Abstract

Type systems have been studied extensively for programs, but not for specifications. This paper presents a type system for the increasingly important class of specification languages based on first-order constraints over relational data models, which have applications in software modelling, architectural description, web ontologies, access control, etc.

The system has rather a different flavour from a traditional type system. From a user’s perspective, the type of an expression is something more than another relational expression approximating its value. Computationally, a type is a relation, and types are constructed from relational operators. Type errors indicate semantic redundancies, in which expressions could be replaced by constants without affecting the meaning of the specification as a whole. The system supports subtypes and union types without the need for downcasts, and a form of overloading that decouples semantics from typing. Its simplicity, it turns out, derives from the power and simplicity of relational join.

Introduction

In a thought-provoking article [14], Lamport and Paulson posed the question “should your specification language be typed?” but not “if so, what kind of type system should it have?” The answer to this second question depends, of course, on what kind of specification language you have in mind.

We address the question for relational constraint languages, an increasingly important class, in which specifications consist of first-order constraints over a relation data model. These languages include modelling notations, such as the Object Constraint Language of UML [16], the language of IBM’s Eclipse Modelling Framework [7], and our own Alloy language [2]; ontology languages such as DAML [5] and OWL [10]; access control directories to its children and a file to its data. An expression like d.contents where d is a directory should be unambiguous. In a programming language like Java, such an expression is a dereference and its disambiguation is trivial. But in a specification language, the syntax of relational expressions is richer, and a more general mechanism is needed. The approach we describe here is simple and effective, and has the advantage that it decouples typing from semantics: although typing does indeed resolve overloading, an unresolved expression has a well-defined meaning, no different from that of the resolved expression.

In the following sections, we (1) explain the rationale for type checking specifications as well as programs; (2) introduce the idea of redundancy as error; (3) outline our own language, Alloy, which is a good representative of the class of relational constraint languages; (4) describe the new type system in detail; (5) illustrate its application with an analysis of type soundness for a tiny subset of Java; and (6) summarize related work.

1 Types for Specification

The reasons for typechecking programs are well known and widely accepted. But do they apply to specifications? Types bring primarily three benefits to programs:

- **Modularity.** Programs written in untyped languages exhibit behaviours – eg, those due to out-of-bound array accesses – that cannot be predicted using the text of the program alone. Such behaviours violate modularity, in the sense that
dependences between modules can no longer be inferred from the sharing of names.

- **Simpler semantics.** In the absence of type checking, a semantics that purports to explain the behaviour of any program accepted by the compiler must handle awkward cases. It must explain, for example, the effect of adding a string to an integer, or of dereferencing a field that does not exist. This may involve extending the behaviours to include special kinds of failure that were not to occur would not need to be described at all. On the other hand, the independence of typing and semantics has merit: in a typed setting, determining how a program will behave may require knowing first how it types.

- **Early error detection.** Dynamic typing causes programs to fail earlier in their execution, making diagnosis easier and often preventing more damaging failures. Static typing allows errors of a certain class to be detected before running the program, and guarantees their absence from all possible executions.

- **Implementation.** Type information can be used to improve runtime performance of a program.

For specifications, the benefit of modularity does not apply. Since the very purpose of specifications is to describe behaviour, specification languages are fully explicit, with no hidden mechanisms.

Simplicity of semantics is often a concern; for example, predicate subtyping can ensure that functions are never applied outside their domains, so that notions of undefined expressions can be avoided. For a relational constraint language without function application, however, the familiar semantics of first-order logic, sets and relations can be applied to an untyped specification, fully decoupling types from semantics. As explained below (in Section 3.2), our semantics permits expressions of mixed or zero arity, which the type system rules out, but this is a minor complication.

Early error detection is the key reason for type checking specifications. Although the distinction between compile-time and runtime does not carry over exactly, in most specification approaches there are two distinct phases of analysis: a first phase in which fast and shallow analyses are applied, and a second phase in which the analysis proper is performed, more deeply and at greater cost. Type analyses for programs are sound, meaning that a passing program will execute without certain runtime errors. For specifications, there is no notion of runtime error, but we can still ask whether an analysis finds all errors in a given class.

Although specifications are not run like programs, types can be used to improve the performance of deeper analyses. In Alloy, for example, type information is used in the exploitation of symmetry [19]. The subtyping introduced here offers further opportunities for optimization, for which there is insufficient space to discuss.

Over the last decade, with the development of deeper automated analyses (such as model checking), and an increasing appreciation of partial models of systems, a shift in perspective has occurred. For analysis of both programs and specifications, the focus is now on error-detection rather than verification. With this shift comes an inversion: since the purpose of an analysis is now to find errors, a sound analysis is one that never reports spurious errors and a complete analysis is one that finds all errors in a given class. The type analysis we describe here is sound but incomplete, even for the limited class of errors it detects.

## 2 Errors, Anomalies and Redundancy

What constitutes a type error? If a program is checked against explicit annotations, type errors are simply failures to satisfy those annotations. An analogous notion is fruitful for specifications too, although we do not pursue it here. Instead, we assume a specification without explicit type annotations in which types must be inferred. In this case, type errors represent anomalies: failures to satisfy generic properties that are expected to hold universally. For a program, anomalies correspond to runtime failures: adding integers to strings, out-of-bound accesses, accessing deallocated memory, data races, etc.

For a specification, it is less clear what constitutes an anomaly. Inconsistency is always a problem; if a specification $S$ is inconsistent, logically denoting false, then a claim $S \Rightarrow P$ that property $P$ follows from it (and thus from any system satisfying the specification) is vacuously true; and the claim $M \Rightarrow S$ that the system $M$ satisfies it is vacuously false for all systems but an inconsistent one that does nothing.

Unfortunately, inconsistency is hard to detect, and is the subject of ongoing research [20,3]. But it suggests a more general notion of anomaly that includes some simpler cases. An inconsistent specification does not make sense because the details were not worth writing: the specifier could simply have written false instead. So we shall regard as anomalous any case in which an expression or formula is redundant, because it can be reduced to a constant without affecting the meaning of the specification in which it occurs.

## 3 An Overview of Alloy

Alloy is a relational constraint language designed for lightweight modelling of software systems [12]. It is amenable to a form of automatic analysis based on reduction to SAT, in which simulated executions and counterexamples to putative theorems are found by exhaustive search within user-defined bounds on the base types of the model [11]. It has been used in a variety of applications, and has been taught at 15 or so universities; details can be found online [2].

A merit of Alloy as a subject of study is that it hews closely to conventional logic; indeed, it can be viewed as a syntax for a first order logic with relational operators. It has no built-in notions of state machines or execution (and yet can be used to model dynamic behaviour without too much trouble). Features of a type system designed for Alloy are thus likely to be applicable to other languages based on first order logic.

