Subtyping in Alloy

by

Emina Torlak

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 20, 2004

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by . . . .
Daniel N. Jackson
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Subtyping in Alloy

by

Emina Torlak

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Computer Science and Engineering

ABSTRACT

A type system for the Alloy modelling language is described that supports subtypes and allows overloading of relation names. No special syntactic features needed to be added to the language to support the type system; there are no casts, and the meaning of a model can be understood without reference to types. Type errors are associated with expressions that are irrelevant, in the sense that they can be replaced by an empty relation without affecting the value of their enclosing formula. Relevance is computed with an abstract interpretation that is relatively insensitive to standard algebraic manipulations. The typechecker for the system is presented in the context of Alloy Analyzer 3.0. Its architecture is explained in terms of key data abstractions, algorithms, and complexity analyses.

Acknowledgements

This thesis is a product of almost two years of research. Its completion would not have been possible without the guidance and support I have received from my colleagues, friends, and family.

I would like to thank Jonathan Edwards who came up with the idea of adding subtypes to Alloy and who developed much of the theoretical underpinnings of the Alloy 3.0 type system. Jonathan's energetic personality, sharp intellect, and great sense of humor made our collaboration a wonderful academic and social experience.

I would also like to acknowledge all the help I have received from Greg Dennis. Greg's work has been instrumental in discovering many of the bugs in the Alloy 3.0 typechecker. Our whiteboard sessions about the Analyzer architecture have been an invaluable venue for shaping new ideas and revealing hidden problems.

Special thanks go to my mother, Edina, and my sister, Alma. Their boundless love, support and caring gave me the courage to follow my dreams. Thanks also to my friends Rishi Kumar, Stephanie Wang, Yogishwar Maharaj, Mandana Vaziri, and Philip Kong.

Finally, I would like to thank my thesis advisor, Prof. Daniel Jackson, for his insights and guidance on both my academic and research endeavors. His active participation in the Alloy 3.0 project has yielded several key ideas and many interesting discussions. Most importantly, his encouragement, enthusiasm, and understanding have helped me rediscover why I came to MIT in the first place.

Contents

List of Figures

1 Introduction

Most researchers agree that judicious use of formal methods is the key to producing high quality software [1]. Unfortunately, formal methods have yet to gain popularity in the software industry. Most practitioners believe that, for all but safety-critical systems, the resources needed to formalize a product's specification are better spent in the later phases of the development cycle [22]. As a result, a lot of research has been directed toward the development of *lightweight formal methods* that provide the benefits of formalism at a small price.

The Alloy language and its Analyzer are the products of the MIT Software Design Group's efforts to develop a versatile lightweight modelling kit. The project started in 1998, and the kit has since grown both in sophistication and popularity. Researchers at MIT and other universities have used the language and the Analyzer to investigate a variety of problems ranging from brainteasers to the UML core metamodel [10].

Since its first release, the Alloy modelling kit has matured through continuous revision and improvements. This document describes the typing system of the latest version of the language, Alloy 3.0, and its implementation by the Analyzer. In the remainder of this chapter, we explain the rationale for typechecking specifications and discuss the limitations of Alloy 2.0's type system. Chapters 2 and 3 provide an overview of Alloy 3.0 and a formal description of its typing rules. The Alloy Analyzer 3.0 and the typechecker are discussed in Chapters 4. Finally, Chapter 1 gives an overview of related work.

1.1 Types for Specifications

In their article on typing of specification languages [15], Lamport and Paulson conclude that types are beneficial to specification languages with computerized tool support. Such specification languages derive similar benefits from having a type system as do statically typed programming languages. In particular, types can help simplify language semantics, provide early detection of a certain class of errors, and improve runtime performance.

The semantics of an untyped programming language must give meaning to awkward constructions such as the addition of a string and an integer or dereferencing of a non-existent field. A typed programming language eliminates the need to provide meaning for these special kinds of failures by rejecting them during the typechecking phase. Types are used in a similar manner to simplify the semantics of specification languages. For example, predicate subtypes of PVS [18] ensure that functions are never applied outside their domains thereby avoiding the notion of undefined expressions.

Early error detection is the main reason for type checking programs and specifications. Although the distinction between compile-time and runtime is not applicable to specification languages, most tools for checking specifications perform two distinct phases of analysis: (1) fast, shallow analysis which eliminates blatant errors, and (2) deep, expensive analysis which detects subtler bugs. The shallow analysis phase is analogous to a program's compile-time whereas the deep analysis roughly corresponds to its runtime. In the case of a program, it is desirable to catch as many errors as possible at compile-time thus avoiding potentially costly (and certainly frustrating) runtime failures. The same is true for a specification: the time and resource consuming deep analysis is better saved for truly insidious bugs.

Although a specification is not run like a program, type information can be used to reduce the cost of its deep analysis. In Alloy 2.0, for example, type information is used in the exploitation of symmetry [19]. The subtyping introduced in Alloy 3.0 offers further opportunities for optimization, which are discussed in [7].

1.2 Limitations of Alloy 2.0 Type System

Alloy is a powerful modelling notation that has been successfully used to gain insight into a wide range of problems. However, the type system of Alloy 2.0 limits the usefulness of the Analyzer and the expressiveness of the language in many ways. These limitations are discussed below in the context of the following snippet of an email client's model:

```
1  sig Id {}                                            // identifier
2  sig String {}                                        // string
3  sig Obj { id : Id }                                  // object
4  disj sig MBox extends Obj { contents : set Obj }     // message box
5  disj sig Msg extends Obj { content : set String }    // message
6  fact SomeNonempty {
7     some MBox.contents
8     some Msg.contents
9  }
```

The first three lines introduce three sets and their corresponding types named `Id`, `String`, and `Obj`. Lines 4-5 declare sets (but not types) `Mbox` and `Msg` as disjoint subsets of `Obj`. Lines 3-5 also define several relations between pairs of sets; for example, the relation `id` maps the elements of the set `Obj` to those of the set `Id`. Lines 6-9 (are intended to) state that at least one mailbox in the universe is non-empty and that at least one message contains some text.

1.2.1 Coarse Control over Scopes

The Alloy Analyzer works by translating an Alloy model into a Boolean satisfiability problem, which is then given to a SAT solver. The translation requires that the user provide numerical bounds, or *scopes*, on the size of all *types* defined in the model. Since Alloy 2.0 has no notion of subtyping, this means that the user can specify scopes only at the root signature level. In our example, the user would be allowed to set the scope for `Obj` but not for `MBox` and `Msg`.

The coarse-grained specification of scopes can seriously hurt the Analyzer's performance. For example, suppose that the scope of `Obj` is *n*. The Analyzer would translate the relation `contents` into an $n \times n$ matrix of Boolean variables and the set `MBox` into a $1 \times n$ matrix. The translation of the relational join on line 7 entails multiplying the matrices representing `MBox` and `contents`, which requires $O(n^2)$ scalar multiplications. Now, suppose what the user really wanted was an instance of the model with n/m message boxes and $(n * (m-1))/m$ messages. The ability to set the desired scopes on `MBox` and `Msg` would allow the references to `MBox` and `contents` to be translated into $1\times (n/m)$ and $(n/m)\times n$ matrices, respectively. As a result, the translation of line 7 would now involve $O(n^2/m)$ scalar multiplications. Taking $n = m$ (i.e. there is one message box), it is easy to see how finer grained scoping could improve the Analyzer's performance.

1.2.2 Namespace Sharing

The type system of Alloy 2.0 promotes subsets' fields to the root signature, effectively forcing all descendents of the same root to share the same namespace. This hinders modular design and limits the effectiveness of the Analyzer 2.0's typechecking algorithm. The user cannot add a field to a subset without knowing the contents of other subsets in the same set hierarchy. Worse yet, he may accidentally write a vacuous expression that will typecheck. For instance, line 8 of our model would compile in Alloy 2.0 since the type of the expression `Msg` and the left type of the expression `contents` are the same. However, the expression `Msg.contents` is necessarily empty since the relation `contents` maps message boxes to objects. This could by easily discovered by a typechecker if lines 4 and 5 introduced the types `MBox` and `Msg` as disjoint subtypes of `Obj`.

