,d)[m_0,m_k] \\ &= c\sum_{k\geq i>0} m_i - l_i \\ &= c\sum_{k\geq i>0} (m_i - (m_i - V[m_{i-1},m_i]/d) \\ &= c\sum_{k\geq i>0} V[m_{i-1},m_i]/d \\ &= cV[m_0,m_k]/d \\ &= cV[l,r]/d \end{aligned}$$

which implies that $c = d$. □

Lemma 12. *Let P be any piece with boundary points entirely within M then*

$$U_V(M,d)(P) = V(P). \tag{10}$$

Proof. First, write $M = \{m_0, m_1, \ldots, m_k\}$ where $0 = m_0 \leq m_1 < \cdots < m_k = 1$. We have for any $1 \leq i \leq k$,

$$\begin{aligned} U_W(M,d)[m_{i-1},m_i] &= d(m_i - (m_i - W[m_{i-1},m_i]/d)) \\ &= d(W[m_{i-1},m_i]/d) \\ &= W[m_{i-1},m_{i+1}] \end{aligned}$$

and since $[m_i, m_{i'}]$ is the disjoint union of $[m_{j-1}, m_j]$ for $i < j \leq i'$, we have by disjointness,

$$U_W(M,d)[m_i,m_{i'}] = W[m_i,m_{i'}]$$

for any $0 \leq i \leq i' \leq k$. Since P has boundary points entirely within M, we can write $P = [m_{i_1}, m'_{i_1}] \cup \cdots \cup [m_{i_n}, m'_{i_n}]$, for $m_{i_1} \leq m'_{i_1} \leq \cdots \leq m_{i_n} \leq m'_{i_n}$. This means

$$\begin{aligned} U_W(M,d)(P) &= U_W(M,d)([m_{i_1},m'_{i_1}] \cup \cdots \cup [m_{i_n},m'_{i_n}]) \\ &= U_W(M,d)[m_{i_1},m'_{i_1}] + \cdots + U_W(M,d)[m_{i_n},m'_{i_n}] \\ &= W[m_{i_1},m'_{i_1}] + \cdots + W[m_{i_n},m'_{i_n}] \\ &= W([m_{i_1},m'_{i_1}] \cup \cdots \cup [m_{i_n},m'_{i_n}]) \\ &= W(P). \end{aligned}$$

□

We now extend both the above definition and lemma to valuation sets. We define for $\overline{V}$ a valuation set, $\max \overline{V}/M \triangleq \max_{a\in\mathbb{A}} \max \overline{V}_a/M$.

Definition 14. *Let $\overline{V}$ be a valuation set, $M \supseteq \{0,1\}$ a set of points, and $d \geq \max \overline{V}/M$. Then $U_{\overline{V}}(M,d)$ is the valuation set with $U_{\overline{V}}(M,d)_a \triangleq U_{\overline{V}_a}(M,d)$ for all $a \in \mathbb{A}$.*

Let $d \geq \max \overline{V}/M$. In particular, this means for each $a \in \mathbb{A}$, $d \geq \max \overline{V}_a/M$. Thus, by Lemma 12, $U_{\overline{V}}(M,d)$ and $\overline{V}$ agree on M. We now show that $U_{\overline{V}}(M,d)$ is easily replaceable on M. If for each $m \in M^{\setminus 0}$, we set $l_a(m_i) \triangleq m_i - V[m_{i-1}, m_i]/d$, then

$$P(U_{\overline{V}}(M,d)_a) = \bigcup_{m \in M^{\setminus 0}} [l_a(m), m],$$

which satisfies condition (1). To satisfy condition (2), we note that by Lemma 11, $c(U_{\overline{V}}(M,d)_a) = d$ for all a. This shows that $U_{\overline{V}}(M,d)$ is easily replaceable on M.

7.12 Formula simplification

Before proving any theorems about simplification, we write the definition for $\#\mathsf{Pt}(t)$ and $\#\mathsf{Pt}(f)$ more carefully.

