# A Declarative Extension of IDL-based Type Definitions within Open Distributed Environments

Arno Puder

Department of Computer Science
University of Frankfurt
D-60054 Frankfurt
Germany
*puder@informatik.uni-frankfurt.de*

**Abstract**

An open distributed environment can be perceived as a service market where services are freely offered and requested. Any infrastructure which seeks to provide appropriate mechanisms for such an environment has to include some mediator functionality to bring together matching service requests and service offers. The matching algorithm that the mediator must perform commonly builds upon an IDL-based type definition for service specification. In order for the matching algorithm embedded in this mediator to succeed, the types of various services have to be standardized and distributed to all interested parties. We argue that those well defined "standards"[1] are too inflexible and even contradict the idea of an open service market. Therefore we propose a new way to augment an arbitrary IDL-based type definition by a semantic tag. Whereas several other authors have used pre- and post-conditions to add the notion of behavior description, we use the well established field of Horn clauses as a semantic extension. The advantage lies in the nature of the declarative semantic and it will be shown that the need for a well defined standard is abolished within so called type families.

**Keywords:** Open Systems, Trading, Subtyping, IDL, Declarative Semantics

## 1 Introduction

There have been several approaches to type theory, depending on which aspects of a problem are to be modeled. One common employment of type theory derives from the need to recognize programming errors in a high level language. Given the simple assignment statement $x{:=}a$ in a typed programming language, both entities, the variable $x$ as well as the expression

[1]The term "standard" is not to be confused with ANSI- or ISO-like standards. For the context of this paper a standard is an agreement (or convention) between client and server implementors about what the services are going to be called and what the arguments will be.

$a$ are typed. The compiler can use the redundant type information to ensure that the type of $a$ is compatible with the type of $x$. A type here resembles much a "suit of armor" [Card85], where it protects an underlying data item from invalid operations. The simplest requirement in the above example would be to check that the type of the left hand side of the assignment statement is identical to the type of the right hand side.

With the rise of the object paradigm, the definition of a type has changed due to the nature of the object model. One important feature of the object model is the encapsulation of data and code, which consequently decouples an object's interface from it's actual implementation[2]. The most general characteristics of any type definition are that a type denotes a set of objects with time-invariant, externally visible properties. *Time-invariant* means that the properties hold during the whole lifetime of an object and *externally visible* refers to the requirement of encapsulation, whereby implementation specific details are hidden within the object (see [Amer91]).

In order to define a type, the relevant properties have to be identified with respect to assumptions about the object model and the environment under consideration. This paper focuses on the implications of a type definition within open, distributed object-based environments which are characterized by an extensive network of independent vendors and customers. Service providers (vendors) and service requestors (customers) are modeled by objects (called server and client objects, respectively). As will be shown later, a type here is not only a basis for a "treaty" between a client and server object but also the only means for a client to find a suitable server in the first place. Therefore a type has the flavor of a standard which all participating parties have to agree upon *a priori*.

Within the context of this paper we assume that a type definition is based on some *interface definition language* (IDL for short) as the CORBA-IDL or DCE-IDL (see [OMG91] and [OSF92] respectively). The grammer defined by an IDL can be used for an interface specification for a set of objects. The main goal of this paper will be to extend an arbitrary IDL-based type definition by a declarative description which will weaken the need for an a priori, tight standard.

The rest of this paper is organized as follows: the next section defines the environment which serves both as a basis for this paper and as a motivation for the introduction of declarative types. Section 3 defines the notion of a type family which will be the starting point for a declarative extension to IDL-based type definitions. In section 4 an example is given which is followed by a formal definition of a declarative type. Section 6 presents an implementation for hierarchical programs (the proof of correctness is given in Appendix B). A summary and outlook conclude this paper.

# 2 Environment

In this section we present a basic model called the *object graph* to motivate some consequences on type definitions within open distributed systems. As the name implies, the framework builds upon the object model. The results given at the end of this section serve as a starting point for the so called *type families*, which will be discussed in detail in the following section.

[2]Languages like C++ do not make a distinct difference between type and class concept. Strictly speaking, a class is one possible implementation for a type.

## 2.1 Object based computations

The environment, which serves as a basis for this paper, builds upon the classical definition of an object (see [Booch91]): "An object has state, behavior and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable." Using this definition, a problem domain may be decomposed as a set of interacting and co-operating objects. A snapshot of such an object-based computation may be visualized as a directed graph, where nodes represent objects and arcs represent references. A reference (or arc) is therefore a referral of an object's identity. The direction of the arc determines whose identity is known to whom. For an object to hold a reference to another object means to know about the existence of this particular instance, allowing operation invocations (also commonly called method invocations). Thus a directed arc between two nodes (objects) represents the ability to invoke operations along the direction of this arc (i.e. the service provider is at the arc head, and the requester is at the tail). Service providers are also called server objects and service requesters are called client objects, respectively. The directed graph will be called an *object graph*.

