On Formal Semantics and Analysis of Typed Modeling Languages: An Analysis of Ascend

Hemant K. Bhargava *
Code SM-BH
Naval Postgraduate School
Monterey CA 93943
bhargava@nps.navy.mil

Ramayya Krishnan
The Heinz School
Carnegie Mellon University
Pittsburgh PA 15213
rk2x+@andrew.cmu.edu

Peter Piela
Aspen Technology
piela@aspentec.com

Abstract

The use of strong typing, exemplified in the Ascend modeling language, is a recent phenomenon in executable modeling languages for mathematical modeling. It is also one that has significant potential for improving the functionality of computer-based modeling environments. Besides being a strongly typed language, Ascend is unique in providing operators that allow dynamic type inference, a feature that has been shown to be useful in assisting model evolution and reuse. We develop formal semantics for the type system in Ascend—focusing on these operators—and analyze its mathematical and computational properties. We show that despite the strong interactions between various statements involving the operators, the language does possess certain desirable mathematical and computational properties. Further, our analysis identifies general issues in the design and implementation of type systems in mathematical modeling languages. The methods used in the paper are applicable beyond Ascend to a class of typed modeling languages that may be developed in the future.

* *INFORMS Journal on Computing.* File: ascend-joc-1996.tex. Last revised February 2, 1997.
Correspondence to H.K. Bhargava. Phone: (408) 656-2264.

On Formal Semantics and Analysis of Typed Modeling Languages: An Analysis of Ascend

Executable modeling languages (EMLs [18]) have been instrumental in the development of modern modeling environments [21]. The use of strong typing is a recent phenomenon in such languages, but one that is likely to grow in importance in the future [54]. In this paper, we describe a method for the formal analysis of a class of typed modeling languages. These are languages which implement first-order type systems with inclusion subtyping [15] and which provide operators for dynamic type inference (see §1.3). We explain our method by developing the semantics for, and analyzing the properties of, the type system used in one such fully implemented language, *Ascend* [50].

The main objectives of the work described here are twofold. First, in the context of Ascend itself, the formal specification and analysis of the semantics of its type system allows us to establish representational as well as computational properties of the language. They should also enable users to understand precisely the implications of statements made in the language, and guide designers in making improvements to the language. In this context, it is important to note that certain aspects of Ascend's type system, not found in other modeling languages and little discussed in the model management literature, have been shown to be particularly useful in assisting model reuse [39]. Second, we hope that by providing an approach for the rigorous analysis of a type system designed to assist modeling, this paper will guide research on major design issues and trade-offs peculiar to the use, current or future, of typing in modeling languages.

We elaborate on both of these objectives, their relevance to modeling environments, and related work in type systems and type inference, in §1. We describe the essential features of Ascend in §2 with the objective of illustrating representation features of its underlying type system. In §3.1 we precisely state what we mean by formalizing the type system in Ascend, and in §3.2, we lay out our method for doing so. This involves partitioning Ascend into an object-level (corresponding to a class-based language) and a meta-level language. We describe the formal semantics of the Ascend type system in detail in §4 and analyze its properties in §5. The implications of these results on the design and implementation of typed modeling languages are discussed in §6.

1 Modeling Languages: Typing and Semantics

In the last two decades a number of declarative executable modeling languages have been developed to facilitate mathematical modeling. These include general-purpose languages for mathematical programming modeling (e.g., GAMS [10], AMPL [19], MODLER [28]), languages for mathematical modeling in specific domains (e.g., PM* [38] for production planning), graph-based modeling languages based on graph grammars [34, 35], a modeling language (SML [24, 25]) to support the *structured modeling* [20] framework, mathematical extensions to database languages (e.g., SQLMP [16] extends SQL) and languages rooted in formal logic (e.g., $L^{\uparrow}$ and $L_{\downarrow}$ [7, 5]).

1.1 Trends in Modeling Languages

The earlier EMLs focused on representing model algebra—the information necessary to solve the model. These languages classified objects and expressions into such general categories as variables, parameters, constraints, and objective functions [54]. This categorization enabled certain simple kinds of semantic validation (e.g., a variable used in the objective function must appear in a constraint). Later research, aiming to support much more than model representation and solution, has led to the use of deeper categorization—or typing—of objects, and a broader range of reasoning over this categorization. Bradley and Clemence [14, 13] describe a type calculus (involving dimensional information about variables) for modeling languages and its application in model validation and integration. Bhargava et al. [9] type variables via their quiddities—a formal representation of semantic information—and discuss its use in model integration. Muhanna describes the use of data typing to validate interconnected models [43]. SML [24, 25], the leading implementation of Geoffrion's structured modeling

framework [20], significantly exploits the advantages that strong typing[1] provides in determining various sorts of errors or inconsistencies in models. Finally, a type system in which type determines model structure forms the core of the strongly typed modeling language Ascend [50].[2] Models in Ascend (see §2) define *types*, and variables in the language may have multiple associated types, many of them inferred. Further, Ascend has type manipulation operators which enhance the reusability and integration of models [39].

A second noticeable trend in modeling languages is in the area of formal semantics of these languages. Since expressions in the earlier EMLs closely mirrored conventional algebraic expressions, these languages automatically had a commonly understood interpretation. The semantic restrictions required to enforce model correctness were rarely formalized [54]. However, as EMLs are extended with sublanguages for indexed expressions (see [24, 25]), dimensional analysis [14, 4]) and structured data modeling [8, 54], the importance of a formal semantic specification has increased. Regarding indexed expressions, Neustadter [45] has addressed the formal semantics of algebraic expression sublanguages using a generic algebraic modeling language. Hong and Mannino [32] provide denotational semantics of their modeling language L_U, which is based on measurement theory. Vicuna [54] uses attribute grammars to specify declaratively the semantic restrictions in SML. However, little has been done to date on the semantics of modeling-oriented type systems—such as Ascend's—which allow objects to have multiple types and provide operators that employ inference to manipulate types, and in which important attributes of a model depend on the type of its variables.

1.2 Focus and Motivation

This paper falls at the convergence of these two recent trends in executable modeling languages: the incorporation of methods for *typing*, and the formalization of declarative *semantics* of these languages. EMLs make it easier for modelers to represent, verify, debug, and solve mathematical models. It is agreed that, to make further progress in supporting the modeling lifecycle, EMLs must be extended to capture various kinds of qualitative information about models, which can then be used by inference procedures to provide additional functionality in modeling systems [7]. Recent research, cited above, indeed provides evidence that such extensions are consistent with a stronger role for typing and methods of type inference. The lack of a semantic formalization, particularly for typed languages, severely limits the error detection capabilities that modeling languages can provide to facilitate correct formulation of models [54].

By developing and analyzing the semantics of one particular typed modeling language, Ascend, we hope to make a contribution both to the development of typed modeling languages, as well as to a formal analysis of these languages. Specifically, in the context of Ascend, our method explicates the semantics of the language in an implementation-independent way. While Ascend has a well-developed *operational* semantics, we believe there are three advantages in carefully developing the declarative semantics.

First, our formal description of Ascend (§4) specifies precisely the semantics of a fully implemented and distinctive typed modeling language. In the absence of such semantics, one must appeal to intuition to understand the effects of various statements in Ascend; this can be non-trivial and confusing. Second, it describes the Ascend language in a notation and formalism that is well-understood and widely used for language design and specification. Third, it permits a principled investigation of extensions to the language, and suggests directions for improving or extending the language.

Why choose *Ascend*—rather than a generic typed language—to demonstrate our method? Because Ascend is a useful, working, modeling language which has a significant user group in industry. Because it is the only modeling language which incorporates both strong typing and operators that can alter, dynamically, the declared types of an object. And because of the importance of these features in assisting model reuse via model interconnection, which is recognized as an important and complex problem in modeling systems [37, 43].

The ideas presented in this paper generalize beyond Ascend in two ways. First, the method that we use to specify and analyze the semantics of Ascend is applicable to a broad class of typed modeling languages (see

[1]A strongly typed language is one in which all elements have an associated type. The type information is used during compilation to ensure that all expressions in the language are type consistent.

[2]Ascend supports equational modeling, which includes paradigms such as mathematical programming, simultaneous equations, and differential equations.

§3.2). Second, our analysis (§5) establishes a set of desirable properties which might serve as a benchmark for type systems for model management languages. Such properties can, and should, be investigated in all modeling languages. Since the design of Ascend's type system is entirely compatible with current algebra-oriented executable modeling languages, this paper should be extremely useful to those interested in extending these languages.

1.3 Related work

There is an extensive literature on typing in programming languages, some of which is relevant to the issues we consider in this paper. A slightly dated but excellent survey of typing and the role it plays in enabling data abstraction and polymorphism in programming languages is provided in [15]. Type systems are classified both in terms of the set of types that can be represented and in the kinds of relationships such as equivalence or inclusion that can be computed. Computing such relationships between types is fundamental to the analysis and implementation of type checking and type inference algorithms, a topic that is of considerable interest to the programming language commmunity [2, 27]. Given the recent trend toward incorporating strong typing in modeling languages, this body of work is directly relevant to implementing language environments. Thus, for instance, the fact that Ascend's type system is a first-order type system with single inheritance subtyping (as in Cardelli and Wegner's classification [15]) gives us certain assurances about type checking.

However, analogs of Ascend's distinctive type modification operators—which were developed to meet *modeling* needs—have not been studied in the programming language literature. This leads to our focus on the analysis of a type system—such as Ascend's—in which type modification operators play an important role. Ideas related to our work may be found in the areas of type inference in programming languages [3, 15, 40, 36], typing in object-oriented languages [29] (as a particular example, consider the type system in the object-oriented programming language Eiffel [42]), and subsumption in concept definition languages [56]. We briefly discuss each of these topics and the relation of results in these areas to our work.

1.3.1 Type Inference in Programming Languages

Type systems were introduced in the programming language literature several years ago and now come in various forms. [15]. The objective of typing is to ensure that functions (or equivalently, programs) are applied to appropriate arguments. Programming languages can perform type checking either at "compile time" or at "run time." Languages that perform compile time type checking are *statically typed*, while languages that perform run time type checking are *dynamically typed*. ML [47] is an example of a statically typed language while LISP [55] is an example of a dynamically typed language. It is obvious that the benefits of type checking (particularly, static type checking) are best realized when every element of the program has an assigned type.

To relieve the burden on the programmer in making type assignments, programming languages such as ML rely on a *type assignment system*. A type assignment system is a set of rules for associating types, in general the *most general type*, to programs *without assuming typed declarations of variables*. *Type inference*, in this literature, refers to the algorithmic implementation of a type assignment system. An example is the inference in ML of the type of the function *reverse* that reverses a list [46]. Given the definition of the *reverse* function—this definition contains no mention of types—ML's type assignment system is able to infer the most general type of the function to be $list(t) \rightarrow list(t)$ where t is the type variable. This, for instance, permits the *reverse* function to be used to reverse a list of integers (i.e., the type variable t takes on the value *integer*).

Thus, our use of the term type inference in the context of inferring the type associated with a model property (i.e., an Ascend variable) is fundamentally different from the use of the term in the programming language community. First, it is not our aim to infer the type of a function or program phrase *from a definition which makes no mention of the type of variables.* Second, in languages such as ML type inference involves determining the most general type of a function from its *definition*, while our inference procedures establish the most refined type of a model property from a collection of declarations that either *state or modify its type.*

Finally, there is another crucial distinction between Ascend's type system and conditional typing in programming languages. Ascend's type manipulation operators implement a specific and fixed semantics, considered

relevant in mathematical modeling; in this paper, we articulate and explain these semantics using predicate logic. Programming languages with conditional typing, on the other hand, provide a specialized logic which is still general enough to specify and implement, perhaps with significant programming, particular operators (such as those of Ascend) which impose various types of type constraints. While these languages do not offer such operators as "built-in" or primitive operators for development of mathematical models, Ascend does.

1.3.2 Concept and Frame Description Languages

Concept description languages and frame-based representation languages have been an active area of research in the knowledge representation community. These languages provide representational features and inferential services. For example, frame-based languages permit the definition of a collection of structures called frames that denote classes of objects that have certain attributes. Each frame has a number of data elements called slots, each of which corresponds to an attribute that members of its class can have. Representational features in Ascend (e.g., atom and model types) are similar to these features of frame-based languages. Frames also support procedural features thereby permitting procedures to be attached to slots for computing values of slots or to propagate side effects when a slot is filled. While Ascend does support frame-like structures, it does not allow procedural attachments to slots; on the other hand, unlike typical frame-based languages, Ascend has type modification operators which are powerful tools in model building.

