A Typed Language for Truthful One-Dimensional Mechanism Design

Lapets, Andrei


http://hdl.handle.net/2144/3754

Boston University
A Typed Language for Truthful One-Dimensional Mechanism Design

Andrei Lapets  
Computer Science Dept.,  
Boston University,  
Boston MA  
lapets@bu.edu

Alex Levin  
Massachusetts Institute of Technology,  
Cambridge MA  
levin@mit.edu

David C. Parkes  
SEAS,  
Harvard University,  
Cambridge MA  
parkes@eecs.harvard.edu

ABSTRACT

We first introduce a very simple typed language for expressing allocation algorithms that allows automatic verification that an algorithm is monotonic and therefore truthful. The analysis of truthfulness is accomplished using a syntax-directed transformation which constructs a proof of monotonicity based on an exhaustive critical-value analysis of the algorithm. We then define a more high-level, general-purpose programming language with typical constructs, such as those for defining recursive functions, along with primitives that match allocation algorithm combinators found in the work of Mu'alem and Nisan [10]. We demonstrate how this language can be used to combine both primitive and user-defined combinators, allowing it to capture a collection of basic truthful allocation algorithms. In addition to demonstrating the value of programming language design techniques in application to a specific domain, this work suggests a blueprint for interactive tools that can be used to teach the simple principles of truthful mechanism design.

1. INTRODUCTION

Mechanism design (MD) considers the situation of multi-agent systems with private information and self-interest. A common example of a problem in MD is presented by a combinatorial auction in which a set of items are to be allocated to bidders with values on subsets of the items. Such auctions find application in many domains, including for expressive sourcing of goods and logistics such as transportation, and for wireless spectrum allocation [2,13]. One is often interested in the design of truthful, or non-manipulable mechanisms so that the best strategy for participants is to report private information truthfully, regardless of the strategies and valuations of other bidders.

In designing truthful mechanisms, an especially well-studied environment is presented by one-dimensional settings, in which the private information of each participant is restricted to a single value that indicates his or her value for an "acceptable" allocation, already known to the auctioneer. This provides a well understood special case because there is an especially simple characterization for the allocation rules that can be implemented within a truthful mechanism. These are the monotonic allocation rules, wherein if a bidder is a winner for some reported value then he or she will also be a winner for all higher reported values.

This context of single-minded mechanism design provides the setting for our work on the automatic verification of truthfulness. In particular, we are inspired by the paper of Mu'alem and Nisan [10], who considered the use of modular design for the construction of mechanisms in the domain of known single-minded (KSM) combinatorial auctions. In a KSM auction, every bidder wants a particular bundle of goods and the bundle (but not the bidder's value) is known to the auctioneer. The authors define a small language of modular combinators for constructing truthful mechanisms, and characterize the conditions under which these combinators can be combined to form truthful allocation mechanisms. In doing so, they were able to construct polynomial-time truthful mechanisms with improved approximation ratios for several cases of KSM combinatorial auctions. However, these conditions had to be proven manually for various combinations by appealing to the properties of the basic combinators.

In this paper, we adopt techniques from programming language theory and programming language design to provide a straightforward means for defining allocation algorithms whose truthfulness can be verified automatically. This is done in two ways. First, we define a very simple programming language of deterministic, convergent allocation algorithms, and provide a method for automatically determining whether an algorithm in this language is monotonic. We then define a more high-level, general-purpose programming language with typical constructs, such as those for defining recursive functions, along with primitives that match some of the combinators for manipulating bids and outcomes found in Mu'alem and Nisan [10]. We use standard programming language techniques to demonstrate that for any algorithm written in this language, it is possible to automatically verify that the conditions under which the primitives are used and combined preserve truthfulness. This verification is done by inductively analyzing the definition of an algorithm, and keeping track of desirable algorithm characteristics. The language can be used to combine both primitive and user-defined combinators to define truthful allocation algorithms. We demonstrate this extensibility of the language, and the ease with which one can build a variety of
1.1 Related Work

There is a substantial amount of literature on domain-specific languages (DSLs) (for a general survey see [15]) that deals with both concrete examples of languages designed for a specific application domain (such as languages for modular interpreters and graphical constructs), as well as the fundamental characteristics that they might possess. Hudak [5] provides an overview of some embedded domain-specific languages that inherit a type system from their host language, and reviews several characteristics which make domain-specific languages valuable. One such characteristic relevant to our work is the declarative value of a DSL—a language does not need to be used merely as a way to compute a result [4]. It can be a way to communicate and describe the process used to obtain that result in a clear manner that leverages the contextual knowledge of an expert of the application domain. More importantly, when the description of a computation is viewed this way, it is possible to formally reason about the properties the computation might possess by analyzing the description, and to prove invariants about transformations on the description.

There is one striking example from the work of Jones et al. [8] in which these techniques have been used to establish formal properties in market contexts. The authors present a declarative language for describing financial and insurance contracts by defining a collection of typed combinators for building a large variety of contracts. The declarative nature of the language plays a crucial role in their work, and allows them to define an inductive algorithm for transforming contracts into functions that compute the possible values of those contracts as a function of time. In the first language we define, we take advantage of similar ideas to transform an allocation algorithm of reports about value and selects an outcome. Let $v_j \in Bid$ for bidder $j$ is a claim about the value $v_j \in \mathbb{R}_+$ the bidder has for an acceptable allocation.