Definition 15. *Let $t \in R_{\texttt{Intvl}}$. Then $\#\mathsf{Pt}(t) = \{t_1, t_2\}$, where $t = [t_1, t_2]$ and $t_1, t_2 \in R_{\texttt{Intvl}}$. Let $t \in R_{\texttt{Piece}}$. Then*

$$\#\mathsf{Pt}(t) \triangleq \#\mathsf{Pt}(t_1) \cup \cdots \cup \#\mathsf{Pt}(t_n)$$

where $t = \cup t_1, \ldots, t_n$ for $t_i \in R_{\texttt{Intvl}}$. Let $t \in R_{\mathtt{s}}$. We define $\#\mathsf{Pt}(t)$ as the union of all $\#\mathsf{Pt}(t')$ for t' an interval or point in t. For a simplified formula f, we let $\#\mathsf{Pt}(f)$ denote the union of $\#\mathsf{Pt}(t)$ for all terms t in f. We refer to the elements of $\#\mathsf{Pt}(f)$ as point atoms.

We first aim to prove the following theorem:

Theorem 6. *Let f be a simplified formula and let S be a piecewise uniform replacement such that $\#\mathsf{Pt}(f) \subseteq S$. Let μ be an assignment and $\overline{U}$ a piecewise uniform valuation set. If $S \xrightarrow{\mu} \overline{U}$ then $\mathcal{A}_{\overline{U}}, \mu \vDash f \iff \mathcal{A}_{\overline{U}}, \mu \vDash S(f)$.*

This requires the following lemma.

Lemma 13. *Suppose we have $S \xrightarrow{\mu} \overline{U}$. If $t \in R_{\texttt{Piece}}$ and $\#\mathsf{Pt}(t) \subseteq S$, let $\underline{S}|_t = \{y \in S|_t \mid \forall y' \in S.\, \mu(y) = \mu(y') \Rightarrow y \leq_S y'\}$. Then*

$$\overline{U}_a([\![t]\!]^\mu) = c(\overline{U}_a) \sum_{y \in \underline{S}|_t} \mu(y) - \mu(z_{a,y}).$$

Proof. Before proceeding, we give a formula for $\overline{U}_a([\![t]\!]^\mu)$. Write $t = \cup[y_1, y_1'], \ldots, [y_n, y_n']$ and let $\mu(S)|_t = \{m \in \mu(S) \mid \mu(y_i') \geq m > \mu(y_i)$ for some $1 \leq i \leq n\}$. Since $[\![t]\!]^\mu = \bigcup_{i=1}^n [\mu(y_i), \mu(y_i')]$ and $\#\mathsf{Pt}(t) \subseteq S$, we have that $\partial[\![t]\!]^\mu \subseteq \mu(S)$. Therefore by (1) and by Equation (3), we have

$$\overline{U}_a([\![t]\!]^\mu) = c(\overline{U}_a) \cdot \sum_{m \in \mu(S)|_t} (m - l_a(m)).$$

To complete this argument, it suffices to show that μ gives a bijection between $\mu(S)|_t$ and $\underline{S}|_t$, for if $\mu(y) = m \in M$ and $y = \min\{y' \in S \mid \mu(y') = m\}$ then $\mu(y) - \mu(z_{a,y}) = m - l_a(m)$ according to (2), agreeing with the summand above.

Let $y \in \underline{S}|_t$. As $y \in S|_t$, we have $y_i' \geq_S y >_S y_i$ for some $1 \leq i \leq n$. By (4), $\mu(y_i') \geq \mu(y) \geq \mu(y_i)$. It also must be that $\mu(y) > \mu(y_i)$, as $y = \min\{y' \in S \mid \mu(y') = m\}$. Therefore $\mu(y) \in \mu(S)|_t$. Thus, $\mu|_{\underline{S}|_t} : \underline{S}|_t \to \mu(S)|_t$. It remains to show that $\mu|_{\underline{S}|_t}$ is injective and surjective.

Injectivity is almost immediate. Since $>_S$ is a total order, $\min\{y' \in S \mid \mu(y') = m\}$ is unique for each $m \in \mu(S)$. Therefore $\underline{S}|_t$ contains at most one y such that $\mu(y) = m$ for each $m \in \mu(S)$.