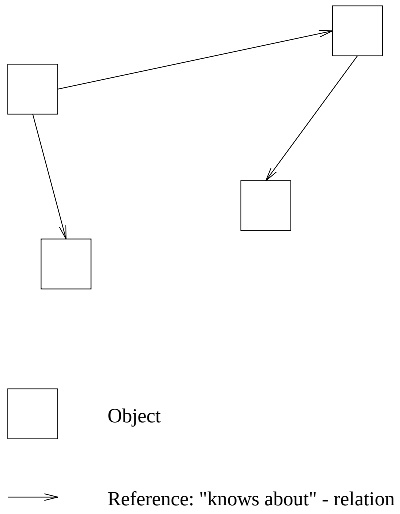


**Figure 1:** Directed graph as a snapshot of an object based computation

Both references and objects are typed and the definition of a type is assumed to be based upon some IDL. This view contributes to an elementary implication of the object model: the encapsulation of data and code. The definition of a type as an interface specification emphasises the decoupling of an object's implementation and the way it is accessed. Implementation details of a server object are irrelevant to a client. From a client object's perspective (the one holding the reference) a reference guarantees a treaty that the server object must fulfill (treaty is meant in terms of allowed method invocations). Polymorphism here occurs when the type of the reference is a super-type of the object to which it points. The server object therefore is a specialization to what the client expects, if it can fulfill the treaty[3].

[3] Polymorphism is often referred to as the "principle of substitutability" where an object of type A may be substituted by an object of type B without anything "bad" happening.

In this paper we are not concerned with questions such as how the nodes of such an object graph are mapped onto a physical environment or how to implement an object's identity (i.e. whether objects (nodes) span machine boundaries; the programming language used to implement the object behavior; the problems with object identity and object migration, etc.). Our primary interest are the mechanisms by which the structure of the object graph may change over time. The following elementary operations can induce a change of the graph:

1. object creation,
2. object deletion,
3. passing of references as parameters of method invocations.

Operations 1 and 2 change the node set and possibly the arc set and operation 3 alters the arc set only. It is obvious that the client who invokes a method on another object may only pass those references that it has stored locally (i.e. "references do not come from nothing"). The only way to pass on another object's identity is by including a reference as a parameter of a method invocation.

## 2.2 Trading and the role of types

As has been motivated in the previous section, we impose no constraints as to how the object graph is actually mapped onto a physical environment but we do assume that the object graph is only loosely coupled. Client and server objects are not only decoupled in terms of locality, but also in terms of development domains. A client object may be implemented by one company, while a server object may be implemented by another. This scenario implies that a client object in general does not control the instantiation of a particular service, but rather has to browse the object graph for an appropriate service provider. These considerations have led to software components like the ODP Trader or CORBA's Request Broker (see [ISO93] and [OMG91] respectively), which serve as a mediator between service requesters and service providers.

The abstract functionality of such a mediator may be modeled within the object graph as a *lookup object* with a well known identity. Each newly created object is implicitly given a reference to this lookup object. The interface of the lookup object usually contains at least two methods: registration of a type and query for an instance of a particular type. Internally the lookup object maintains a database whose entries map from type descriptions to object identities. The parameter for a query therefore contains a type description of the desired service, which the lookup object must match against those stored in its database.

The implementation of the lookup object is based on numerous design decisions: caching of unsatisfiable queries; best match of two types according to some metric, etc. These issues are of no importance here. What is of importance is the matching algorithm itself, independent of any qualitative design decisions (i.e. according to the requirements of the subtype relation, this matching algorithm is simply supposed to return YES or NO).