An allocation algorithm $A(v) \in \mathcal{O}$ takes a vector $v = (v_1, \ldots, v_n)$ of reports about value and selects an outcome. Let $A_j(v) \in \{0, 1\}$ denote whether or not bidder $j$ is allocated in the outcome. As usual, let $v_{-i} = (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$.

Definition 1. An allocation algorithm $A$ is monotone when for any bidder $j$ and any vector of bids $v_{-j}$, if $A_j(v) = 1$ then we have $A_j(v'_{j}, v_{-j}) = 1$ for any $v'_{j} \geq v_j$.

For any monotonic allocation algorithm $A$, for any $v_{-j}$ there exists a single critical value for bidder $j$, denoted $\theta_j(v_{-j}) \in \mathbb{R} \cup \{\infty\}$, such that for all $v_j > \theta_j(v_{-j})$, bidding $v_j$ will lead to bidder $j$ being allocated, and for all $v_j < \theta_j(v_{-j})$, bidder $j$ is not allocated.

A mechanism couples an allocation algorithm $A$ with a payment algorithm $p(v) \in \mathbb{R}_+$ that determines the payment made by each bidder given bids $v$. We assume that bidders have quasi-linear utility, such that bid $j$ with value $v_j$ has utility $v_j A_j(v'_{j}, v_{-j}) - p_j(v'_{j}, v_{-j})$ given mechanism
We address this problem by considering critical intervals instead of individual thresholds, which allows us to verify a larger collection of monotonic algorithms. In fact, it is possible to determine automatically whether any algorithm in our simple language is monotonic.

3.1 Introducing an Abstract Syntax

We will consider a collection of simple algorithms that can be constructed using a language in which the conditions of branching if statements are allowed to depend only on bid values, and in which only a single bidder can be allocated at a time. Thus, these algorithms can only be used to represent single-item auction mechanisms.

A definition of a programming language consists of its syntax, a grammar (usually represented using BNF notation) describing syntactically correct programs, usually called expressions. We present the syntax for the language.

natural \( i \in \mathbb{N} \)

bid vector \( v \in \mathbb{R}^n \)

primitive \( p ::= \text{alloc} \mid \text{value} \)

expression \( e ::= i \mid v \mid p \)

\( | e_1 \ e_2 \mid \text{if} \ e_1 \geq e_2 \ \text{then} \ e_3 \ \text{else} \ e_4 \)

Naturals are used as indices into the bid vector. Note that the \( e_1 \ e_2 \) rule is partial application of a function in its curried form. To understand partial application, consider the primitives \( \text{value} \) and \( \text{alloc} \), that represent functions which take an index (a natural) and a bid vector, and output the value and desired outcome, respectively, of the bid under that index. These functions only return their results after they have been applied to both arguments in succession, e.g. \((\text{value} \ 1 \ v)\) is the \( \text{value} \) function applied first to 1, and then to \( v \), in that order. The result of this expression is then defined to be \( v_1 \), and in general,

\[
((\text{value} \ i) \ v) = v_i, \\
((\text{alloc} \ i) \ v) \in \mathcal{O}.
\]

We will typically simplify this to \( \text{value} \ 1 \ v \), so such sequences should be assumed to be left-associative, i.e. \( e_1 \ e_2 \ e_3 \) means \((e_1 \ e_2) \ e_3\).

We now present an example of an expression in this language, an algorithm that allocates to either the first or second bidder, depending on which of the two bid values is greater:

\[
\text{if \ value} \ 1 \ v \geq \text{value} \ 2 \ v \ \text{then alloc} \ 1 \ v \ \text{else alloc} \ 2 \ v
\]

Note that in its unrestricted form, the syntax does not prevent us from constructing an expression whose result is not defined (for example, by forgetting to supply arguments to \( \text{value} \)). It also does not prevent us from constructing an expression that defines a non-monotonic allocation algorithm (we could reverse the indices inside the if condition in the above example to obtain a non-monotonic algorithm). We will address the first issue by defining a type system, and the second by analyzing the critical interval functions corresponding to an expression.

3.2 Type System

The type of an expression is a symbol that represents the mathematical domain of that expression (for example, \( 2 \in \mathbb{N} \), so \( \mathbb{N} \) could be the type of the expression 2). A type
Bid Vector
If the example is a straightforward exercise. Then, the if for each of the four subexpressions of the statement must both also be of type \( O \), and its condition must be a well-defined application of the \( \text{value} \) primitive to two arguments. We can thus view the first case as the leaf of the decision tree, and the second as the node of the tree. ☐

While this type system ensures that well-typed expressions have some appropriate structure, it is still possible to construct non-monotonic allocation algorithms in this language. We address this issue below.

### 3.3 Critical Interval Functions

We can check the monotonicity of a well-typed expression in this simple language by transforming that expression into a critical interval function \( \theta_i \) for each bidder \( i \). This function \( \theta_i \) takes the other bidders’ values \( v_{-i} \) and produces a collection of intervals. The intervals indicate exactly the range of the bidder’s bid value \( v_i \) for which the bidder will be allocated. If for each bidder \( i \), for all bid vectors \( v_{-i} \), the collection of critical intervals is equivalent to a single threshold function, we know that the algorithm the expression represents must be monotonic.