Let $m \in \mu(S)|_t$. Let $y_m = \min \mu|_S^{-1}(m)$. If we show that $y_m \in \underline{S}|_t$, surjectivity is established. As $m \in \mu(S)|_t$, $\mu(y_i') \geq m > \mu(y_i)$ for some $1 \leq i \leq n$. Hence $\mu(y_i') \geq \mu(y_m) > \mu(y_i)$, so by (4), $y' \geq_S y_m >_S y$. This means $y_m \in S|_y^{y'} \subseteq S|_t$. As $y_m = \min \mu|_S^{-1}(m)$, $y_m \in \underline{S}|_t$. This completes the proof. □

Proof (Theorem 6). Let $\mathcal{A} = \mathcal{A}_{\overline{U}}$. As f is simplified, it only consists of predicates of the form given in Corollary 2. S does nothing to predicates of the form $\sum_i r_i d_i \geq \sum_j r_j d_j$, so we focus on those of the form

$$\sum_i r_i \mathtt{V}_{a_i}(t_i) \geq \sum_j r_j \mathtt{V}_{a_j}(t_j),$$

for t_i,$t_j \in R_{\mathtt{Intvl}} \cup R_{\mathtt{Piece}}$. Without loss of generality, we can assume that $t_i, t_j \in R_{\mathtt{Piece}}$, for if $t \in R_{\mathtt{Intvl}}$, then $[\![\mathtt{V}_a(t_i)]\!]^\mu_{\mathcal{A}} = [\![\mathtt{V}_a(\cup t_i)]\!]^\mu_{\mathcal{A}}$. We know that $\#\mathsf{Pt}(f) \subseteq S$ so $\#\mathsf{Pt}(t_i) \subseteq S$ and $\#\mathsf{Pt}(t_j) \subseteq S$ for all i and j. Let $P_i = [\![t_i]\!]^\mu$ and $P_j = [\![t_j]\!]^\mu$. Let $\underline{S}|_t = \{y \in S|_t \mid \forall y' \in S.\, \mu(y) = \mu(y') \Rightarrow y \leq_S y'\}$. We have

$$\begin{aligned}
[\![S(\mathtt{V}_{a_i}(t_i))]\!]^\mu_{\mathcal{A}} &= \sum_{y \in S|_{t_i}} \mu(y) - \mu(z_{a_i,y}) \\
&= \sum_{y \in \underline{S}|_t} \mu(y) - \mu(z_{a_i,y}) + \sum_{y \in S|_{t_i} \setminus \underline{S}|_t} \mu(y) - \mu(z_{a_i,y}) \\
&= \sum_{y \in \underline{S}|_t} \mu(y) - l_{a_i}(\mu(y)) + \sum_{y \in S|_t \setminus \underline{S}|_t} \mu(y) - \mu(y) \\
&= \overline{U}_{a_i}(P_i)/d
\end{aligned}$$

where we use $S \xrightarrow{\mu} \overline{U}$ (2) and (3) for the third equality, and Lemma 13 for the fourth equality. Similarly,

$$[\![S(\mathtt{V}_{a_j}(t_j))]\!]^\mu_{\mathcal{A}} = \overline{U}_{a_j}(P_j)/d.$$

Now

$$\begin{aligned}
&\mathcal{A}, \mu \vDash \sum_i r_i \mathtt{V}_{a_i}(t_i) \geq \sum_j r_j \mathtt{V}_{a_j}(t_j) \\
&\iff [\![\sum_i r_i \mathtt{V}_{a_i}(t_i)]\!]^\mu_{\mathcal{A}} \geq [\![\sum_j r_j \mathtt{V}_{a_j}(t_j)]\!]^\mu_{\mathcal{A}} \\
&\iff \sum_i r_i \overline{U}_{a_i}(P_i) \geq \sum_j r_j \overline{U}_{a_j}(P_j) \\
&\iff \sum_i r_i [\![S(\mathtt{V}_{a_i}(t_i))]\!]^\mu_{\mathcal{A}} \geq \sum_j r_j [\![S(\mathtt{V}_{a_j}(t_j))]\!]^\mu_{\mathcal{A}} \\
&\iff \mathcal{A}, \mu \vDash S\left(\sum_i r_i \mathtt{V}_{a_i}(t_i) \geq \sum_j r_j \mathtt{V}_{a_j}(t_j)\right).
\end{aligned}$$

This completes our argument. □

We now aim to prove the following theorem:

