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

.

allocation algorithms without manually proving their truthfulness, by providing as examples the Exst_k algorithm of Mu'alem and Nisan [10] and the profit extraction technique of Goldberg et al. [3].

More broadly, these results demonstrate how certain kinds of programming techniques and programming language features can be applied effectively to a specific domain. This work also provides a ready blueprint for interactive tools that can be used to teach the principles of economic mechanism design.

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 algorithm description into a collection of functions that can then be analyzed in isolation, and relate that analysis back to a property of the original algorithm. Our second language can be viewed as a narrow, domain-specific *logic* that can be embedded in a pre-existing language for verifying pure functional programs, such as the calculus of constructions [11].

The analysis in the first language we present is related to the approach taken in work by Tadjouddine and Guerin [14]. There, the `Alloy` modeling language [7], which is based on first order relational logic [6], is used to model two simple auctions (a two-bidder Vickrey auction and a quantity-restricted multi-unit auction). Then, the model checker is used to prove assertions regarding the truthfulness of these two auctions, which is done by negating those assertions and exhaustively searching a finite, non-exhaustive space of possible scenarios (in terms of the number of bidders, the number of items, and the number of values) to find a counterexample to the negation. This does not constitute a proof of truthfulness, but rather suggests a strong likelihood that because the algorithm is truthful for a finite collection of test cases, it will be truthful in all other cases. While the verification technique we introduce for our first language similarly involves an exhaustive analysis, the analysis is over the collection of all possible evaluations of an algorithm description, and generates a complete proof of truthfulness if and only if such a proof exists. To make this analysis efficient, we restrict the kinds of algorithms our first language can describe, though the language is expressive enough to capture an n-bidder Vickrey auction for any fixed n. Furthermore, both languages we define do not need to be translated into a separate modeling language such as `Alloy`, because our analyses are performed by directly transforming descriptions of algorithms. For a mechanism designer, this obviates the need to learn a separate modeling language, and like any effective DSL, a specialized language for the purpose of mechanism design can more easily take into consideration the specific knowledge and goals of the designer.

2. PRELIMINARIES

The required terminology and notational conventions from programming language theory related to type theory, inference rules and syntax will be introduced as needed. The ideas that are adopted here can be found in most standard texts on formal concepts of programming languages; see for instance Pierce [12]. For those familiar with the simply-typed *lambda calculus* [9], it may be worth noting that the simply-typed lambda calculus is a subset of the general-purpose language we define in a later section.

For now, we review some terminology, relevant definitions, and results for mechanism design in the context of *one-dimensional* private information.

We consider allocation domains. Each bidder $j \in \{1, \ldots, n\}$ has private information $v_j \in \mathbb{R}_+$ about its *value* for an acceptable allocation. The set of acceptable allocations to the bidder is known to the mechanism. An outcome $o \in \mathcal{O}$ is a description of the allocation, with $o_j \in \{0, 1\}$ to indicate whether or not bidder j is satisfied (and has value) for the allocation. For instance in a single-item auction, then $o_j = 1$ if and only if bidder j receives the item in the outcome of the mechanism.

DEFINITION 1. *A* bid $\hat{v}_j \in$ `Bid` *for bidder j is a claim about the value $\hat{v}_j \in \mathbb{R}_+$ the bidder has for an acceptable allocation.*

An *allocation algorithm* $A(v) \in \mathcal{O}$ takes a vector $v = \langle v_1, \ldots, v_n \rangle$ 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 $v_{-i} = \langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \rangle$.

DEFINITION 2. *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 $p(v) \in \mathbb{R}^n_+$ that determines the payment made by each bidder given bids v. We assume that bidders have quasi-linear utility, such that bidder j with value v_j has utility $v_j A_j(v'_j, v_{-j}) - p_j(v'_j, v_{-j})$ given mechanism

$M = (A, p)$ and bids (v'_j, v_{-j}). Our focus in this paper is on truthful mechanisms:

DEFINITION 3. *Mechanism $M = (A, p)$ is truthful when*

$$v_j A_j(v_j, v_{-j}) - p_j(v_j, v_{-j}) \geq v_j A_j(v'_j, v_{-j}) - p_j(v'_j, v_{-j}), \quad (1)$$

for all v_j, all $v'_j \neq v_j$, and all v_{-j}.