The syntax of the language for critical interval functions is provided below. Note that the base primitives are intervals, and that set operators for union and intersection have been introduced.

\[
\begin{align*}
\text{natural } i & : \in \mathbb{N} \\
\text{primitives } p & : = v_i \mid \max p_1, p_2 \mid \min p_1, p_2 \\
\text{base interval } \varphi & : = [p_1, p_2] \mid (p_1, p_2) \mid [p_1, p_2] \\
\text{critical interval } \theta & : = \emptyset \mid \varphi \mid \theta_1 \cap \theta_2 \mid \theta_1 \cup \theta_2 \\
\text{if } v_i \geq v_j \text{ then } \theta_1 \text{ else } \theta_2
\end{align*}
\]

In order to isolate each possible collection of critical intervals at the leaves of the expression (and ensure that nodes consist only of if statements), we treat \( \cap \) and \( \cup \) as functions, and transform the critical intervals into normal forms by applying the familiar set equations in Figure 2. The notation \( e \downarrow e' \) indicates that the expression \( e \) evaluates to expression \( e' \), and \( c \) is shorthand notation for conditions. For brevity, we use only intervals of the form \( [p_1, p_2] \), though the rules apply to any combination of interval forms.

Most of the rules are familiar distribution laws and applications of identity and transitivity. The only rules worth noting are those that deal with if statements and allow for the distribution of both intersections and unions across the two branches of an if statement, e.g.

\[
\begin{align*}
\text{(if } c \text{ then } \theta_1 \text{ else } \theta_2) & \downarrow (\theta_1 \cup \theta) \downarrow \text{if } c \text{ then } (\theta_1 \cup \theta) \text{ else } (\theta_2 \cup \theta)
\end{align*}
\]

The syntax of the normal forms of critical intervals is as
indicate that the other bidders. In the inductive case, suppose that the lemma holds for the two branches \( \theta_1 \) and \( \theta_2 \).

We defne \( N \) inductively in Figure 3. In the base case \( N \) alloc. \( N \) produces a vector with one full critical interval corresponding to \( i \), indicating that bidder \( i \) will be allocated for any bid value in \([0, \infty)\), and empty critical intervals for the other bidders. In the inductive case \( N\)-if, the premises indicate that \( N \) makes recursive calls on the two branches \( e_1 \) and \( e_2 \), and obtains vectors of critical intervals for each. It then constructs new critical intervals for each bidder based on the condition, and returns the new vector \( (\theta'_1, \ldots, \theta'_n) \).

\[
\begin{align*}
\emptyset \cap \varphi & \downarrow 0 \\
\varphi \cap \emptyset & \downarrow 0 \\
([p_1, p'_1] \cap [p_2, p'_2]) & \downarrow \max(p_1 p_2, \min(p'_1 p'_2)) \\
(\theta_1 \cup \theta_2) \cap \emptyset & \downarrow (\theta_1 \cap \emptyset) \cup (\theta_2 \cap \emptyset) \\
\theta_1 \cap (\theta_1 \cup \theta_2) & \downarrow (\theta_1 \cap \theta_1) \cup (\theta_1 \cap \theta_2) \\
\text{if } c \text{ then } \theta_1 \text{ else } \theta_2 & \downarrow \text{if } c \text{ then } (\theta_1 \cap \theta_3) \text{ else } (\theta_2 \cap \theta_3) \\
\theta_3 \cap (\text{if } c \text{ then } \theta_1 \text{ else } \theta_2) & \downarrow \text{if } c \text{ then } (\theta_1 \cap \theta_3) \text{ else } (\theta_2 \cap \theta_3) \\
\emptyset \cup \theta & \downarrow \theta \\
\theta \cup \emptyset & \downarrow \theta \\
(\theta_1 \cup \theta_2) \cup \theta_3 & \downarrow \theta_1 \cup (\theta_2 \cup \theta_3) \\
\text{if } c \text{ then } \theta_1 \text{ else } \theta_2 & \downarrow \text{if } c \text{ then } (\theta_1 \cup \theta) \text{ else } (\theta_2 \cup \theta) \\
\theta \cup (\text{if } c \text{ then } \theta_1 \text{ else } \theta_2) & \downarrow \text{if } c \text{ then } (\theta_1 \cup \theta) \text{ else } (\theta \cup \theta_2)
\end{align*}
\]

\[\text{Figure 2: Set Reduction Equations}\]

According to the reduction rules for intersection and union, this would reduce to:

\([v_2, \infty)\),

which is exactly the range for \( v_1 \) that ensures allocation for bidder 1. We can compute the critical interval function for bidder 2 in a similar manner, and the result of applying \( N \) to our example would be:

\([\langle v_2, \infty \rangle, \langle v_1, \infty \rangle]\)

These are indeed the critical value intervals for bidders 1 and 2, as for any fixed \( v_2 \), the algorithm will allocate to bidder 1 for all bid values \( v_1 \in [v_2, \infty) \), and for any fixed \( v_1 \), it will allocate to bidder 2 for all bid values \( v_2 \in (v_1, \infty) \).