### 3.1 Gross Structure

The syntax of MicroAlloy, a subset of the language that captures all its essential features, is shown in Figure 1. A MicroAlloy specification consists of a sequence of paragraphs:
- signatures, which introduce sets and relations;
- predicates, which give names to formulas that can be used elsewhere;
- facts, which are formulas that represent assumptions that always hold; and
- assertions, which are formulas intended to follow from the facts.

Here is a model inspired by Paul Simon’s 1973 song 'One man's ceiling is another man's floor':

3.1  sig Platform {}
3.2  sig Man (ceiling, floor: Platform)
3.3  pred Above (m, n: Man) {m.floor = n.ceiling}
3.4  fact (all m: Man | some n: Man | Above (n,m))
3.5  assert (all m: Man | some n: Man | Above (m,n))

The signature (3.1) introduces a set Platform. The signature (3.2) introduces a set Man, and with it two relations with that set as their first column, one called ceiling, and one called floor, both from Man to Platform. These relations are actually constrained to be total functions, so every man has exactly one ceiling and one floor. (This detail is explained in Appendix A.)

The predicate (3.3) associates the name Above, and two arguments m and n, with a formula that is true when m’s floor is n’s ceiling. The fact (3.4) states the premise of the song: that every man m has a man n above him. Finally the assertion claims a corollary: that every man m has a man n below him. In other words, the assertion claims that from ‘One man’s ceiling is another man’s floor’ one can conclude that ‘One man’s floor is another man’s ceiling.’

The assertion is invalid, and Alloy’s tool finds a counterexample such as this:

Man = {M0, M1}
Platform = {P0, P1}
floor = {(M0,P1),(M1,P0)}
ceiling = {(M0,P0),(M1,P1)}

Part of the problem is that man M0’s ceiling is his own floor; our formalization did not pay attention to the word another.

Here is an example (due to David Gries) of another song, with an unexpected consequence:

3.6  sig Person (loves: set Person)
3.7  assert {
3.8    all me, baby: Person |
3.9    (all p: Person | baby in p.loves) and baby.loves = me
3.10   => me = baby
3.11  }

The signature (3.6) introduces a set Person and a relation loves from Person to Person. Note the keyword set which indicates that the relation is not necessarily a function, but may map a Person to a set of Person’s of any finite cardinality. The assertion states that from the premise ‘Everybody loves my baby, but my baby loves only me’ (3.9) one can logically conclude ‘I am my baby’ (3.10).

3.11

specification ::= paragraph*
paragraph ::= sigDecl | predDecl | factDecl | assertDecl

sigDecl ::= [abstract] sig sigName [extends sigName] { decl,* }
predDecl ::= pred predName (argDecl,*) { formula* }
factDecl ::= fact (argDecl,*) { formula* }
avertDecl ::= assert { formula }

decl ::= var : [ set | option ] expr

formula ::= elemFormula | compFormula | quantFormula | letFormula | quantExpr
elemFormula ::= expr in expr | expr = expr
compFormula ::= not formula | formula logicop formula
logicop ::= and | or | =>
quantFormula ::= quantifier decl | formula
letFormula ::= let var = expr | formula
quantExpr ::= quantifier expr
quantifier ::= all | some | no

expr ::= sigName | var | unin | none | expr binop expr | unop expr
binop ::= - | + | & | . | ->
unop ::= ^ | ~
sigName ::= identifier
var ::= identifier

Figure 1: Syntax of MicroAlloy

3.2 Expressions and Formulas

Expressions and formulas take their standard form, but are given a slightly unconventional interpretation. Every expression denotes a relation. Thus a set is viewed as a unary relation, and a scalar is viewed as a singleton set. A quantifier thus binds its variable to relation values.

The keyword in denotes subset, so that e in s is true in general when the relation e is a subset of the relation s, and in particular when e is a unary, singleton relation and s is a unary relation, corresponding to the statement that the scalar e is a member of the set s.

The operators +,-,& are the standard set operators of union, difference and intersection, applied to relations regarded as sets of tuples. The operators ~ and ^ are the standard transpose and transitive closure operators on binary relations. The constant unin denotes the standard universal relation that maps every atom to every atom; none denotes the empty relation.

The remaining two operators, dot and arrow, are standard operators generalized to arbitrary arity. The dot operator is a generalized relational composition. Given expressions p and q, the expression p.q contains the tuple (p₁, …, pᵣ, qᵣ, …, qₙ) when p contains (p₁, …, pᵣ), q contains (qᵣ, …, qₙ), and pᵣ = q₁.

When p is a set and q is a binary relation, the composition p.q is the standard relational image of p under q (giving a simple semantics to ‘navigation expressions’). When p and q are both binary relations, p.q is standard relational composition.

The arrow operator is cross product: p ↠ q is the relation containing the tuple (p₁, …, pᵣ, qᵣ, …, qₙ) when p contains (p₁, …, pᵣ), and q contains (qᵣ, …, qₙ).

Two technicalities are worth noting. The first is that these definitions are intended to apply even if p and q do not have uniform
arieties. A relation can contain tuples of different lengths, including tuples of zero length. The type system will eliminate this leniency, and will only permit specifications in which all relational expressions have a uniform arity greater than zero.

The second is that the treatment of scalars as relations, and the first order nature of the language, allows us to dispense with function application. Rather than writing

\[ f(x) = y \]

to say that the function \( f \) maps the scalar \( x \) to the scalar \( y \), we write

\[ x.f = y \]

which will be true when \( x \) is in the domain of the relation \( f \) and is mapped to exactly one atom denoted by \( y \). Represented in this way, application of a partial function outside its domain results simply in the empty set. This sidesteps many of the semantic problems usually associated with partial functions, although sometimes the semantics is not what one expects. For example, the formula

\[ \text{no } p: \text{Person} \mid p.\text{wife in } p.\text{siblings} \]

not only rules out incest, but also requires that every person have a wife, because the body of the formula is vacuously true when \( p.\text{wife} \) is empty.

The quantifiers are all (universal), some (existential) and no (not exists). The last two can be applied to expressions. The formulas no \( e \) and some \( e \) are true when the expression \( e \) denotes an empty and non-empty relation respectively.

Let formulas can be regarded as syntactic sugar:

\[ \text{let } x = e \mid f \]

is equivalent to the formula \( f \) with its free occurrences of \( x \) replaced by the expression \( e \).

3.3 Relations, Extensions and Overloading

In MicroAlloy, the expressions that appear on the right-hand side of relation declarations in signatures may contain only the names of signatures, and not other relation names. Interpreting these declarations is therefore straightforward. The declaration

\[ \text{sig } S \{ \text{r: } E \} \]

introduces a set \( S \) and relation \( r \), and constrains \( r \) to be a subset of the relation given by the expression \( S \rightarrow E \). The expression \( E \) is a relational expression over the signature names of the model (including \( S \) itself). Multiplicity markings give a convenient shorthand for further constraining the relation to be a function, total, etc; these are explained in the appendix.