Concept description languages support a different world view from that of frame-based systems. The most influential among these is the so called KL-ONE family [56] which includes languages such as CLASSIC [11] and LOOM [41]. They have well developed and clear criterial semantics [56] and require concepts to be defined in terms of other previously defined concepts. For example, the concept "children" is related to the concept "sons." These concept definitions are *automatically classified* and organized using the subsumption relation which is stated intensionally unlike in Ascend, where the taxonomic hierarchy is set up extensionally using the *refines* relation. For example, in concept description languages, the concept "woman with children" can be inferred to be more general than (i.e., it subsumes) the concept "women with sons;" this, by virtue of the relationship between "children" and "sons;" such inference is not supported in Ascend. From a practical point of view this means that in concept description languages, the modeler or the programmer does not declare *kind of* or *subtype* relations in order to create taxonomic structures whereas in Ascend, the modeler must handcraft and maintain the hierarchy. There has been considerable work on defining these subsumption algorithms and studying the tradeoff between the expressiveness of the concept description language and the tractability of these algorithms.

To summarize, Ascend shares some similarities with the declarative, representational features of frame-based languages. The procedural extensions in frame-based languages and the concept of automatic classification of concepts in concept description languages are not seen in Ascend. Similarly, the type modification operators in Ascend which modify the type of a property find no analog in concept description or frame description languages.

2 The Ascend Modeling Language

Ascend was originally designed to support the declarative and structured specification of large systems of constraints that arise in engineering design [48]. See [50] for an introduction to the language and [39] for a discussion of its application in OR/MS applications in general and model management in particular. The language builds on concepts used in object-oriented programming and conventional strongly-typed languages such as Pascal. Specifically, it supports a well defined type system, type manipulation operators, and monotonic inheritance. It is not a pure object-oriented language since it does not subscribe to either the "information hiding" or the "explicit message passing" paradigms used in languages such as Smalltalk [30].

Modeling in the Ascend language consists of declaring structured types. There exist 4 unstructured *elementary* types that are basic to the language and need not be declared. Structured types are categorized into *Atom* types (Atoms) and *Model* types (Models), and are typically composed from other types using certain operators in the language. This section has two objectives: a) to give the reader an idea of the utility and complexity

of Ascend's type modification operators, and b) to provide enough details about the type system to facilitate understanding of the later sections that develop and analyze formal semantics for typed languages.

[Figure 1 about here.]

Throughout this section, we illustrate Ascend concepts via the well-understood simple transportation problem (Figure 1). The fundamental modeling philosophy in Ascend—which is consistent with its unique operators (particularly the "ARE-THE-SAME" operator) for inheritance and type modification—is a "building block approach": create components of a complex model, then put these and any pre-existing components together via operators in the language. For example, we first build `plant` and `customer` models, then we create the `transporation` model by declaring certain relationships between plants and customers; the complete collection of Ascend statements in given in Appendix A. With regard to Ascend's support for reusability, note that the `plant` and `customer` models could later be used, perhaps with some specialization, to develop models for vehicle routing or traveling salesman problems as well. To further illustrate how the operators facilitate a building block approach, we combine the plant-customer `transporation` model with a warehouse-plant `transportation` model to create the integrated transhipment model.

2.1 Elementary Types

There are four elementary types: real, integer, boolean, and symbol. Informally each of these types can be thought of as names that refer to the set of real numbers, integers, booleans, and symbols respectively.

2.2 Atom Types

Atoms, in contrast to elementary types, are structured, i.e., they contain a number of named *properties* each of which has a value that is an instance of some type. In the case of Atoms, values of all properties are required to be instances of elementary types. The definition of a commonly used Atom in Ascend—`solver_var`, that models an algebraic variable—is given below. The precise meaning of the statements is indicated in the corresponding first-order logic statements on the right; this topic is discussed in more detail in [6].

`ATOM solver_var;`	
`value IS_A real;`	∀x solver-var(x) →
`lower_bound IS_A real;`	∃y (value(x)=y, real(y))
`upper_bound IS_A real;`	∃y' (lower-bound(x)=y', real(y'))
`END solver_var;`	∃y" (upper-bound(x)=y", real(y"))

This definition declares a set of individuals of type `solver_var` each with three properties `value`, `lower_bound`, and `upper_bound`. The IS_A operator is used to associate a given type with a given property, e.g., the statement `value IS_A real` associates the type `real` with the property `value`.

`ATOM supply_capacity REFINES solver_var;`	∀x supply-capacity (x) →
`low_bound := 0;`	solver-var(x),
`END supply_capacity;`	low-bound(x) = 0

Atoms can be organized into refinement hierarchies using the REFINES operator. These refinement hierarchies specify subtypes and enable inheritance. Thus, for instance, as shown above, `supply_capacity` is specified to be a subtype of `solver_var`. Hence it inherits the properties `value`, `low_bound` and `upper_bound` from the definition of `solver_var`. Since mathematical models formulated in Ascend are stated in terms of algebraic variables, a typical formulation will specify several refinements of atom types such as `solver_var`.

We can depict Ascend objects and relationships between them in a diagram. Figure 2 gives an example diagram which shows a fragment of the types `solver_var`, `supply_capacity` and `real`. We encourage the reader to make use of such diagrams to understand some of the more complicated examples in this paper.

[Figure 2 about here.]

2.3 Model Types

Models, like Atoms, are structured; however, the values of their properties are not restricted to be instances of elementary types, and can be instances of Atoms, Models, or elementary types. Models also permit specification of algebraic relationships between properties that have elementary or atomic type, and are defined within the scope of the model. Qualified names are used to reference properties that are not locally defined.

[Figure 3 about here.]

To continue building our transportation model, we now define the model of a `plant`; the definition is given in Figure 3 with the lines numbered for ease of explanation. The model `plant` denotes a set of individual plants, each of which supplies a set of customers, denoted by their identifiers, `customerId`. This is stated in Lines 1–3 above and is derived from our conceptualization of a plant as shown in Figure 3.

Each plant has several properties: `sup` (Line 4) denotes its supply capacity; `fc`, an array (Line 5), models the flow from the plant to each of the customers that it supplies; and `cost` (Line 6) represents the unit cost of supplying each of the customers. Note that the types of these properties (e.g., `supply_capacity`) are atoms that are presumed to have been developed earlier. These properties are related algebraically and specified compactly using arrays indexed over sets of symbols (e.g., `fc`). For example, the `totalFlow` (Lines 7–8) from a plant to the customers it supplies should not exceed the available supply capacity `sup`. In general, as indicated in Lines 7–10, arrays in Ascend may be indexed over sets of symbols or integers or refinements of these elementary types. These index sets can also be declared implicitly (e.g., `[1..maxcustomer]`) and computed from other sets using operations common to algebraic modeling languages.

[Figure 4 about here.]

The `customer` model—a mirror image of the plant model—is described in a similar way. As shown in Figure 4, a customer can be supplied from several plants. This conceptualization leads to the model specification shown below. Each customer has a demand `dem` which is an instance of an atom `demand` and has a property `fp` which models the flow into a customer from the set of plants `plantId` that supply it. Paralleling the constraint on plants, the flow into a customer is required to be equal to the demand at the customer.

2.4 Equivalencing model properties: The ARE_THE_SAME operator

Recall that our `plant` model was based on the concept of having a plant supply a set of customers. Similarly, the `customer` model is based on the concept of having a customer being supplied by a set of plants. But, how do we relate specific plants to the customers they supply and vice versa? In a transportation context, as illustrated in Figure 5, these plants and customers have a specific linkage; in Ascend terms, the flows emanating from a plant are *equivalent* to flows incident on a customer. This linking is accomplished using the `ARE_THE_SAME` operator which establishes equivalence between two or more model properties. ATS declarations define equivalence classes in which all members are equivalent [12, 53]. Equivalenced properties are forced to have the same type(s) and values.

[Figure 5 about here.]

This process also completes the development of our `transportation` model. It contains an array of plant and customer models—`p` and `c` respectively (Lines 1–3). The set of plants that supply a given customer is stated in Line 5; this also states, inversely, the set of customers that a given plant supplies.

The key step—Line 9—in constructing the model is to specify that the jth flow `fc[j]` from plant `p[i]` is *equivalent* (ATS) to the ith flow *fp[i]* into customer `c[j]`. We have now created the connected network transportation structure. This ATS statement, via type inference, also results in the value, upper bound and lower bound of the `fp` and `fc` properties to be equivalenced. Thus, the effect of an ATS declaration is not limited to the properties for which the declaration is made. Since any property independently may participate in other

ATS or other operator declarations, the overall implications—and even the validity of the entire collection of declarations—are not quite obvious.

The validity of an ATS declaration depends on the structure of the type hierarchies created with the REFINES relationship (see below). ATS declarations can be made only for properties whose types are *comparable*. Two types are comparable if one lies on the *root path* [1] of the other in a type hierarchy. Given two types on the same root path, the type that is farther from the root is referred to as the *more refined type*. When an ATS declaration relates two model properties with different types, the type of one of the properties is altered to be of the more refined type. Thus, an ATS declaration can associate more than one type with a model property.

2.5 Defining model hierarchies; the REFINES operator

As with Atoms, Models can also be organized into inheritance hierarchies using the REFINES operator. In all cases, inheritance is purely monotonic in the sense that all properties and relationships between them are inherited from the parent, and none can be deleted.

To see the utility of this operator, let's consider a fairly common scenario. The demand `dem` is specified as an exogenous input in our `customer` formulation above. Now we desire the demand to be obtained as a result from a *demand forecasting model*. To incorporate this extension, we make use of a general `forecast` model which we adapt, using the REFINES operator, into an `expForecast` model for our purposes. This way we begin developing our model hierarchy into which, refining the `customer` model, we will add a `customer_with_forecasted_demand` model.

```
MODEL forecast;
   Tf IS_A integer;
   D[1..Tf] IS_A demand;
   E[1..Tf] IS_A expectedValue;
   S[2..Tf] IS_A smoothedValue;
   F[2..Tf] IS_A forecastedValue;
END forecast;
```

The `expForecast` model defines an exponential smoothing forecasting model with `alpha` as a smoothing parameter. Clearly, other refinement of the base forecasting model `forecast` are possible (e.g., a moving average model).

```
MODEL expForecast REFINES forecast;
   alpha IS_A dimensionlessConstant;
   E[1] = D[1]
   FOR i IN [2..Tf] CREATE
      E[i] = alpha*D[i] + (1-alpha)*E[i-1];
      F[i] = E[i] + S[i]/alpha;
   END;
   S[2] = E[2] - E[1];
   FOR i IN [3..Tf] CREATE
      S[i] = alpha*(E[i]-E[i-1]) + (1-alpha)*S[i-1];
   END;
END expForecast;
```

Now we refine the `customer` model to create the `customer_with_forecasted_demand`. The demand `dem` at the customer is specified to be equal to the forecasted value.

```
MODEL customer_with_forecasted_demand REFINES customer;
   F IS_A forecast;
   dem = F.E[F.tf];
END customer_forecast;
```

Apart from the purpose already served by the REFINES operator, model hierarchies, in conjunction with other Ascend operators (ARE_THE_SAME, ARE_ALIKE and IS_REFINED_TO), prove useful in defining a variety of more complex models. We discuss this feature next.

2.6 Declaring similarity between model properties: The ARE_ALIKE operator

The transportation model developed thus far consisted of an array of plant and customer models. Now consider a scenario in which we apply the forecasting models developed above. That is, we want to define a new transportation model, say transportation_model_with_customer_demand_forecasts in which all the customers are instances of the customer_with_forecasted_demand model.