\[\text{Lemma 3. Suppose that } e \text{ is a well-typed expression of type } O, \text{ and } \theta_i \text{ is component } i \text{ of the vector of critical interval functions } N(e). \text{ Then for all } v_{-i}, \text{ if the value } v_i \text{ of the bid represented by } (\text{value } i b) \text{ is within one of the critical intervals described by } \theta_i v_{-i}, \text{ alloc } i b \text{ will the the outcome of the entire mechanism when applied to the vector } v.\]

\[\text{Proo. We will prove this by induction over the structure of the well-typed expression } e. \text{ In the base case, for any outcome in which bidder } i \text{ is allocated nothing, the interval is }, \text{ and for an outcome in which } i \text{ is allocated, the interval is the whole range of bid values } [0, \infty). \text{ For our inductive hypothesis, suppose that the lemma holds for the two subexpressions of the if statement. If the condition is independent of the bid value of bidder } i, \text{ we simply wrap the intervals obtained from the two subexpressions in this conditional expression:}

\[\theta_i'' = \text{if } v_j \geq v_k \text{ then } \theta_i \text{ else } \theta_i'.\]

Given \( v_{-i} \), we know that one of the branches will be chosen, for each of which the lemma holds by our inductive hypothesis. If the condition does depend on the bid value of bidder \( i \), assume without loss of generality that it is of the form \( v_i \geq v_j \). We know by our inductive hypothesis that \( v_i \in \theta_i \), iff bidder \( i \) is allocated, so if \( v_i \in [v_j, \infty) \), bidder \( i \) is allocated iff \( v_i \in (\theta_i \cap [v_j, \infty)) \). Likewise, if \( v_i \in [0, v_j) \), bidder \( i \) is allocated iff \( v_i \in \theta_i \cap [0, v_j) \). We return the union of these two intervals, \( (\theta_i \cap [v_j, \infty)) \cup (\theta_i \cap [0, v_j)) \), which accounts for both possible ranges for \( v_i \), and the lemma holds,
3.5 Checking for Monotonicity

In order to check that a critical interval function can be represented by an equivalent critical value function, we define a predicate Mon. To accomplish this, we define a context. A context is any suitable representation of a strict order on reals and a partial order on bid values $v_i$. We denote a context by $G$ and an empty context by $\emptyset$. We also assume there exists a function $\delta$ that takes a context and a condition, and extends the partial order on bids. Particularly, $\delta$ can extend any $G$ with an ordering on two bids, $G, \delta(G, v_j) \vdash v_i \geq v_j$, and this should be done in a way that ensures that reflexivity and transitivity are enforced:

$$\forall G, G \vdash v_i \geq v_j \quad G \vdash v_i \geq v_j \quad G \vdash v_j \geq v_k$$

We now define Mon recursively over a critical interval function:

- $\text{MON-}\emptyset \quad G \vdash \emptyset : \text{Mon}$
- $\text{MON-}\varphi \quad G \vdash q = \infty \quad G \vdash [p, q) : \text{Mon}$
- $\text{MON-}\chi \quad G \vdash \exists \ell \; q_{\ell} = \infty \land \forall j \neq \ell \exists j' \; g_j \geq p_{j'} \quad G \vdash [p_{\ell}, q_{\ell}) \cup \ldots \cup [p_{x}, q_{x}) : \text{Mon}$
- $\text{MON-}\Phi \quad G \vdash \text{if } c \text{ then } \theta_1 \text{ else } \theta_2 : \text{Mon}$

Lemma 4. A critical interval $\theta$ satisfies Mon if and only if the interval it describes is connected and infinite on the right (and thus, is equivalent to a critical value function).

Proof. The cases for a single interval and an empty interval are trivial. Each of the non-empty collections of intervals must satisfy two conditions. First, we must be able to conclude that for every interval $[p_1, p_2)$ there exists some interval $[p'_1, p'_2)$ such that given the partial ordering on bids $G$, we can conclude that $p_2 \geq p'_1$. Second, there must exist at least one interval of the form $[p, \infty)$. It is evident that both are satisfied if and only if there exists a single threshold for the space of bids that ensures allocation.

For the if case, we observe that so long as Mon holds for the two subexpressions under their respective extended contexts, every base case list of intervals for these two subexpressions must have a critical value, so Mon must hold for the entire expression. □

Example 3. For instance, consider a different example in which a critical interval function is of the following form:

$$\text{if } v_1 \geq v_2 \text{ then } [4, v_1) \cup [v_2, \infty) \text{ else } [4, v_2) \cup [v_1, \infty).$$

We see that the first branch, $v_1 \geq v_2$ implies that the right-hand border of the interval $[4, v_1)$ overlaps the left-hand border of $[v_2, \infty)$, and because there is an interval $[v_2, \infty)$ which goes on to infinity. We see that a similar fact is true about the second branch under the condition $v_2 > v_1$. Thus, Mon holds for this critical interval function.

Theorem 2. For any well-typed expression $e$, the algorithm that $e$ represents is monotonic if and only if every component $\theta_i$ in the vector of critical interval functions $N(e)$ satisfies Mon.