In a truthful mechanism it is a dominant-strategy equilibrium for each bidder to report his or her true (private) value for an acceptable outcome. A simple characterization for truthful allocation mechanisms is available in the one-dimensional domains considered in this paper:

THEOREM 1. *Let a mechanism $M = (A, p)$ be such that every bidder that does not receive an acceptable allocation pays nothing. Then M is truthful if and only if its allocation algorithm is monotone and it collects the critical value for bidder j from any bidder j that receives an acceptable allocation.*

Mu'alem and Nisan [10] also introduce the additional property of *bitonicity*, stronger than monotonicity, which is useful for checking whether the combination of algorithms forms a mechanism that is truthful. Let $w_A(v) = \sum_j A_j(v)v_j$ denote the *welfare* from allocation algorithm A, i.e. the total value of the allocation.

DEFINITION 4. *[10] An allocation algorithm A is **bitonic** if it is monotone, and if for every bidder j and any v_{-j}, the welfare function $w_A(v_{-j}, v_j)$ is a non-increasing function of v_j for $v_j < \theta_j(v_{-j})$, and a non-decreasing function of v_j for $v_j \geq \theta_j(v_{-j})$.*

3. A SIMPLE LANGUAGE FOR MONOTONIC ALLOCATION ALGORITHMS

Verifying the truthfulness of an allocation algorithm in a one-dimensional domain can be reduced to the problem of verifying that the algorithm is monotonic in all of its bid values. Let us suppose that there exists a language of combinators or other constructs that can be combined, making it possible to inductively assemble allocation algorithms. One way to verify the monotonicity of such algorithms is to attempt to inductively construct the critical value threshold *functions* (which are themselves represented for each bidder i by an *algorithm* parameterized by the bid value vector v_{-i}) corresponding to the allocation algorithms. If such functions be constructed for every bidder, the algorithm is monotonic. However, a significant limitation with such an approach is that some monotonic algorithms are rejected because the approach requires that every component of an algorithm is itself monotonic under *all* contexts. This is not required for monotonicity of the combined algorithm because one of the components may only be used in a limited number of contexts in which it is monotonic:

EXAMPLE 1. *Consider an algorithm A which allocates to bidder 1 if $v_1 \in [0, 1] \cup [2, \infty)$. Now, suppose a larger algorithm A' only calls A if $v_1 > 2$, and otherwise, allocates to no one. Clearly, A' is monotonic, but if we first checked the components of A' for monotonicity, we would find that A is not monotonic, and would reject A'.*

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	::=	`alloc` | `value`
expression	e	::=	i | v | p
		|	$e_1\ e_2$ | `if` $e_1 \geq e_2$ `then` e_3 `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 `value` and `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. ((`value` 1) v) is the `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,

$$\begin{aligned} ((\texttt{value}\ i)\ v) &= v_i, \\ ((\texttt{alloc}\ i)\ v) &\in \mathcal{O}. \end{aligned}$$

We will typically simplify this to `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:

`if value` 1 $v \geq$ `value` 2 v `then alloc` 1 v `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 `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*

$$\textsc{App}\ \dfrac{\textsc{App}\ \dfrac{\textsc{Value}\ \dfrac{}{\vdash \texttt{value} : \mathbb{N} \to V \to \mathbb{R}} \quad \textsc{Natural}\ \dfrac{1 \in \mathbb{N}}{\vdash 1 : \mathbb{N}}}{\vdash \texttt{value}\ 1 : V \to \mathbb{R}} \quad \textsc{Bid Vector}\ \dfrac{v \in \mathbb{R}^n}{\vdash v : V}}{\vdash \texttt{value}\ 1\ v : \mathbb{R}}$$

Figure 1: Example of a Derivation

system restricts the space of syntactically correct expressions to the subset of expressions that can be assigned a type. The system consists of a syntax for types, and a collection of *inference rules* that can be used to assign types to expressions inductively. A syntax of types is as defined below, where $\mathcal{O}$ represents the space of outcomes, V represents the space of bid vectors, and $\tau \to \tau'$ represents a function that takes arguments of type τ and returns results of type τ'.

$$\begin{aligned} \text{base type } \varsigma &::= \mathbb{N} \mid \mathbb{R} \mid \mathcal{O} \mid V \\ \text{type } \tau &::= \varsigma \mid \tau \to \tau \end{aligned}$$

An inference rule consists of a collection of *premises* above the line, and a *judgment* below the line. The turnstile operator $\vdash$ can be viewed as an assertion that the judgment which follows it can be proven. We present two example inference rules that assign types to the `value` and `alloc` primitives.

$$\textsc{Value}\ \frac{}{\vdash \texttt{value} : \mathbb{N} \to V \to \mathbb{R}}$$

$$\textsc{Alloc}\ \frac{}{\vdash \texttt{alloc} : \mathbb{N} \to V \to \mathcal{O}}$$