1.3 Contributions

The type system of Alloy 3.0 retains the desirable features of its predecessor without suffering from the drawbacks discussed in the previous section. The principle contributions of this work[1] are [7]:

- A treatment of subtyping that utilizes subtype information but requires no casts and no changes to the underlying semantics of the language. For example, the expression `Obj.contents` is legal in the context of the email client model: we do not require the user to downcast `Obj` to `MBox`. Alloy 3.0 subtypes also enable fine-grained control over scopes which we exploit to make constraint solving more efficient [8]. In the case of the email client model, Alloy 3.0 would allow the user to set bounds on the scopes of signatures `MBox` and `Msg`.

- A scheme for resolving references to overloaded names that does not burden the semantics of the language with operational details or distinguish terms that should be equivalent. For example, suppose that we rename the field `content` in signature `Msg` to `contents` (which is legal since `Msg` and `MBox` have separate namespaces in Alloy 3.0). Our typechecker would infer that the reference to the name `contents` in the expression `MBox.contents` on line 7 refers to the field `contents` of `MBox` rather than that of `Msg`. Furthermore, the equivalent expression, `~contents.MBox`, would also successfully typecheck in Alloy 3.0. Alloy 2.0, on the other hand, would report that the name `contents` is ambiguous in this context and would require the user to rewrite the expression as `~MBox$contents.MBox`.

- A treatment of union types that allows more flexibility in modelling but which detects gratuitous, erroneous unions. For instance, we could make the definition of the field `contents` in `MBox` more precise by declaring it as `contents:  set MBox + Msg`. However, the typechecker would not accept the expression `(Id + MBox).contents` since the set `Id` could be

[1] The theory and formalism of the Alloy 3.0 type system have been developed jointly by Jonathan Edwards, Daniel Jackson and myself. I have designed and implemented the Alloy 3.0 typechecker.

replaced by the empty set without changing the meaning of the expression: i.e. the set of values represented by the expression `(Id + MBox).contents` is the same as that represented by the expression `MBox.contents`.

Although our type system was developed in the context of the Alloy modelling language, the system itself is applicable to any first order logic with relations. In particular, we have found it to be an invaluable tool for understanding and manipulating object models [7]. As object models continue to gain in popularity among the software engineering community, we hope that the work presented here will have a significant impact on the way in which practitioners reason about and build software.

2 An Overview of Alloy 3.0

Alloy is a relational constraint language designed for lightweight modelling of software systems [11]. The kernel of the language, MicroAlloy, is simply a syntax for first order logic with relational operators. The full language provides additional syntactic constructs which improve readability and succinctness of Alloy specifications. However, since all constructs of the full language can be expressed in terms of MicroAlloy primitives, this chapter will focus on the syntax and semantics of MicroAlloy 3.0.

2.1 Gross Structure

The syntax of MicroAlloy is shown in Figure 2-1. A MicroAlloy specification consists of a sequence of paragraphs. Each type of paragraph encapsulates a modularity mechanism:

- *signatures* introduce sets and relations
- *predicates* name and parameterize formulas that can be used elsewhere
- *functions* name and parameterize expressions that can be used elsewhere
- *facts* are formulas representing assumptions that always hold; and
- *assertions* are formulas intended to follow from the facts and implicit constraints.

The following model, inspired by Paul Simon’s 1973 song “One man’s ceiling is another man’s floor,” showcases each kind of paragraph:

```
sig Platform {}
sig Man { ceiling, floor : Platform }
fun down(m : Man) : Platform { m.floor }
fun up(m : Man) : Platform { m.ceiling }
pred Above(m, n : Man) { down(m) = up(n) }
fact Song { all m : Man | some n : Man | Above(n, m) }
assert BelowToo { all m : Man | some n : Man | Above(m, n) }
```

The signature `Platform` introduces a set Platform. The signature `Man` introduces a set of the same name and two relations, `ceiling` and `floor`, which have Man as their first column and Platform as their second. These relations are implicitly constrained to be total functions, giving every man exactly one ceiling and one floor.

The function on line 3 associates the names `down` and an argument `m` with an expression

specification	::=	paragraph*
paragraph	::=	sigDecl **|** predDecl **|** funDecl **|** factDecl **|** assertDecl
sigDecl	::=	[**abstract**] **sig** sigName [**extends** sigName] **{** decl,* **}**
	|	**sig** sigName **in** sigName **{** decl,* **}**
predDecl	::=	**pred** predName (argDecl,*) **{** formula* **}**
funDecl	::=	**fun** funName (argDecl,*) : [**set** **|** **option**] expr **{** expr **}**
factDecl	::=	**fact** [factName] **{** formula* **}**
assertDecl	::=	**assert** [assertName] **{** formula* **}**
decl	::=	var **:** [**set** | **option**] expr
formula	::=	elemFormula | compFormula | quantFormula | quantExpr
elemFormula	::=	expr **in** expr | expr **=** expr
compFormula	::=	**not** formula | formula logicop formula
logicop	::=	**and** | **or** | **=>**
quantFormula	::=	quantifier decl **|** formula
quantExpr	::=	quantifier expr
quantifier	::=	**all** | **some** | **no**
expr	::=	sigName | var **|** **univ** **|** **none** | expr binop expr | unop expr
binop	::=	**–** | **+** | **&** | **.** | **–>**
unop	::=	**^** | ***** | **~**
var	::=	identifier
sigName	::=	identifier

Figure 2-1 MicroAlloy Syntax

that evaluates to the Platform serving as `m`'s floor. Similarly, the function `up` evaluates to the Platform that is its argument's ceiling. The predicate `Above` is a two argument abstraction of a formula that is true when `down` of its first argument is the same as the `up` of its second argument. The fact `Song` states the title of the song: that every man `m` has some man `n` above him.

Finally, the assertion `BelowToo` claims a corollary of `Song`: that every man `m` is above a man `n`. In other words, `BelowToo` asserts that the conclusion "One man's floor is another man's ceiling" follows from the fact "One man's ceiling is another man's floor." Although seemingly true, this conclusion is invalid, and the Alloy Analyzer finds a counterexample such as this:

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

The problem is that `M0`'s ceiling is also his floor, since the model does not formalize the notion that one man's ceiling is *another* man's floor. Indeed, we need an additional premise which, together with `Song`, will make `BelowToo` valid: different men do not share the same floor or the same ceiling.

2.2 Expressions and Formulas

Expressions and formulas are given a slightly unconventional interpretation in Alloy. Every expression denotes a relation. Thus a set is treated as a unary relation and a scalar as a singleton set. A consequence of this interpretation is that a quantifier always binds its variable to a relation value.

The keyword `in` denotes the subset relation. In general, *e* `in` *s* is true when the relation *e* is a subset of the relation *s*. When *e* is a unary, singleton relation and *s* is a unary relation, the statement *e* `in` *s* has the usual set theoretic meaning 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 viewed as sets of tuples. The arrow operator takes the cross product of its arguments: the relation p –> q contains the tuple $\langle p_1, \ldots, p_m, q_1 \ldots, q_n\rangle$ when p contains $\langle p_1, \ldots, p_m\rangle$ and q contains $\langle q_1 \ldots, q_n\rangle$. The operators ~ and ^ are the transpose and transitive closure operators on binary relations. The constant `univ` represents the standard universal relation that maps every atom to every atom, and the constant `none` denotes the empty relation.

The remaining binary operator, dot, denotes a generalized relational composition. The composition of expressions p and q contains the tuple $\langle p_1, \ldots, p_{m\text{-}1}, q_2 \ldots, q_n\rangle$ when $\langle p_1, \ldots, p_m\rangle \in p$, $\langle q_1 \ldots, q_n\rangle \in q$, and $p_{\mathrm{m}} = q_1$ for an appropriately defined notion of equality. In the specific case when p is a set and q is a binary relation, the composition $p.q$ is the standard relational image of p under q. If p and q are both binary relations, $p.q$ is standard relational composition.