Proof. We know from Lemma 4 that Mon holds if and only if there exists a critical value function for each one of the bidders representing the minimum bid value that will ensure allocation. Thus, we know it represents an allocation algorithm that is monotonic in every bid value. □

3.6 Issues with Generalizing the Approach

Suppose we want to extend the approach to a more expressive language, so that we may describe a larger collection of allocation algorithms. While the translation algorithm $N$ runs in polynomial time for the limited collection of expressions we have considered, extending this approach to a more expressive language would require the analysis of an exponentially large number of contexts. For example, suppose $e_1$ and $e_2$ are expressions, $v_1$ through $v_4$ are bid values, and maxval returns the maximum of two values, and that we have an expression of the form below.

$$\text{if } (\text{maxval } (v_1, v_2) \geq \text{maxval } (v_3, v_4)) \text{ then } e_1 \text{ else } e_2$$

There are 2 possible outcomes for each max expression, which makes for 4 possible combinations for the condition. Thus, the intervals generated for the expression $e_1$ must be checked under contexts in which $v_1 \geq v_2 \geq v_3 \geq v_4$, $v_1 \geq v_2 \geq v_4 \geq v_3$, and $v_2 \geq v_1 \geq v_3 \geq v_4$. We can observe that in general, given $k$ maxval expressions in a condition, we would have up to $2^k$ contexts under which to check the subexpression’s intervals.

Suppose we want to extend this approach to domains in which there are multiple items to allocate. It would then make sense to have conditions in which sums of bid values may occur. This presents another complication, as contexts would then need to encode relations of the form $v_1 + v_2 \geq v_3$. 

as the union of these two intersections is exactly the set of intervals on which bidder $i$ would be allocated. □
4. AN EXTENDED LANGUAGE

In order to allow a greater variety of interesting algorithms, and to allow algorithms that apply to multiple item settings, we modify the language in two ways. First, we make it more general by introducing recursion, functions, and variables. Second, because the complications described in the previous section occur only when conditions depend on bid values, we use types to prevent well-typed conditions from depending on bid values. To compensate for this limitation, we introduce special primitives that are similar to the combinators presented in the work by Mu'alem and Nisan [10]. We assign appropriate types to these primitives, and augment the type system so that an expression is well-typed only if the primitives are used in ways that preserve monotonicity. We also demonstrate that the type system allows us to extend the language with new primitive combinators simply by providing appropriate types for those combinators, which makes it possible to use the language to define an even larger collection of monotonic allocation algorithms, including algorithms for multiple-item settings.

Because we are using special primitives with appropriate types, it is no longer necessary to generate explicit witnesses (i.e. critical value functions) to prove an algorithm is monotonic. However, the pricing functions can still be found for any polynomial-time monotonic allocation algorithm using binary search in polynomial time [10]. We will assume that the algorithms written in this language cannot explicitly construct a bid or bid vector. An outcome can only be constructed using one of a set of primitive functions: alloc takes a bid and produces an outcome where only that bid is satisfied; max takes two outcomes and returns the one with higher welfare. We consider a simple expression equivalent to our previous example:

$$\text{max (alloc 1 v) (alloc 2 v)}$$

This algorithm allocates to either the first or the second bidder in the bid vector v, depending on whose bid is greater. Thanks to the introduction of functions and recursion, we can also construct a generalization of this example, which allocates to the highest bidder out of n bids:

$$\text{maxAll = arg v . arg n . fix (arg f . arg i . \begin{array}{l} \text{if } (>= i n) \text{ then} \\ \text{alloc i v} \\ \text{else} \\ \text{max (alloc i v) (f (+ 1 i))} \end{array})}$$

The function maxAll takes the bid vector v, and applies the fixpoint of the function in the outermost parentheses to 1. The function inside the parentheses simply takes a counter and checks whether the counter has reached its limit n. If it has not, it applies max to the outcome that the bid indexed by the counter specifies, and the result of the function call on the rest of the vector.

4.2 Type System

We now present the syntax of type expressions for the language. The only extension is the introduction of proposition annotations for base types that indicate whether a computed result of a type depends monotonically on at least one bid value. Booleans, naturals, and reals can either be independent of all bid values, or can depend monotonically on a bid value. We assume that indices and reals have the usual ordering relation, and that True ≥ False. Outcomes might also depend bitonically on a bid value, and thus we include one additional annotation for outcomes, Biton.

- proposition $P ::= \text{Mon} \mid \text{Indep}$
- outcome prop. $Q ::= P \mid \text{Biton}$
- base type $\varsigma ::= \text{Bool}_P \mid \mathbb{N}_P \mid \mathbb{R}_P \mid V \mid \mathcal{O}_Q$
- type $\tau ::= \varsigma \mid \tau \rightarrow \tau$

Primitive operators can only be applied to certain kinds of bid-dependent values, and their result types propagate these dependencies. We assign types to all the primitives:

- alloc : $\mathbb{N}_\text{Indep} \rightarrow V \rightarrow \mathcal{O}_\text{Indep}$
- value : $\mathbb{N}_\text{Indep} \rightarrow V \rightarrow \mathbb{R}_\text{Mon}$
- max : $\mathcal{O}_\text{Biton} \rightarrow \mathcal{O}_\text{Biton} \rightarrow \mathcal{O}_\text{Biton}$
- + : $\mathbb{N}_\text{Indep} \rightarrow \mathbb{N}_\text{Indep} \rightarrow \mathbb{N}_\text{Indep}$
- ≥ : $\mathbb{N}_\text{Indep} \rightarrow \mathbb{N}_\text{Indep} \rightarrow \text{Bool}_\text{Indep}$
- or : $\text{Bool}_\text{Indep} \rightarrow \text{Bool}_\text{Indep} \rightarrow \text{Bool}_\text{Indep}$