Theorem 7. *Suppose e is well-formed and $\cdot \vdash e : \mathsf{Piece}^{\mathbb{A}}$. Then $e \vDash E(x)$ if and only if*

$$\vDash \bigwedge_{b \in B(e)} \bigwedge_{S \in S_b} \forall \mathcal{Y}_b. S(\mathtt{R}(c(b) \wedge \psi(S) \Rightarrow E(\rho(b)))). \tag{4}$$

Like with Theorem 2, we generalize to well-formed properties:

Theorem 10. *Suppose e is well-formed and $\cdot \vdash e : \tau$. Let F be a well-formed property such that $x : \mathtt{s}_\tau; \cdot \vdash F : \mathit{Formula}$. Then*

$$\vDash \bigwedge_{b \in B(e)} \bigwedge_{S \in S_b} \forall \mathcal{Y}_b. S(\mathtt{R}(c(b) \wedge \psi(S) \Rightarrow F\{\rho(b)/x\})) \tag{11}$$

if and only if $e \vDash F$.

Here we state and prove some results which help connect points in programs to points in their constraints.

Lemma 14. *For any interpretation $\mathcal{A}$, if $S \xrightarrow{\mu} \overline{U}$, then*

$$\mathcal{A}, \mu \vDash \psi(S).$$

Proof. Straightforward from the construction of $U_{\overline{V}}(M, d)$ and the definition of $S \xrightarrow{\mu} \overline{U}$. □

Lemma 15. *Suppose $D : b \Downarrow_{\overline{V}} v$. Then $M(D) = \{r\#\mathsf{Pt} \mid \mathsf{mark}_a(b_1, b_2)\#s \Downarrow r\#\mathsf{Pt}$ occurs in $D\} \cup M(b)$.*

Proof. Let $M(D)' \triangleq \{r\#\mathsf{Pt} \mid \mathsf{mark}_a(e_1, e_2)\#s \Downarrow r\#\mathsf{Pt}$ occurs in $D\} \cup M(e)$. This argument proceeds by induction on D.

[E-Val] First note that $\{r \mid r\#\mathsf{Pt}$ appears in $v\} = M(v)$. And then observe that if $D : v \Downarrow v$, then $\{r \mid r\#\mathsf{Pt}$ appears in $v\} = \{r \mid r\#\mathsf{Pt}$ appears in v, v appears in $D\}$. Therefore, $M(D) = M(v) = M(e)$ in this case.

[E-Ops] Then $e = o(e_1, \ldots, e_n)$, $D_i : e_i \Downarrow v_i$, and $v = [\![o]\!](v_1, \ldots, v_n)$ for some $o \in O$. Inspection of the allowed operations enables us to claim that $r\#\mathsf{Pt}$ only occurs in v if it occurs also in v_i for some i. Therefore, $M(D) = M(D_1) \cup \cdots M(D_n)$. Also, $M(D)' = M(D_1)' \cup \cdots \cup M(D_n)'$. By induction, $M(D_i) = M(D_i)'$, concluding this case.

[E-Mark] Then $e = \mathsf{mark}_a(e_1, e_2)\#s$, $v = r\#\mathsf{Pt}$, $D_1 : e_1 \Downarrow \overline{[r_1, r_1']}$, $D_2 : e_2 \Downarrow \overline{V}_{a'}(P)$, and $\overline{V}_a[r_1, r] = \overline{V}_{a'}(P)$. Let $\#\mathsf{Pt}(P)$ be the set of points within P. Now $M(D) = M(D_1) \cup M(D_2) \cup \{r_1\#\mathsf{Pt}, r_1'\#\mathsf{Pt}, r\#\mathsf{Pt}\} \cup \#\mathsf{Pt}(P)$. Since $D_2 : e_2 \Downarrow \overline{V}_{a'}(P)$, $\#\mathsf{Pt}(P) \subseteq M(D)'$. Since $D_1 : e_1 \Downarrow \overline{[r_1, r_1']}$, we have $r_1, r_1' \in M(D_1)$. Therefore, $M(D) = M(D_1) \cup M(D_2) \cup \{r\#\mathsf{Pt}\}$. We also have $M(D)' = M(D_1)' \cup M(D_2)' \cup \{r\#\mathsf{Pt}\}$. By induction, $M(D_1) = M(D_1)'$ and $M(D_2) = M(D_2)'$. This completes the case.