Note that there are no premises above the line, because these are primitives. We now present the remaining rules. We will adopt the convention that premises above the line should be read from left to right, and from top to bottom.

$$\textsc{Bid Vector}\ \frac{v \in \mathbb{R}^n}{\vdash v : V} \quad \textsc{Natural}\ \frac{i \in \mathbb{N}}{\vdash i : \mathbb{N}}$$

$$\textsc{If}\ \frac{\vdash e_1 : \mathbb{R} \quad \vdash e_2 : \mathbb{R} \quad \vdash e_3 : \mathcal{O} \quad \vdash e_4 : \mathcal{O}}{\vdash \texttt{if}\ e_1 \geq e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 : \mathcal{O}}$$

$$\textsc{App}\ \frac{\vdash e_1 : \tau_2 \to \tau_1 \quad \vdash e_2 : \tau_2}{\vdash e_1\ e_2 : \tau_1}$$

Definition 5. *A syntactically correct expression e is* well-typed *if there exists a type derivation for e built using the type inference rules.*

Let us consider our earlier example:

`if value` $1\ v \geq$ `value` $2\ v$ `then alloc` $1\ v$ `else alloc` $2\ v$

As illustrated in Figure 1, the derivation for the subexpression `value` 1 is an instance of the App rule applied to instances of the Value and Index rules, and the subexpression `value` $1\ v$ is another application of the App rule, along with the Bid Vector rule. Constructing such a derivation for each of the four subexpressions of the `if` statement in the example is a straightforward exercise. Then, the If rule can be applied to the four derivations to establish a type for the entire example.

Lemma 1. *Any non-trivial expression e that is well-typed according to the above type system and has type $\mathcal{O}$ will necessarily be a decision tree with comparisons between bid values at its nodes and outcomes at its leaves.*

Proof. Let us consider every possible expression e that has a derivation that concludes $e : \mathcal{O}$. According to the type rules, we see that one such derivation would occur for an expression which is an application of the `alloc` primitive to its two arguments. The only other possibility is that the expression is an `if` statement, and the inference rule for the `if` statement requires that the two branches of the `if` statement must both also be of type $\mathcal{O}$, and its condition must be a well-defined application of the `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 θ_i for each bidder i. This function θ_i takes the other bidders' values v_{-i} and produces a collection of intervals. The intervals indicate exactly the ranges 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.

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

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. $(\texttt{if}\ c\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2) \cup \theta \Downarrow \texttt{if}\ c\ \texttt{then}\ (\theta_1 \cup \theta)\ \texttt{else}\ (\theta_2 \cup \theta)$.

The syntax of the normal forms of critical intervals is as

$$
\begin{array}{rcl}
\emptyset \cap \varphi & \Downarrow & \emptyset \\
\varphi \cap \emptyset & \Downarrow & \emptyset \\
([p_1, p_1') \cap [p_2, p_2')) & \Downarrow & [\max p_1 p_2, \min p_1' p_2') \\
(\theta_1 \cup \theta_2) \cap \theta_3 & \Downarrow & (\theta_1 \cap \theta_3) \cup (\theta_2 \cap \theta_3) \\
\theta_3 \cap (\theta_1 \cup \theta_2) & \Downarrow & (\theta_3 \cap \theta_1) \cup (\theta_3 \cap \theta_2) \\
(\texttt{if } c \texttt{ then } \theta_1 \texttt{ else } \theta_2) \cap \theta_3 & \Downarrow & \texttt{if } c \texttt{ then } (\theta_1 \cap \theta_3) \texttt{ else } (\theta_2 \cap \theta_3) \\
\theta_3 \cap (\texttt{if } c \texttt{ then } \theta_1 \texttt{ else } \theta_2) & \Downarrow & \texttt{if } c \texttt{ then } (\theta_3 \cap \theta_1) \texttt{ else } (\theta_3 \cap \theta_2)
\end{array}
$$

$$
\begin{array}{rcl}
\emptyset \cup \theta & \Downarrow & \theta \\
\theta \cup \emptyset & \Downarrow & \theta \\
(\theta_1 \cup \theta_2) \cup \theta_3 & \Downarrow & \theta_1 \cup (\theta_2 \cup \theta_3) \\
(\texttt{if } c \texttt{ then } \theta_1 \texttt{ else } \theta_2) \cup \theta & \Downarrow & \texttt{if } c \texttt{ then } (\theta_1 \cup \theta) \texttt{ else } (\theta_2 \cup \theta) \\
\theta \cup (\texttt{if } c \texttt{ then } \theta_1 \texttt{ else } \theta_2) & \Downarrow & \texttt{if } c \texttt{ then } (\theta \cup \theta_1) \texttt{ else } (\theta \cup \theta_2)
\end{array}
$$