We assume that there exists a function $T$ mapping primitives to their types. In presenting the typing rules, starting with the primitives, we employ $\Gamma$, a list of variables and the types to which they are bound, to keep track of variable
types in an expression: 1

\[
\text{Primitive} \quad \frac{T(p) = \tau}{\Gamma \vdash p : \tau} \quad \text{Bool} \quad \frac{e \in \{\text{True, False}\}}{\Gamma \vdash e : \text{Bool}_{\text{Indep}}}
\]

\[
\text{Num} \quad \frac{s = \mathbb{N} \text{ if } |r| = r, \quad s = \mathbb{R} \text{ if } w}{\Gamma \vdash r : \text{Indep}} \quad \text{Bids} \quad \frac{v : V}{\Gamma \vdash v : V}
\]

Any natural can be a real number. Also, any independent expression is obviously bitonic, and any bitonic expression is by definition monotonic (thus, \(\text{Indep} \subseteq \text{Biton} \subseteq \text{Mon}\)).

We provide the rules that establish these relationships:

\[
\text{Ind} \Rightarrow \text{Bit} \quad \frac{\Gamma \vdash e : \text{Indep}}{\Gamma \vdash e : \text{Biton}} \quad \text{Bit} \Rightarrow \text{Mon} \quad \frac{\Gamma \vdash e : \text{Biton}}{\Gamma \vdash e : \text{Mon}}
\]

Finally, we provide the typing rules for the remaining language constructs. Note that in the \(\text{IF} \) rule, we have assumed that the boolean expression used for branching must be independent of bid values. The \(\text{Var} \) rules state that under an environment in which \(\Gamma\) extended with the variable \(x\) of type \(\tau\), \(x\) has type \(\tau\), and the \(\text{Fun} \) rule indicates that if an expression has type \(\tau_2\) under an environment in which \(x\) has type \(\tau_1\), the function which takes \(x\) as an argument has type \(\tau_1 \rightarrow \tau_2\).

\[
\text{Var} \quad \frac{\Gamma \vdash x : \tau \vdash x : \tau}{\Gamma \vdash \text{fix } e : \tau} \quad \text{Fix} \quad \frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \text{fix } e : \tau}
\]

\[
\text{Fun} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x.e : \tau_1 \rightarrow \tau_2}
\]

\[
\text{App} \quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{app } e_1 \; e_2 : \tau_2}
\]

\[
\text{If} \quad \frac{\Gamma \vdash e_1 : \text{Bool}_{\text{Indep}} \quad \Gamma \vdash e_2 : \tau' \quad \Gamma \vdash e_3 : \tau'}{\Gamma \vdash \text{if } e_1 \; \text{then } e_2 \; \text{else } e_3 : \tau'}
\]

Because recursion is allowed in this language, one major cause for concern is divergence (i.e. non-termination). If a winning bidder could cause an algorithm to diverge by raising its bid, the mechanism would no longer be monotonic.

**Lemma 5.** Any well-typed algorithm that diverges for any specific vector of bid values will diverge for all possible vectors of bid values.

**Proof.** Because the conditions inside if expressions must be independent of bid values, there is no way a bid value can influence how an if expression is evaluated. The only way a bid value can influence how an expression is evaluated is in an occurrence of the \(\text{max} \) operator. However, both argument expressions for \(\text{max} \) must always be evaluated fully before their overall values can be determined and compared. Thus, \(\text{max} \) diverges only if at least one of its arguments diverges. This means that even if changing a bid would change which of its two arguments \(\text{max} \) selects as its result, this could not affect whether or not an algorithm diverges.

It is possible to construct a diverging algorithm of any type, and it is not clear how the type should be interpreted in such a case. However, the interpretation of diverging algorithms is not our main focus, and because we have established that algorithms that diverge do so for all bid value vectors, we can restrict our main result to algorithms that do not diverge.

**Theorem 3.** Any non-diverging well-typed expression of type \(Q\), where \(\varsigma\) is a base type and \(Q\) is one of the three possible propositions, represents an algorithm whose result is obtained in a manner that corresponds to the proposition \(Q\) (for example, if \(Q = \text{Mon}\), the result the algorithm chooses to return depends monotonically on the bid values).

**Proof.** We argue by induction over the type derivation for such an expression.

It is obvious that real and boolean constants are trivially independent. The operators \(+, \geq, \text{and or} \) return independent results when applied to independent arguments. For \(\text{alloc}\), an outcome that allocates a single independently present bid specified by an independent index is trivially independent. The \(\text{value} \) operator produces a real value that is indeed monotonic in at least one bid value (even if it is independent in all others). The \(\text{max} \) operator can be applied only to two bitonic outcomes, and itself produces a bitonic outcome for such arguments (Theorem 3 in [10]).

We observe that the subtyping rules relating \(\text{Indep}, \text{Mon}, \text{and Biton} \) only allow promotion from an independent type, or from a bitonic to a monotonic type. Obviously, any independent algorithm is both bitonic and monotonic, and any bitonic algorithm is monotonic.