It is important to think about the role of a type in such an environment. In order for the match algorithm within the lookup object to succeed, a type description must conform to a standard, which all participating parties have to agree upon *a priori*. This standard has to be

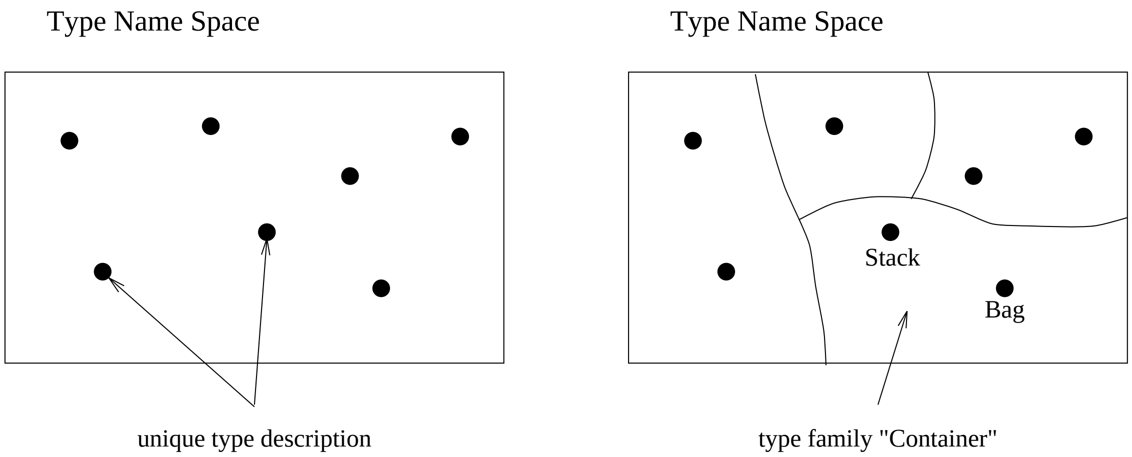


**Figure 2:** Partitioning of the type name space in type families

defined well enough to be matched unambiguously against other types. Current mediators, like the aforementioned ODP Trader, base their match algorithm mainly upon syntactic features of the IDL. The implication is that the exact syntactic structure of a particular service IDL must be communicated to all parties.

We propose a semantic extension to these purely syntactic based IDL's to allow for a certain "fuzziness" in type descriptions. Although the idea of introducing semantics to a type definition is not new (see for example [Amer91], [Lisk93] or [Larch93]), this paper develops a new method based upon the well known field of declarative semantics. The advantages, as will be seen in the following sections, are that the semantic extension is not based upon pre- and post-conditions. Therefore it won't be necessary to change the underlying IDL. Furthermore due to the nature of declarative semantics this will introduce the desired kind of flexibility of type specifications, which weakens the requirement for an a priori, well defined standard. In order to accomplish this task, the next section will introduce the notion of a type family.

# 3 Type Families

This section will establish the notion of a type family which serves as a basis for a declarative extension. In the following the term *type* is to denote a type definition based on some IDL. These IDL's (like DCE-IDL or CORBA-IDL) describe the interface of an object without explicit reference to their underlying semantics. The *semantics* associated with a type is implicit with respect to the behavior intended by the programmer. In order to distinguish between different types, there have to be very strict *syntactic* naming conventions. These conventions correspond to standards, which all participating parties have to agree upon *a priori*. Therefore the name space of all possible types consists of syntactically well distinguished type descriptions, as depicted on the left hand side of figure 2.

It is important to see that although some types denote different behavior they are similar in the way they are accessed by clients. For example consider these two type definitions based on the CORBA-IDL:

```
interface Bag {                        interface Stack
     Bag();                                 Stack();
     ~Bag();                                ~Stack();
     void GetChar( [out] char c );          void PopChar( [out] char c );
     void PutChar( [in] char c );           void PushChar( [in] char c );
     bool IsEmpty();                        bool IsEmpty();
}                                      }
```

From an abstract point of view these two types denote different containers, capable of storing and retrieving data items. The `GetChar()` method of a bag is non-deterministic in the sense that it is unknown in advance which data item `GetChar()` will return. In contrast the stack's `PushChar()` and `PopChar()` methods work according to the LIFO principle and therefore exhibit a deterministic behavior.

Although different in their actual behavior, these two types are used by client objects in a similar way (relative to the naming conventions like for example method names[4]). The informal definition of a type family is to partition the type name space according to these similar usage patterns. This grouping of types to a type family depends highly on those types currently defined in the type name space. Each type family has one *generic type*, representing the usage patterns of all it's members (a generic type for the bag and stack types for example would define a retrieval and deposit method). Note that although this generic type may be specified in terms of the underlying IDL, there is some loss of information. What is missing is some kind of "tag" to the generic type to denote the difference between the various members of a family. This tag will be defined in terms of a declarative Horn clause program.

A *declarative type* therefore identifies the family and the specific members of the family by means of a declarative description based on well known Horn clauses. The logic programming paradigm offers the advantage, that it describes *what* is particular about a type, not *how* this behavior is to be accomplished. Two parties may describe one member of a family differently; yet the lookup object is able to match these two descriptions. The need for a tight standard is reduced to a way to distinguish between different type families and a declarative description methodology to differentiate between various members of one family.