MicroAlloy provides three logical quantifiers: `all` (universal), `some` (existential), and `no` (not exists). The last two can be applied to expressions. The formulas `some` *e* and `no` *e* are true when the expression *e* denotes a non-empty and empty relation respectively.

The definitions of MicroAlloy operators given above are intended to apply even if their arguments do not have uniform arities. Technically, an overloaded relation can contain tuples of different lengths, including tuples of length zero. The type system eliminates this leniency and only permits specifications in which all relational expressions have a uniform arity greater than zero.

Another technicality worth noting is that Alloy's treatment of scalars as relations, and the first order nature of the language, allows us to sidestep many of the semantic problems usually associated with partial functions. Specifically, rather than writing $f(x) = y$ to say that the function f maps the scalar x to the scalar y, we can 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 of its domain results simply in the empty set.

2.3 Relations, Extensions, Subsets 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. Thus interpreting signature paragraphs is straightforward. The declaration

```
sig S {r : E}
```

introduces a set `S` and a relation `r`, which is constrained to be a subset of the relation given by the expression `S -> E`. The expression `E` is a relational expression over the signature names of the specification, including `S` itself.

Multiplicity markings provide a shorthand for constraining the multiplicity properties of a relation. A binary relation `r` declared as

```
sig S {r : T}
sig T {}
```

is a total function. Adding the keywords `option` or `set` to the declaration

```
sig S {r : [option | set] T}
```

makes `r` into a partial function or an arbitrary relation respectively. Hence, the default is to implicitly constraint relations to be total functions; the keywords `option` and `set` eliminate the implicit constraints.

Special multiplicity symbols are used in the declarations of higher arity relations. For example, the declaration of a ternary relation `q`

```
sig S {q : T -> U}
sig T {}
sig U {}
```

implies nothing about its multiplicity properties. However, declaring `q` as

```
sig S {q : T ->? U}
```

constrains the binary relation `s.q` to be a partial function for all `s` in `S`. If another question mark is added to the left of the arrow,

```
   sig S {q : T ?->? U},
```

then the relation `s.q` becomes an injection for each `s` in `S`. In general, declaring `q` to be

```
   sig S {q : T n->m U}
```

constrains the relation `s.q` to map each atom of `T` to *m* atoms of `U`, and each atom of `U` to *n* atoms of `T`; *m* and *n* stand for the symbols ?, !, +, meaning zero or one, exactly one, and one or more, respectively.

In addition to introducing sets and relations into a model, signature declarations also introduce *types*. For example, the declaration

```
   sig S {}
```

introduces the set `S` and the type `S`. The relationship between the set `S` and the type `S` introduced by a signature is simple: every element of `S` is considered to be of type `S`. Signature extension thus defines one signature to be a subset of another and its type to be a subtype of the superset's type. For instance, the paragraphs

```
   sig Candy {}
   sig Chocolate extends Candy {}
```

introduce the sets `Candy` and `Chocolate` and the types `Candy` and `Chocolate`. The set `Chocolate` is a subset of `Candy`, and the type `Chocolate` is a subtype of `Candy`. Hence, every element of the set `Chocolate` belongs to the set `Candy` and has the property of being a specific subtype of `Candy` called `Chocolate`.

All extensions of a signature are mutually disjoint. Writing

```
   sig Vehicle {}
   sig Car extends Vehicle {}
   sig Plane extends Vehicle {}
```

creates the sets `Car` and `Plane` which are disjoint subsets of `Vehicle`. Marking a signature as `abstract` implies that its extensions exhaust it. In particular, prefacing the declaration of `Vehicle` with the keyword `abstract` would imply that all `vehicle`s are either `car`s or `plane`s.

A signature can be declared to be `in` another signature or a union of signatures. The semantics of `in` differs from the semantics of `extends` in two ways. First,

```
sig S {}
sig T in S {}
sig U in S {}
```

introduces the sets `T` and `U` but not the type `T` and `U`. Hence, all elements of `T` and `U` have the type `S`. Second, the sets `T` and `U` are not mutually disjoint; the above definition does not precludes the existence of some `s` that belongs to both `T` and `U`.

The rule bounding relations applies equally to those whose first column is a signature that either `extends` or is `in` another signature. For example,

```
sig S {}
sig T extends S {r : E}
sig U in S {q : E}
```

creates the relations `r` and `q` which are subsets of the expressions `T –> E` and `U –> E` respectively. Given a scalar `s` in the set `S`, `s.r` will denote the empty set if `s` is not in `T`.

The same name can be used for two different relations, so long as the types of the signatures in which they are declared are neither direct nor indirect subtypes of each other. The meaning of an occurrence of a relation name in an expression is the union of all the relations of that name. For instance, given the declarations

```
sig S {}
sig T extends S {r : E}
sig U extends S {r : E}
```

the expression `x.r` denotes $\texttt{x.(}r_T + r_U\texttt{)}$ where r_T stands for the relation `r` defined in `T` and r_U for the relation `r` defined in `U`. As we shall see in chapter 3, if `x` can be determined to have the type `T`, the relation `r` in `x.r` will be resolved to r_T. The meaning of the expression `x.r`, given as a sum of overloadings, will then be equivalent to the meaning of the resolved expression $\texttt{x.}r_T$.

2.4 Formal Semantics

The meaning of an Alloy formula is its *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 *basic models* whose free variables are the sets and relations declared in the signatures. The relevant formula for basic models is the conjunction of the formulas in all the facts and the formulas implied by the signature and relation declarations.

The models of a predicate are obtained by augmenting the basic models with the parameters of the predicate and eliminating any bindings that do not also satisfy the predicate's constraints; a function's models are obtained in the same way. The models of an assertion are the subset of the basic models which satisfy the formulas of the assertion. An assertion is *valid* if every basic model is also a model of the assertion; a *counterexample* is a basic model that does not satisfy the formulas of the assertion.

Typically, the predicates and functions of an Alloy specification represent properties of a system such as invariants (over a single state), transitions (over a pair of states), and traces (over a sequence of states). Assertions record intended consequences of the design of the system. Usually, an assertion is used to check whether a collection of specified properties implies some other, often more fundamental, property. A counterexample of an assertion demonstrates that the design does not always exhibit the desired behavior.

A partial semantics for MicroAlloy is given in Figure 2-2. 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. The operators appearing on the right hand side of the equations should be interpreted as operators in the meta theory, distinct from the operators with the same name appearing on the left, which belong to the language being defined.

M	:	Formula$\to$Binding$\to$Boolean
E	:	Expression$\to$Binding$\to$RelationValue

M[**not** f]b	=	$\neg$M[f]b	M[**all** x: e | f]	=	$\wedge$\{M[f] (b $\oplus$ x$\rightarrow$v) | v $\subseteq$ E[e]b $\wedge$ #v=1\}
M[f **and** g]b	=	M[f]b $\wedge$ M[g]b	M[**all** x: **set** e | f]	=	$\wedge$\{M[f] (b $\oplus$ x$\rightarrow$v) | v $\subseteq$ E[e]b\}
M[f **or** g]b	=	M[f]b $\vee$ M[g]b	M[**some** x: e | f]	=	$\vee$\{M[f] (b $\oplus$ x$\rightarrow$v) | v $\subseteq$ E[e]b $\wedge$ #v=1\}
M[f **=>** g]b	=	M[f]b $\Rightarrow$ M[g]b	M[**some** x: **set** e | f]	=	$\vee$\{M[f] (b $\oplus$ x$\rightarrow$v) | v $\subseteq$ E[e]b\}
M[p **in** q]b	=	E[p]b $\subseteq$ E[q]b	M[**some** e]b	=	E[e]b $\supset \varnothing$
M[p **=** q]b	=	E[p]b = E[q]b	M[**no** e]b	=	E[e]b = $\varnothing$
E[**none**]b	=	$\varnothing$	E[**univ**]b	=	U(b)
E[p**.**q]b	=	E[p]b . E[q]b	E[**~**p]b	=	\{(p_2, p_1) | (p_1, p_2) $\in$ E[p]b\}
E[p**–>**q]b	=	E[p]b $\rightarrow$ E[q]b	E[**^**p]b	=	μX.(X $\cup$ X.\{(p_1, p_2) | (p_1, p_2) $\in$ E[p]b\})
E[p**+**q]b	=	E[p]b $\cup$ E[q]b			
E[p**&**q]b	=	E[p]b $\cap$ E[q]b			
E[p**–**q]b	=	E[p]b \ E[q]b			

variables	:	E[x]b = b(x)
signames	:	E[s]b = b(s)
relations	:	E[r]b = $\cup$ \{b(r_i) | r_i has name r\}

Figure 2-2 Semantics of Expressions and Formulas

A binding must include an explicit set that represents the universal set of all atoms. For a binding b, the meta expression U(b) denotes the universal set. The value assigned to the variable x by b is obtained by applying b to x. The expression b $\oplus$ x$\rightarrow$v creates a new binding that is like b but which binds the value v to the variable x.