Finally, observe that the rules \(\text{Var}, \text{Fix}, \text{Fun}, \) and \(\text{App} \) cannot be used to modify the proposition governing a type, so as long as all subexpressions have appropriate annotations, any combination of expressions built according to any of these rules will also have an appropriate annotation. In the case of \(\text{App} \), note that the type \(\tau_2\) of the second expression must match the expected argument of the function of type \(\tau_2 \rightarrow \tau_1\), so if a function such as \(\text{max} \) expects an expression of type \(\text{Biton}\), \(\text{max} \) can only be applied to such an expression. Finally, the \(\text{IF} \) rule only allows a boolean condition that is independent. Assuming the two branches have appropriate annotations, the type of the entire \(\text{if} \) expression will have an appropriate annotation as well.

The type annotations can only express coarse characteristics about values (e.g. independence from all variables, monotonicity in one variable), so the type inference rules reject some algorithms that are monotonic.

**Remark 1.** There exist expressions \(e\) that are monotonic allocation algorithms, but do not type check.

**Example 4.** We can consider our previous method for representing the two-bidder Vickrey auction:

\[
\text{if } \geq (\text{value } 1 \text{ v}) (\text{value } 2 \text{ v}) \text{ then alloc } 3 \text{ v else alloc } 4 \text{ v, that is not well-typed because the arguments of } \geq \text{ must be of type } \mathbb{R}_{\text{Indep}}, \text{ while here, they are of type } \mathbb{R}_{\text{Mon}}.
\]

However, recall that we can represent the two-bidder Vickrey auction if we modify the description appropriately:

\[
\text{max} (\text{alloc } 1 \text{ v}) (\text{alloc } 2 \text{ v})
\]

This example illustrates that in order to verify the truthfulness of some algorithms, it is necessary to rewrite them in an appropriate form. However, there exist algorithms which cannot be represented.
Example 5. Consider the following algorithm which allocates to either bidder 3 or bidder 4, depending on the bids of bidders 1 and 2:

$$s f \geq (\text{value } 1 v)(\text{value } 2 v) \text{ then alloc } 3 v \text{ else alloc } 4 v$$

For the same reason as the previous example, this algorithm is not well-typed.

In general, thresholds cannot be computed explicitly for each bidder, because it is not possible to check all execution paths of a recursive algorithm. This is the quod pro quo for not needing to check the exponentially many execution paths found in most interesting algorithms; the simple type annotations allow the type inference algorithm to make only a single pass over the expression.

Nonetheless, the simple language of decision trees defined in Section 3 does allow an algorithm to depend on values of specific bidders, and we may want to recover this expressive power in at least a limited way. We can do this by adding a single type inference rule, which ensures that all of the well-typed, monotonic expressions from the language of decision trees are assigned the type $V \rightarrow \mathcal{O}_{\text{Mon}}$.

**Decision Tree**

$$N(e) \downarrow (\theta_1, \ldots, \theta_n) \quad \forall i, e \vdash \theta_i : \text{Mon}$$

This addition will not lead to a violation of Lemma 5 because the decision trees always converge and it will not break Theorem 3 because any such $e$ is indeed monotonic by Theorem 2.

### 4.3 More Primitives and Example Algorithms

The definition of the language is easy to extend with additional primitives, so long as we assign appropriate types to these primitives that ensure that both Lemma 5 and Theorem 3 do not break. To demonstrate how one can extend the language in such a manner, we provide a collection of example extensions that allow us to encode other allocation algorithms.

First, assume that bidders are bidding for one item of which there is an unlimited supply, so there is no need to worry about infeasible outcomes or collisions between bidders' interesting sets. In our language, bidders who are not allocated in a bitonic outcome do not have an effect on welfare, and the sum of two such bitonic welfare functions is still bitonic. Thus, we can generalize the type of `combine` under these conditions.

```
maxallk = \arg \forall i v n d k o.
  if (or (\geq k d) (\geq n i)) then
    max
      \text{// try remaining branches 'i' up to 'n'}
      (maxallk v (+ i 1) n d k o)
    \text{// add to accumulator, go deeper in the tree}
    \text{// note that 'i' is reset back to '1'}
    (maxallk v i n (+ d 1) k (combine (alloc i v) o))
  else
    if (feasible o) then o else noalloc
```

To compute the maximum-welfare outcome $\text{Exst}_k$ for a bid vector of length at most $n$, we can use the expression below:

$$\text{Exst}_k k v = \text{maxallk } v 1 n 1 k \text{ noalloc}$$

#### 4.3.2 Profit Extraction

Next, assume that bidders are bidding for one item of which there is an unlimited supply, so there is no need to worry about infeasible outcomes or collisions between bidders' interesting sets. In our language, bidders who are not allocated in a bitonic outcome do not have an effect on welfare, and the sum of two such bitonic welfare functions is still bitonic. Thus, we can generalize the type of `combine` under these conditions.

```
combine : \mathcal{O}_{\text{Biton}} \rightarrow \mathcal{O}_{\text{Biton}} \rightarrow \mathcal{O}_{\text{Biton}}
```

We also add `thresh`, which takes any bitonic outcome and returns an empty allocation if and only if the welfare of that outcome is below a specified threshold.

```
thresh : \mathcal{R}_{\text{Indep}} \rightarrow \mathcal{O}_{\text{Biton}} \rightarrow \mathcal{O}_{\text{Biton}}
```

It is clear that `thresh` converges so long as its arguments converge, and adding a threshold based on welfare to a bitonic outcome results in a bitonic outcome.