Signature extension defines one signature set to be a subset of another. All extensions of a signature are mutually disjoint. So if we write

\[ \text{sig } S \{ \} \]
\[ \text{sig } T \text{ extends } S \{ \} \]
\[ \text{sig } U \text{ extends } S \{ \} \]

then \( T \) and \( U \) are disjoint subsets of \( S \). Marking a signature as \textit{abstract} says that its extensions exhaust it.

The rule bounding relations applies equally to those declared in signatures that extend other signatures. Thus in this model

\[ \text{sig } S \{ \} \]
\[ \text{sig } T \text{ extends } S \{ \text{r: } E \} \]
\[ \text{sig } U \text{ extends } S \{ \} \]

the relation \( r \) is a subset of \( T \rightarrow E \). Given a scalar \( s \) in the set \( S \), the expression \( s.r \) will simply denote the empty set if \( s \) is not in the set \( T \). If this is always true — eg, because \( s \) is declared to have type \( U \) — the expression will actually be deemed to be ill-typed, although its meaning is perfectly well defined.

The same name can be used for two different relations, so long as they are declared in different signatures, and one does not extend the other, directly or indirectly. The meaning of an occurrence of a relation name in an expression is the union of all the relations of that name. For example, given the declarations

\[ \text{sig } S \{ \} \]
\[ \text{sig } T \text{ extends } S \{ \text{r: } E \} \]
\[ \text{sig } U \text{ extends } S \{ \text{r: } E \} \]

the expression \( x.r \) denotes \( x.(r_\text{T} \cup r_\text{U}) \) where \( r_\text{T} \) and \( r_\text{U} \) are the relations called \( r \) declared in the \( T \) and \( U \) respectively. As we shall see, if \( x \) can be determined to have the type \( T \), for example, the relation \( r \) in \( x.r \) will be resolved to \( r_\text{T} \). The meaning of the expression \( x.r \), given as a sum of overloads, will then be equivalent to the meaning of the resolved expression \( x.r_\text{T} \).

3.4 Formal Semantics

The meaning of an Alloy formula is its \textit{models}: the set of bindings of free variables to relation values for which the formula evaluates to true. The signatures and facts define a set of \textit{basic models} whose free variables are the sets and relations declared in the signatures, and for which the relevant formula is the conjunction of the formulas in all the facts, and the formulas implicit in the relation declarations and signature extensions.

The models of a predicate are then obtained by extending these models with the parameters of the predicate, and eliminating any bindings that do not also satisfy the formulas of the predicate. The models of an assertion are the subset of the basic models obtained by eliminating bindings that do not also satisfy the formulas of the assertion. An assertion is \textit{valid} if every basic model is also a model of the assertion; a \textit{counterexample} is a basic model that is not a model of the assertion.

Typically, the predicates of an Alloy specification represent properties of a system: invariants (over a single state), transitions (over a pair of states), or traces (over a sequence of states). Assertions record intended consequences of the design of the system, usually that a collection of properties characterizing the design implies some other (often more fundamental) properties, and their counterexamples are behaviours that show the design does not always satisfy the desired properties.

A partial semantics is given in Figure 2. It assumes that we have already derived a set of bindings from the signature declarations, and for any such binding, gives the meaning of each class of expression and formula in the standard denotational style. As usual, there are some puns: the operators appearing on the right-hand side of the equations (including dot and arrow, as defined above in Section 3.2) are operators in the meta theory, distinct
from the operators with the same name appearing on the left, which belong to the language being defined.

A small technicality: a binding must include an explicit set that represents the universal set of all atoms. For a binding \( b \), we write \( U(b) \) for that set, and \( b(x) \) for the value the binding assigns to variable \( x \). To extend a binding, we write \( b \odot \{x \mapsto v\} \) for the binding that is like \( b \) but which binds the value \( v \) to the variable \( x \).

Note that the semantics does not require relations to have uniform arity. An expression can denote a relation containing tuples of different lengths, including zero. Transitive closure may be applied to non-binary relations. Applied to a relation of arity \( 3 \), where the tuples are interpreted as labelled edges in a graph, the closure will yield the set of all paths in the graph, consisting of a starting node, a sequence of labels, and an ending node. The closure of a finite relation may thus be infinite. These anomalies will be excluded by the type system.

### Figure 2: Semantics of Expressions and Formulas

\[
\begin{align*}
M[\text{not} \quad f]b & = \neg M[f]b \\
M[f \quad \text{and} \quad g]b & = M[f]b \land M[g]b \\
M[f \quad \text{or} \quad g]b & = M[f]b \lor M[g]b \\
M[f \Rightarrow g]b & = M[f]b \Rightarrow M[g]b \\
M[\text{all} \ x: \ e \mid f]b & = \land (M[f](b \odot x \mapsto v) \mid v \subseteq E[e]b \land \#v=1) \\
M[\text{all} \ x: \ \text{set} \mid f]b & = \land (M[f](b \odot x \mapsto v) \mid v \subseteq E[e]b) \\
M[\text{some} \ x: \ e \mid f]b & = \lor (M[f](b \odot x \mapsto v) \mid v \subseteq E[e]b \land \#v=1) \\
M[\text{some} \ x: \ \text{set} \mid f]b & = \lor (M[f](b \odot x \mapsto v) \mid v \subseteq E[e]b) \\
M[p \quad \text{in} \quad q]b & = E[p]b \subseteq E[q]b \\
M[p = q]b & = E[p]b = E[q]b \\
M[\text{let} \quad v = e \mid f]b & = M[f](b \odot v \mapsto E[e]b) \\
M[\text{none}]b & = \emptyset \\
M[\text{univ}]b & = U(b) \\
M[\text{some} \ e]b & = E[e]b \cup \emptyset \\
M[\text{no} \ e]b & = E[e]b = \emptyset \\
E[p,q]b & = E[p]b \cdot E[q]b \\
E[p \rightarrow q]b & = E[p]b \rightarrow E[q]b \\
E[p \land q]b & = E[p]b \land E[q]b \\
E[p \land q]b & = E[p]b \land E[q]b \\
E[p \land q]b & = E[p]b \land E[q]b \\
E[p \land q]b & = E[p]b \land E[q]b \\
E[\land]b & = \mu X.(E[p](b \odot X \Rightarrow X \cdot E[p])) \\
\text{variables:} \ E[x]b & = b(x) \\
\text{signames:} \ E[s]b & = b(s) \\
\text{relations:} \ E[r]b & = \cup \{b(r) \mid r \text{ has name } r\}
\end{align*}
\]

### 4 The Type System

#### 4.1 Types As Relations

In a conventional type system, the language of type expressions is different from the language of value expressions. In our system, the type expressions are themselves relational expressions, forming the sublanguage of value expressions whose variables are restricted to signature names, and whose operators are restricted to union and product. By exploiting the algebraic properties of these operators, we can represent a type as a relational value. The type inference rules can thus make use of relational operators to construct types.