The syntax chosen here for a declarative description is derived from PROLOG, which itself stems from a subset of first order predicate calculus known as Horn clauses. Although the data structures found in PROLOG serve practical needs, we restrict the following example to the pure theory of Horn clauses.

# 4 Extended example

This section will present an extended example on how to use Horn clauses as a semantic tag for members of a type family. The example builds upon the aforementioned type family of containers. As has been explained in the previous section, there must be a generic type which serves as a general interface to any member of this type family. The following paragraphs will show how to use the declarative semantic to specify the difference between two members

[4] See [KONS93] for some hints on how to accomplish this mapping of various types.

of the type family Container, namely a stack and a bag. Note that the major difference between a stack and a bag is the non deterministic behavior of the retrieval method of a bag. The generic type for a type family of all containers is assumed to be defined as (using CORBA-IDL):

```
interface Container {
     Container();
     ~Container();
     void RetrieveChar( [out] char c );
     void DepositChar( [in] char c );
     bool IsEmpty();
}
```

It is important to note that the behavior description of various members of a type family is at an abstract level, sufficient for describing the *difference* in behavior and not a *complete* method by method specification (as done in [Lisk93] and [Larch93]). In the above example, the difference between a bag and a stack may be sufficiently described by choosing two arbitrarily distinct data items. Although the container will in general be able to store more than two kinds of elements, abstracting the behavior with only two elements will suffice for the declarative extension.

Two distinct elements, which can be placed and removed from a container, may be represented by different 1-ary function symbols. Let $f$ and $g$ be two such function symbols. A container with one $f$ and three $g$'s may be represented by the term $g(f(g(g(\epsilon))))$. Note that the constant symbol $\epsilon$ is used to terminate a term. The symbol $\epsilon$ by itself denotes an empty container. In order to provide a framework for a declarative description of the store and retrieval methods, the two predicate symbols *put* and *get*, both with three arguments, are introduced.

To see how a "semantic tag" to the generic type for a *bag* translates to a declarative program, consider the following six definite clauses, denoted by $P_B$ (see Appendix A for the terminology):

$$\begin{aligned}
P_B : get(f(x), x, f(\epsilon)) &\leftarrow && (1)\\
get(g(x), x, g(\epsilon)) &\leftarrow && (2)\\
get(f(x), f(y), z) &\leftarrow \; get(x, y, z) && (3)\\
get(g(x), g(y), z) &\leftarrow \; get(x, y, z) && (4)\\
put(x, f(x), f(\epsilon)) &\leftarrow && (5)\\
put(x, g(x), g(\epsilon)) &\leftarrow && (6)
\end{aligned}$$

Note that the predicate names *get* and *put* are chosen arbitrarily and have in particular no relation to the generic type method names or signatures. Furthermore the behavior of the additional method `IsEmpty()` defined for the generic type is not given, as it's semantic is assumed to be the same for all members of the type family container. This is the most distinguished difference to other approaches for semantic extensions to IDL-based type definitions: not the complete behavior of every method is given, rather only the part of the behavior sufficient for distinction between various members of one type family.

The informal semantics for *put* is that if an element (third argument of the predicate) is added to a container (first argument), the result is a new container with the added element (second argument). A legal ground instance from the Herbrand base $B_L$ would be $put(f(g(f(\epsilon))), f(f(g(f(\epsilon)))), f(\epsilon))$. Here $f$ is added to a container currently holding two $f$'s and one $g$. An illegal ground instance would be $put(f(\epsilon), f(\epsilon), g(\epsilon))$. The informal semantics for *get* is analogous. The element to be removed is denoted by the third argument of the *get*-predicate. The element will be removed from a container (first argument) and resulting in a new container (second argument). In the following example, the legal ground instance $get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon))$ (which is an element for the least Herbrand model $M_{P_B}$ of $P_B$) is derived in a bottom up technique from clauses in $P_B$:

Clause (3): Substitute: $x$ with $g(f(\epsilon))$
$y$ with $f(\epsilon)$
$z$ with $g(\epsilon)$

$$get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon)) \leftarrow get(g(f(\epsilon)), f(\epsilon), g(\epsilon))$$

Clause (2): Substitute: $x$ with $f(\epsilon)$

$$get(g(f(\epsilon)), f(\epsilon), g(\epsilon)) \leftarrow$$

Modus Ponens

$$get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon)) \leftarrow$$

Substitutions of variables with terms are legal operations on clauses due to the universal closure. The last derivation step is allowed because of the Modus Ponens rule. All ground instances in $M_{P_B}$ can be constructed in the same fashion. $M_{P_B}$ now holds all "legal ground instances" (without proof), and thus divides $B_L$ in two partitions. If the outermost function symbol in a term is to denote the last inserted element, then it can be seen that the above declarative program models a bag. Both ground atoms $get(f(g(\epsilon)), f(\epsilon), g(\epsilon))$ and $get(f(g(\epsilon)), g(\epsilon), f(\epsilon))$ are elements of $M_{P_B}$. The former atom would not represent a valid behavior for a stack. The fact that both ground atoms belong to $M_P$ for $P_B$ is the result of the non determinism of the *get*-operation.