The semantics allows relations with multiple arities. An expression can denote a relation containing tuples of different lengths, including zero. The transpose and transitive closure operators may be applied to non-binary relations. However, such an application has meaning only for the binary relations contained in the set of relations denoted by the argument expression.

3 The Type System: Formal Description

3.1 Types as Relations

In Alloy 3.0 type system, the type expressions are relational expressions. They form a simple sublanguage of value expressions: the variables of the type language are restricted to signature names and its operators to union and product. The algebraic properties of union and product enable us to represent types as relational values. The type inference rules can therefore use relational operators to construct types.

All type computations are performed on *atomic types* that partition the universe. This allows us to eliminate subtype comparisons from the type inference rules in favor of exact matching. However, as we will see in chapter 4, the type system is implemented with subtype comparisons rather than with operations on atomic types, since the former approach is more space-efficient and facilitates construction of user-friendly error messages.

The atomic types include the types of *leaf signatures* and *remainder types*. A leaf signature S is not extended by any other signature, and the atomic type S denotes the same set of elements as the signature S itself. A remainder type S denotes the set of elements in a compound non-abstract signature S which belong to S but not to the signatures that extend S.

The intuition behind the idea of using atomic types can be gleaned from the following example:

```
sig Coffee {}
sig Chocolate {}
sig Flavor {}
sig Bold extends Flavor {}
sig Mild extends Flavor {}
sig GiftBasket {
   …
   holds: (Coffee + Chocolate) -> Flavor
}
```

The relation `holds` is a ternary relation of type `GiftBasket -> (Coffee + Chocolate) -> Flavor`, which itself is an expression that bounds the value of `holds`. The type of `holds` atomizes to

```
GiftBasket -> (Coffee + Chocolate) -> (Flavor + Bold + Mild)
```

where the atomic type `Flavor` represents the set

```
Flavor – (Bold + Mild).
```

Since product distributes over union, we can rewrite any type as a sum of products. Hence, the atomized type of `holds` becomes

```
   GiftBasket -> Coffee -> Flavor
+  GiftBasket -> Coffee -> Bold
+  GiftBasket -> Coffee -> Mild
+  GiftBasket -> Chocolate -> Flavor
+  GiftBasket -> Chocolate -> Bold
+  GiftBasket -> Chocolate -> Mild
```

Given that both product is associative, and union is both associative and commutative, any type can be represented as a set of tuples of atomic types, as the form of the above expression suggests. Types are therefore relations, and can be compared and combined using relational operators.

3.2 Inference Rules

The inference rules in Figure 3-1 are interpreted over the set of valid types. A valid type is constructed as follows:

$$\mathtt{Type} = \cup\{b_1 \rightarrow \ldots \rightarrow b_n \mid n \geq 1 \wedge ((\overset{n}{\underset{i\geq 1}{\forall}} b_i \in \mathtt{Base}) \vee (\overset{n}{\underset{i\geq 1}{\forall}} b_i = \mathtt{none}))\}$$

The set `Base` includes a model's atomic types and `none` is a special atomic type denoting an implicit "empty" signature associated with the keyword `none`. This construction rules out zero-arity relations, forbidding expressions such as `Coffee.Coffee` which are semantically well-defined but useless. It also rules out nonsensical expressions such as `none–>Coffee`.

$\mathtt{Base}_n \quad : \quad \wp(\mathtt{Base}^n)$

$\beta_n \quad : \quad \mathtt{Base}_n \cup \wp(\{\mathtt{none}\}^n)$

$$\frac{\begin{array}{c}\Gamma \vdash p : P, \Gamma \vdash q : Q, \\ P\neq\emptyset \wedge Q\neq\emptyset \Rightarrow \exists n\geq 1\ \{\beta_n \cap P\} \neq\emptyset \wedge \{\beta_n \cap Q\} \neq\emptyset\end{array}}{\Gamma \vdash p \mathbin{\mathbf{+}} q : P + Q} \qquad \frac{\Gamma \vdash p \mathbin{\&} q : T}{\Gamma \vdash p \mathbin{\mathbf{=}} q : T}$$

$$\frac{\Gamma \vdash p : P, \Gamma \vdash q : Q}{\Gamma \vdash p \mathbin{\mathbf{->}} q : P -> Q} \qquad \frac{\Gamma \vdash p \mathbin{\&} q : T}{\Gamma \vdash p \mathbin{\mathbf{in}} q : T}$$

$$\frac{\Gamma \vdash p : P, \Gamma \vdash q : Q}{\Gamma \vdash p \mathbin{\mathbf{\&}} q : P \mathbin{\&} Q} \qquad \frac{\Gamma \vdash e : T, \Gamma, x{:}T \vdash f}{\Gamma \vdash \mathbf{all}\ x{:}\ e \mathbin{|} f}$$

$$\frac{\Gamma \vdash p : P, \Gamma \vdash q : Q}{\Gamma \vdash p\mathbf{.}q : P.Q} \qquad \frac{\Gamma \vdash e : f, \Gamma \vdash g}{\Gamma \vdash f \mathbin{\mathbf{and}} g}$$

$$\frac{\Gamma \vdash p : P, \Gamma \vdash q : Q, P \mathbin{\&} Q \neq \emptyset}{\Gamma \vdash p \mathbin{\mathbf{-}} q : P} \qquad \frac{}{\vdash \mathbf{none} : \{\mathrm{none}\}}$$

$$\frac{\Gamma \vdash p : P}{\Gamma \vdash \mathbf{\sim}p : \sim\{\mathtt{Base}_2 \cap P\}} \qquad \frac{}{\vdash \mathbf{univ} : \mathtt{Base}_1}$$

$$\frac{\Gamma \vdash p : P}{\Gamma \vdash \mathbf{\hat{}}p : \hat{}\{\mathtt{Base}_2 \cap P\}}$$

Figure 3-1 Type Inference Rules

The empty type, denoted as ∅, is also included in the set of valid types. The empty type represents the type of an erroneous expression such as `Coffee.holds`. Another way to look at it is that the empty type indicates a redundancy: an expression whose type is ∅ can be removed from the specification without affecting its meaning. If a type of an expression or formula cannot be deduced using the rules in Figure 3-1, then the expression or formula is deemed to be of type ∅.

Types are inferred for elementary formulas as well as expressions. This enables detection of errors due to comparisons of expressions whose values are disjoint. For example, the formula `Coffee in Chocolate` is always false since the sets `Coffee` and `Chocolate` do not intersect.

Judgments for the signature types are assumed, with an axiom giving a type in atomized form for each signature name. The type of a relation `r` defined in signature `S` as

```
   sig S {r: e}
```

is deduced from an additional inference rule

$$\frac{\Gamma \vdash S\text{–>}e : T}{\Gamma \vdash r_S : T}$$

Together, these rules allow typing of signature declarations, even though they might be mutually recursive. The relation `r` is labeled by its signature since it may be overloaded.

The rules given apply to formulas appearing in facts. The extension to predicates and functions is obvious: their bodies are checked in an environment Γ that binds, additionally, their arguments.