Figure 2: Set Reduction Equations

follows. Note that i, p, ϕ, and c are defined as previously.

$$
\begin{array}{rrcl}
\text{interval union} & \chi & ::= & \varphi \mid \varphi \cup \chi \\
\text{critical interval} & \theta & ::= & \emptyset \mid \chi \mid \texttt{if } c \texttt{ then } \theta_1 \texttt{ else } \theta_2
\end{array}
$$

We can view θ as a set of decision trees with unions of intervals at the leaves (a leaf is either $\emptyset$ or some list of intervals χ), and with conditions at the nodes (an `if` statement is a node with two branches θ_1 and θ_2).

LEMMA 2. *Every critical interval function can be converted into a normal form.*

PROOF. We observe that $\cap$ can be eliminated completely by the first set of equations. The only two non-trivial cases then become `if` and $\cup$. The second set of rules ensures that there are no occurrences of `if` inside a $\cup$ expression. Finally, because all expressions with $\cup$ are converted into right-associative form, they must form chains of intervals. □

3.4 Obtaining Critical Interval Functions

We now define an algorithm N that recursively transforms a well-typed expression of type $\mathcal{O}$ into a critical interval function. We observe that a well-typed expression of type $\mathcal{O}$ must either be `alloc` applied to two arguments, or an `if` expression that wraps two subexpressions, both of type $\mathcal{O}$.

We define 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 $\langle \theta_1'', \ldots, \theta_n'' \rangle$.

EXAMPLE 2. *Suppose we apply N to our simple example.*

if value 1 $v \geq$ ***value*** 2 v ***then alloc*** 1 v ***else alloc*** 2 v

Let us compute only the critical interval function for bidder 1*. The first branch of the* ***if*** *statement would yield a full interval, $[0, \infty)$. The second branch would yield the empty interval, $\emptyset$. The condition indicates that these two intervals would be combined to obtain:*

$$([0, \infty) \cap [v_2, \infty)) \cup (\emptyset \cap [0, v_2)).$$

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:*

$$\langle [v_2, \infty), (v_1, \infty) \rangle.$$

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 $v_2 \in (v_1, \infty)$.*

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

PROOF. We will prove this by induction over the structure of the well-typed expression e. In the base case, for any outcome in which bidder i is allocated nothing, the interval is , and for an outcome in which i is allocated, the interval is the whole range of bid values $[0, \infty)$. 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, we simply wrap the intervals obtained from the two subexpressions in this conditional expression:

$$\theta_i'' = \texttt{if } v_j \geq v_k \texttt{ then } \theta_i \texttt{ else } \theta_i'.$$

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,

$$\text{N-Alloc}\ \frac{}{N((\texttt{alloc}\ i)\ b) \Downarrow \langle \emptyset, \ldots, \emptyset, \underbrace{[0,\infty)}_{\theta_i}, \emptyset, \ldots, \emptyset \rangle}$$

$$\text{N-If}\ \frac{\begin{array}{c} N(e_1) \Downarrow \langle \theta_1, \ldots, \theta_n \rangle \\ N(e_2) \Downarrow \langle \theta'_1, \ldots, \theta'_n \rangle \\ \theta''_i = (\theta_i \cap [v_j, \infty)) \cup (\theta'_i \cap [0, v_j)) \\ \theta''_j = (\theta'_j \cap [v_i, \infty)) \cup (\theta_j \cap [0, v_i)) \\ \forall k \text{ s.t. } k \neq i \wedge k \neq j,\ \theta''_k = \texttt{if}\ v_i \geq v_j\ \texttt{then}\ \theta_k\ \texttt{else}\ \theta'_k \end{array}}{N(\texttt{if}\ ((\texttt{value}\ i)\ b) \geq ((\texttt{value}\ j)\ b)\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Downarrow \langle \theta''_1, \ldots, \theta''_n \rangle}$$

Figure 3: Definition of the Algorithm N

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

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 $\bullet$. We also assume there exists a function δ that takes a context and a condition, and extends the partial order on bids. Particularly, δ can extend any G with an ordering on two bids,

$$\forall G,\ \delta(G, v_i \geq v_j) \vdash v_i \geq v_j,$$

and this should be done in a way that ensures that reflexivity and transitivity are enforced:

$$\forall G, G \vdash v_i \geq v_i \qquad \frac{G \vdash v_i \geq v_j \qquad G \vdash v_j \geq v_k}{G \vdash v_i \geq v_k}$$