The declarative program for a *stack*, denoted by $P_S$, is the same as for $P_B$, except that it lacks clauses 3 and 4. As can be seen, the least Herbrand model $M_{P_S}$ for $P_S$ lacks exactly those *get*-ground instances, which describe the non-deterministic behavior of a bag. Therefore we have the following set inclusion property: $M_{P_S} \subseteq M_{P_B} \subseteq B_L$. This gives a hint on how the subtype relation for declarative types will be defined in the next section.

To give an idea of the declarative nature of this type system, consider the following alternate program for a stack (denoted by $P_{S'}$):

$$\begin{aligned}
P_{S'}: get(x, y, f(\epsilon)) &\leftarrow equal(x, f(y)) && (1)\\
get(x, y, g(\epsilon)) &\leftarrow equal(x, g(y)) && (2)\\
equal(x, x) &\leftarrow && (3)\\
put(x, f(x), f(\epsilon)) &\leftarrow && (4)\\
put(x, g(x), g(\epsilon)) &\leftarrow && (5)
\end{aligned}$$

Although the pure syntactic structure of $P_S$ and $P_{S'}$ differ, it can easily be verified that the least Herbrand model of $P_{S'}$ contains exactly the same *get*-ground instances as those of $P_S$ (without proof). What has happened here is that $P_{S'}$ contains a new predicate *equal* which denotes syntactical equivalence. The program $P_{S'}$ is used to describe the same type of object: a stack. This example shows the expressiveness of declarative types. Although the two programs $P_S$ and $P_{S'}$ differ in their actual syntactical representation, they mean the same (except for the *equal*-ground instances).

The program $P_{S'}$ suggests that a programmer should be given the freedom to use as many predicates as necessary. These "helping"-predicates however are not subject to the subset relationship between the least Herbrand model of two programs. It is therefore inevitable to specify the relevant predicates, i.e. those predicates whose ground instances will be compared by a matching algorithm.

# 5 Formal definition of a declarative type

In this section a formal definition of a declarative type will be established. The term declarative type here is meant as a declarative extension to an IDL-based type notation. As has been mentioned earlier, the syntactic structure of the underlying IDL is not modified, rather the declarative extension is to be understood as an orthogonal augmentation (i.e. merely a semantic tag). For that reason, it is not necessary to tailor a declarative type for some specific IDL and the term IDL is meant to be treated as a generic expression for any IDL notation.

**Def.: Declarative type** $DT$**:** Let $I$ be an IDL-based notation and $L$ a first order language. A declarative type $DT$ is a triple $(S, P, R)$, where $S$ is an interface specification based on $I$, $P$ is a finite set of definite clauses based on $L$ and $R$ is a finite set of relevant predicate symbols based on $L$, such that for each predicate symbol in $R$ exists a definition in $P$.

**Def.: Subtype relation** $\leq_{DT}$**:** Let $T_1 = (S_1, P_1, R_1)$ and $T_2 = (S_2, P_2, R_2)$ be two declarative types, where $S_1$ and $S_2$ are based on $I$. Let $M_{P_1}$ be the least Herbrand model of $P_1$ and $M_{P_2}$ the least Herbrand model of $P_2$ respectively. Let $\tilde{M}_{P_1} := \{p \in M_{P_1} \mid$ the predicate symbol of the ground instance $p$ is an element of $R_1\}$. Definition of $\tilde{M}_{P_2}$ respectively. $T_1$ is a subtype of $T_2$ (denoted by $T_1 \leq_{DT} T_2$) iff

1. $S_1 =_{IDL} S_2$
2. $R_1 = R_2$
3. $\tilde{M}_{P_1} \subseteq \tilde{M}_{P_2}$

Note that the above two definitions can only be applied to positive atoms. The subtype relation still has to be extended for negation. The condition $S_1 =_{IDL} S_2$ of the subtype relation for declarative types represents the traditional interface matching algorithm found in common trader implementations. The equality relation $=_{IDL}$ may therefore be defined as syntactic equivalence. However the equality relation could be defined in terms of a more general equivalence relation which would match syntactically different types (like the stack and bag examples from a previous section). Work on object oriented interoperability (see [Kons93]) may serve as a basis for this equivalence relation.