All types are manipulated in an atomized form over a set of base types that partition the universe. This will allow us to eliminate subtype comparisons from the type inference rules in favour of exact matching.

The base types include all leaf signatures - that is, those that are not extended by others - and, additionally, for each non-abstract compound signature, a ‘remainder’ base type of the same name. Note a subtle shift in interpretation. If signature \( S \) is a leaf, the base type \( S \) denotes the same set of elements as the signature \( S \) itself. But if \( S \) is a compound signature, the remainder type \( S \) denotes the elements that belong to \( S \) but not to its subtypes.

Consider, for example, the following (slightly modified) excerpt from the specification discussed below (in Section 5.4):

```
4.1 \text{sig} \ Variable \{\}
4.2 \text{sig} \ Value \{\}
4.3 \text{sig} \ Object \text{ extends} \ Value \{\ldots\}
4.4 \text{sig} \ Null \text{ extends} \ Value \{\}
4.5 \ldots
4.6 \text{sig} \ State \{\}
4.7 \ldots
4.8 \text{holds:} \ (\text{Slot} + \text{Variable}) \rightarrow \text{Value}
4.9 \}
```

The right-hand side of the declaration of \( \text{holds} \) (4.8) is both the type of this.\( \text{holds} \) (for a scalar this in the set \( \text{State} \)) and an expression that bounds its value. Recall that \( \text{holds} \) itself is a ternary relation of type

\[
\text{State} \rightarrow \text{Slot} + \text{Var} \rightarrow \text{Value}
\]

In atomized form, this becomes

\[
\text{State} \rightarrow (\text{Slot} + \text{Var}) \rightarrow (\text{Value} + \text{Object} + \text{Null})
\]

where the base type \( \text{Value} \) represents the set

\[
\text{Value} \rightarrow (\text{Object} + \text{Null})
\]

Since product distributes over union, we can rewrite any type as a sum of products. In this case, we obtain:

```
\text{State} \rightarrow \text{Slot} \rightarrow \text{Null}
+ \text{State} \rightarrow \text{Slot} \rightarrow \text{Object}
+ \text{State} \rightarrow \text{Slot} \rightarrow \text{Value}
+ \text{State} \rightarrow \text{Variable} \rightarrow \text{Null}
+ \text{State} \rightarrow \text{Variable} \rightarrow \text{Object}
+ \text{State} \rightarrow \text{Variable} \rightarrow \text{Value}
```

Now because product is associative, and union is associative and commutative, we can represent a type as a set of tuples of base types (as the layout of the above expression suggests). Types are
thus relations, and can be compared and combined by relational operators, in contrast to the traditional approach in which types are combined by syntactic operators.

4.2 Inference Rules

The inference rules (Figure 3) are interpreted over a set of valid types constructed as follows. Given a set Base of base types, the set of types of arity-\( n \) relations is

\[
\text{Type}_n = \{ \text{Type}_n \mid n \geq 1 \}
\]

and the set of all types is

\[
\text{Type} = \bigcup \{ \text{Type}_n \mid n \geq 1 \}
\]

There are no types corresponding to mixed-arity relations; this will effectively impose the constraint that all expressions denote conventional relations of a uniform arity. The omission of \( \text{Type}_0 \) rules out zero-arity relations, forbidding, for example, expressions such as \text{State}.\text{State} which are well-defined semantically, but not useful in practice.

The empty type, which we denote as \( \varnothing \), is included, however. It plays an important role in detecting vacuity; an expression whose type is \( \varnothing \) can be replaced by the constant \( \text{none} \) without affecting the meaning of the specification.

An inference rule is assumed to be applicable only if the computed type belongs to \( \text{Type} \), the set of valid types. This allows us to omit arity side-conditions from the rules for union (that arities match), join (that arities sum to 3 or more), and transitive closure (that the arity is 2).

Types are inferred for elementary formulas as well as expressions; this allows us to detect vacuity errors due to comparisons of expressions whose values are disjoint.

Judgments for the signature types themselves are assumed, with an axiom giving a type in atomized form for each signature. And for a relation \( r \) defined in a signature \( S \) with a range expression \( e \)

\[
\text{sig } S \{ r : e \}
\]

we assume an additional inference rule

\[
E \vdash S \rightarrow e : T \\
E \vdash r : T
\]

Together, these rules allow us to type the signature declarations, even though they are mutually recursive. The relation \( r \) may be overloaded, so we label the relation by its signature.

The rules given apply to formulas appearing in facts. The extension to predicates is obvious: their formulas are checked in an environment that binds, additionally, the arguments of the predicate.

4.3 The Type Checking Process

To resolve overloading, a separate type analysis of the entire specification is performed for each possible way in which all overloaded names may be resolved. A \textit{resolution} of the original specification is a new specification obtained by replacing every occurrence of a relation name with a labelled version of that name. Thus, a relation name \( \tau \) is replaced by \( r_{\tau} \), where \( S \) is one of the signatures that declares a relation named \( \tau \).

![Figure 3: Type Inference Rules](image)

Before type checking, we assume that a prepass has checked that there are no failures due to undeclared variables, relations or signatures. Every resolution is analyzed; we shall say a resolution \textit{passes} if all its formulas are well typed. The specification as a whole passes if all resolutions pass, and all but one contain an expression typed as empty.

A specification may fail for different reasons:

- If any resolution fails, there is an \textit{arity error}, such as an attempt to take the union of a set and a relation, or take the closure of a non-binary relation.
- If more than one resolution passes without inference of the empty type for any expression, there is an \textit{ambiguity error}; a relation name was resolvable to more than one relation.
- If no resolution passes without inference of the empty type for an expression, there is a \textit{vacuity error}; an expression is
equivalent to \texttt{none} however overloading are resolved, and can therefore be replaced by \texttt{none} without affecting the meaning of the specification.

### 4.4 Examples of Overloading

Given the following signature declarations

\[
\begin{align*}
\text{sig } & \text{A } {\text{t}} : \text{set } \text{A} \\
\text{sig } & \text{B } {\text{t}} : \text{set } \text{B} \\
\text{sig } & \text{C } {\text{r}} : \text{A } \rightarrow \text{B}
\end{align*}
\]

let’s consider type checking a variety of formulas:

- \textbf{all } \text{a : } \text{A } | \text{some } \text{a.r : } \text{The variable a has the type A, and } \text{r}_k \text{ is the only resolution of } \text{r} \text{ that gives a non-empty type to a.r. So the formula passes.}

- \textbf{all } \text{b : } \text{B } | \text{some } \text{r.a : } \text{The relation } \text{r}_k \text{ is the only resolution of } \text{r} \text{ whose right-hand type is B, so the formula passes.}

- \textbf{all } \text{b : } \text{B } | \text{some } \text{r.b : } \text{Both relations } \text{r}_k \text{ and } \text{r}_l \text{ have B as their right-hand type, and give non-empty types to r.b. So the formula is rejected due to ambiguity.}

- \textbf{all } \text{b : } \text{B } | \text{b in } \text{r.b : } \text{If } \text{r} \text{ is resolved to } \text{r}_k, \text{ the expression r.b} \text{ has mixed arity. So the formula is rejected due to mixed arity.}

- \textbf{all } \text{b : } \text{B } | \text{c : } \text{C } | \text{some } \text{c.(r.b) : } \text{The expression c.(r.b) has a non-empty type only when } \text{r} \text{ is resolved to } \text{r}_l \text{ so the formula passes.}

These examples illustrate a few key points:

- Although usually a relation is resolved in a join by its leftmost type, the scheme is more general: resolution can exploit any part of its type. The declaration of relations within signatures (and the rule that two relations within a signature cannot have the same name) merely ensures that no two relations have the same type.