Recall that the customer_with_forecasted_demand is a refinement of the customer model. If we declare (using the ARE_ALIKE (AA) operator) that all the instances c of the customer component of the transportation model are structurally similar, any change to the type of one of the elements of c would result in the desired change being propagated to all customer model instances. The model fragment used to declare that all the instances of the customer model "are alike" is shown below. Such a fragment would have to be added to the transportation model specification of the previous section.

```
FOR i,i+1 in customerId CREATE
c[i],c[i+1] ARE_ALIKE;
```

As with the ATS declaration, a modeler can declare two model properties to be alike only if their types are comparable. An AA declaration makes the types of the properties the same, but does not *equivalence* them as does ATS. Since the type of a property determines its structure, a structural change (i.e., the refinement of the type of a property) made to any one property is propagated automatically to the group of properties declared to be alike.

2.7 Modifying the type of a property: The IS_REFINED_TO operator

To continue with the example from the previous section, consider a refinement of the transportation model which consists of customers which are instances of the customer_with_forecasted_demand model. The model specification is shown below.

```
MODEL transportation_model_with_customer_demand_forecasts REFINES
transportation;
   customerId := {NY, LA, PHIL};
   c[‘‘LA’’] IS_REFINED_TO customer_with_forecasted_demand;
   c[‘‘LA’’].F IS_REFINED_TO expForecast;
END trans_forecast;
```

The first statement uses the IS_REFINED_TO (IRT) operator to modify the type of c from customer to its subtype customer_with_forecasted_demands. Since the elements of c have been declared to be similar using the AA operator, this results in all elements of c becoming instances of the more refined type. Recall that the property F of a customer model is of type forecast not its subtype expForecast. The second statement uses the IRT operator to modify the type of the forecast property to the exponential forecast model, expForecast. This second application of the IRT operator is particularly useful in a scenario in which we have multiple forecasting models specified and are interested in generating variants of the base transportation model integrated with different types of customer demand forecasting models. This ability to *combine* type modification (using IRT) with propogation of structure (using AA) or equivalencing of structure and value (using ATS) is a unique and powerful feature of the Ascend modeling language, and facilitates an evolutionary approach to modeling and model reuse.

2.8 An example of model integration in Ascend

Finally, in this section, we demonstrate how the operators in Ascend facilitate reuse via model integration. We use the integration of two transportation models to define a transhipment model as an illustrative example.

```
IMPORT transportation;

MODEL transhipment;
   (* P = production
      D = distribution *)

   P, D IS_A transportation;
   warehouseId IS_A set OF symbol;

   P.customerId, D.plantId, warehouseId ARE_THE_SAME;

   FOR i IN warehouseId CREATE
      P.c[i].dem >= D.p[i].totalFlow;
   END;

   P.obj.included := FALSE;
   D.obj.included := FALSE;

   obj : MINIMIZE
            SUM(P.p[P.plantId].shipmentCost) +
            SUM(D.p[warehouseId].shipmentCost);

END transhipment;
```

Integration begins with the incorporation of two instances of the transportation model (P and D respectively) into the transhipment model. The set of customer identifiers of P (P.customerId) and the set of plant identifiers of D (D.plantId) are equivalenced using the ARE_THE_SAME operator. The material balance constraint at the transhipment node which requires that the flow into a transhipment node to be greater than or equal to the flow out the node is defined for the identifiers in the merged set. Finally, the integration is completed by "unincluding" the objective functions of the individual transportation models (by assigning the included attribute a value of FALSE) and introducing a new objective function. This sort of compact integration is made possible due to the ability to specify and reuse structured component models and support for powerful operators such as ARE_THE_SAME.

2.9 Summary

To summarize, the Ascend language has three important features.

1. A modular approach using the IS_A operator to building complex model types from simpler model types, atom types and elementary types.
2. Organization using the REFINES operator of atom types and model types into inheritance hierarchies.
3. Type modification using the ARE_THE_SAME, ARE_ALIKE, and IS_REFINED_TO operators of model properties in order to accomplish specific goals such as model reuse, structure propagation and integration.

Given that type systems and type inference play a central role in model statement, and that the complex interactions between model (type) declarations and operator (type modification) declarations are a key reason why Ascend operators provide powerful support for modeling, it is important to understand the behavior of the

Ascend type system. In particular, given the complex interactions between model and operator declarations which define and modify the type of a model property, it is important to establish if the type system has desirable features such as the existence of computable and uniquely definable *most refined type* corresponding to each property in an Ascend model. It is our goal in formalizing the semantics of Ascend to make these tasks simpler, and the meaning of each operator declaration clearer.

3 Developing Formal Semantics of Type Systems

What exactly do we mean by investigating the formal semantics of Ascend? The part of Ascend concerned with the mathematical relationships between objects is not the issue here; the semantics are similar to those of other executable modeling languages. What's distinctive about Ascend (or of any typed modeling language), as we stated before, is its type system, and the fact that statements involving Ascend's operators (AA, ATS, and IRT) can be used to change the declared types of objects. The interactions between these statements can be very complex, and hence it is not obvious a) whether certain operator statements are valid from the point of view of consistent typing (i.e., that the type of the operands satisfies semantic constraints), and b) what changes they induce to types of previously declared objects. That is what we are concerned with in this paper. The problem is stated more precisely in §3.1, and our solution strategy is summarized in §3.2.

3.1 The Problem Statement

Let us conceptualize a statement about the type T of an Ascend object, θ, in the context of a model M (or of an atom M), as

$$\mathrm{type}(\theta, M) = T \tag{1}$$

Each instantiation of the schema described by Equation 1 corresponds to a type declaration for an object. There are two ways in which types may be stated. The first is user-declarations of types, made in atom or model definitions. The second is via inference (e.g., over statements involving an AA, ATS, or IRT operator). Since there can now be several type statements for a given object, what then—if any—is its "true" type? The answer lies in the relationship of type refinement, and the fact that types are organized in a hierarchy. That one type refines another is stated in Ascend solely via user-declarations.

Type refinement is significant in Ascend in two ways. First, statements involving AA, ATS, or IRT are valid only when the concerned objects have types such that one is a refinement of the other. Second, the "true" type of an ascend object is defined to be the *most refined type* (MRT), computed using all instances of Statement 1. This true type is required in order to apply various other computational procedures in Ascend. Let us conceptualize the MRT of an object θ as

$$\mathrm{mrt}(\theta, M) = T' \tag{2}$$

Our primary concern in developing a formal semantics for Ascend is to develop a method for examining the truth of instances of these two statements. That is, given statements such as ($\mathit{type}(\theta, M) = T$) or ($\mathrm{mrt}(\theta, M) = T'$), we wish to have a clear, unambiguous, and implementation-independent way of determining whether they are true or not. This is important in a typed modeling language since the type of an object determines various attributes such as its mathematical structure and its definitional dependencies on other objects.

3.2 Strategy: A Bilevel Conceptualization

We distinguish the "class-based" or *object-level* fragment of Ascend from its *meta-level* type modification operators. The class-based fragment consists of the three kinds of types (*elementary types*, *atom types* and *model types*), the IS_A operator (used to declare types of slots) and the REFINES operator (used to establish type

hierarchies). The *meta-level* operators (ATS, AA, IRT) operate on, and draw inferences about, these object level declarations in order to alter type information about the slots. [3]

This bilevel conceptualization has two principal advantages. First, it characterizes the class of languages to which our methods are applicable (i.e., a language with first-order types, inclusion subtyping [15], and operators which permit reasoning over the types of objects). Second, as shown below, the distinction made between the class-based fragment of the language and the operator declarations permits us to define a simple translation from Ascend statements into predicate logic.

Since atom and model declarations in Ascend define classes of individuals which share certain properties, we formalize models and atoms as sets. The same is true for elementary types such as real, integer, boolean, and string which also refer to sets of individuals. Each type is viewed as a first-order predicate which denotes all the elements of the type. The refinement relationship between types also translates into a first-order statement. For example, REFINES$(T1, T2)$ is equivalent to $\forall x (T1(x) \rightarrow T2(x))$. The named slots in atom and model declarations are viewed as functions, whose domain is the model or atom they appear in and whose range is the type declared by the IS_A operator. For example, the statement *p1 IS_A point* in the model strut defines a function p1 whose domain is the set of all struts and whose range is the set of points. The definition of a function F with a domain M and range T is viewed as a first-order assertion funDef(F, D, R). For better readability, we write this predicate as: $F : M \mapsto T$. The class based fragment of Ascend can therefore be formalized as a collection of sets, refinement relationships between sets, and functions declared on the sets. Further, it facilitates an accurate description of the semantics of the meta-level operators in Ascend which are applied to functions (i.e., the slots) in the class-based fragment of the language. In particular, the treatment of functions, their domain and their range (i.e., type) as first class objects permits us to state, within a single formal system, both the semantical constraints on making statements involving the operators and the rules for making inferences with these statements.

4 Logical Formalization of Ascend's Type System

Recall that in formalizing Ascend's semantics, our focus is on the truth-value of statements of the forms $F : M \mapsto T$ (i.e., type $(F, M) = T$) and mrt$(F, M) = T'$. We will reconstruct the essential aspects of Ascend's type system axiomatically as statements in first-order logic. Then, any particular statement about the type of an Ascend object (i.e., a slot or property) will be true if and only if it can be derived, using standard logical inference, in our formalization[4]. A new assertion will be valid (consistent with all the previous ones) if and only if it does not lead to a contradiction in our formal system, and its "implications" (on other objects) will be exactly those statements, not derivable earlier, that can be derived once the assertion is added.

We present the axioms in three categories: 1) axioms defining the mathematical properties of the IRT, AA, and ATS operators, 2) axioms that state the "rules" for drawing inferences about the type of a slot, and 3) axioms that represent integrity constraints for declarations involving IRT, AA, and ATS operators. The inference rules are the most significant in making Ascend's type systems as "powerful" as it is, since they can trigger a series of deductions starting from a single user declaration. The integrity constraints play an important role in ensuring that the language has certain desirable mathematical properties.

Before we describe the axioms, we must deal with some issues of notation and terminology. We use upper case letters as logical variables in our axioms: we will usually use F for functions (recall that properties are treated as functions in the formalization), T for types, and M for models (recall, though, that models are types). All free variables are assumed to be universally quantified, unless we explicitly use the existential quantifier. For better readability, we also state a few definitions—though these could also be thought of as axioms in our formalization.

Definition 1 *Type of Function*

[3]Certain other aspects of Ascend (e.g., equations, dimensional information, and instantiation procedures) can be ignored since they do not affect the type of an object.

[4]In first-order logic, a proof-theoretic derivation of a statement guarantees its truth in a semantic sense as well.

The *type* of a function F in a model M is defined as follows:

$$(\text{type}(F, M) = T) \leftrightarrow (F : M \mapsto T)$$

Statements of the form $(F : M \mapsto T)$ can either be based on user declarations of type or be generated via inference using the axioms described in this section. Hence, a function may be associated with several types.