# 6 Transformation Algorithm

Here, we will present an algorithm which checks the subtype relation $\leq_{DT}$ for a subset of the Horn clauses given two declarative types[5]. Unfortunately there exists no algorithm for $\leq_{DT}$ based on the complete Horn clause set. The (informal) reason for this is, that testing the subset property $\tilde{M}_{P_1} \subseteq \tilde{M}_{P_2}$ is equivalent to a logical implication (the conjunction of all clauses from $P_1$ implies the conjunction of all clauses from $P_2$). Thus allowing arbitrarily Horn clauses would be equivalent of testing satisfiability of first order expressions, which is undecidable. Therefore we have to restrict the set of all Horn clauses to obtain a decidable subset. The subset chosen here are the hierarchical programs (see Appendix A for a definition). The following algorithm transforms a hierarchical program to a set of unit clauses (i.e. facts), which are semantically equivalent:

**Algorithm:** Transformation of a hierarchical program to a set of unit clauses.
**Input:** $P$: a finite, hierarchical definite program with $T_P \uparrow \omega = T_P \downarrow \omega$.
**Output:** $P_u$: semantically equivalent program, containing only unit clauses.

1. $k := 0; P_0 := P$
2. If all clauses in $P_k$ are unit clauses, goto 6.
3. Let $A \leftarrow B_1, \ldots, B_n$ be a non-unit clause in $P_k$, where the definition of each $B_i$ consists only of unit clauses in $P_k$.
4. Let $\tilde{P} := \{A' \leftarrow \mid$ there exists a mgu $\sigma$, such that $A' = A\sigma$ and for each $B_i$ there exists a unit clause $\{C \leftarrow\} \in P_k$ with $B_i\sigma = C\sigma\}$. Note that $\tilde{P}$ is finite.
5. Let $P_{k+1} := (P_k \setminus \{A \leftarrow B_1, \ldots, B_n\}) \cup \tilde{P}; k := k+1$; goto 2.
6. Let $P_u := P_k$; stop.

Clearly the algorithm terminates as step 5 eliminates one non-unit clause per iteration. The application of steps 3-5 replace the non-unit clause by a finite set of semantically equivalent unit clauses. A formal proof of the correctness of these transformation steps is given in Appendix B. Once two hierarchical programs have been transformed according to the above algorithm, the set-inclusion property $\tilde{M}_{P_1} \subseteq \tilde{M}_{P_2}$ of the $\leq_{DT}$ relation can simply be verified via the subsumption principle on the transformed programs $P_{1u}$ and $P_{2u}$.

# 7 Conclusion and future work

In this paper we have motivated and defined a declarative extension to IDL-based type definitions. Within open distributed environments a type resembles much a standard, by which a client object is able to locate a suitable server object via some mediator. So far this standard has to be very specific in the sense that the precise syntactic structure and semantics of a type have to be known by all programmers in advance. The purpose for a declarative extension is to weaken the need for an a priori tight standard, which is of

[5]The algorithm presented here has been implemented in C++ using LEX and YACC and is available via anonymous ftp at *diamant.vsb.cs.uni-frankfurt.de:/pub/dt/horn_clause/dt_imp.tar.Z.*

particular benefit for open, distributed environments to reduce communication overhead between implementors of various objects.

It has been shown that a declarative program can describe the difference in behavior within members of a type family. The benefits from using Horn clauses lie in the nature of declarative semantics which describes *what* constitutes a particular behavior, not *how* to implement it. For this reason the declarative extension to an IDL-based type notation weakens the need for an a priori standard (of course only with respect to members of one type family).

The future work will go in two directions: first of all, the grouping of type families other than containers will be investigated with respect to practical issues. For the other direction we will pursue the theory of declarative types to replace the transformation algorithm for stronger subsets of Horn clauses, which are more expressive than hierarchical programs.

# Appendix A: Definitions

This appendix establishes well known definitions from the field of logic programming. A thorough introduction with more complete explanations may be found for example in [Lloyd87].

Def.: A *term* is defined inductively as follows:

- a variable is a term
- a constant is a term
- if $f$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term

Def.: If $p$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is called an *atomic formula* or, more simply, an *atom*.

Def.: A *substitution* $\theta$ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each $v_i$ is a variable, each $t_i$ is a term distinct from $v_i$ and the variables $v_1, \ldots, v_n$ are distinct. Each element $v_i/t_i$ is called a binding for $v_i$.