We now define Mon recursively over a critical interval function:

$$\text{Mon-}\emptyset\ \frac{}{G \vdash \emptyset : \text{Mon}} \qquad \text{Mon-}\varphi\ \frac{G \vdash q = \infty}{G \vdash [p, q) : \text{Mon}}$$

$$\text{Mon-}\chi\ \frac{G \vdash \exists \ell\ q_\ell = \infty \wedge \forall j \neq \ell\ \exists j'\ q_j \geq p_{j'}}{G \vdash [p_1, q_1) \cup \ldots \cup [p_k, q_k) : \text{Mon}}$$

$$\text{Mon-If}\ \frac{\delta(G, c) \vdash \theta_1 : \text{Mon} \qquad \delta(G, \neg c) \vdash \theta_2 : \text{Mon}}{G \vdash \texttt{if}\ c\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2 : \text{Mon}}$$

Lemma 4. *A critical interval θ 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:*

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

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 context $v_2 > v_1$. 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 θ_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.

$$\texttt{if}\ (\texttt{maxval}\ (v1, v2) \geq \texttt{maxval}\ (v3, v4))\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

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 $v_1 \geq v_2 \geq v_3 \geq v_4$, $v_1 \geq v_2 \geq v_4 \geq v_3$, $v_2 \geq v_1 \geq v_3 \geq v_4$, and $v_2 \geq v_1 \geq v_4 \geq v_3$. 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 $v_1 + v_2 \geq v_3$

(for example, to represent an algorithm in which either both the first and second bidders are allocated, or only the third bidder is 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 this is the method used to calculate payment functions for allocation algorithms.

4.1 Abstract Syntax

We extend the language by adding higher-order functions (i.e. λ abstractions, denoted using `arg` $x.e$, which are used to represent a function that takes an argument, binds it to x, and evaluates e) and a *fixpoint* operator for recursion. We introduce a restriction, as well, by no longer allowing bid values to appear inside `if` conditions. The only way to represent algorithms that depend on bid values is to use special primitives, with types used to reject any use of these primitive combinators that violates the monotonicity of an allocation algorithm. In order to illustrate this in a simple manner, we will begin with a language that is extended only with the `max` primitive, corresponding to the welfare MAX operator from Mu'alem and Nisan [10], which takes two outcomes and returns the one with greater welfare. This language is not yet expressive enough to define algorithms for multi-unit domains, but later in the section, we will show how adding a few simple primitives removes this limitation.

$$
\begin{array}{rrcl}
 & x & \in & Vars \\
\text{boolean} & b & ::= & \texttt{True} \mid \texttt{False} \\
 & r & \in & \mathbb{R} \\
 & i & \in & \mathbb{N} \\
\text{bid vector} & v & \in & \mathbb{R}^n \\
\text{primitive} & p & ::= & \texttt{alloc} \mid \texttt{value} \mid \texttt{max} \mid \texttt{+} \mid \geq \mid \texttt{or} \\
\text{expression} & e & ::= & b \mid r \mid i \mid v \mid p \\
 & & \mid & e_1\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & & \mid & x \mid \texttt{arg}\ x.e \mid \texttt{fix}
\end{array}
$$

Note 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:

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

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:

```
maxAll = arg v . arg n .
  fix (
    arg f . arg i .
      if (>= i n) then
        alloc i v
      else
        max (alloc i v) (f (+ 1 i))
  ) 1
```

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 $\texttt{True} \geq \texttt{False}$. Outcomes might also depend bitonically on a bid value, and thus we include one additional annotation for outcomes, Biton.

$$
\begin{array}{rrcl}
\text{proposition} & P & ::= & \text{Mon} \mid \text{Indep} \\
\text{outcome prop.} & Q & ::= & P \mid \text{Biton} \\
\text{base type} & \varsigma & ::= & \texttt{Bool}_P \mid \mathbb{N}_P \mid \mathbb{R}_P \mid V \mid \mathcal{O}_Q \\
\text{type} & \tau & ::= & \varsigma \mid \tau \to \tau
\end{array}
$$

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.