Definition 2 *Applicable Functions*

Given a model M and a function (slot) F, we say that F is an applicable function for M (written appl-func(M,F)) if there is a type T such that $F : M \mapsto T$. Formally,

$$\text{appl-func}(M, F) \ \leftrightarrow \ \exists T (F : M \mapsto T)$$

Definition 3 *Type Comparability*

Two *types* are comparable if and only if one of them refines the other. Type comparability is commutative.

$$\text{comp}(T1, T2) \leftrightarrow \ (\text{refines}(T1, T2) \ \lor \ \text{refines}(T2, T1))$$

Definition 4 *Type Conformability*

Two *functions* are type conformable in the context of a model M *if and only if* all their types in M are comparable. Type conformability is commutative.

$$\text{conf}(F1, F2, M) \ \leftrightarrow \ \forall T1 \ \forall T2 \left(\begin{array}{r} ((F1 : M \mapsto \ T1) \ \land \ (F2 : M \mapsto \ T2)) \\ \rightarrow \ \text{comp}(T1, T2)) \end{array} \right)$$

4.1 Mathematical Properties of Ascend Operators

Axiom 1 *refines*: The *refines* relationship is a partial order.

$$\begin{array}{rrr} \text{Reflexivity:} & & \text{refines}(T, T) \\ \text{Anti-symmetry:} & (\text{refines}(T1, T2) \ \land \ \text{refines}(T2, T1)) \ \rightarrow & T1 = T2 \\ \text{Transitivity:} & \exists T \ (\text{refines}(T1, T) \ \land \ \text{refines}(T, T2)) \ \rightarrow & \text{refines}(T1, T2) \end{array}$$

Axiom 2 *are-the-same*: ATS is an equivalence relation; it is transitive, symmetric and reflexive.

$$\begin{array}{rrr} \text{Reflexivity:} & & \text{ats}(F, F, M) \\ \text{Symmetry:} & (\text{ats}(F1, F2, M) \ \rightarrow & \text{ats}(F2, F1, M)) \\ \text{Transitivity:} & \exists F \ (\text{ats}(F1, F, M) \ \land \ \text{ats}(F, F2, M)) \ \rightarrow & \text{ats}(F1, F2, M)) \end{array}$$

Axiom 3 *are-alike*: AA is an equivalence relation; it is transitive, symmetric and reflexive.

$$\begin{array}{rrr} \text{Reflexivity:} & & \text{aa}(F, F, M)) \\ \text{Symmetry:} & (\text{aa}(F1, F2, M)) \ \rightarrow & \text{aa}(F2, F1, M)) \\ \text{Transitivity:} & \exists F \ (\text{aa}(F1, F, M) \ \land \ \text{aa}(F, F2, M)) \ \rightarrow & \text{aa}(F1, F2, M)) \end{array}$$

4.2 Reasoning about Types

The following axioms describe type inferencing and manipulation in Ascend. The axioms are depicted concisely in Figure 6; this figure may also be meaningful in understanding the key logical idea underlying each axiom. The first two rules tell us that types of functions can be inferred via refinement and composition. The remaining rules specifically concern the semantics of Ascend operators, and state how statements involving operators result in the association of new types with various functions. Note that a rule having a schema $\psi \rightarrow (\phi \rightarrow \gamma)$ is the same as $(\psi \wedge \phi) \rightarrow \gamma$. We often write it in the former form to indicate that it can be read as: Suppose ψ; then ϕ implies γ.

[Figure 6 about here.]

Function Applicability

Axiom 4 Models inherit the functions of other models that they refine.

$$(\text{refines}(T, T1) \;\wedge\; (F : T1 \mapsto T2)) \rightarrow (F : T \mapsto T2)$$

Axiom 5 Functions are applicable via composition.

$$\exists T \;\; ((F1 : T1 \mapsto T) \;\wedge\; (F2 : T \mapsto T2)\;) \rightarrow ((F1 \circ F2) : T1 \mapsto T2)$$

Implications of Declarations involving Operators

Axiom 6 An IRT declaration associates a more refined type with an applicable function.

$$((F : M \mapsto \; T1) \;\wedge\; \text{irt}(F, M, T2) \;\wedge \text{refines}(T2, T1)) \rightarrow (F : M \mapsto \; T2)$$

Axiom 7 AA relationships are inherited via refinement. i.e., if two functions are-alike in a given model, they also are-alike in all refinements of the model.

$$(aa(F1, F2, M) \wedge \; \text{refines}(T, M)) \rightarrow aa(F1, F2, T)$$

Axiom 8 ATS relationships are inherited via refinement. i.e., if two functions are-the-same in a given model, they also are-the-same in all refinements of the model.

$$(ats(F1, F2, M) \wedge \; \text{refines}(T, M)) \rightarrow ats(F1, F2, T)$$

Axiom 9 Implications, on applicable functions, of an AA declaration: If $F1$ and $F2$ are-alike, and if $F1$ has a type that is more refined than a type of $F2$, then the type of $F1$ gets associated with $F2$.

$$\begin{aligned}&\text{aa}(F1, F2, M) \;\rightarrow \\ &\quad (\forall T1\, \forall T2 \; ((F1 : M \mapsto \; T1) \;\wedge\; (F2 : M \mapsto \; T2) \;\wedge \text{refines}(T1, T2)) \\ &\qquad \rightarrow (F2 : M \mapsto \; T1))\end{aligned}$$

Note that no inference is possible if the functions are not type conformable. In fact, such a declaration will be disallowed—see the integrity constraints in §4.3.

Axiom 10 Implications, on applicable functions, of an ATS declaration: This is analogous to the previous AA axiom.

$$\begin{aligned}&\text{ats}(F1, F2, M) \rightarrow \\ &\quad (\forall T1\ \forall T2\ ((F1 : M \mapsto T1) \wedge (F2 : M \mapsto T2) \wedge \text{refines}(T1, T2)) \\ &\qquad \rightarrow (F2 : M \mapsto T1))\end{aligned}$$

Axiom 11 If two functions *are alike* in a model T, then they *are alike* in all instances of T.

$$\begin{aligned}&\text{aa}(F1, F2, T) \rightarrow \\ &\quad (\forall F\ \forall M\ ((F : M \mapsto T) \rightarrow \text{aa}(F \circ F1, F \circ F2, M)))\end{aligned}$$

Axiom 12 If two functions *are-the-same* in a model T, then they *are-the-same* in all instances of T. This is in analogous to the previous AA axiom.

$$\begin{aligned}&\text{ats}(F1, F2, T) \rightarrow \\ &\quad (\forall F\ \forall M\ ((F : M \mapsto T) \rightarrow \text{ats}(F \circ F1, F \circ F2, M)))\end{aligned}$$

However, an ATS declaration results in additional inferences about the type of an object as shown below.

Axiom 13 Recursive propogation of the effects on an ATS declaration: If ats$(F1, F2, M)$, then for every applicable function F, it also follows that ats$(F1 \circ F, F2 \circ F$,M).

$$\begin{aligned}&\text{ats}(F1, F2, M) \rightarrow \\ &\quad (\forall F\ \forall T\ ((F1 : M \mapsto T) \wedge \text{appl-func}(F, T)) \\ &\qquad \rightarrow \text{ats}(F1 \circ F, F2 \circ F, M))\end{aligned}$$

Notice that for an ATS declaration between any two particular functions $f1$ and $f2$, the above axiom is universally quantified over M, F, and T associated with $F1$ (and with $F2$ since ATS is commutative). In particular, it applies to all applicable functions of T and for all types T[5].

Determining Most Refined Type

Axiom 14 *most refined type* (mrt): Since a function may be associated (via inference) with several types, we write its *most refined type* in the context of a model M as mrt(F,M), and define it as follows:

$$\begin{aligned}&\exists T'\ ((F : M \mapsto T') \wedge \forall T\ ((F : M \mapsto T) \rightarrow \text{refines}(T', T))) \\ &\qquad \rightarrow (\text{mrt}(F, M) = T')\end{aligned}$$

[5]In addition to equivalencing the *type* of the properties that are operands of an ATS declaration, the operator also equivalences their values. We do not discuss this aspect any further since it is not relevant to type semantics.

4.3 Integrity Constraints for Operator Declarations

Not all syntactically correct statements are legal in Ascend. For example, if two functions are declared are-alike, they must already be type conformable. In our formalization, such preconditions for making Ascend declarations are modeled via integrity constraints. The essential requirement for consistency of user declarations is type conformability of the functions. We note that ensuring this may be non-trivial, since various rules—applied to existing user assertions—can result in several inferred assertions about types. We restate the formal definition of type conformability, made earlier in this section, for the reader's convenience.

$$\mathrm{conf}(F1, F2, M) \leftrightarrow \forall T1\, \forall T2 \begin{pmatrix} ((F1 : M \mapsto T1) \wedge (F2 : M \mapsto T2)) \\ \rightarrow \mathrm{comp}(T1, T2)) \end{pmatrix}$$

The integrity constraints ensure that various user declarations are consistent with each other. Our objective is that the system of axioms, combined with user declarations, should result in a logical contradiction if and only if the declarations are inconsistent. Therefore, not only will user-declared statements satisfy the constraints, but so also will any deductions made using the inference rules (Theorem 1 in §5). Note that the constraints are stated in the form $\phi \rightarrow \perp$ where $\perp$ is always false (indicates a contradiction). For example, assume that there are functions f_1 and f_2 such that not all of their types are comparable (let's say these types are t' and t''). Then, if a user asserts $\mathrm{aa}(f_1, f_2)$—which is not allowed in Ascend—the AA integrity constraint (Axiom 15) results in a contradiction.

Our intent is that each integrity constraint be enforced only for the objects for which a relation is being asserted, rather than for all objects for which that relation is true. In fact, we will prove (in Theorem 1) that the latter, wider, kind of check is unnecessary—its satisfaction is guaranteed in our formalization by the former kind of check.

Axiom 15 Two functions can be declared *are-alike* only if they are type conformable.

$$(\mathrm{aa}(F1, F2, M) \wedge \neg\mathrm{conf}(F1, F2, M)) \rightarrow \perp$$

Axiom 16 Two functions can be declared *are-the-same* only if they are type conformable.

$$(\mathrm{ats}(F1, F2, M) \wedge \neg\mathrm{conf}(F1, F2, M)) \rightarrow \perp$$

Axiom 17 The type of a function can directly (including with IRT) be declared only to one that is comparable with every declared or derivable type of the function.

$$\begin{pmatrix} \mathrm{irt}(F, M, T) \vee \\ F : M \mapsto T \end{pmatrix} \wedge \exists T' \begin{pmatrix} F : M \mapsto T' \wedge \\ \neg\mathrm{comp}(T, T') \end{pmatrix} \rightarrow \perp$$

Axiom 18 Each type in an Ascend type hierarchy can have at most one parent. That is, the hierarchy does not admit multiple inheritance. Therefore, if it is stated (or true via type refinement) that a type M refines two other types, then those types must themselves be comparable.

$$\begin{pmatrix} \mathrm{refines}(M, T1) \wedge \\ \mathrm{refines}(M, T2) \wedge \\ \neg\mathrm{comp}(T1, T2) \end{pmatrix} \rightarrow \perp$$

Axiom 19 A type M may be declared to refine a type T only if, for all functions that are applicable to both M and T, all their associated types in M are comparable to all their types in T.

$$\mathrm{refines}(M, T) \wedge \exists T1\, \exists T2 \begin{pmatrix} (F : M \mapsto T1) \wedge \\ (F : T \mapsto T2) \wedge \\ \neg\mathrm{comp}(T1, T2) \end{pmatrix} \rightarrow \perp$$

Axiom 20 A statement of the form T refines M cannot be compatible with a statement that F has type T in model M. This also covers the case where $T = M$ since "refines" is reflexive.

$$\left(\begin{array}{c} \mathrm{irt}(F, M, T) \vee \\ F : M \mapsto T \end{array}\right) \wedge \ \mathrm{refines}(T, M) \rightarrow \bot$$

Axiom 21 A statement of the form $F : M \mapsto T$ or irt(F,M,T) cannot be made if F can be composed with another function such that the domain and range of the composed function are the same.

$$\left(\begin{array}{c} \mathrm{irt}(F, M, T) \vee \\ F : M \mapsto T \end{array}\right) \wedge \exists F' \exists T' \left(\begin{array}{l} (F \circ F' : T' \mapsto T') \vee \\ (F' \circ F : T \mapsto T) \end{array}\right) \rightarrow \bot$$

5 Analysis

We have formalized Ascend's type system with a collection of axioms described in the previous section. As stated in §4, the type inference rules provide tremendous deductive power, which is particularly useful in model reuse. The integrity constraints restrict the sorts of things that can be expressed in Ascend, with the aim of endowing the type system with certain desirable mathematical and computational properties. Having presented the formalization, we can now investigate whether the type system indeed has these properties.

In particular, there are three properties which we wish to investigate. First, do Ascend's type inference rules (Axioms 4–14) preserve consistency with respect to the integrity constraints (Axioms 15–21)? This is a minimal requirement for any formal system, and we claim that our construction satisfies this requirement (Theorem 1). This property has an important computational implication in that it frees the implementor of the type system from having to check integrity constraints after each inferential step thus permitting greater inferential efficiency.

Second, given that for any function F several deductions $F : M \mapsto T$ can be made as an application of many different inference rules, can we guarantee that the most refined type a) exists uniquely—i.e., a function does not have two types that cannot be compared?, and b) can be determined in a finite number of inference steps? These are also critical properties of any type system. Again, we claim that our formalization satisfies these properties (Lemma 4 and Theorems 2 and 3). These properties also have an important computational implication beyond the obvious importance of finiteness. Once a most refined type (mrt) is computed, the inference engine can halt since mrt is known to be unique.