Def.: Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution and $E$ an expression. Then $E\theta$, the *instance* of $E$ by $\theta$, is the expression obtained from $E$ by simultaneously replacing each occurence of the variable $v_i$ in $E$ by the term $t_i$.

Def.: Let $S$ be a finite set of simple expressions. A substitution $\theta$ is called a *unifier* for $S$ if $S\theta$ is a singleton. A unifier for $S$ is called a *most general unifier* (mgu) for $S$ if, for each unifier $\sigma$ of $S$, there exists a substitution $\gamma$ such that $\sigma = \theta\gamma$.

Def.: A *definite program clause* is a formula of the form $\forall x_1 \ldots \forall x_s (A \leftarrow B_1 \wedge \ldots \wedge B_n)$ where $A, B_1, \ldots, B_n$ are atoms and $x_1, \ldots, x_s$ are all the variables occurring in the formula. Throughout this paper, a definite program clause will be denoted by $A \leftarrow B_1, \ldots, B_n$.

Def.: A *unit clause* is a definite program clause of the form $A \leftarrow$; i.e. clause with an empty body.

The informal semantics of $A \leftarrow B_1, \ldots, B_n$ is "for each assignment of each variable, if $B_1, \ldots, B_n$ are all true, then $A$ is true."

Def.: A *definite program* is a finite set of definite program clauses.

Def.: In a definite program, the set of all program clauses with the same predicate symbol $P$ in the head is called the *definition of* $P$.

Def.: A *level mapping* of a program is a mapping from its set of predicate symbols to the non-negative integers. We refer to the value of a predicate symbol under this mapping as the level of that predicate symbol.

Def.: A program is *hierarchical* if it has a level mapping such that, in every program statement $A \leftarrow B_1, \ldots, B_n$, the level of every predicate symbol $B_i$ is less than the level of $A$.

Def.: Let $L$ be a first order language. The *Herbrand universe* $U_L$ for $L$ is the set of all ground terms, which can be formed out of the constants and function symbols appearing in $L$.

Def.: Let $L$ be a first order language. The *Herbrand base* $B_L$ for $L$ is the set of all ground atoms, which can be formed by using predicate symbols from $L$ with ground terms from the Herbrand universe $U_L$ as arguments.

Def.: A *pre-interpretation* of a first order language $L$ consists of the following:

- a non-empty set $D$, called the domain of the interpretation,
- for each constant in $L$, the assignment of an element in $D$,
- for each n-ary function symbol in $L$, the assignment of a mapping from $D^n$ to $D$.

Def.: Let $L$ be a first order language. The *Herbrand interpretation* for $L$ based on a pre-interpretation is given by the following:

- the domain of the pre-interpretation is the Herbrand universe $U_L$,
- constants in $L$ are assigned themselves in $U_L$,
- if $f$ is an n-ary function symbol in $L$, then the mapping from $(U_L)^n$ into $U_L$ defined by $(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n)$ is assigned to $f$.

Def.: Let $P$ be a definite program and $\{M_i\}_{i \in I}$ be a non-empty set of all Herbrand models for $P$. Then $\bigcap_{i \in I} M_i$ is the *least Herbrand model* for $P$ denoted by $M_P$.

As with every Herbrand interpretation, the least Herbrand model is a subset of the Herbrand base ($M_P \subseteq B_L$). Although the programmer might have a different interpretation in mind, there is a strong reason to view $M_P$ as the natural interpretation of the program: the atoms in $M_P$ are precisely those that are logical consequences of the program.

Def.: Let $P$ be a definite program. The mapping $T_P : Pot(B_P) \rightarrow Pot(B_P)$ is defined as follows: Let $I$ be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P \mid A \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in $P$ and $\{B_1, \ldots, B_n\} \subseteq I\}$.

Def.: Let $L$ be a complete lattice and $T : L \rightarrow L$ be a mapping. We say $a \in L$ is the *least fixpoint* (lfp) of $T$, if $a$ is a fixpoint (that is, $T(a) = a$) and for all fixpoints $b$ of $T$, we have $a \leq b$.

Theorem: *(Fixpoint Characterisation of the least Herbrand model)* Let $P$ be a definite program. Then $M_P = lfp(T_P) = T_P \uparrow \omega$.

# Appendix B: Proof

Let $P$ be a hierarchical program, containing not only unit clauses. Let $P'$ be constructed from $P$ according to steps 3-5. Let this clause be $A \leftarrow A_1, \ldots, A_n$. Then $lfp(T_P) = lfp(T_{P'})$.