$$
\begin{array}{rcl}
\texttt{alloc} & : & \mathbb{N}_{\text{Indep}} \to V \to \mathcal{O}_{\text{Indep}} \\
\texttt{value} & : & \mathbb{N}_{\text{Indep}} \to V \to \mathbb{R}_{\text{Mon}} \\
\texttt{max} & : & \mathcal{O}_{\text{Biton}} \to \mathcal{O}_{\text{Biton}} \to \mathcal{O}_{\text{Biton}} \\
\texttt{+} & : & \mathbb{N}_{\text{Indep}} \to \mathbb{N}_{\text{Indep}} \to \mathbb{N}_{\text{Indep}} \\
\geq & : & \mathbb{N}_{\text{Indep}} \to \mathbb{N}_{\text{Indep}} \to \texttt{Bool}_{\text{Indep}} \\
\texttt{or} & : & \texttt{Bool}_{\text{Indep}} \to \texttt{Bool}_{\text{Indep}} \to \texttt{Bool}_{\text{Indep}}
\end{array}
$$

We assume that there exists a function T mapping primitives to their types. In presenting the typing rules, starting with the primitives, we employ Γ, a list of variables and the types to which they are bound, to keep track of variable

types in an expression:[1]

$$\textsc{Primitive}\ \frac{T(p) = \tau}{\Gamma \vdash p : \tau} \qquad \textsc{Bool}\ \frac{e \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash e : \texttt{Bool}_{\text{Indep}}}$$

$$\textsc{Num}\ \frac{\varsigma = \mathbb{N} \text{ if} \lfloor r \rfloor = r,\ \varsigma = \mathbb{R} \text{ o/w}}{\Gamma \vdash r : \varsigma_{\text{Indep}}} \qquad \textsc{Bids}\ \frac{}{\Gamma \vdash v : V}$$

Any natural can be a real number. Also, any independent expression is obviously bitonic, and any bitonic expression is by definition monotonic (thus, $\mathcal{O}_{\text{Indep}} \subset \mathcal{O}_{\text{Biton}} \subset \mathcal{O}_{\text{Mon}}$). We provide the rules that establish these relationships:

$$\textsc{Ind}{\Rightarrow}\textsc{Bit}\ \frac{\Gamma \vdash e : \varsigma_{\text{Indep}}}{\Gamma \vdash e : \varsigma_{\text{Biton}}} \qquad \textsc{Bit}{\Rightarrow}\textsc{Mon}\ \frac{\Gamma \vdash e : \varsigma_{\text{Biton}}}{\Gamma \vdash e : \varsigma_{\text{Mon}}}$$

$$\mathbb{N} \subset \mathbb{R}\ \frac{\Gamma \vdash e : \mathbb{N}_P}{\Gamma \vdash e : \mathbb{R}_P}$$

Finally, we provide the typing rules for the remaining language constructs. Note that in the If rule, we have assumed that the boolean expression used for branching must be independent of a bid value. The Var rules states that under an environment Γ extended with the variable x of type τ, x has type τ, and the Fun rule indicates that if an expression has type τ_2 under an environment in which x has type τ_1, the function which takes x as an argument has type $\tau_1 \to \tau_2$.

$$\textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Fix}\ \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{fix}\ e : \tau}$$

$$\textsc{Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{arg}\ x.e : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\textsc{If}\ \frac{\Gamma \vdash e_1 : \texttt{Bool}_{\text{Indep}} \qquad \Gamma \vdash e_2 : \tau' \qquad \Gamma \vdash e_3 : \tau'}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau'}$$

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 `max` operator. However, both argument expressions for `max` must always be evaluated fully before their overall values can be determined and compared. Thus, `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 `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 ς_Q, where ς 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 `+`, $\geq$, and `or` return independent results when applied to independent arguments. For `alloc`, an outcome that allocates a single independently present bid specified by an independent index is trivially independent. The `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 `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 Indep, Mon, 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 Var, Fix, Fun, and 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 App, note that the type τ_2 of the second expression must match the expected argument of the function of type $\tau_2 \to \tau_1$, so if a function such as `max` expects an expression of type $\mathcal{O}_{\text{Biton}}$, `max` can only be applied to such an expression. Finally, the If rule only allows a boolean condition that is independent. Assuming the two branches have appropriate annotations, the type of the entire `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:*

$$\texttt{if}\ \geq (\texttt{value}\ 1\ v)(\texttt{value}\ 2\ v)\ \texttt{then alloc}\ 3\ v\ \texttt{else alloc}\ 4\ v,$$

that is not well-typed because the arguments of $\geq$ must be of type $\mathbb{R}_{\text{Indep}}$, 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:

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

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.