Finally, given that the AA and ATS relations are equivalence relations, it is interesting to inquire—from a computational point of view—whether there is any relationship between the MRT of different elements in an equivalence class. We will show, in fact, that all elements of an AA or ATS equivalence class will have the same MRT (Lemma 7). Going further, we will show that elements of the union of intersecting equivalence classes (referred to here as a *clique*) also have the same MRT (Theorem 4). This result implies that an efficient implementation of the type system should first compute the *clique* before computing the most refined types of the elements.

We now proceed to present a collection of preliminary results (lemmas 1–7) which are used to establish the three theorems which establish the properties of the Ascend type system that were discussed above. To help the reader discern the relationships between the various lemmas and theorems we present the map (see Figure 7) shown below. All proofs are given in Appendix B.

[Figure 7 about here.]

5.1 Preliminary Results: Functions and Types

Definition 5 (Range-Recursion in Functions) *A type association $F : M \mapsto T$ is considered to be* range-recursive *if T refines M. Otherwise it is* non-range-recursive.

A function F is non-range-recursive if each of its type associations is non-range-recursive.

A collection of atomic functions is non-range-recursive if a) each atomic function is non-range-recursive and b) each possible composition (according to Axiom 5) results in a non-range-recursive function.

Lemma 1 *Given a consistent collection of user-supplied assertions in Ascend, each inferred type association $F : M \mapsto T$ will be consistent with integrity constraints represented by Axioms 20 and 21. That is, if the initial type associations are consistent, all the type associations inferred via the type inference rules will also be non-range-recursive.*

Lemma 2 *Any composite function in a consistent collection of user-supplied Ascend assertions must be composed of atomic functions each of which appears only once in the composition.*

Corollary 1 (To Lemma 2) *Any valid composite function in Ascend must have a finite composition length. Precisely, it can be written in the form $f_1 \circ \ldots \circ f_n$ (for some finite n) where all the f_i's are distinct atomic functions.*

Lemma 3 (Finite number of functions) *Given any consistent collection of user-supplied assertions in Ascend, the total number of functions (atomic and composite) is finite.*

Lemma 4 (Comparability of Types) *Given a consistent collection of user-supplied assertions in Ascend, for any function f and any model m, all the types associated with f in m are comparable.*

5.2 Consistency of Type System

Lemma 5 (Type conformability for AA equivalence classes) *Given a consistent collection of user-supplied assertions in Ascend, all functions in an AA equivalence class are type conformable. In other words, if $aa(f_1', f_2', t)$ can be deduced for functions f_1' and f_2' and some type t, then, f_1' and f_2' must be type conformable in t.*

Lemma 6 (Type conformability for ATS equivalence classes) *Given a consistent collection of user-supplied assertions in Ascend, all functions in an ATS equivalence class are type conformable. In other words, if $ats(f_1', f_2', t)$ can be deduced for functions f_1' and f_2' and some type t, then, f_1' and f_2' must be type conformable in t.*

Now we can establish the first part of our claim about MRT computation in Ascend.

Theorem 1 (Type inference rules preserve consistency) *Given a collection of user-supplied assertions that is consistent with all the integrity constraints (Axioms 15–21), all deductions will also be consistent with all constraints. That is, the type inference rules preserve consistency with respect to the integrity constraints.*

5.3 Most Refined Types and Equivalence Classes

Theorem 2 (Uniqueness of MRT) *Given any consistent collection of user-supplied assertions in Ascend, each function has a unique* most refined type.

Theorem 3 (Finiteness of MRT computation) *Given any consistent collection of user-supplied assertions in Ascend, the* most refined type *of any function can be computed in a finite number of steps, as can any other deduction in Ascend.*

Definition 6 (Clique) *A clique, with respect to a model M, is a union of equivalence classes (AA and ATS) of applicable functions for that model, such that each class has a non-empty intersection with at least one other distinct class in that union. In other words, every pair of elements X and Y in the clique satisfies one of the following 3 conditions: 1) $aa(X, Y, M)$, 2) $ats(X, Y, M)$) or 3) there is some element Z in the clique such that the pairs (X, Z) and and (Z, Y) both satisfy one of the 3 conditions.*

Note that, by definition of equivalence classes, any two distinct AA classes (or two distinct ATS classes) must have an empty intersection. So, in order to determine a clique, we must look for AA classes that have a non-empty intersection with ATS classes.

Lemma 7 (MRT of AA or ATS Equivalence Class) *All elements of an (AA or ATS) equivalence class have the same MRT.*

Theorem 4 (MRT of a Clique) *All elements of a clique of functions will have the same MRT.*

6 Discussion

We have formalized the type system of Ascend using a bi-level strategy which distinguishes between the class-based fragment of the language (which in the type system literature may be characterized as a first-order system with single inheritance subtyping [15]) and the type modification operators. The result has been the development of an axiomatic system which precisely specifies the semantics of both the class-based fragment and the operators using predicate logic. Given that the type modification operators can associate multiple types with an object, we also proved several desirable properties of the formal system. In particular, we stated and proved three results relevant to the questions we posed in §3.1 about most refined type. Specifically, we show that our formal system guarantees that the most refined type of each object in Ascend is unique, that it can be computed in a finite number of steps, and that each step preserves the type integrity constraints.

In [17], Davis articulates 6 connections between formalizations of domain theories and the actual implementation. Two of these connections are relevant to the formalization of Ascend (the domain is Ascend's worldview of modeling), and we discuss them in this context.

1. "The axioms of the domain may be used directly in symbolic form as input to a theorem-prover..." This is certainly possible with our formalization, but such a system would be inefficient from a computational point of view. However, it is useful—and we have indeed used it—as an abstract formal system of which we can investigate various properties. Some of these are important in determining the validity of the system, and others have implications for implementing it. For example, Theorem 1 (consistency of the type inference rules) makes it unnecessary, in any implementation, to check that every deduction from a consistent collection of statements itself satisfies all the integrity constraints. Similarly, due to Theorem 2, the MRT computation can stop once *a* most refined type is found, and Theorem 3 guarantees that it will indeed be found finitely. Theorem 4 gives us an important implementation insight: since all elements of a clique must have the same MRT, it tells us that we should first compute all cliques, and then compute the MRT for one element in each clique.

2. "After [a] program has been written, a *post hoc* logical analysis may aid in debugging, understanding, presenting, and extending it." There are several features of the type system in Ascend for which extensions can be investigated using this formalization:

   (a) Type conformability of objects that are operands of the type modification operators is a constraint on all operations. This requires that the organization of the type hierarchy either anticipate operations one may want to perform (i.e., arrange the types of properties we want to operate on along the same root path) or require modifications after a need to apply a type modification operator has been identified. The formalization will allow relaxations of the type conformability to be investigated in a systematic and principled manner. Specifically, we can replace the type conformability integrity constraint with an alternative and determine if the desirable representational and computational properties (e.g., uniqueness and finiteness of mrt) of the formal system still hold true.

   (b) Closely related to type conformability is subtyping with multiple inheritance. Currently, the type system only permits subtyping with single inheritance. The addition of multiple inheritance would offer a useful representational feature and the implications for the computational and representational

properties can be investigated using the formalization by adding axioms that specify the semantics of multiple inheritance.

(c) The integrity constraints on function composition disallow the representation of recursive type structures which are both elegant and useful representational structures to provide in a modeling language. However, these constraints are required to guarantee properties about MRT computation—that the MRT can indeed be computed in finite time. Clearly, other alternative formulations are possible that allow recursive structures (e.g., introduction of variant types and associated case operations) but impose additional constraints. Our formalization can serve as a foundation for the investigation of these formulations and their impact on representational and computational properties of Ascend.

(d) In contrast to the previous items which have advocated investigating *additions* to the representational features of the language, we can also investigate trading away expressive power to realize gains in compilation efficiency. For instance, if certain type of operator declarations which trigger an iterative inferential process (e.g., when operands are part of both an AA declaration and an inferred ATS statement), and therefore increase the computational complexity of the MRT algorithm, were disallowed, what would be the gains in computational efficiency of the MRT algorithm?

Designers of any modeling language must make tradeoffs between the language's representation power, deductive power (which influences representational economy), and computational properties. A formalization such as the one presented in this paper can facilitate the investigation of such tradeoffs. More generally, such an approach can lead to a principled comparison of different typed modeling languages. One aspect of such a comparison is that alternative languages (or type systems) use different terms for similar concepts or similar-seeming terms for different concepts (e.g., terms such as IS-A, REFINES, and INSTANCE-OF). Such terminology issues can be examined by developing and comparing the formal semantics for all the terms. More importantly, the formal semantics and analysis will make it possible to compare alternative type systems on the basis of their representational and computational properties. We strongly believe that type systems have an importance place in modeling languages and that many more type systems—some different from Ascend's, some an extension of it—will be developed in the future. We hope that this paper will stimulate this process and, eventually, lead to a generally applicable standards and theory of typing that can be incorporated into various modeling languages.

Acknowledgements

The authors acknowledge funding by the U.S. Army Research Office (grants DAAH04-94-G-0239 and DAAH04-96-10385) and the U.S. Marine Corps (grant M9545097WRR7AHZ).

References

[1] A. Aho, J. Hopcroft, J. Ullman, 1974. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, Massachusetts.

[2] A. Aho, R. Sethi, J. Ullman, 1986. *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts.

[3] A. Aiken, E.L. Wimmers, T. K. Lakshman, 1994. "Soft Typing with Conditional Types," *Proceedings of the ACM conference on Principles of Programming Languages*, 163-173.

[4] H.K. Bhargava, "Dimensional Analysis in Mathematical Modeling Systems: A Simple Numerical Method," 1993. *ORSA Journal on Computing* **5: 1**, 33-39.

[5] H.K. Bhargava and S.O. Kimbrough, 1994 "On Embedded Languages, Meta-level Inference and Computer-aided Modeling," in *Computer Science and Operations Research: The Impact of Emerging Technology*, Stephen Nash and Ariela Sofer (eds.), Kluwer. *Proceedings of the ORSA CSTS conference*, Williamsburg, VA.

[6] Bhargava, H. K., R. Krishnan, P. Piela, "Formalizing the Semantics of ASCEND," January 1994. *Proceedings of the Twenty-Seventh International Conference on System Sciences* (Maui, HI), 505–516.

[7] H.K. Bhargava, and S. O. Kimbrough, 1993. "Model Management: An Embedded Languages Approach,"*Decision Support Systems* **10:3**, 277–300.

[8] H.K. Bhargava, R. Krishnan, S. Mukherjee, 1993. "On the Integration of Data and Mathematical Modeling Languages,", *Annals of Operations Research 38*, 69-1996.

[9] H. K. Bhargava, S.O. Kimbrough, R. Krishnan, 1991. "Unique Names Violations: A Problem for Model Integration,"*ORSA Journal of Computing* **3:2**, 107-120.

[10] J. Bisschop, A. Meeraus, 1982. "On the Development of a General Algebraic Modeling Language," *Mathematical Programming Study 20*, 1-29.

[11] A. Borgida, R. Brachman, D. Mcguiness, L. Resnick, 1989. "CLASSIC: A Structural Data Model for Objects," Proceedings of SIGMOD, 58-67.

[12] A. Borning, 1979. "ThingLab: A Constraint Oriented Simulation Laboratory", Ph. D. Thesis, Stanford University.

[13] G.H. Bradley, and R.D. Clemence, Jr., 1988. "Model Integration with a Typed Executable Modeling Language," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, Vol. III, Decision Support and Knowledge Based Systems Track,* Benn R. Konsynski, ed., 403–410.

[14] G.H. Bradley, and R.D. Clemence, Jr., 1987. "A Type Calculus for Executable Modeling Languages," *IMA Journal of Mathematics in Management 1*, 277-291.

[15] L. Cardelli and P. Wegner, 1985. "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys 17*, 471-522.

[16] J. Choobineh, 1991. "SQLMP: A data sublanguage for the representation and formulation of linear mathematical models,"*ORSA Journal of Computing 3*.

[17] E. Davis, 1990. *Representations of Commonsense Knowledge,* pp. 12–14, Morgan Kuafmann Publishers, San Mateo, CA.

[18] R. Fourer,1983. "Modeling Languages versus Matrix Generators for Linear Programming," *ACM Transactions on Mathematical Software 9*, 143–183.

[19] R. Fourer, D. Gay, B. W. Kernighan, 1990. "A Mathematical Programming Language," *Management Science 36*, 519-554.