3.3 Relevance Rules

Recall that at the end of section 2.3, we claimed that Alloy 3.0 typing rules resolved overloaded field references. Specifically, it was stated that, given the declarations

```
   sig S {}
   sig T extends S {r : E}
   sig U extends S {r : E},
```

the relation `r` in the expression `x.r` would be resolved to r_T if `x` can be determined to have the type `T`. Furthermore, the previous section indicated that the type of an expression is set to ∅ if it is possible to use type information to deduce that the given expression is redundant.

The type inference rules in Figure 3-1 fulfill neither of these claims. They provide no way to resolve reference overloading, and they do not set the types of all provably redundant expressions to ∅. Consider, for example, the expression `(S + T).(S->S)` in the context of the declarations given above: the expression `T` is clearly redundant since only the type of `S` is relevant to the join of `(S + T)` and `(S->S)`. Yet, the inference rules will type the signature reference `T` in `(S + T).(S->S)` as {`T`} rather than ∅.

These deficiencies are corrected by applying the rules in Figure 3-2 to the types inferred using the rules from the previous section. The Reduction Rules work in a top-down fashion to reduce the inferred type of each expression to its *relevance type*. The relevance type of an expression is the least upper bound on the expression's value that can be inferred from the context of the enclosing expression. In particular, the following relationship holds between every Alloy expression `x` and its inferred and relevance types:

x ⊆ *relevance_type*(`x`) ⊆ *inferred_type*(`x`).

The type judgment $\langle \Gamma, X[] \rangle \vdash e \downarrow T$ says that T is the relevance type of e in the type environment Γ and the context X (where X is the formula or expression enclosing e). If the relevance type of an expression e cannot be determined from the rules in Figure 3-2, then e is deemed to have the empty relevance type, resulting in a redundancy error. The relevance type of an overloaded field reference r must intersect with the type of exactly one field named r; otherwise, we can conclude that the overloading cannot be resolved from the context and signal an ambiguity error.

Again, the given rules apply to formulas inside facts. The extension to predicates and functions is the same as before. The bodies of quantified formulas are handled like the bodies of predicates with the additional requirement that Γ binds the quantified variables to their relevance types.

$$\frac{\langle \Gamma, X[]\rangle \vdash p \mathbin{\mathbf{+}} q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\langle \Gamma, X[[]+q]\rangle \vdash p \downarrow P\&T \quad \langle \Gamma, X[p+[]]\rangle \vdash q \downarrow Q\&T}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash p \mathbin{\mathbf{-\!>}} q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\begin{array}{c}\langle \Gamma, X[[]\text{–>}q]\rangle \vdash p \downarrow \{\langle P_1, \ldots, P_n\rangle \in P \mid \exists\langle Q_1, \ldots, Q_m\rangle \in Q, \langle P_1, \ldots, P_n, Q_1, \ldots, Q_m\rangle \in T\} \\ \langle \Gamma, X[p\text{–>}[]]\rangle \vdash q \downarrow \{\langle Q_1, \ldots, Q_m\rangle \in Q \mid \exists\langle P_1, \ldots, P_n\rangle \in P, \langle P_1, \ldots, P_n, Q_1, \ldots, Q_m\rangle \in T\}\end{array}}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash p\mathbf{.}q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\begin{array}{c}\langle \Gamma, X[[].q]\rangle \vdash p \downarrow \{\langle P_1, \ldots, P_n\rangle \in P \mid \exists\langle Q_1, \ldots, Q_m\rangle \in Q, P_n=Q_1 \wedge \langle P_1, \ldots, P_{n-1}, Q_2, \ldots, Q_m\rangle \in T\} \\ \langle \Gamma, X[p.[]]\rangle \vdash q \downarrow \{\langle Q_1, \ldots, Q_m\rangle \in Q \mid \exists\langle P_1, \ldots, P_n\rangle \in P, P_n=Q_1 \wedge \langle P_1, \ldots, P_{n-1}, Q_2, \ldots, Q_m\rangle \in T\}\end{array}}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash p \mathbin{\mathbf{\&}} q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\langle \Gamma, X[[]\&q]\rangle \vdash p \downarrow P\&T \quad \langle \Gamma, X[p\&[]]\rangle \vdash q \downarrow Q\&T}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash p - q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\langle \Gamma, X[[]-q]\rangle \vdash p \downarrow P\&T \quad \langle \Gamma, X[p-[]]\rangle \vdash q \downarrow Q\&T}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash \mathbf{\sim}p \downarrow T,\ \Gamma \vdash p : P}{\langle \Gamma, X[\sim[]]\rangle \vdash p \downarrow (\sim T)\&P}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash \mathbf{\hat{}}p \downarrow T,\ \Gamma \vdash p : P}{\langle \Gamma, X[\hat{}[]]\rangle \vdash p \downarrow \{\langle P_1, P_2\rangle \in P \mid \exists\langle T_1, T_2\rangle \in T, \mathit{Path}_T(T_1,P_1) \wedge \mathit{Path}_T(P_2,T_2)\}}$$

where $\mathit{Path}_T(x, z)$ iff $x = z \vee (x, z) \in T \vee \exists y_1, \ldots, y_n \{\langle x, y_1\rangle, \langle y_1, y_2\rangle, \ldots, \langle y_{n-1}, y_n\rangle, \langle y_n, z\rangle\} \subseteq T$

$$\frac{\langle \Gamma, X[]\rangle \vdash p \mathbin{\mathbf{=}} q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\langle \Gamma, X[[]=q]\rangle \vdash p \downarrow P\&T \quad \langle \Gamma, X[p=[]]\rangle \vdash q \downarrow Q\&T}$$

$$\frac{\langle \Gamma, X[]\rangle \vdash p \mathbin{\mathbf{in}} q \downarrow T,\ \Gamma \vdash p : P,\ \Gamma \vdash q : Q}{\langle \Gamma, X[[] \text{ in } q]\rangle \vdash p \downarrow P\&T \quad \langle \Gamma, X[p \text{ in } []]\rangle \vdash q \downarrow Q\&T}$$

Figure 3-2 Type Reduction Rules

not be resolved in the smallest enclosing subexpression. In `~(r.b).c`, the variable `c` resolves the overloading of `r`. Finally, the redundancy of an expression is not necessarily obvious from its local context. The redundancy of `B` in the last example is implied by a constraint at the top of the formula (namely, that the type of `a` is `A`).

4 Architecture of Alloy 3.0 Typechecker

In the following pages, we will briefly review the Alloy Analyzer's functionality, limitations, and internal structure, and then delve into the details of the Analyzer's typechecking mechanism. Section 4.1 describes the context in which the typechecker operates and the existing design constraints to which the implementation of the typechecker had to conform. The Alloy Analyzer framework and its translation and parsing facilities were developed largely by Ilya Shylakhter and Manu Sridharan. The typechecking software was developed by the author. Its design and analysis are discussed in sections 4.2 and 4.3. Detailed explanations of other major Analyzer components—atomizer and translator—can be found in [8], [12], and [13].

4.1 An Overview of the Alloy Analyzer 3.0

4.1.1 Functionality and Limitations

In keeping with the lightweight modelling philosophy [14], Alloy's desingers[1] have developed a tool that fully automates the semantic analysis of the language. The Alloy Analyzer can *simulate* executions of a given model and *check* assertions about its properties. Simulation and checking are essential in helping the user incrementally arrive at a desired specification [10].

Since Alloy is not decidable, the Analyzer cannot perform a sound and complete analysis. Instead, given a model *M* which defines basic types T_1 ... T_N, the tool searches for an example (or counterexample) of *M* in the set of all possible instances of *M* which use at most S_i atoms of T_i. The parameter S_i, also known as the *scope* of type T_i, is determined by the user.

[1] Alloy 2.0 was designed by Daniel Jackson, with assistance from Ilya Shylakhter and Manu Sridharan. Alloy 3.0 was born with additional assistance from Jonathan Edwards, Emina Torlak, Vincent Yeung, Gregory Dennis and Mandana Vaziri.