[1] The sole purpose of Γ is to make it possible to provide types for expressions which bind values to variables, and in which bound variables occur.

EXAMPLE 5. *Consider the following algorithm which allocates to either bidder* 3 *or bidder* 4*, depending on the bids of bidders* 1 *and* 2*:*

$$\texttt{if}\ \geq(\texttt{value}\,1\,v)(\texttt{value}\,2\,v)\ \texttt{then alloc}\,3\,v\ \texttt{else alloc}\,4\,v$$

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

In general, a 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 *quid 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 $V \to \mathcal{O}_{\text{Mon}}$.

$$\textsc{Decision Tree}\ \frac{N(e) \Downarrow (\theta_1, \ldots, \theta_n) \qquad \forall i, \bullet \vdash \theta_i : \text{Mon}}{\Gamma \vdash e : V \to \mathcal{O}_{\text{Mon}}}$$

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 presented in the literature by Mu'alem and Nisan [10] and Goldberg et al. [3]. The simplicity of the primitives makes it relatively easy to verify the accuracy of their type annotations, and this can be done either manually or with the help of some other formal logic or proof system that enables automatic verification. The type system can then be used to organize these components into more complex algorithms whose monotonicity can be verified automatically by using the inference rules presented in the previous section.

To improve legibility in the examples, we will adopt syntactic sugar for multiple arguments (e.g. `arg x1 x2 x3. e`) and will write recursive functions in closed form, omitting the `fix` operator.

4.3.1 Exhaustive Search

If we want to allow the creation of some algorithms for a multiple-item setting, we can add a few relevant primitives and give them the types below.

$$\begin{aligned}
\texttt{noalloc} &: \mathcal{O}_{\text{Indep}} \\
\texttt{combine} &: \mathcal{O}_{\text{Indep}} \to \mathcal{O}_{\text{Indep}} \to \mathcal{O}_{\text{Indep}} \\
\texttt{feasible} &: \mathcal{O}_{\text{Indep}} \to \texttt{Bool}_{\text{Indep}}
\end{aligned}$$

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 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 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.

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

To compute the maximum-welfare outcome Exst_k for a bid vector of length at most n, we can use the expression below:

```
Exst_k k v = maxAllk v 1 n 1 k 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.

$$\texttt{combine} \ : \ \mathcal{O}_{\text{Biton}} \to \mathcal{O}_{\text{Biton}} \to \mathcal{O}_{\text{Biton}}$$

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.

$$\texttt{thresh} \ : \ \mathbb{R}_{\text{Indep}} \to \mathcal{O}_{\text{Biton}} \to \mathcal{O}_{\text{Biton}}$$

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 k v n i.
  if (>= n i) then
    combine (thresh k (alloc i v))
            (filter k v n (+ i 1))
  else
    noalloc
```

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.

```
profitExtract = arg R v n k maxk.
  if (>= maxK k) then
    max (thresh R (filter k v n 1))
        (profitExtract R v n (+ k k) maxk)
  else
    noalloc
```

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/kth 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

[1] G. Barthe, M. Frade, E. Gimenez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions, 2003.

[2] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. MIT Press, January 2006.

[3] A. Goldberg, J. Hartline, A. Karlin, M. Saks, and A. Wright. Competitive auctions. *Games and Economic Behavior*, 55:242–269, 2006.

[4] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[5] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[6] D. Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM.

[7] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[8] S. P. Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[9] R. Loader. Notes on simply typed lambda calculus. Technical report, LFCS, University of Edinburgh, February 1998.

[10] A. Mu'alem and N. Nisan. Truthful approximation mechanisms for restricted combinatorial auctions. In *Proc. 18th National Conference on Artificial Intelligence (AAAI-02)*, 2002.

[11] C. Parent-Vigouroux. Verifying programs in the calculus of inductive constructions. *Formal Aspects of Computing*, 9(5-6):484–517, 1997.

[12] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[13] T. Sandholm. Expressive commerce and its application to sourcing: How we conducted $35 billion of generalized combinatorial auctions. *AI Magazine*, pages 45–58, 2007.

[14] E. M. Tadjouddine and F. Guerin. Verifying dominant strategy equilibria in auctions. In H.-D. Burkhard, G. Lindemann, R. Verbrugge, and L. Z. Varga, editors, *CEEMAS*, volume 4696 of *Lecture Notes in Computer Science*, pages 288–297. Springer, 2007.

[15] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

APPENDIX

A. TYPE DERIVATION EXAMPLE

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

```
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'
     (f v (+ i 1) n d k o)
     // add to accumulator, go deeper in the tree
     // note that 'i' is reset back to '1'
     (f v 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 $\mathcal{O}_{\text{Biton}}$. 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.