- In the type system as described here, arity is not used to resolve overloading. In our implementation, however, it is; \textbf{all } \text{b : } \text{B } | \text{b in } \text{r.b} \text{ would pass.}

- A relation name need not be resolved in the smallest enclosing subexpression. In \textit{c.}(r.b), the variable \textit{c} resolves the overloading of \textit{r}.

### 4.5 Type Soundness

Our types are simple abstractions. Evaluated as a relational expression, an expression’s type must always contain its actual value. To formalize this, we first define the meaning of each base type in a given binding:

\[
\begin{align*}
\text{D : Base, Binding } & \rightarrow \text{ RelationValue} \\
\text{D[ } \text{t} \text{]b } & = \text{ b(t) } \cup \text{ (b(s) } | \text{ s } \subseteq \text{ t)}
\end{align*}
\]

where \( s \subseteq t \) if \( s \) extends \( t \) directly or indirectly. This captures the interpretation of base types explained in Section 4.1; the meaning of the base type would simply be \( b(t) \) were it not for base types that correspond to the remainders of non-abstract supertypes.

Now we interpret the meaning of a type as a relation:

\[
\text{B[T]b} = \cup \text{ D[ } \text{t}_1 \text{]b } \times \cdots \times \text{ D[ } \text{t}_n \text{]b } | \text{ (t}_1 \text{, …, t}_n \text{) } \in \text{T}
\]

The key to soundness is that the abstraction is an upper bound:

\textbf{Lemma: Approximation.} The type of an expression is an upper bound on its value: if \( \text{i- e : T} \text{ then, for every binding } \text{b}, \text{E[e]}\text{b} \subseteq \text{B[T]b}.\)

\textit{Proof.} By induction on the structure of the expression \( e. \)

We can now state the soundness criteria, and give informal arguments to support them.

\textbf{Theorem 1: Vacuity of expressions.} An expression for which an empty type is inferred can be replaced by the constant \texttt{none} without a change in meaning.

\textit{Proof (informal).} By the approximation lemma, If the empty type is inferred for \( e \), its value is empty in any binding, so it can be replaced by \texttt{none}.

\textbf{Theorem 2: Vacuity of containment.} If the empty type is inferred for \( p \text{ in } q \), \( q \) can be replaced by the constant \texttt{none} without a change in meaning.

\textit{Proof (informal).} If the types of \( p \) and \( q \) do not intersect, then since they bound the values of \( p \) and \( q \) in any binding, there is no non-empty value that \( p \) can take that is a subset of any non-empty value \( q \) can take. The meaning of the formula therefore depends only on whether \( p \) is equal to \texttt{none}. If so, the formula is true; otherwise it is false.

Note that the same theorem does not apply to the formula \( p = q \). In that case, the formula is true iff \( p \) and \( q \) are both empty.

The treatment of overloading has a soundness criterion too. We claim that although types are used to identify failures to resolve overloading, they are not needed for semantics. If overloading are unresolved, the specification’s meaning is perfectly well defined (although in practice we reject it). If all overloading are resolved, the meaning is the same, whether we leave the overloaded relation name unresolved, or replace it by a name that identifies the appropriate relation without ambiguity.

\textbf{Theorem 3: Resolution of overloading.} A specification that type checks has the same meaning as the one resolution that type checks with no inference of empty types.

\textit{Proof (informal).} Consider an occurrence of a relation name \( r_i \) with overloading \( r_i \) for \( i \in 1..n \). The meaning of \( r \) is given by the union of the \( r_i \) (Section 3.3). Assume without loss of generality that the passing resolution replaces \( r_i \) by \( r_i \). If all other resolutions identify vacuities, then, by Theorem 1, each \( r_i \) for \( i \neq 1 \) can be replaced by \texttt{none}. The expression \( r \) is thus equivalent to the expression \( r_i \).

### 4.6 Extensions

More vacuities could be identified. Consider the expression

\((a + b) \& c\)
where $a$ and $c$ both have the type $A$, and $b$ has type $B$, where $A$ and $B$ are disjoint. Although the expression $b$ is not redundant in the context of the subexpression $a + b$, it is redundant in the larger context of the entire expression. Had we written the expression instead as

$$(a \& c) + (b \& c)$$

the type system we have described would have derived the empty type for $b \& c$ and reported a vacuity error. It is possible to extend the type system to reject ‘inessential unions’ like $a + b$.

In the scheme we have described, overloading is resolved by vacuity and not by arity. It is straightforward to allow arities to resolve overloading (as done by our implementation).

### 4.7 Implementation

Like all type systems, ours is to be viewed as a specification and not an implementation of a type checking algorithm. Interpreted literally, it suffers from an obvious problem: the need to type check all resolutions.

Our implementation works in a single pass, bottom up. The type of a relation is represented with a structure that distinguishes its individual overloading. On reaching a top-level expression (or a transitive closure operator), it attempts to resolve all overloading it has seen so far. For each relation name for which no resolution is found, the algorithm lists the multiple possible overloading. Arity and vacuity errors that do not arise from overloading are reported when encountered.

The implementation fails to meet the specification in two respects. First, it allows arity to be used to resolve overloading. Second, vacuity errors are reported for equality formulas, since although not technically vacuous, such formulas are very likely to be wrong, and can be easily rewritten in a more perspicuous form: as $\text{no } p \text{ and no } q$ for example, rather than $p = q$.

## 5 An Example: Type Soundness in Java

To give a sense of the practicalities of using this type system, we illustrate it with a longer and fuller example. We will model the operational semantics of elementary statements in Java, and show that the types assigned to variables and fields of objects by the Java type system bound the values they may take at runtime.

The example uses a few features of the full Alloy language that are not included in MicroAlloy, but which are explained in context.

### 5.1 Java Type Hierarchy

We start with some signatures that describe the elements of the type hierarchy:

```alloy
5.1 abstract sig Type {
5.2   sig Interface extends Type {
5.3     xtends: set Interface
5.4   }
5.5   sig Class extends Type {
5.6     xtends: option Class,
5.7     implements: set Interface,
5.8     fields: set Field
5.9   }
```

The abstract marking on the signature `Type` indicates that the signatures that extend `Type` subsume it; thus every type is an interface or a class. An interface is declared to extend zero or more interfaces; a class extends at most one other class (hence the keyword `option`). An class implements some set of interfaces, and has a set of fields, each of which has a declared type.