[E-EvalPc] Then $e = \mathsf{eval}_a(e')$, $v = \overline{V}_a(P)$, and $D' : e' \Downarrow P$. Let $\#\mathsf{Pt}(P)$ be the set of points in P. Then $M(D) = M(D') \cup \#\mathsf{Pt}(P)$. Since $D' : e' \Downarrow P$, $\#\mathsf{Pt}(P) \subseteq M(D')$. Therefore $M(D) = M(D')$. Now $M(D)' = M(D')'$. By induction, $M(D')' = M(D')$, so we can conclude this case.

[E-EvalIntvl] The argument is very similar to the argument for [EvalPc].

[E-Div] Then $e = \mathsf{divide}(e_1, e_2)$, $v = ([r_1, r_2], [r_2, r_1'])$, $D_1 : e_1 \Downarrow [r_1, r_1']$, and $D_2 : e_2 \Downarrow r_2\#\mathsf{Pt}$. Then $M(D) = M(D_1) \cup M(D_2) \cup \{r_1, r_1', r_2\}$. Since $D_1 : e_1 :\Downarrow [r_1, r_1']$, $\{r_1, r_1'\} \subseteq M(D_1)$, and as $D_2 : e_2 \Downarrow r_2\#\mathsf{Pt}$, we have $r_2\#\mathsf{Pt} \in M(D_2)$. Therefore $M(D) = M(D_1) \cup M(D_2)$. Now $M(D)' = M(D_1)' \cup M(D_2)'$. By induction, $M(D_1) = M(D_1)'$ and $M(D_2) = M(D_2)'$. This case is then complete.

[E-Split] Then $e = \mathsf{let}\ x_1, \ldots, x_n, w_1, \ldots, w_{n'} = \mathsf{split}\ e_1\ \mathsf{in}\ e_2$, $D_1 : e_1 \Downarrow (v_1, \ldots, v_{n+n'})$, $D_2 : S_{\mathcal{X}} S_{\overline{\mathcal{W}}} S_{\mathcal{W}}(e_2) \Downarrow v$, where

$$S_{\mathcal{X}} \triangleq \{x_i \mapsto v_i \mid 1 \leq i \leq n\}$$
$$S_{\overline{\mathcal{W}}} \triangleq \{\overline{w_i} \mapsto \overline{v_{n+i}} \mid 1 \leq i \leq n'\}$$
$$S_{\mathcal{X}} \triangleq \{w_i \mapsto v_{n+i} \mid 1 \leq i \leq n'\}.$$

Now $M(D) = M(D_1) \cup M(D_2)$, and $M(D)' = M(D_1)' \cup M(D_2)' \cup M(e_2)$. Since $M(D_2)' \supseteq M(S_{\mathcal{X}} S_{\overline{\mathcal{W}}} S_{\mathcal{W}}(e_2))$ and $M(S_{\mathcal{X}} S_{\overline{\mathcal{W}}} S_{\mathcal{W}}(e_2)) \supseteq M(e_2)$, we have $M(D)' = M(D_1)' \cup M(D_2)'$. By induction, $M(D_1) = M(D_1)'$ and $M(D_2) = M(D_2)'$, completing this case. □

Lemma 16. *Suppose that $D : b \Downarrow v$. Suppose that μ is a variable assignment that agrees with D. Then $M(D) \subseteq M(b) \cup \mu(\{y_{\#s} \in \mathcal{Y} \mid \#s \in Id(b)\})$.*

Proof. According to Lemma 15, $M(D) = M(b) \cup \{r \mid \mathsf{mark}_a(b_1, b_2)\#s \Downarrow r\#\mathsf{Pt}\ \text{occurs in}\ D\}$. Therefore we just have to show

$$\{r \mid \mathsf{mark}_a(b_1, b_2)\#s \Downarrow r\#\mathsf{Pt}\ \text{occurs in}\ D\} \subseteq \mu(\{y_{\#s} \in \mathcal{Y} \mid \#s \in \mathrm{Id}(b)\}).$$