[20] A.M. Geoffrion, 1987. "An Introduction to Structured Modeling", *Management Science 33*, 547-588.

[21] A.M. Geoffrion, 1989. "Computer-Based Modeling Environments," *European Journal of Operational Research 41* 33-43.

[22] A.M. Geoffrion, 1990. "Reusing Structured Models Via Model Integration", Proceedings of the Twenty Second Hawaii International Conference on the System Sciences, IEEE Press, 601-611.

[23] A.M. Geoffrion, 1991. "FW/SM: A Prototype Structured Modeling Environment", *Management Science 37*, 1513-1538.

[24] A.M. Geoffrion, 1992. "The SML Language for Structured Modeling: levels 1 and 2," *Operations Research 40*, 38-57.

[25] A.M. Geoffrion, 1992. "The SML Language for Structured Modeling: levels 3 and 4," *Operations Research 40*, 58-75.

[26] M. Genesereth, N. Nilsson, 1987. *Logical Foundations of Artificial Intelligence*, Morgan Kaufman, Los Altos, CA.

[27] M. J. C. Grodon, 1988. *Programming Language Theory and its Implementation*, Prentice-Hall, Englewood Cliffs, NJ.

[28] H.J. Greenberg, 1993. "MODLER: Modeling by object-driven linear elemental relations," *Annals of Operations Research 38*, 239-280.

[29] C. A. Gunter, J. C. Mitchell, 1994. *Theoretical Aspects of Object-oriented Programming: Types, Semantics, and Language Design*, The MIT Press, Cambridge, MA.

[30] A. Goldberg, 1985. *Smalltalk-1980: The Language and its Interactive Environment*, Addision-Wesley, Reading (Mass.), 1985.

[31] A. C. Hindmarsh, 1980, "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers," *ACM-Signum Newsletter 15*, 10-11.

[32] S.M. Hong, M. Mannino, 1995. "Formal Semantics of the Unified Modeling Language," *Decision Support Systems, 13*, 263-293.

[33] P. Jackson, H. Reichgelt, F. van Harmelen, 1989. *Logic-Based Knowledge Representation,* The MIT Press, Cambridge, MA.

[34] C. Jones, 1990. "An Introduction to Graph Based Modeling Systems, Part I: Overview," *ORSA Journal on Computing 2*, 136-151.

[35] C. Jones, 1991. "An Introduction to Graph Based Modeling Systems, Part II: Graph Grammars and the Implementation," *ORSA Journal on Computing 3*, 180-206.

[36] D. Katiyar, D. Luckham, J. Mitchell, 1994. "A type system for prototyping languages," *Proceedings of the ACM conference on Principles of Porgramming Languages*, 138-150.

[37] J. Kotteman, D. Dolk, 1991. "Model Integration and a Theory of Models", *Decision Support Systems 9*, 51-63.

[38] R. Krishnan, 1990. "A Logic Modeling Language for Model Construction," *Decision Support Systems, 6*, 123-152.

[39] R. Krishnan, P. Piela, A. Westerberg, 1993. "Reusing Mathematical Models in Ascend," In: C. Holsapple and A. Whinston, Eds., *NATO ASI on Decision Support Systems*, Springer-Verlag, New York, NY, 275-294.

[40] B. Kristoffersen, 1995. "A critical analysis of Standard ML," Working Paper 193, Institute for Informatics, University of Oslo.

[41] R. Macgregor, 1990. "The evolving technology of classification-based knowledge representation systems," in *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, John Sowa, Ed.

[42] B. Meyer, 1988. *Object-oriented Software Construction*, Prentice Hall International (UK), Ltd.

[43] W. A. Muhanna, 1992. "On the Organization of Large Shared Model Bases", *Annals of Operations Research 38*, 359-396.

[44] B. A. Murtagh, M. A. Saunders, 1985. "MINOS User's Guide", Technical Report SOL 83-20, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University, Palo Alto, CA.

[45] L. Neustadter, 1994. "A Formalization of Expression Semantics for an Executable Modeling Language," *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA, 492-504.

[46] A. Ohori, P. Buneman, 1994. "Static Type Inference for Parametric Classes," in [29].

[47] L. Paulson, 1991. *ML for the Working Programmer*, Cambride University Press.

[48] P. Piela, 1989. "Ascend: An Object-Oriented Computer Environment for Modeling and Analysis", Ph. D. Dissertation, Carnegie Mellon University.

[49] P. Piela, T. Epperly, K. Westerberg, A. Westerberg, 1991. "Ascend: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language," *Computers and Chemical Engineering 15*, 53-72.

[50] P. Piela, R. McKelvey, A. Westerberg, 1992. "An Introduction to Ascend: Its Language and Interactive Environment," *Journal of Management Information Systems 9*, 91-121.

[51] P. Piela, B. Katzenberg, R. Mckelvey, 1993. "Integrating the User into Research on engineering Design Systems," *Research in Engineering Design*, forthcoming.

[52] R. Reiter, 1984."Towards a logical reconstruction of relational database theory,", in *On Conceptual Modeling*, Eds., Brodie, Mylopoulos, and Schimdt, Springer Verlag, New York.

[53] I. Sutherland, 1963. "Sketchpad: A Man-Machine Graphical Communication System", Technical Report No. 296, MIT Lincoln Laboratory.

[54] F. Vicuna, 1990. "Semantic Formalization in Mathematical Modeling Languages," Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, CA.

[55] P. Winston, B. Horn, 1995. *LISP*, Addision Wesley Publsishing Company.

[56] W. A. Woods, J. Schmolze, "The KL-ONE family," *Computers and Mathematics with Applications 22*, special issue on Semantic Networks in Artificial Intelligence.

A Transportation model (and components) in Ascend

```
ATOM customerId REFINES string;
END customerId;

ATOM plantId REFINES string;
END plantId;

ATOM demand REFINES solver_var;
   low_bound := 0;
END demand;


ATOM supply_capacity REFINES solver_var;
   low_bound := 0;
END demand;

ATOM flow REFINES solver_var;
   low_bound := 0;
END flow;

ATOM unitCost REFINES solver_var;
   low_bound := 0;
END unitCost;

ATOM cost REFINES solver_var;
   low_bound := 0;
END cost;


MODEL plant;
   sup IS_A supply_capacity;
    customerId IS_A set OF symbol;
   maxCustomer IS_A integer;

   CARD(customerId) <= maxCustomer;

   f[customerId], totalFlow IS_A flow;
   cost[customerId] IS_A unitCost;
   totalFlow = SUM(f[customerId]);
   totalFlow <= sup;

   shipmentCost IS_A cost;
   shipmentCost = SUM(f[i]*cost[i] | i IN customerId);
END plant;

MODEL customer;
   dem IS_A demand;
   plantId IS_A set OF symbol;
   f[plantId] IS_A flow;
```

```
   SUM(f[plantId]) = dem;
END customer;

MODEL transportation;
   plantId, customerId IS_A set OF symbol;
   p[plantId] IS_A plant;
   c[customerId] IS_A customer;
   FOR i IN customerId CREATE
      c[i].plantId := [j IN plantId | i IN p[j].customerId];
   END;
   FOR i IN plantId CREATE
      FOR j IN p[i].customerId CREATE
         p[i].f[j], customer[j].f[i] ARE_THE_SAME;
      END;
   END;
   obj : MINIMIZE
            SUM(p[i].shipmentCost | i IN plantId);
END transportation;
```

B Proofs

B.1 Proofs: Preliminary Results

Lemma 1 *Given a consistent collection of user-supplied assertions in Ascend, each inferred type association $F : M \mapsto T$ will be consistent with integrity constraints represented by Axioms 20 and 21. That is, if the initial type associations are consistent, all the type associations inferred via the type inference rules will also be non-range-recursive.*

Proof: Our proof strategy is to show that the application of each type inference rule preserves consistency with respect to Axioms 20 and 21. We need to demonstrate this only for those rules (Axioms 4–6 and 9–10) from which we derive new type associations. The remaining axioms result in inferences of AA or ATS relationships; since we are not proving their consistency, we will not assume in our proof that any AA or ATS statements (derived or declared) are consistent with the integrity constraints.

1. First, consider Axiom 4. We are given that the type association $F : T1 \mapsto T2$ in the LHS of the rule satisfies constraint 20 (i.e., $T2$ does not refine $T1$) and 21. For the *conclusion* to be in violation of Axiom 20, it must be the case that $T2$ refines T. But, we are given (LHS of the axiom) that T refines $T1$. From transitivity of refinement, it follows that $T2$ refines $T1$, which contradicts our assumption. Similarly, the possibility of the conclusion (function F) violating Axiom 21 is easily dismissed since we already assumed that F (in the LHS) is consistent with it.

2. To show consistency of Axiom 5 with respect to constraint 20, we only need to show that $T2$ cannot refine $T1$. That follows simply because we know that $F1$ and $F2$ (in the LHS) satisfy Axiom 21 so that the composition $F1 \circ F2$ cannot have a range ($T2$) that is a subtype of its domain ($T1$). Similarly, the composite function cannot violate Axiom 21 since that would imply that $F1$ or $F2$ (or both) violate it too.

3. For the application of Axiom 9, the violation of Axiom 21 is dealt with as in Axiom 4. Similarly, the conclusion cannot violate Axiom 20, since that would also mean that $F1$ is non-range-recursive which we are given to be false.

4. The proof for the consistency of Axiom 10 is exactly the same as that for Axiom 9.

Since each of the type inference rules that produces a type association ensures that this association is consistent with Axioms 20 and 21, given that it operates on a consistent database, it follows that the entire set of rules preserves consistency of these two constraints. ⋄

Lemma 2 *Any composite function in a consistent collection of user-supplied Ascend assertions must be composed of atomic functions each of which appears only once in the composition.*

Proof: We will prove this result by refutation. Let us suppose that there is a composite function in which the same atomic function f appears twice. Therefore, it must contain a segment of the form $f \circ f$ or $f \circ g \circ f$, where g may be composite. For the first form to be a valid composition, f must have the same domain and range, but that violates the integrity constraint represented by Axiom 20. The second case violates the integrity constraint represented by Axiom 21 since there must be some m and t such that $f : m \mapsto t$ and $g : t \mapsto m$. Hence it is not possible for the same function name to occur twice in the definition of a composite function. ⋄

Corollary 1 (To Lemma 2) *Any valid composite function in Ascend must have a finite composition length. Precisely, it can be written in the form $f_1 \circ \ldots \circ f_n$ (for some finite n) where all the f_i's are distinct atomic functions.*

Proof: This is obvious since there is only a finite number of atomic functions, and each atomic function can occur at most once in any composition. ⋄

Lemma 3 (Finite number of functions) *Given any consistent collection of user-supplied assertions in Ascend, the total number of functions (atomic and composite) is finite.*

Proof: This is easy to see, given the previous results, since the number of atomic functions is finite, and since the remaining functions are composed of a finite sequence of atomic functions. ⋄

Lemma 4 (Comparability of Types) *Given a consistent collection of user-supplied assertions in Ascend, for any function f and any model m, all the types associated with f in m are comparable.*

Proof: We shall prove the theorem by induction on the number of types associated with any function. That is, 1) we know that the result is correct (trivially) if a function has only one associated type; 2) we assume that the result is true for n types $t_1 \ldots t_n$ associated with f in model m; and 3) we need to prove that the $(n+1)^{th}$ type t_{n+1} is comparable with all the other types $t_1 \ldots t_n$. Let us first state our task more precisely. We are given that

$$\begin{array}{ll} f : m \mapsto t_i & \forall i : 1 \ldots n \\ \mathrm{comp}(t_i, t_j) & \forall i \; \forall j : 1 \ldots n \\ f : m \mapsto t_{n+1} & \end{array}$$

and we need to show that

$$\mathrm{comp}(t_i, t_{n+1}) \qquad \forall i : 1 \ldots n \tag{3}$$

Examining our rules for type inference, we can see that there are several ways that $f : m \mapsto t_{n+1}$ could be true: we shall show in each case that the required condition (3) is met.

1. $f : m \mapsto t_{n+1}$ is obtained by refinement—i.e., by combining a declaration $\mathrm{irt}(f, m, t_{n+1})$ with Axiom 6. Then it must satisfy Condition 3 due to the integrity constraint 17, which requires t_{n+1} to be comparable with every existing type t_i of f in m.