```
                    (1) k                                    : N_Indep
                    (2) d                                    : N_Indep
                    (3) >=                                   : N_Indep -> N_Indep -> Bool_Indep
     App (3)  (1) =>  (4) >= k                               : N_Indep -> Bool_Indep
     App (4)  (2) =>  (5) >= k d                             : Bool_Indep
                    (6) n                                    : N_Indep
                    (7) i                                    : N_Indep
     App (3)  (6) =>  (8) >= n                               : N_Indep -> Bool_Indep
     App (8)  (7) =>  (9) >= n i                             : Bool_Indep
                   (10) or                                   : Bool_Indep -> Bool_Indep -> Bool_Indep
     App (10) (5) => (11) or (>= k d)                        : Bool_Indep -> Bool_Indep
     App (11) (9) => (12) or (>= k d) (>= n i)               : Bool_Indep
                   (13) max                                  : O_Biton -> O_Biton -> O_Biton
                   (14) f                                    : V -> N_Indep -> N_Indep -> N_Indep
                                                               -> N_Indep -> O_Biton -> O_Biton
                   (15) v                                    : V
                   (16) 1                                    : N_Indep
                   (17) +                                    : N_Indep -> N_Indep -> N_Indep
    App (17)  (7) => (18) + i                                : N_Indep -> N_Indep
    App (18) (16) => (19) + i 1                              : N_Indep
                   (20) o                                    : O_Indep
    App (14) (15) => (21) f v                                : N_Indep -> N_Indep -> N_Indep -> N_Indep
                                                                -> O_Indep -> O_Biton
    App (21) (19) => (22) f v (+ i 1)                        : N_Indep -> N_Indep -> N_Indep -> O_Indep
                                                                -> O_Biton
    App (22)  (6) => (23) f v (+ i 1) n                      : N_Indep -> N_Indep -> O_Indep -> O_Biton
    App (23)  (2) => (24) f v (+ i 1) n d                    : N_Indep -> O_Indep -> O_Biton
    App (24)  (1) => (25) f v (+ i 1) n d k                  : O_Indep -> O_Biton
    App (25) (20) => (26) f v (+ i 1) n d k o                : O_Biton
    App (17)  (2) => (27) + d                                : N_Indep -> N_Indep
    App (27) (16) => (28) + d 1                              : N_Indep
                   (29) alloc                                : N_Indep -> V -> O_Indep
    App (29)  (7) => (30) alloc i                            : V -> O_Indep
    App (30) (15) => (31) alloc i v                          : O_Indep
                   (32) combine                              : O_Indep -> O_Indep -> O_Indep
    App (32) (31) => (33) combine (alloc i v)                : O_Indep -> O_Indep
    App (33) (20) => (34) combine (alloc i v) o              : O_Indep
    App (21) (16) => (35) f v 1                              : N_Indep -> N_Indep -> N_Indep -> O_Indep
                                                                -> O_Biton
    App (35)  (6) => (36) f v 1 n                            : N_Indep -> N_Indep -> O_Indep -> O_Biton
    App (36) (28) => (37) f v 1 n (+ d 1)                    : N_Indep -> O_Indep -> O_Biton
    App (37)  (1) => (38) f v 1 n (+ d 1) k                  : O_Indep -> O_Biton
    App (38) (34) => (39) f v 1 n (+ d 1) k
                          (combine (alloc i v) o)            : O_Biton

    App (13) (26) => (40) max
                           (f v (+ i 1) n d k o)              : O_Biton -> O_Biton

    App (40) (39) => (41) max
                          (f v (+ i 1) n d k o)
                          (f v 1 n (+ d 1) k
                              (combine (alloc i v) o))       : O_Biton

                   (42) feasible                             : O_Indep -> Bool_Indep
    App (42) (20) => (43) feasible o                         : Bool_Indep
                   (44) noalloc                              : O_Indep
If (43) (20) (44) => (45) if (feasible o) then o else noalloc : O_Indep
    Ind=>Bit (45) => (46) if (feasible o) then o else noalloc : O_Biton

If (12) (41) (46) => (47) if (or (>= k d) (>= n i)) then
                          max
                              (f v (+ i 1) n d k o)
                              (f v 1 n (+ d 1) k
                                 (combine (alloc i v) o))
                      else
                          if (feasible o) then o
                                          else noalloc       : O_Biton
```

Figure 4: Partial type derivation