So let r be in the left hand side. Then $\mathsf{mark}_a(b_1, b_2)\#s \Downarrow r\#\mathsf{Pt}$ occurs in D. We have $\#s \in \mathrm{Id}(b)$ and since μ agrees with D, we have that $\mu(y_{\#s}) = r\#\mathsf{Pt}$. Therefore $r\#\mathsf{Pt}$ is in the right hand side. This completes the argument. □

Lemma 17. *For any path b, $\mathtt{FV}(\rho(b)) \cap \mathcal{Y} \subseteq \mathtt{FV}(c(b)) \cap \mathcal{Y}$, and $\mathtt{FV}(c(b)) \cap \mathcal{Y} = \{y_{\#s} \in \mathcal{Y} \mid \#s \in Id(b)\}$.*

Proof. Straightforward. □

Lemma 18. *Let $\#\mathsf{Pt}(b)$ be the set of point values contained in $c(b)$ and $\rho(b)$. Then $\#\mathsf{Pt}(b) \subseteq M(b)$.*

Proof. This argument goes by induction on the structure of b and is straightforward. □

Proof (Theorem 10). ($\Rightarrow$) Suppose that $e \Downarrow_{\overline{V}} v$. By Proposition 9, there is $b \in B(e)$ such that $D : b \Downarrow_{\overline{V}} v$.

Let μ' be a variable assignment that agrees with D and let S be any piecewise uniform substitution on $\mathcal{Y}_b$ whose order is such that if $\mu'(y) < \mu'(y')$, then $y <_S y'$. By Lemma 16, $\mathcal{Y}_b = \{y_{\#s} \mid \#s \in \mathrm{Id}(b)\}$, so by Lemma 17, $M(D) \subseteq \mu'(\mathcal{Y}_b)$. Let $M = \mu'(\mathcal{Y}_b)$. By Theorem 5, there is a valuation set $\overline{U}$ that agrees with $\overline{V}$ on M and is easily replaceable on M. By Theorem 3, $D : b \Downarrow_{\overline{U}} v$. Now let μ be any assignment that agrees with μ' on all of $\mathcal{Y}_b$, but assigns $z_{a,y}$ for all $a \in \mathbb{A}$ and $y \in S$ such that $S \xrightarrow{\mu} \overline{U}$. This is possible since $\mathcal{Z}$ is disjoint from $\mathcal{Y}$ and for each a and y, there is a unique $z_{a,y} \in \mathcal{Z}$. Let $\mathcal{A} = \mathcal{A}_{\overline{U}}$. By Proposition 13, $[\![\rho(b)]\!]^{\mu}_{\mathcal{A}} = |v|$ and $\mathcal{A}, \mu \vDash c(b)$. By Proposition 11, $[\![\mathtt{R}(\rho(b))]\!]^{\mu}_{\mathcal{A}} = |v|$ and $\mathcal{A}, \mu \vDash \mathtt{R}(c(b))$. By Lemma 14, $\mathcal{A}, \mu \vDash \psi(S)$. Then we have

$$\mathcal{A}, \mu \vDash S(\mathtt{R}(c(b) \wedge \psi(S))) \qquad \text{(Theorem 6)}$$
$$\mathcal{A}, \mu \vDash S(\mathtt{R}(F\{\rho(b)/x\})) \qquad \text{(Equation (11))}$$
$$\mathcal{A} \vDash \mathtt{R}(F\{\rho(b)/x\}) \qquad \text{(Theorem 6)}$$
$$\mathcal{A} \vDash F\{\rho(b)/x\} \qquad \text{(Proposition 11)}$$

and by Lemma 10 as $[\![\rho(b)]\!]^{\mu}_{\mathcal{A}} = |v|$,

$$\mathcal{A} \vDash F\{\mathtt{v}/x\}.$$

Now $\cdot \vdash F\{\mathtt{v}/x\} : \mathrm{Formula}$ so that $\cdot \vdash F : \mathrm{Formula}$. Since F is well-formed, F contains no point values. Therefore all point values are contained within $\mathtt{v}$. This allows us to apply Theorem 4 to obtain