2. $f : m \mapsto t_{n+1}$ is a user declaration. Then, as above, it must satisfy Condition 3 due to the integrity constraint 17.

3. $f : m \mapsto t_{n+1}$ is inferred via refinement as an application of Axiom 4. For that to be true, it must be the case (using the LHS of Axiom 4) for some t' that $(f : t' \mapsto t_{n+1})$ and that refines(m, t'). From integrity constraint 19, the statement refines(m, t) can be true only if for each applicable function of t' and m—in particular for function f—all its existing types in t' are comparable with all its existing types in m. Specifically, this means that t_{n+1} (a type of f in t') must be comparable with all types t_i ($i \in [1, n]$) of f in m. That leads to a satisfaction of Condition 3.

4. $f : m \mapsto t_{n+1}$ is inferred as an application of Axiom 9 combined with a declaration aa(f', f, m). For that to be the case, it must be true that $(f' : m \mapsto t_{n+1})$. From the AA integrity constraint (Axiom 15), for aa(f', f, m) to be true, it must be the case that f and f' are type conformable. That is, all of their existing associated types must be comparable. In particular, t_{n+1} (a type of f') must be comparable with all the types t_i ($i \in [1, n]$) of f. Again, Condition 3 is satisfied.

5. $f : m \mapsto t_{n+1}$ is inferred as an application of Axiom 10 combined with a declaration ats(f', f, m). This is analogous to the previous case.

6. Finally, there is the possibility that Axiom 5 (function composition) was applied in inferring $f : m \mapsto t_{n+1}$. Now f can either be an atomic function or a composite of two or more atomic functions. If it is atomic, this possibility does not arise (i.e., one of the previous 5 cases must be valid). Therefore, our proof is now complete for atomic functions.

   If f is composite, it follows that f must be of the form $f_1 \circ f_2$ where (without loss of generality) f_1 is atomic. For Axiom 5 to have been applied then, it must be true that there is some type t such that $f1 : m \mapsto t$ and $f2 : t \mapsto t_{n+1}$. Since f is the same as $f_1 \circ f_2$, and we know that f has types t_i ($i \in [1, n]$) in m, it follows (Axiom 5) that there must be types τ_i ($i \in [1, n]$) such that $f_1 : m \mapsto \tau_i$ and $f_2 : \tau_i \mapsto t_i$. Since f_1 is atomic, all of its types in m (i.e., all τ_i and t) must be conformable; let Γ be the most refined of these (i.e., refines(Γ, τ_i) and refines(Γ, t) for all i).

   Next, since a) $f_2 : \tau_i \mapsto t_i$, b) refines(Γ, τ_i), c) $f_2 : t \mapsto t_{n+1}$, and d) refines(Γ, t), it follows (from Axiom 4) that $f_2 : \Gamma \mapsto t_i$ ($i \in [1, n]$) and $f_2 : \Gamma \mapsto t_{n+1}$. Therefore, if we can show that all the types of f_2 in Γ are comparable (in particular, that t_{n+1} is comparable) with the rest), we are done. If f_2 is atomic, we have already shown that all its types must indeed be comparable. If it is composite, then by the same process we can infer that there is an f_2' that has these same types in some Γ'. Again, if f_2' is atomic, we're done; else, we recurse through the proof. From Lemma 1, it follows that this process will terminate in a finite number of steps with an atomic function that has the types t_i ($i \in [1, n+1]$), and that these types are all comparable.

That concludes our proof of the comparability of all types associated with a function in a model. ◇

B.2 Proofs: Consistency of Type System

Lemma 5 (Type conformability–AA) *Consider functions f_1 and f_2 and a type t such that $aa(f_1, f_2, t)$ is deduced from a consistent collection of user-supplied assertions in Ascend. Then, f_1 and f_2 must be type conformable in t.*

Proof: To prove the result, we need to show that for any types t_1 and t_2 such that $f_1 : t \mapsto t_1$ and $f_2 : t \mapsto t_2$, t_1 and t_2 are comparable. Our proof relies on the fact that a statement such as aa(f_1, f_2, t) can only be inferred using either Axiom 7 or Axiom 11. Further, if f_1 and f_2 were atomic, it must be that Axiom 7 was applied.

1. First, consider the case where both f_1 and f_2 are atomic. Since Axiom 7 must have been used, it must be true that aa(f_1, f_2, t') for some type t' where t refines t'. Since the number of types is finite, we can assume that aa(f_1, f_2, t') is a declared statement and, hence, satisfies type conformability (Axiom 15). Since $f1'$ and $f2'$ are type conformable in t', all their types in t' are comparable. Let t^* be the most refined of these. Axiom 9 (applied to aa(f_1, f_2, t') ensures that t^* is associated with both f_1 and f_2 in t'.

Now, consider the types t_1 and t_2 of f_1 and f_2, respectively, in t. Either $t*$ refines both t_1 and t_2 (which are not required to be types of f_1 and f_2, respectively, in t') or at least one of them refines t^*. For the former to be true, t_1 and t_2 must be comparable (due to constraint 18), and we're done. For the latter to be true, assume (without loss of generality) that t_1 refines t^*. Then, applying Axiom 9 (since $\text{aa}(f_1, f_2, t)$, $f_1 : t \mapsto t_1$, and $f_2 : t \mapsto t^*$), we get $f_2 : t \mapsto t_1$. But, since we already know that $f_2 : t \mapsto t_2$, it must be the case (using Lemma 4) that t_1 and t_2 are comparable.

2. Next, consider the case where at least one of f_1 and f_2 is composite. We've already shown that Axiom 7 preserves type conformability. If Axiom 11 were used in inferring $\text{aa}(f_1, f_2, t)$, there must be a) functions f, f_1' and f_2' such that $f_1 = f \circ f_1'$ and $f_2 = f \circ f_2'$, and b) a type m such that $\text{aa}(f_1', f_2', m)$ and $f : t \mapsto m$. Again, since the composite functions f_1 and f_2 must have a finite length (Corollary 1), repeated application of Axiom 11 would leave us with atomic functions that are-alike. Without loss of generality, let us assume that f_1' and f_2' are these atomic functions that are-alike in m. From Part 1 of the proof, it follows that they must be type conformable.

   Now we know that $f_1 : t \mapsto t_1$ and $f_2 : t \mapsto t_2$. Since $f_1 = f \circ f_1'$, there must be an m_1 such that $f : t \mapsto m_1$ and $f_1' : m_1 \mapsto t_1$. Similarly, there must be an m_2 such that $f : t \mapsto m_2$ and $f_2' : m_2 \mapsto t_2$. Since f is associated with both m_1 and m_2—as well as m—in t, Lemma 4 ensures that m, m_1 and m_2 are comparable. Either a) both m_1 and m_2 refine m, or b) m refines both m_1 and m_2, or c) one of them (say, m_1 refines m and m refines the other (m_2). We consider each of these cases below.

   (a) Both m_1 and m_2 refine m: Let's assume, without loss of generality, that m_1 refines m_2. Since m_1 refines m we get from Axiom 7 that $\text{aa}(f_1', f_2', m_1)$. Since the two functions are atomic, they must be type conformable in m_1 as well. Since m_1 refines m_2 and $f_2' : m_2 \mapsto t_2$, Axiom 4 gives us $f_2' : m_1 \mapsto t_2$. Since f_1' and f_2' are type conformable in m_1, all their types—including t_1 of f_1' and t_2 of f_2'—must be comparable.

   (b) m refines both m_1 and m_2: Since m refines m_1 and $f_1' : m_1 \mapsto t_1$, it follows from Axiom 4 that $f_1' : m \mapsto t_1$. Similarly, we get $f_2' : m \mapsto t_2$. But we know that f_1' and f_2' are conformable in m. Hence it follows that t_1 and t_2 are comparable.

   (c) m_1 refines both m which refines m_2: Since m refines m_2, Axiom 4 gives $f_2' : m \mapsto t_2$. Since m_1 refines m, Axiom 4 gives $f_2' : m_1 \mapsto t_2$ and Axiom 7 gives $\text{aa}(f_1', f_2', m_1)$. Therefore, f_1' and f_2' are conformable in m_1. Hence all their types—including t_1 of f_1' and t_2 of f_2'—must be comparable.

We have shown that arbitrary types t_1 and t_2 of f_1 and f_2, respectively, are comparable. That establishes that any two functions in an AA equivalence class are type conformable. ⋄

Lemma 6 (Type conformability–ATS) *Consider functions f_1 and f_2 and a type t such that ats(f_1, f_2, t) is deduced from a consistent collection of user-supplied assertions in Ascend. Then, f_1 and f_2 must be type conformable in t.*

Proof: The proof is an extension of the proof for Lemma 5. Given $\text{ats}(f_1, f_2, t)$, we need to show that for any types t_1 and t_2 chosen as before, t_1 and t_2 are comparable. In this case, the proof relies on the fact that $\text{ats}(f_1, f_2, t)$ can only be inferred via Axioms 8, 12, and 13.

1. If f_1 and f_2 are atomic, the proof is entirely analagous to Part 1 of the previous proof.

2. If f_1 and f_2 are composite, $\text{ats}(f_1, f_2, t)$ must have been involved—in the general case—repeated application of Axioms 12 and 13. Given that any composite function has a finite length, the statement must have been derived from $\text{ats}(f_1', f_2', m)$ for some atomic functions f_1' and f_2'. If Axiom 12 were applied, the proof that type conformability is preserved is analagous to Part 2 of the previous proof. If Axiom 13 were applied, we know that $m = t$ and there is an applicable function f (of m, or t) such that $f_1 = f_1' \circ f$ and $f_2 = f_2' \circ f$.

Now consider the types t_1 and t_2. Since $f_1 : m \mapsto t_1$, there must be an m_1 such that $f_1' : m \mapsto m_1$ and $f : m_1 \mapsto t_1$. Similarly, since $f_2 : m \mapsto t_2$, there must be an m_2 such that $f_2' : m \mapsto m_2$ and $f : m_2 \mapsto m_2$. Since ats(f_1', f_2', m), all their types—in particular m_1 and t_2—in m are comparable (from Part 1 of the proof). Assume that m_1 refines m_2. By applying Axiom 4 to $f : m_2 \mapsto t_2$, we get $f : m_1 \mapsto t_2$. But we already know that $f : m_1 \mapsto t_1$. Since all types of f must be comparable (Lemma 4), it follows that t_1 and t_2 are comparable.

That concludes our proof that all functions in an ATS class are type conformable. $\diamond$

Theorem 1 (Type inference rules preserve consistency) *Given a collection of user-supplied assertions that is consistent with all the integrity constraints (Axioms 15–21), all deductions will also be consistent with all constraints. That is, the type inference rules preserve consistency with respect to the integrity constraints.*

Proof : We prove this result by proving the impossibility of violation for each integrity constraint. The result follows trivially from Lemma 5, in the case of Axiom 15, and from Lemma 6 for Axiom 16. In the case of Axiom 17, note that a) IRT statements are never inferred (so cannot cause a violation), and b) Lemma 4 established that all types of a function are comparable (and so any inferred type association cannot violate the constraint). Similarly, Axiom 18 can not be violated since type refinement is never inferred except by transitivity (which, by definition, guarantees comparability). The same argument applies to ensure that Axiom 19 can never be violated. Finally, Lemma 1 established that the constraints represented by Axioms 20 and 21 are preserved by the type inference rules. $\diamond$

B.3 Proofs: MRT and Equivalence Classes

Theorem 2 (Uniqueness of MRT) *Given any consistent collection of user-supplied assertions in Ascend, each function has a unique* most refined type.