Note that both `Interface` and `Class` have a relation called `xtends` (not extends, to distinguish it from the Alloy keyword), but that these two relations represent different albeit similar notions.

Let’s constrain the type hierarchy further, by requiring that no class or interface directly or indirectly extends itself. To say this, we declare a predicate that determines whether a relation is acyclic, and apply it to the two relations:

```alloy
5.13 pred Acyclic (: univ -> univ) {no x: univ | x in x.^r}
5.14 fact {
5.15     Acyclic (Interface <-: xtends)
5.16     Acyclic (Class <-: xtends)
5.17 }
```

The operator $<$: is domain restriction: $s<$ is the subset of $t$ whose tuples have first elements in the set $s$. This operator is convenient for disambiguating overloaded relations. It’s important to realize that there is no special casts or other type-related syntax here; restriction has a simple meaning for the untyped language.

The function applications type because `univ` intersects with every type.

### 5.2 Java Values

Values are either objects or null:

```alloy
5.18 abstract sig Value {}
5.19 sig Null extends Value {}
5.20 sig Object extends Value {
5.21     type: Class,
5.22     slot: Field ?->? Slot
5.23 } {slot.Slot = type.fields}
5.24 sig Slot {}
```

An object has a type corresponding to its instantiated class, and a slot for holding a value associated with each of its fields. The multiplicity markings (explained in the appendix) in the declaration of slot make it an injection, so that a field has at most one slot, and a slot belongs to at most one field.

The signature fact (5.23) records constraints about individual members of the signature more succinctly than if it were written explicitly:

```alloy
5.25 fact {all this: Object | this.slot.Slot = this.type.fields}
```

The shorthand is similar to the use of self references in an object-oriented programming language. Note how each field of the signature (or in fact any signature that it intersects with) is treated as a dereference. The fact says that fields of an object that have slots are those fields that are prescribed by the object’s type.
5.3 Java Statement and Expression Syntax

We'll consider only two simple forms of program statement, assignments and field setters:

\[
\begin{align*}
5.25 & \textbf{abstract sig} \text{ Statement } \\
5.26 & \text{ sig Assignment \text{ extends} Statement } \\
5.27 & \text{ var: Variable,} \\
5.28 & \text{ expr: Expr} \\
5.29 & \} \\
5.30 & \text{ sig Setter \text{ extends} Statement } \\
5.31 & \text{ field: Field,} \\
5.32 & \text{ expr, reexpr: Expr} \\
5.33 & \}
\end{align*}
\]

For expressions, we'll include variables, constructor calls and field getters:

\[
\begin{align*}
5.34 & \textbf{abstract sig} \text{ Expr } \\
5.35 & \text{ type: Type,} \\
5.36 & \text{ subexprs: set Expr} \\
5.37 & \} \{ \text{subexprs = this + this.'@expr} \\
5.38 & \text{ sig Variable \text{ extends} Expr } \\
5.39 & \text{ declType: Type} \\
5.40 & \} \{ \text{type = declType} \\
5.41 & \text{ sig Constructor \text{ extends} Expr } \\
5.42 & \text{ class: Class} \\
5.43 & \} \\
5.44 & \text{ sig Getter \text{ extends} Expr } \\
5.45 & \text{ field: Field,} \\
5.46 & \text{ expr: Expr} \\
5.47 & \} \{ \text{type = field.declType} \\
\end{align*}
\]

The fact for signature Expr (5.37) defines the subexpressions of an expression (which will be handy later). The keyword this is bound to the instance of the signature that is implicitly universally quantified (as shown in the previous section). The @ sign preempts implicit dereferencing; had we written ^expr instead, it would be expanded to ^*(this.expr) which is not what we meant, and would be caught by the type checker. Note how the fact in Expr uses a relation from Getter, a signature that extends it, without need for a downcast.

The other signature facts define the types of expressions. The type of a variable is just its declared type (5.40). The type of a getter (5.47) is the declared type of its field. Note that the relations type and declType are both overloaded, being used also in the signatures Class and Field respectively, but are easily resolved.

5.4 Java Execution State

Our final signature represents the states of the executing program:

\[
\begin{align*}
5.48 & \textbf{sig} \text{ State } \\
5.49 & \text{ objects: set Object,} \\
5.50 & \text{ reaches: Object -> Object,} \\
5.51 & \text{ vars: set Variable,} \\
5.52 & \text{ holds: (Slot + Var) -> Value,} \\
5.53 & \text{ val: Expr -> Value} \\
5.54 & \} \\
5.55 & \text{ all o: Object | o.reaches = holds[o.slot[Field]] & Object} \\
5.56 & \text{ holds.Value & Variable = vars} \\
5.57 & \text{ objects = holds(vars).\text{preorder}} \\
5.58 & \text{ all e: Expr | let v = val[e] } \\
5.59 & \text{ e in Variable => v = holds[e]}
\end{align*}
\]

Associated with a state we have the set of objects reachable in the heap (5.49), a reachability relation between objects (5.50), and a set of variables in scope (5.51). The relation holds gives the current value of slots and variables (5.52); in terms of it, we define val to give values to expressions.

The first three constraints say that an object reaches all the objects held in its fields' slots (5.55); that every variable in scope holds a value (5.56); and that the objects in the heap are those reachable from the variables (5.57). The last constraint defines the values of expressions, for example that the value of a variable is the value it holds (5.59). The value of a constructor expression is constrained only to be some object of the appropriate type (5.61). This is a simplification, but freshness of new objects does not affect type soundness.

Note the use of the union in the declaration of holds. Without unions, we would have to split the relation into two separate relations, or create a spurious supertype of slots and variables.

5.5 Java Type Checking

The purpose of static type checking in Java is to rule out runtime type errors. If we can establish that the runtime value of an expression always belongs to the expression's declared type, it is a small step to ensuring that called methods always exist. To capture this notion, we write a predicate:

\[
\begin{align*}
5.63 & \textbf{pred} \text{ RuntimeTypesOK } (s: \text{ State}) \{ \\
5.64 & \text{ all o: s.objects, f: o.type.\text{fields} |} \\
5.65 & \text{ let v = s.holds [o.slot [f]] | HasType (v, f.\text{declType})} \\
5.66 & \text{ all var: s.vars |} \\
5.67 & \text{ let v = s.holds [v] | HasType (v, var.\text{declType})} \\
5.68 & \}
\end{align*}
\]

A state is acceptable if every field of every reachable object (5.65) and every variable in scope (5.67) holds a value of its declared type.