The user cannot make any logically sound conclusions when no instance is found for a model. However, experience shows that many models have a small-scoped instance if one exists. Therefore, the user can view the Analyzer's failure to find an instance as heuristic evidence that an assertion is valid or an invariant inconsistent [13].

4.1.2 A Sketch of the Analyzer Architecture

The Analyzer is effectively an Alloy compiler [13]. It converts Alloy source code into an intermediate form which is then translated into boolean logic. The generated boolean formula is handed to an off-the-shelf SAT solver, such as Chaff [16]. If the solver finds an instance of the boolean formula, the solution is translated into an instance of the analyzed relational formula and presented to the user.

The process of converting Alloy source into the intermediate form consists of three stages. First, the source is parsed to generate an *abstract syntax tree* (AST) which is a Composite [9] of *Nodes* representing Alloy expressions and formulas. Then, the AST is typechecked and systematically re-written in terms of the Alloy kernel language primitives. Finally, the simplified AST is *atomized* to form a *translatable AST*. Atomization refactors the user-defined type hierarchy into a flat collection of disjoint *atomic types* and then decomposes all the constraints into equivalent constraints involving smaller relations or predicates over the atomic types [8].

The translation of the translatable AST into boolean logic relies on the following simple idea: fixing the scope enables us to represent the value of a relation $r : S \text{ -> } T$ as a bit matrix R such that $R[i, j] = 1$ if and only if r relates the i^{th} atom of S to the j^{th} atom of T [13]. We can express all possible values of r by assigning a boolean variable to each cell of R. Any constraint on r can be written as a formula in these boolean variables. Thus, we can translate a relational formula into a boolean formula by expressing each relational variable as a matrix of boolean variables [12].

The boolean formula produced by the translation algorithm is guaranteed to be satisfiable if and only if the original relational formula has a solution in the given scope. As a result, the SAT solver output is interpreted in the obvious way. If the solver fails to produce a solution, the user is informed that none exists in the specified scope. If a solution is found, it is converted into an instance of the original relational formula and presented to the user as a graph, tree, or a string of characters.

4.2 Description of Key Type Abstractions

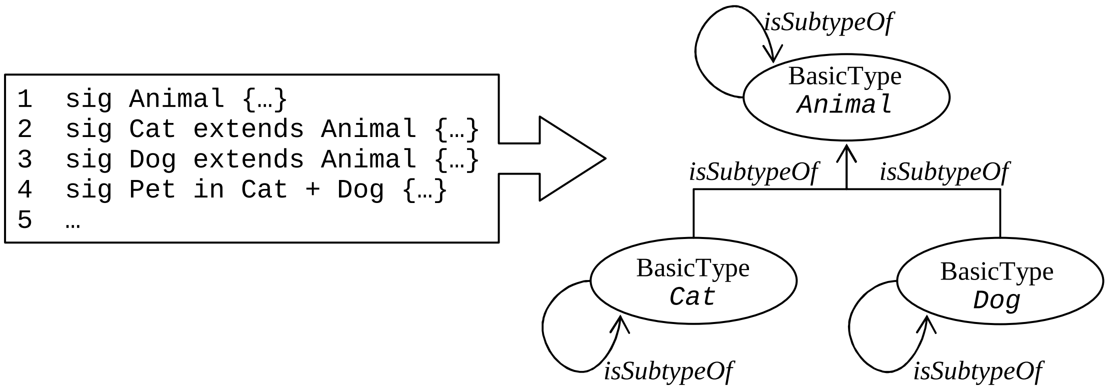
The Analyzer's typechecking mechanism rests on a few key abstractions: *BasicType*, *RelationType*, and *UnionType*. These abstractions encapsulate the functionality needed by the typechecking algorithms to calculate the type of an Alloy expression. In the remainder of this chapter, the BasicType corresponding to the type of an Alloy signature will be typographically designated with italicized Courier font (e.g. *A*). A RelationType will be represented as a sequence of arrow-separated BasicTypes enclosed in square brackets (e.g. [*A*→*B*]). UnionTypes will be denoted as a set of plus-separated RelationTypes enclosed in curly braces (e.g. {[*A*→*B*] + [*B*→*A*]}).

4.2.1 BasicType

The interface BasicType (Figure 4-1) represents Alloy types introduced by type-generating Alloy signatures. These signatures include the built-in signature `Int` and all user-defined signatures except those declared with the keyword `in`. A signature declared to be `in` another signature creates a new set but not a new Alloy type. Hence, in Figure 4-2, the signatures `Animal`, `Dog`, and `Cat` introduce new BasicTypes with the corresponding names, but the signature `Pet` does not.

```
package alloy.type;

public interface BasicType {

    public boolean isEmpty();

    public boolean isSubtypeOf(BasicType other);

    public BasicType intersect(BasicType other);
}
```

Figure 4-1. BasicType Interface



```
sig Animal {…}
sig Cat extends Animal {…}
sig Dog extends Animal {…}
sig Pet in Cat + Dog {…}
…
```

Figure 4-2 Assigning Types to Alloy Signatures

Concrete implementations of the BasicType interface are immutable classes SigType and EmptyType (Figure 4-3). An instance of the class SigType is a Flyweight [9] which represents the type of a user-defined signature. For example, the BasicType corresponding to the signature `Animal` in Figure 4-2 would be a SigType. EmptyType is a Singleton [9] representation for the type of the Alloy expression `none`.

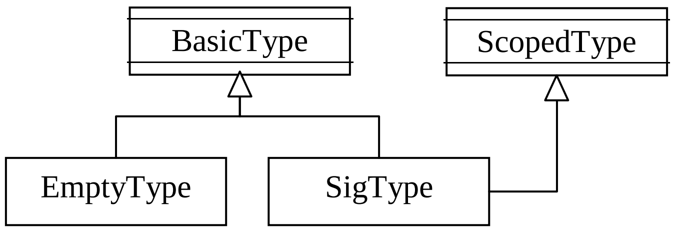


Figure 4-3 BasicType Hierarchy

The user may assign a scope to all signatures that introduce a new type. Hence, a SigType is also a ScopedType, as shown in Figure 4-3.

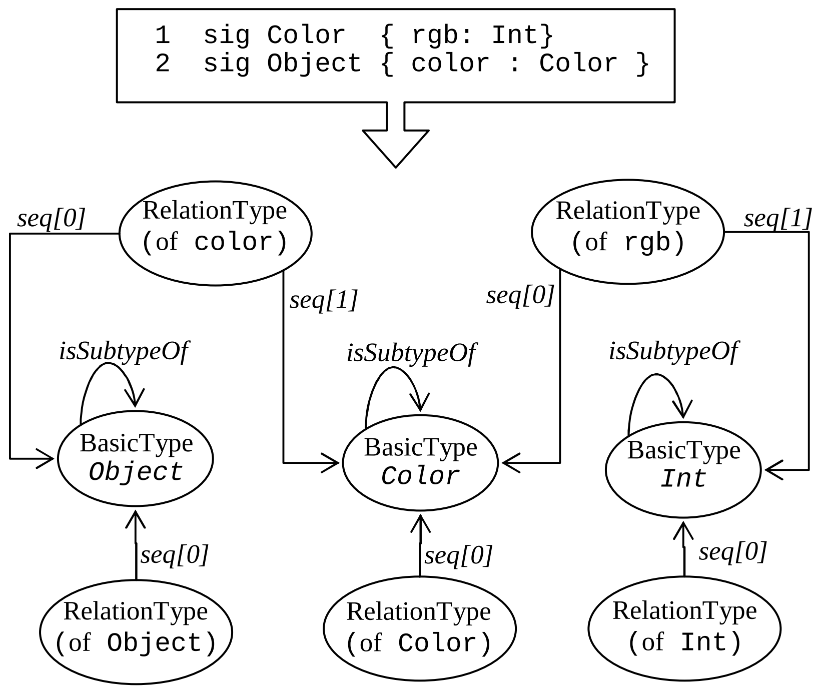
Instances of BasicType can be examined and compared to one another using the operations specified by its interface (Figure 4-1):

- *isEmpty*: tests whether a given BasicType represents the empty set. Consequently, *isEmpty* returns true if and only if it is called on the singleton instance of EmptyType.
- *isSubtypeOf*: determines if two BasicTypes $\mathbf{B}_1$ and $\mathbf{B}_2$ are equal or if $\mathbf{B}_1$ is a descendant of $\mathbf{B}_2$ in the type hierarchy of a given model. For example, in Figure 4-2, *`Dog`* is a subtype of itself and *`Animal`* but not of *`Cat`*. EmptyType is a subtype of all other types.
- *intersect*: returns the intersection of BasicTypes $\mathbf{B}_1$ and $\mathbf{B}_2$. If $\mathbf{B}_1$ is a subtype of $\mathbf{B}_2$, then *intersect* returns $\mathbf{B}_1$, and vice versa. The result of performing *intersect* on two non-intersecting BasicTypes is the EmptyType.