Proof: From the definition of MRT, this requires us to establish, given any model m and any function f applicable to m, that all the types associated with f in m are *comparable*. That is exactly what Lemma 4 establishes. Hence the result. $\diamond$

Theorem 3 (Finiteness of MRT computation) *Given any consistent collection of user-supplied assertions in Ascend, the* most refined type *of any function can be computed in a finite number of steps, as can any other deduction in Ascend.*

Proof: We assume a control strategy based on forward-chaining reasoning as discussed below. The method will consist of several iterations, beginning with the database of user-supplied assertions. Let us denote this database by Δ_0. (It will consist of ground atomic sentences of the form $F : M \mapsto T$, refines$(T1, T2)$, irt(F, M, T), aa$(F1, F2, M)$, and ats$(F1, F2, M)$.) In each iteration i, various inference rules will be applied and inferred results (also ground atomic sentences as above) added to the database. Without loss of generality, assume that the results are added at the end of the iteration. Let Δ_i denote the state of the database at the end of iteration i. The process is terminated when $\Delta_i = \Delta_{i+1}$ for any $i \geq 0$. We will show that each iteration is finite, and that the termination criterion is satisfied in a finite number of iterations. The most critical aspect of this procedure is what happens in each *iteration*, which we define below. It should be evident to the reader that this reasoning procedure is consistent and complete with respect to the axioms.

Each iteration consists of a number of steps, where each step corresponds to one of the type inference rules (Axioms 4–13). These rules are applied in conjunction with Axioms 1–3 that describe the mathematical properties of the Ascend operators. In each step of iteration i, the corresponding rule is applied, over and over again, to the sentences in the database Δ_{i-1} until no more inferences can be generated. Each step will be finite, since the number of objects (type names, function names) and sentences in the database is finite. Therefore, each iteration will terminate finitely.

Next, we note that the total number of ground atomic sentences is bounded since the number of constant terms is finite. Recall that at the end of any iteration, we continue the procedure only if the iteration produced at least 1 new sentence. Therefore, the number of iterations is bounded, in the worst case, by the maximum possible number of sentences in the database. Hence the process will terminate in a finite number of steps. At that time, the MRT of each function can be computed as an application of Axiom 14. In fact, any particular sentence will be true if and only if it is present in the final database of sentences. ⋄

Lemma 7 (MRT of AA or ATS Equivalence Class) *All elements of an (AA or ATS) equivalence class have the same MRT.*

Proof: Lemma 5 (Lemma 6) establishes that all elements (functions) in an AA (ATS) equivalence class are type conformable. That means all their types in the given model are comparable. Let τ be the most refined of these. Axiom 9 (10) ensures that τ will be inferred as an associated type for all the functions in the AA (ATS) equivalence class. Hence, all of them will have the same MRT, τ. ⋄

Theorem 4 (MRT of a Clique) *All elements of a clique of functions will have the same MRT.*

Proof: This follows as a corollary to Lemma 7 and by definition of a clique. ⋄

Contents

1 Modeling Languages: Typing and Semantics **1**
1.1 Trends in Modeling Languages 1
1.2 Focus and Motivation 2
1.3 Related work 3
1.3.1 Type Inference in Programming Languages 3
1.3.2 Concept and Frame Description Languages 4

2 The Ascend Modeling Language **4**
2.1 Elementary Types 5
2.2 Atom Types 5
2.3 Model Types 6
2.4 Equivalencing model properties: The ARE_THE_SAME operator 6
2.5 Defining model hierarchies; the REFINES operator 7
2.6 Declaring similarity between model properties: The ARE_ALIKE operator 8
2.7 Modifying the type of a property: The IS_REFINED_TO operator 8
2.8 An example of model integration in Ascend 9
2.9 Summary 9

3 Developing Formal Semantics of Type Systems **10**
3.1 The Problem Statement 10
3.2 Strategy: A Bilevel Conceptualization 10

4 Logical Formalization of Ascend's Type System **11**
4.1 Mathematical Properties of Ascend Operators 12
4.2 Reasoning about Types 13
4.3 Integrity Constraints for Operator Declarations 15

5 Analysis **16**
5.1 Preliminary Results: Functions and Types 16
5.2 Consistency of Type System 17
5.3 Most Refined Types and Equivalence Classes 17

6 Discussion **18**

A Transportation model (and components) in Ascend **23**

B Proofs **24**
B.1 Proofs: Preliminary Results 24
B.2 Proofs: Consistency of Type System 26
B.3 Proofs: MRT and Equivalence Classes 28

List of Figures

1 Components of a transportation problem. 32
2 Pictorial depiction of Ascend objects 33
3 Model of a plant: conceptualization and statement in Ascend. 34
4 Model of a customer: conceptualization and statement in Ascend. 35
5 Setting up the transportation model with plant and customer models as building blocks. The ATS operator is used to connect plant and customer flows. 36
6 Ascend's Type Inference Rules 37
7 Dependencies between results 38

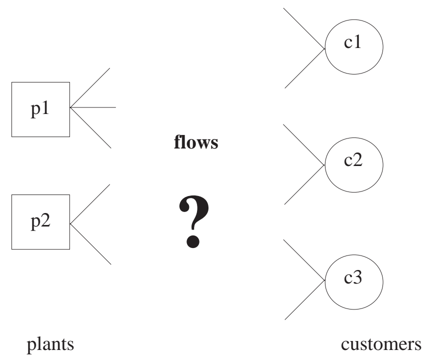


Figure 1: Components of a transportation problem.

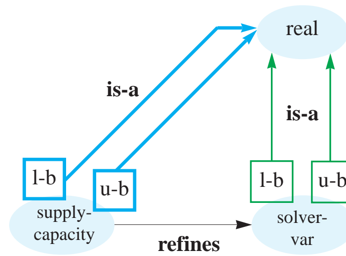
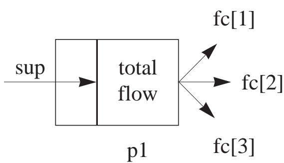


Figure 2: Pictorial depiction of Ascend objects: Types are denoted by shaded ellipses, their properties are grey rectangles (the *value* property of *solver_var* is not shown here). The properties point to their own type (via grey edges). The darker edges are used to depict relationships between types. Finally, inference is depicted with thick lines: note the thick rectangular slots (and thick edges from them). E.g., the node supply-capacity has its slots inferred from the “refines” relation, and the type of these slots is also inherited from solver-var.




```
MODEL plant;

 1 customerId IS_A set OF symbol;
 2 maxCustomer IS_A integer;
 3 CARD(customerId) <= maxCustomer;

 4 sup IS_A supply_capacity;

 5 fc[customerId], totalFlow IS_A flow;
 6 cost[customerId] IS_A unitCost;
 7 totalFlow = SUM(fc[customerId]);
 8 totalFlow <= sup;

 9 shipmentCost IS_A cost;
10 shipmentCost = SUM(fc[i]*cost[i] | i IN customerId);

END plant;
```

Figure 3: Model of a plant: conceptualization and statement in Ascend.

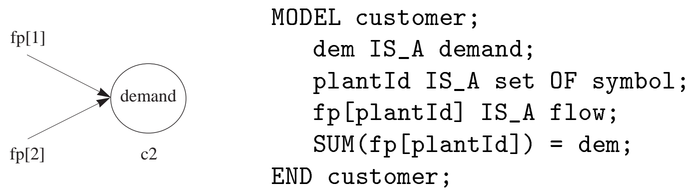
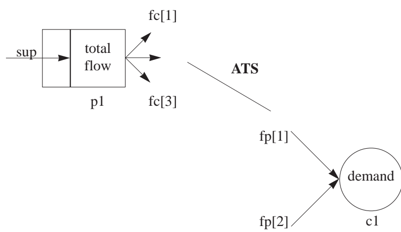


Figure 4: Model of a customer: conceptualization and statement in Ascend.




```
MODEL transportation;
 1 plantId, customerId IS_A set OF symbol;
 2 p[plantId] IS_A plant;
 3 c[customerId] IS_A customer;

 4 FOR i IN customerId CREATE
 5    c[i].plantId := [j IN plantId | i IN p[j].customerId];
 6 END;

 7 FOR i IN plantId CREATE
 8   FOR j IN p[i].customerId CREATE
 9      p[i].fc[j], c[j].fp[i] ARE_THE_SAME;
10   END;
11 END;

12 obj : MINIMIZE
13    SUM(p[i].shipmentCost | i IN plantId);
END transportation;
```

Figure 5: Setting up the transportation model with plant and customer models as building blocks. The ATS operator is used to connect plant and customer flows.

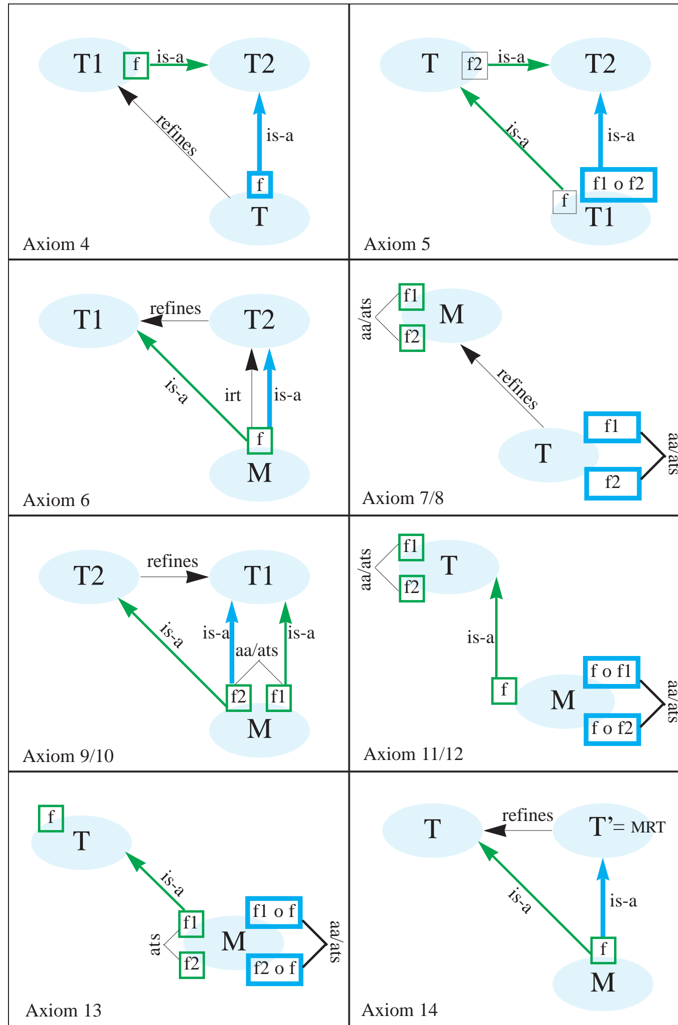


Figure 6: Ascend's Rules for Reasoning with Types: The convention is the same as in Figure 2. The thick rectangles and edges indicate the conclusions resulting from type inference on the statements indicated in the remaining shaded ellipses (types), rectangles (slots), thin black edges (declared relations between types) and thin grey edges (explicitly declared is-a/IRT statements).

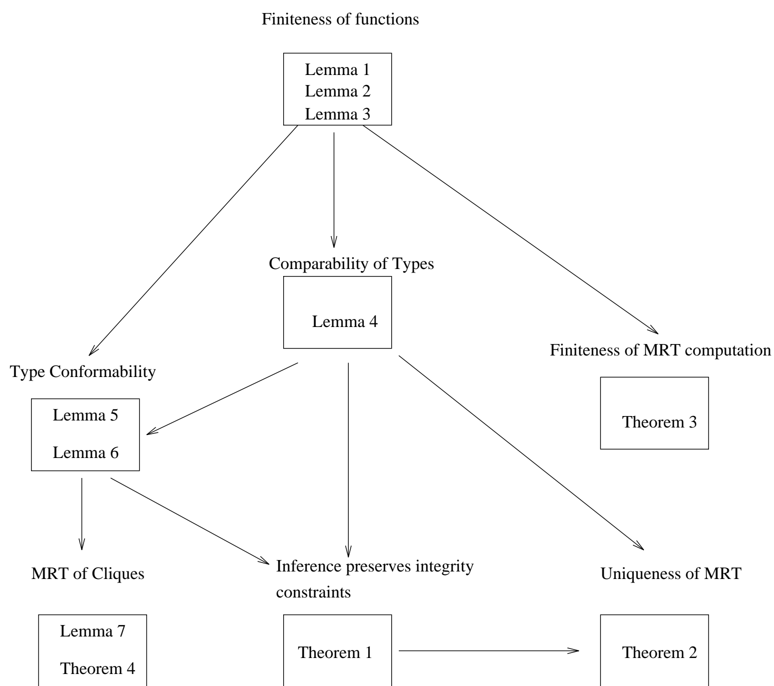


Figure 7: Dependencies between results