$$\mathcal{A}_{\overline{V}} \vDash F\{\mathtt{v}/x\}.$$

($\Leftarrow$) We approach this by assuming Equation (11) does not hold and constructing a piecewise uniform valuation set witness to $e \not\models F$. So suppose that Equation (11) does not hold. Observe that Equation (11) does not contain any $\mathtt{V}$ function symbols. Then there is some μ, $b \in B(e)$, and $S \in S_b$ such that $\mu \vDash S(\mathtt{R}(c(b))) \wedge \psi(S)$ yet $\mu \not\models F\{\rho(b)/x\}$. Write out $S = \{y_1, \ldots, y_n\}$ such that $y_1 <_S \cdots <_S y_n$. Consider the piece

$$[\mu(z_{a,y_1}), \mu(y_1)] \cup \cdots \cup [\mu(z_{a,y_n}), \mu(y_n)].$$

As $\mu \vDash \psi(S)$,

$$\mu(z_{a,y_1}) \leq \mu(y_1) \leq \cdots \leq \mu(z_{a,y_n}) \leq \mu(y_n). \tag{12}$$

Then let U_a be the piecewise uniform valuation on the above piece. Also observe by $\mu \vDash \psi(S)$, for any a, a',

$$\sum_{y \in S} \mu(y) - \mu(z_{a,y}) = \sum_{y \in S} \mu(y) - \mu(z_{a',y}).$$

Therefore we can set d to be the reciprocal of the above sum, and obtain that $d = c(U_a)$ for all a. Then $\overline{U} \triangleq (a \mapsto U_a \mid a \in \mathbb{A})$ is a piecewise uniform valuation set which is easily replaceable on $\mu(S)$. Set $\mathcal{A} = \mathcal{A}_U$.

We can verify that $S \xrightarrow{\mu} U$: (1) is satisfied directly above. (2) is also satisfied by definition. (3) is mildly more difficult. Let $y \in \mathcal{Y}_b$ be such that there is $y' <_S y$ and $\mu(y') = \mu(y)$. Then $\mu(y') \leq \mu(z_{a,y}) \leq \mu(y)$ hence $\mu(z_{a,y}) = \mu(y)$ for all $a \in \mathbb{A}$. (4) is verified by Equation (12) recalling that $y_1 <_S \cdots <_S y_n$.

Then by Theorem 6,

$$\mathcal{A}, \mu \vDash \mathtt{R}(c(b)) \quad \text{and} \quad \mathcal{A}, \mu \not\models \mathtt{R}(F\{\rho(b)/x\}).$$

By Proposition 11,

$$\mathcal{A}, \mu \vDash c(b) \quad \text{and} \quad \mathcal{A}, \mu \not\models F\{\rho(b)/x\}.$$

Therefore by Proposition 14, $b \Downarrow_{\overline{U}} v$ where $|v| = [\![\rho(b)]\!]^{\mu}_{\mathcal{A}}$. By Proposition 9, $e \Downarrow_{\overline{U}} v$. By Lemma 10, $\mathcal{A}, \mu \not\models F\{\mathtt{v}/x\}$. Since $F\{\mathtt{v}/x\}$ has no unbound variables, μ is redundant. Therefore $\mathcal{A} \not\models F\{\mathtt{v}/x\}$. This shows that $e \not\models F$. □

We also have the following generalization of Corollary 1

Corollary 3. *Let e be a well-formed protocol, only using standard operations, and let F a well-formed property. Checking if e satisfies F is decidable.*

7.13 Evaluation of protocols with envy

Non-Envy Free Protocols. To demonstrate the proficiency of our approach for finding envy, we deliberately incorporated errors into select protocols and time how long it takes to find envy, following methodology established by Lester [10]. The results, contained in Table 2, demonstrate that our approach can also efficiently find envy in all protocols tested.

Table 2: Time to find counterexamples to non-envy-free protocols.

Protocol	Time	Description
Cut-Choose	0.020	Agent 1 cuts and chooses.
Cut-Choose	0.021	Slices allocated wrong way round in one branch.
Surplus	0.035	Unsafe trim of slice smaller than reference.
Selfridge-Conway-Surplus	0.034	Allocates trimmings of slice instead of trimmed slice.
Selfridge-Conway-Surplus	0.033	Agent 2 not forced to take trimmed slice if available.
Selfridge-Conway-Full	0.161	Trimmings cut by agent who took trimmed slice.
Aziz-Mackenzie-3	2.594	Did not check if Agent 1 and 2 has the same favorite piece.