The function *isEmpty* runs in constant time. Un-optimized implementations of *isSubtype* and *intersect* will run, at worst, in time proportional to $d_{\max}$, where $d_{\max}$ is the maximum tree depth of a model's type hierarchy forest. This running time could be reduced to $O(\mathbf{log}\ d_{\max})$ at the cost of increasing the amount of storage space allocated to an instance of SigType. Specifically, we could assign a unique numerical identifier to each SigType such that, for distinct SigTypes s_1 and s_2 in the same type tree, $s_1 < s_2 \Rightarrow s_2$ is subtype of s_1. Each SigType could keep track of its ancestors' numerical identifiers in a sorted array. With this mechanism in place, application of *isSubtypeOf* or *intersect* to two SigTypes would reduce to a binary search of the ancestor array. However, $d_{\max}$ tends to be small in practice (generally, $d_{\max} < 10$), making such an optimization unwarranted.

4.2.2 RelationType

The class RelationType represents the type of an Alloy relation during the translation phase. For instance, binary relations `rgb` and `color` in Figure 4-4 would get RelationTypes [*`Color`*→*`Int`*] and [*`Object`*→*`Color`*]. Since sets are viewed as unary relations in Alloy, the type of a signature after atomization is also a RelationType consisting of the single BasicType introduced by that signature (see Figure 4-4).



```
sig Color  { rgb: Int}
sig Object { color : Color }
```

Figure 4-4 Assigning Types to Alloy Relations

Abstractly, a RelationType is an immutable, non-empty sequence of BasicTypes. The implementation of RelationType maintains the representation invariant that the BasicType function *isEmpty* must return the same value for all BasicTypes in a RelationType. This prevents the creation of nonsensical RelationTypes such as [*EmptyType*→`Object`].

RelationTypes support operations *isEmpty*, *isSubsetOf*, and *intersect* (Figure 4-5), which are analogous to the corresponding operations on BasicTypes with the restriction that two RelationTypes of different *arities* (number of BasicTypes) cannot be intersected. The worst case running times for these operations are constant for *isEmpty* and $O(a*d_{\max})$ for *isSubsetOf* and *intersect* where *a* is the common arity of the functions' operands.

```
package alloy.type;

public class RelationType {

   public boolean isEmpty() { … }

   public boolean isSubsetOf(RelationType other) { … }

   public RelationType intersect(RelationType other)
      throws InvalidAritiesException { … }

   public RelationType transpose() { … }

   public RelationType join(RelationType other)
      throws InvalidAritiesException, InvalidJoinException { … }

   public RelationType product(RelationType other) { … }

   public int arity() { … }
}
```

Figure 4-5. Partial Interface of Class RelationType

Additional operations on RelationTypes are:

- *transpose*: given a RelationType r, *transpose* creates a new RelationType r' such that r' is the reversed sequence of BasicTypes in r (e.g. *transpose*($[A \rightarrow B]$) = $[B \rightarrow A]$)
- *join*: given two RelationTypes r_1 and r_2, *join* creates a new RelationType r which is the relational join of r_1 and r_2 (e.g. *join*($[A \rightarrow B]$, $[B \rightarrow A]$) = $[A \rightarrow A]$)
- *product*: given two RelationTypes r_1 and r_2, *product* creates a new RelationType r such that r is the sequence of BasicTypes in r_2 appended to the end of the sequence of BasicTypes in r_1 (e.g. *product*($[A \rightarrow B]$, $[A]$) = $[A \rightarrow B \rightarrow A]$)

The upper bound on the running time of *product*, *join*, and *transpose*, heavily depends on the underlying representation of RelationType. For example, if the underlying representation of RelationType is a doubly linked list, then *transpose* and *product* will run in constant time, and *join* will run in $O(B_{\&})$, where $B_{\&}$ is the worst case running time of BasicType *intersect*. If, on the other hand, the underlying representation is an arraybacked list, then all three operations will have linear worst case running time: *transpose* will run in time proportional to the length of its argument; *product* will finish in $O(a_2)$ where a_2 is the arity of its second argument; and *join* will terminate in $\mathbf{max}(O(B_\&), O(a_2))$.

4.2.3 UnionType

The class UnionType represents the type of an Alloy expression during the typechecking and semantic transformation phases. Specifically, the typechecking and transformation algorithms would view the AST Node corresponding to the signature `Object` from Figure 4-4 as having the UnionType {[*`Object`*]} rather than the RelationType [*`Object`*] or the BasicType *`Object`*. The relationship between UnionTypes, RelationTypes, and BasicTypes is shown in the object model below (Figure 4-6).

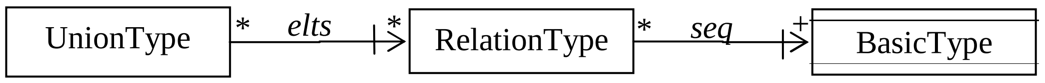


Figure 4-6 UnionTypes, RelationTypes and BasicTypes: Putting it all Together

As evident from Figure 4-6, UnionTypes are immutable sets of RelationTypes. A UnionType with no RelationTypes denotes the result of an illegal operation. RelationTypes within a given UnionType can have differing arities. However, the UnionType of a semantically correct Alloy expression must be uniform in the arity of its elements.

The implementation of the class UnionType ensures that all its instances are *normalized*. A normalized UnionType u has the following properties: (1) u contains no two RelationTypes r_1 and r_2 such that r_1 is a subset of r_2, and (2) u contains no RelationTypes $r_1, r_2, \ldots, r_n$ such that there exists a RelationType r which is semantically equivalent to $r_1 + r_2 + \ldots + r_n$. The normalization invariant is established when an instance of UnionType is created (see 4.3.3 for details), and automatically maintained due to the immutability of UnionTypes.

Since Alloy types approximate actual values of Alloy expressions, their concrete representation must support all the operations applicable to Alloy expressions. Thus, the UnionType interface (Figure 4-7) provides the following operators in addition to *isEmpty*, *isSubsetOf*, *intersect*, *transpose*, *join* and *product* which act analogously to their RelationType counterparts:

- *union*: combines its input UnionTypes u_1 and u_2 to produce a new UnionType u such that the set of atoms described by u is equivalent to the union of the sets of atoms described by u_1 and u_2. For example, *union*({[$A \rightarrow B$] + [$B \rightarrow A$]}, {[A] + [$A \rightarrow B$]}) = {[$A \rightarrow B$] + [$B \rightarrow A$] + [A]}.

- *closure*: computes the transitive closure over all RelationTypes of arity two in the input UnionType. For instance, *closure*({[$A \rightarrow B$] + [$B \rightarrow A$] + [A]}) = {[$A \rightarrow B$] + [$B \rightarrow A$] + [$A \rightarrow A$]}.

- *domainRestrict*: creates a new UnionType u which consists of those RelationTypes from the first argument whose domains intersect the unary RelationTypes from the second argument. So, *domainRestrict*({[$A \rightarrow B$] + [$B \rightarrow A$]}, {[A] + [$A \rightarrow B$]}) would be {[$A \rightarrow B$]}.