Proof: We have to show that $lfp(T_{P'}) \subseteq lfp(T_P)$ and $lfp(T_P) \subseteq lfp(T_{P'})$.

1.) $lfp(T_{P'}) \subseteq lfp(T_P)$

As $P'$ lacks the clause $A \leftarrow A_1, \ldots, A_n$ contained in $P$, but on the contrary possesses more unit clauses as $P$, it suffices to show $T_{P'} \uparrow 1 \subseteq T_P \uparrow \omega$. Let $B \in T_{P'} \uparrow 1$.

Case 1: For some mgu $\sigma$ and some unit clause $\{C \leftarrow\} \in P' \setminus \tilde{P}$ is $B = C\sigma \Longrightarrow \{C \leftarrow\} \in P \Longrightarrow B \in T_P \uparrow 1 \Longrightarrow B \in T_P \uparrow \omega$.

Case 2: For some mgu $\sigma$ and some unit clause $\{C \leftarrow\} \in \tilde{P}$ is $B = C\sigma$. Since $\{C \leftarrow\} \in \tilde{P}$, according to the construction of $\tilde{P}$ in step 4, there is a mgu $\tau$ with $C = A\tau$. Furthermore all atoms in the body of $A\tau \leftarrow A_1\tau, \ldots, A_n\tau$ are unifiable with some unit clauses in $P$ (according to step 4). All ground instances of these unit clauses are elements of $T_P \uparrow 1 \Longrightarrow$ all ground instances of $A\tau$ are elements of $T_P \uparrow 2$. Since $B = C\sigma$ and $C = A\tau \Longrightarrow B = (A\tau)\sigma \Longrightarrow B \in T_P \uparrow 2 \Longrightarrow B \in T_P \uparrow \omega$.

2.) $lfp(T_P) \subseteq lfp(T_{P'})$

Induction over $n$: $T_P \uparrow n \subseteq T_{P'} \uparrow \omega$

$n = 1$: All unit clauses in $P$ are also in $P' \Longrightarrow T_P \uparrow 1 \subseteq T_{P'} \uparrow 1 \subseteq T_{P'} \uparrow \omega$

$n \leadsto n+1$: I.H. $T_P \uparrow n \subseteq T_{P'} \uparrow \omega$
Let $B \in T_P \uparrow (n+1)$. We have to show that $B \in T_{P'} \uparrow \omega$.
$B \in T_P \uparrow (n+1) \Longrightarrow$ there is a ground instance of a clause in $P$, $B \leftarrow B_1, \ldots, B_n$ with $\{B_1, \ldots, B_n\} \subseteq T_P \uparrow n \Longrightarrow \{B_1, \ldots, B_n\} \subseteq T_{P'} \uparrow \omega$.

Case 1: $B \leftarrow B_1, \ldots, B_n$ is a ground instance of some clause in $P' \Longrightarrow B \in T_{P'} \uparrow \omega$.

Case 2: $B \leftarrow B_1, \ldots, B_n$ is a ground instance of $A \leftarrow A_1, \ldots, A_n$ for some mgu $\sigma$. Therefore there is no ground instance of a clause in $P'$. According to the construction rule in step 4, $B_1, \ldots, B_n$ are all ground instances of some unit clauses in $P$ (and therefore in $P'$). $B$ is therefore a ground instance of some unit clause in $\tilde{P}$ (and therefore in $P'$) $\Longrightarrow B \in T_{P'} \uparrow 1 \Longrightarrow B \in T_{P'} \uparrow \omega$.

q.e.d.

# References


[Amer91] P. America. Designing an object oriented programming language with behavioral subtyping. In *REX School/Workshop, LNCS 489*. Springer, 1991.

[Booch91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 1991.

[Card85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.

[ISO93] ISO/IEC. *ODP-Trader, Document Title ISO/IEC JTC 1/SC 21 N 8192*. 1993.

[Kons93] D. Konstantas. Object oriented interoperability. In *ECOOP'93, LNCS 707*. Springer, 1993.

[Larch93] G. T. Leavens and Y. Cheon. Extending CORBA-IDL to specify behavior with Larch. Technical report, ftp.cs.iastate.edu:pub/techreports/TR93-20/larch-corba.txt, 1993.

[Lisk93] B. Liskov and J. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP'93: Object–Oriented Programming*. Springer, 1993.

[Lloyd87] J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer, 1987.

[OMG91] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 1.1*. 1991.

[OSF92] Open Software Foundation. *Distributed Computing Environment, Version 1.0*. 1992.