We can now construct a variation of the profit extraction mechanism from the work of Goldberg et al. [3], which assumes an unlimited supply of an item is available for allocation. Note that if $v$ is a vector of $n$ bids, `filter k v n 1` is an allocation to only the bidders who are willing to pay at least $k$:

```
filter = \arg \forall v n i.
  if (\geq n i) then
    combine (thresh k (alloc i v))
    \text{// note that 'i' is reset back to '1'}
    \text{// add to accumulator, go deeper in the tree}
  else
    if (feasible o) then o else noalloc
```

The first primitive is an outcome in which no bidder is allocated. The second combines two independent outcomes (in each of which some number of bidders might be allocated), and combines them. The third primitive return `True` if an outcome does not allocate any item to more than one bidder, and `False` otherwise. None of these violate Lemma 5 and Theorem 3, as all take independent arguments and return independent results, and this is reflected in their types.

Mu'alem and Nisan [10] define an allocation algorithm $\text{Exst}_k$ that exhaustively iterates over all feasible outcomes in which up to $k$ bidders are allocated, returning the outcome with highest welfare. Its result can be encoded as the maximum with respect to welfare over a collection of individual feasible outcomes, and we can use this observation and implement the $\text{Exst}_k$ algorithm. The function `maxallk` uses an argument $o$ to accumulate allocations. The best way to understand this recursion is to think of a tree of depth $k$ where $d \in [1, k]$ with a branching factor of $n$ where $i \in [1, n]$. Each branch represents a choice of bidder to allocate at that node, and each path to a leaf represents an allocation to $k$ bidders.
Thus, we can let $k = i^2$ for different $i$ and take the maximum over these possibilities. If $k$ exceeds some maximum cut-off, we simply return the empty allocation.

$$\text{profitExtract} = \arg \max R v n k \maxk.$$  
\begin{align*}
\text{if } (\geq \maxk k) & \text{ then} \\
\max (\text{thresh } R (\text{filter } k v n 1)) & (\text{profitExtract } R v n (+ k k) \maxk) \\
\text{else} & \text{noalloc}
\end{align*}

For each $k$, the only way an outcome will be considered is if at least $R/k$ bidders have value above $k$. Thus, we have effectively expressed the requirement that the $R/k$th highest bidder must have a bid value above $k$.

5. CONCLUSIONS

We have shown that it is possible to obtain critical interval functions for a limited collection of well-typed allocation algorithms by using a syntax-directed transformation, and that these can be used to determine whether the algorithm is monotonic. We have also shown how a type system with annotations can benefit a more general language of primitive combinators, and that it is even possible to introduce variables and recursion (using the fixpoint operator) into such a language of combinators while still preserving our ability to verify the monotonicity of some of the algorithms written in the language. We illustrated that it is possible to find collections of basic combinators that allow us to express other allocation algorithms, and we have seen how assigning lightweight, high-level type annotations to these primitive combinators can provide a model of the necessary domain-specific invariants and dependencies in our application domain, making it easy to manage complex relationships in a modular way when assembling algorithms.

The simple type system (even with annotations) and language constructs we presented could be embedded without much difficulty in any typed functional programming language. This ability to embed a domain-specific language in existing systems, as well as the ability to combine modular units in provably correct and appropriate ways, is in the spirit of functional programming and domain-specific language design [5].

If it were possible to define the primitives we have added to our language as algorithms in some other formal logic or proof system that allows the combination of automatically verifiable proofs with function definitions, we could represent the fact that the type annotations of those primitives are accurate, and could avoid manually justifying their types. However, even our existing approach is reasonably flexible if a satisfactory collection of primitive combinators can be established for a particular application domain, or if sufficient conditions can be established on what kinds of primitives can be defined.

One issue that we have already mentioned is the way in which divergence is interpreted for an allocation algorithm. It seems quite reasonable to expect that when writing algorithms in an expressive programming language that allows recursion, a programmer or designer will need to deal with divergence on some inputs. It would be interesting to see whether type-based termination checking [1] can be incorporated into the language, thus eliminating completely the need to consider divergence.

6. REFERENCES

APPENDIX

A. TYPE DERIVATION EXAMPLE

First, note the expression syntax for the exhaustive search allocation algorithm, with no syntactic sugar:

```plaintext
fix (arg f arg v arg i arg n arg d arg k arg o.
if (or (>= k d) (>= n i)) then
  max
  // try remaining branches 'i' up to 'n'
  (fv (+ i 1) n d k o)
  // add to accumulator, go deeper in the tree
  // note that 'i' is reset back to '1'
  (fv 1 n (+ d 1) k (combine (alloc i v) o))
else
  if (feasible o) then o else noalloc
)
```

In Figure 4 we present a type derivation for the body of the exhaustive search allocation algorithm, which has type $O_{\text{Biton}}$. Each base case is numbered, and in addition, each inductive case is labelled on the left-hand side of the => symbol with the name of the inference rule being applied, as well as with the numbers corresponding to the sub-expressions upon which the inference is based.
Figure 4: Partial type derivation