The auxiliary predicate HasTypesays that a value v has a type t if v is null, or if v's declared type is a subtype of t:

\[
\begin{align*}
5.69 & \textbf{pred} \text{ HasType } (v: \text{ Value}, t: \text{ Type}) \{ \\
5.70 & \text{ v in Null or Subtype (v.type, t)} \\
5.71 & \}
\end{align*}
\]

Note the 'downcasing' of t to a class in the first line (5.74) and to an interface in the third line (5.76). Without these casts – 'disambiguations' more accurately – the subsequent occurrences of xtends would be unresolved overloading. From a programming language perspective, it may seem strange that the type checker doesn't 'know' that t has type Class on the right hand side of the first implication (5.74). It would be a mistake, however, to treat the logical operators any differently in the type system from in the semantics. Type checking should be unaffected by, for ex-
ample, transforming \( F \Rightarrow G \) to not \( F \) or \( G \). It would be easy to introduce a special form

\[
\text{case } x : T \mid F
\]
as a shorthand for

\[
x \in T \Rightarrow \text{let } x' = x \land T \mid F
\]

Type soundness can be checked independently for each kind of statement. We’ll use the field setter statement as an illustration:

\[
\begin{align*}
5.78 & \text{pred TypeChecksSetter (stmt: Setter)} \{ \\
5.79 & \text{all } g: \text{Getter } & \text{& stmt.(lexpr+rexpr).subexprs } | \\
5.80 & \text{g.field in } g.\text{expr.type.fields} \\
5.81 & \text{stmt.field in } stmt.\text{lexpr.type.fields} \\
5.82 & \text{Subtype (stmt.rexpr.type, stmt.field.declType)} \\
5.83 & \}
\end{align*}
\]

A setter statement type checks if every getter expression that appears in it is well-typed (5.80); the type of the left-hand expression is a class admitting the given field (5.81); and the type of the right-hand side is a subtype of the type of the field (5.82).

5.6 Java Execution

Execution of a setter statement is described by a predicate whose arguments include a pre-state \( s \) and a post-state \( s' \):

\[
\begin{align*}
5.84 & \text{pred ExecuteSetter (s, s': State, stmt: Setter)} \{ \\
5.85 & \text{stmt.(rexpr+lexpr).subexprs } & \text{& Variable in } s.\text{vars} \\
5.86 & s'.\text{objects } = s.\text{objects } & \text{and } s'.\text{vars } = s.\text{vars} \\
5.87 & \text{let } \text{rval } = s.\text{val[stmt.rexpr]}, \text{lval } = s.\text{val[stmt.lexpr]} \{ \\
5.88 & \text{no lval } & \text{& Null} \\
5.89 & \text{s'.holds } = s.\text{holds } \Rightarrow (\text{lval.slot[stmt.\text{field}]} \Rightarrow \text{rval}) \\
5.90 & \} \\
5.91 & \}
\end{align*}
\]

The first constraint (5.85) is a precondition; it says that the variables appearing in the statement must be in scope. The second constraint (5.86) is a frame condition: executing the statement does no allocation or deallocation of objects or variables. The third constraint includes a precondition, that the value of the left-hand expression cannot be null (5.88), and a postcondition, that the slot associated with the field acquires the value of the right-hand expression (5.89).

The operator ++ is relational override: \( p++q \) includes the tuples of \( q \) and those tuples in \( p \) whose first elements are not the first elements of any tuples in \( q \).

5.7 Java Soundness Conjecture

Putting everything together, we now assert the key inductive claim: that executing a well-typed setter statement in an acceptable state results in another acceptable state:

\[
\begin{align*}
5.92 & \text{assert TypeSoundness } \{ \\
5.93 & \text{all } s, s': \text{State, stmt: Statement } | \\
5.94 & \text{(RuntimeTypesOK (s))} \\
5.95 & \text{stmt in Setter} \\
5.96 & \text{ExecuteSetter (s, s', stmt)} \\
5.97 & \text{TypeChecksSetter (stmt)} \\
5.98 & \} \Rightarrow \text{RuntimeTypesOK (s')} \\
5.99 & \}
\end{align*}
\]

Alloy provides a syntax for commands to instruct the tool to check assertions. The command

\[
5.100 \text{check TypeSoundness for 10 but 1 Statement, 2 State}
\]

instructs it to consider all basic models involving at most 10 types, 10 values, and 10 expressions (but at most 1 statement and 2 states, no more being necessary).

Running the tool exposed a slew of errors in earlier drafts of this specification. Aside from type errors, the tool found some blatant omissions, such as not prohibiting execution of a setter when the left-hand expression evaluates to null, and including only field slots but not variables in characterizing acceptable states. Analysis also exposed some more subtle technicalities, for example that slots must not be shared between fields of different objects (which, if violated, results in a strange kind of aliasing that cannot occur in Java)

\[
\text{fact } \{ \text{all } a, a': \text{Object } | \text{some } a.\text{slot[Field]} \text{& } a'.\text{slot[Field]} \Rightarrow a = a' \}
\]

and that getter expressions cannot contain themselves.

\[
\text{fact } \{ \text{all } g: \text{Getter } | \text{no } g \text{& } g.^{\text{subexprs}} \}
\]

On a 1GHz PowerBook G4, using the Berkmin SAT solver as the Alloy tool's backend, most runs of the command took only a couple of seconds. Running the command on the corrected specification, which finds no counterexamples, takes 72 seconds.

6 Related Work

6.1 Lamport and Paulson

For Lamport and Paulson [14], a ‘specification language’ is a formalism for theorem proving. Most of their concerns arise from the application of partial functions outside their domain, and from higher-order constructs, and therefore do not apply to first-order relational languages.

To type set unions, they advocate the use of disjoint sum types. In relational notations, disjoint sums are not convenient, since the type constructors do not meld well with relational operators. This is why ‘free types’ are rarely used in Z. In our approach, the type of a union of values is simply a union of types.

In the conclusion of their abstract, having weighed the merits of typed and untyped languages, Lamport and Paulson suggest that ‘It may be possible to have the best of both worlds by adding typing annotations to an untyped specification language’. This, we believe, is what our type system achieves, albeit in a first-order setting that is simpler than the higher-order setting they had in mind.

6.2 Alloy 2.0

The type system described here grew from dissatisfaction with the previous version of Alloy [12]. It had no subtypes: a signature \( S \) that extended a signature \( S' \) was regarded as a subset of \( S' \) with the same type as \( S' \). Basic types partitioning the universe were associated with the non-extending signatures. Overloading was resolved by an ad hoc scheme that favoured dot over other operators, and the left side of a relation over the right.

Our new approach improves on this one in several respects:
- It is conceptually simpler. The notion of type is more uniform; there is less coupling between the type system and the semantics; and a number of awkward syntactic elements have been eliminated.
- It is more flexible. The new version offers arbitrary unions, and allows relation names to be overloaded between subtypes of a common supertype.
- It offers prospects for an improvement in the performance of our SAT-based analysis [11], in which fewer boolean variables are allocated to relations because, in a subtype system, their types constrain their domain and range.

One casualty of the new type system has been that the previous version instantiated the type parameters of polymorphic predicates and signatures automatically. Combining subtype and parametric polymorphism, not surprisingly, proved tricky, so we decided to view parametrically polymorphic modules as templates that are instantiated prior to subtype checking.

6.3 OCL

OCL [23] is the constraint language of UML, the Unified Modeling Language, a collection of notations for software modelling that has been standardized by the Object Management Group (OMG). The most recent version of OCL [16] is described in a response to the OMG’s request for proposals for UML version 2.0, which was approved by the OMG in March 2003.

Like Alloy, OCL is based on a first-order relational data model, and organizes sets in a subtype hierarchy. Because of multiple inheritance, subtypes do not have unique least upper bounds, and thus do not form a lattice. For this reason, and because set operators are provided by a library module that is parameterized on a single type, a union such as Person + Man is not legal without a typecast on at least one (and sometimes both) subexpressions.

Overloading in OCL is resolved only by the left type of a relation. There are no relational operators (such as join, closure or transpose). The dot operator is interpreted as function application rather than relational image, so, following the type systems of programming languages, OCL does not permit a relation \( r \) to appear in the expression \( x.r \) unless the type of \( x \) is a subtype of the left type of \( r \). Downcasts are provided to alleviate this, but their role is uncertain. They do not prevent out of domain function applications, since it is still possible for \( r \) to be partial, and for the scalar \( x \) to be outside its domain, in which case the result is a special undefined value. When \( r \) is an arbitrary relation rather than a function, the expression \( x.r \) denotes a set, which is not easily handled, since downcasts apply only to scalars.

These problems have been noted by others [eg, 21], but they appear to have no clean solution that does not involve extensive changes to the language semantics.

6.4 Z

The formal specification language Z [22, 24] has a simple type system without subtypes. The fundamental unit of specification, the schema, can be used to elaborate the description of the state of a system or the structure of objects incrementally. But schema types admit only exact matching, so it is not possible to write an operation on an arbitrary file system object, for example, and apply it to a directory.

Object-Z [6] is an extension of Z that adds notions of classes and inheritance. As in Alloy 2.0, however, there is no true subtyping. Dereferencing an object as if it belonged to the subclass is permitted without a downcast, but may result in an undefined expression.

6.5 Other Languages

Ontology languages, despite their emphasis on classification, tend to be untyped or to have simple flat type systems. The OWL language [10] is the current focus of efforts to develop a standard web ontology language, and is based on earlier languages such as DAML [5]. OWL is untyped, but could benefit from a type system to find a class of errors without resorting to more elaborate analyses.

RCL 2000 [1] is a specification language for role-based authorization constraints. It is equivalent to a restricted form of first-order logic. Although the language is untyped, it should not be hard to apply Alloy-style typing to it, since authorization constraints are readily translated into Alloy [18].

Architectural description languages, such as Darwin [15], ACME [8] and xADL [13], allow the description not only of particular systems, but of architectural ‘styles’, usually characterized by first-order constraints on the topology of the system. Alloy has been used to express architectural constraints; a recent paper [9], for example, translates Darwin to Alloy to enable analysis. These languages tend either to be untyped or to have simple type systems without subtypes or unions.

In an early attempt to add types to LISP, Cartwright [4] advocated a type system similar to ours in two ways. Types were unions of disjoint base types, and function application required only that the type of the formal and actual intersect. This eliminated the burden of explicit casts, but required other means (such as theorem proving or runtime checks) to ensure lack of runtime errors.

Predicate subtypes [17] are used in the PVS theorem prover largely to ensure that partial functions are not applied outside their domain. Subtypes are characterized by arbitrary formulas. This gives great expressive power, of course, but a severe loss in tractability, and a blurring of the phase distinction between type checking and deeper analysis.

Conclusions

We have presented a simple type system for specification. It offers unions and subtypes without the usual complications that they bring in the context of programs. The key idea is to exploit the use of relational join rather than function application.

Function application is essential to programming. In a specification, by encoding scalars as singleton sets, we can use relational join to the same effect (and for other purposes too). Since applying a function outside its domain is always an error, a type system for functions must perform a subsumption check for each application. A relational join, however, may legitimately yield an empty set. But if does so in every case, a specification
error is likely. In a type system for join, therefore, an intersection check is more natural than a subsumption check.

For relational constraint languages, the system we have described gives more flexibility than existing type systems and catches errors missed by a system lacking subtypes. The subtype information should also make more efficient SAT-based analysis possible, a research direction we are currently pursuing.

Acknowledgments

Manu Sridharan designed and implemented a 'subscoping' scheme for Alloy that included some of the subtyping structure described in this paper. Mandana Vaziri suggested the key idea that an overloaded relation should be viewed semantically as the union of its resolutions. The paper benefited from comments on an earlier draft from Derek Rayside and Robert Seater.

This work was supported by grant 0086154 from the ITR program of the National Science Foundation, and by a grant from the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References


Appendix: Multiplicities

Alloy uses some simple shorthands to constrain the multiplicity properties of relations. A binary relation \( r \) declared as

\[
\text{sig } S \{ r: T \}
\]

\[
\text{sig } T \{ \}
\]

is a total function. Declared, however, like this

\[
\text{sig } S \{ r: \text{option } T \}
\]

\[
\text{sig } T \{ \}
\]

it becomes a partial function, and like this

\[
\text{sig } S \{ r: \text{set } T \}
\]

\[
\text{sig } T \{ \}
\]

an arbitrary relation. The default is thus to make the relation a total function; the keywords \text{set} and \text{option} eliminate implicit constraints. For relations of higher arity, special multiplicity symbols are used. The declaration of a ternary relation \( q \)

\[
\text{sig } S \{ q: T \rightarrow U \}
\]

\[
\text{sig } T \{ \}
\]

\[
\text{sig } U \{ \}
\]

for example, ascribes no particular multiplicity properties to it. If we write

\[
\text{sig } S \{ q: T \rightarrow? U \}
\]

\[
\text{sig } T \{ \}
\]

\[
\text{sig } U \{ \}
\]

however, the binary relation \( s.q \) is constrained to be a partial function for all \( s \) in \( S \); like this

\[
\text{sig } S \{ q: T \rightarrow? U \}
\]

\[
\text{sig } T \{ \}
\]

\[
\text{sig } U \{ \}
\]

it becomes an injection too. Put informally, if the expression in the declaration of \( q \) is

\[ n \ T \rightarrow m \ U \]

the relation \( s.q \) maps each \( T \) to \( m \) elements of \( U \), and \( n \) elements of \( T \) to each \( U \), where \( m \) and \( n \) are \(?\), \(!\), \(+\), meaning respectively zero or one, exactly one, and one or more. This notation is a bit ad hoc, but it is simple and convenient to use in practice.