- *rangeRestrict*: generates a new UnionType u which consists of the RelationTypes from the first argument whose ranges intersect the unary RelationTypes from the second argument. As an example, applying *rangeRestrict* to {[$A \rightarrow B$] + [$B \rightarrow A$]} and {[A] + [$A \rightarrow B$]} would yield {[$A \rightarrow B$]}.

The upper bound on the running time of *isEmpty* and *transpose* is $O(|u|)$, where u is the input UnionType and $|u|$ is the number of RelationTypes in u. Applying the remaining UnionType operations to inputs u_1 and u_2 will result in the worst case running time of

```
package alloy.type;

public class UnionType {

    public boolean isEmpty() { … }

    public boolean isSubsetOf(UnionType other) { … }

    public UnionType intersect(UnionType ut) { … }

    public UnionType transpose() { … }

    public UnionType join(UnionType other) { … }

    public UnionType product(UnionType other) { … }

    public UnionType union(UnionType other) { … }

    public UnionType closure() { … }

    public UnionType domainRestrict(UnionType other) { … }

    public UnionType rangeRestrict(UnionType other) { … }

    public Set arities() { … }

    public int size() { … }

}
```

Figure 4-7. Partial Interface of Class UnionType

$O(|u_1|*|u_2|*f_x(u_1, u_2))$[1], since these operators could take up to $O(f_x(u_1, u_2))$ time to combine each RelationType in u_1 with each RelationType in u_2. The function f_x is defined as $f_x = \max_{r_1 \in u_1, r_2 \in u_2} C_x(r_1, r_2)$ where C_x is the cost of applying the RelationType operator x to its arguments. For example, the worst case cost of *join*ing u_1 and u_2 would be $O(|u_1| * |u_2| * f_{join}(u_1, u_2))$.

[1] This bound applies to the unary operator *closure* as well since taking the *closure* of a UnionType u is equivalent to computing the *union* of repeated *join*s of u.

4.3 Description of Key Typechecking Algorithms

The typechecking process of an Alloy AST consists of two stages. First, Nodes that represent expressions are assigned potentially overloaded types in a bottom-up fashion by the type assignment Visitor [9]. Then, the overloaded types are resolved in a top-down manner by the type resolution Visitor. Both the assignment and resolution Visitors report errors to the user as they are detected. Normalization of UnionTypes ensures that the types reported in the error messages are in a concise canonical form.

4.3.1 Assignment of Overloaded Types

The type assignment Visitor computes types of AST Nodes that are instances of the *HasType* interface. Concrete implementations of HasType include a representation for each kind of Alloy expression. This section will focus on the Visitor's handling of the most commonly used expressions: binary, unary, variable, and function invocation expressions.

A binary expression is represented by an instance of the concrete class *BinaryExpr*. The Visitor computes the type of a BinaryExpr by performing a corresponding UnionType operation on the types of its subexpressions. For example, if we assume the type hierarchy from Figure 4-4, then the type of the binary expression `Object.color` would be computed as follows: (1) call the Visitor recursively on the left and right subexpressions, and (2) assign to the entire expression the type *join*(type(`Object`), type(`color`)) = *join*({[*`Object`*]}, {[*`Object`*→*`Color`*]}) = {[*`Color`*]}. If the UnionType of a binary expression turns out to contain no RelationTypes (i.e. the UnionType {}), an error is reported to the user if and only if neither subexpression is typed as {}. This ensures that error messages are not propagated up the tree.

A unary expression corresponds to an instance of the class *UnaryExpr* in the AST. The type of a UnaryExpr is the result of applying an appropriate UnionType operator to the type of its subexpression. For instance, the type of the UnaryExpr corresponding to the unary expression ~`color` would be recursively computed as *transpose*(type(`color`)) =

transpose({[`Object`→`Color`]}) = {[`Color`→`Object`]}. The Visitor checks that the UnionType of a UnaryExpr's subexpression contains at least one RelationType of arity 2 (unless its type is {}). For closure operators * and ^, the Visitor also ensures that a UnaryExpr's subexpression is assigned a type with intersecting domain and range.

An instance of *VariableExpr* represents a reference to a locally bound variable or a field. The type assignment Visitor first attempts to obtain the type of a VariableExpr from the local environment. If the lookup fails, the algorithm tries to type the expression as a field reference. The VariableExpr corresponding to a field reference *f* is given the UnionType that is the union of types of all fields named *f*. The typing of field references results in *overloading* if field names are not unique. Finally, if a VariableExpr cannot be typed as either a bound variable or a field reference, it is given the type {} and an error is reported to the user.

InvocationExpr is the AST encoding of a function invocation. The Visitor first computes the types of an InvocationExpr's arguments and then ensures that those types intersect with the types of the formal parameters of the invoked function. The arguments of an InvocationExpr cannot have overloaded types. Hence, their type is determined in two passes, using fresh instances of the type assignment and overloading resolution visitors. The type of the entire expression is taken to be the return type of the invoked function.

4.3.2 Resolution of Overloaded Types

The type resolution Visitors systematically reduces the types computed by the type assignment Visitor to their relevant components. The process of type resolution uncovers two kinds of errors: *redundancy* and *overloading* errors. A redundancy error arises when one or more subexpressions make no contribution to the relevance type of the entire expression. An overloading error occurs when a field reference cannot be uniquely resolved from the context. This section will demonstrate how both types of errors are detected in the process of resolving the types of binary, unary, and variable expressions.

```
void opResolveChildren(UnionType parentType, Expr left, Expr right)
let lt = type(left)  // the unresolved type of left child
let rt = type(right) // the unresolved type of right child
let lt′ = {}
let rt′ = {}
$\forall$leftRT $\in$ lt do
   $\forall$rightRT $\in$ rt do
      let essence = intersect(op(leftRT, rightRT), parentType)
      if essence $\neq$ {} then
         lt′ = lt′ + {opInverseLeft(essence, leftRT, rightRT)}
         rt′ = rt′ + {opInverseRight(essence, leftRT, rightRT)}
set type of left to lt′
set type of right to rt′
```

Figure 4-8 Type Resolution Algorithm for Binary Expressions

For all binary operators except union, the type resolution process follows the same basic template. Since types are resolved by propagating relevance types down the AST, the Visitor assumes that the type of the parent BinaryExpr has already been reduced to its relevant component. Using this information, the Visitor executes the algorithm presented in Figure 4-8 to determine the relevant contribution of the BinaryExpr's subexpressions. The metavariable *op* stands for a binary operator on UnionTypes; the functions *opInverseLeft* and *opInverseRight* are operator-specific procedures for inferring the left and right relevance type components from *essence, leftRT* and *rightRT*. For example, *op* for a product expression would be the UnionType operator *product*. The *opInverseLeft* function would extract the first |*leftElem*| BasicTypes from all RelationTypes in *essence*, form them into new RelationTypes, and sum them together. The *opInverseRight* function would perform the same operation using the last |*rightElem*| BasicTypes from all elements of *essence*.

In the case of a binary union, the resolution procedure is much simpler. Each child's relevant contribution is taken to be the intersection of the parent's relevance type and the child's type as computed by the assignment Visitor. If the intersection is empty and neither the parent nor the child is empty, this indicates a redundancy error. In other words, the child's type had no influence on the parent's relevance type, and the value of the entire expression would be unchanged if the child were removed.

The resolution Visitor includes two procedures for determining the relevance type of a UnaryExpr's child. If a UnaryExpr was created by the use of the transpose operator, the resolved type of the child is simply the transpose of the intersection between the parent's relevance type and the transpose of the child's unresolved type. In the case of the closure operators, the Visitor executes the algorithm in Figure 4-9 to compute the child's resolved type (*childType′* in the figure). The algorithm creates a directed graph *G* whose vertices are the BasicTypes present in *parentType* and *childType*. The graph contains an edge between two vertices *b* and *b′* if they intersect or if there is a RelationType in *parentType* which consists of *b* and *b′*. Hence, a particular RelationType *r* from *childType* contributed to the relevance type of *parentType* if a path in *G* includes *r*. It is possible for a *childType* not to contribute any RelationTypes to the non-empty *parentType* of a UnaryExpr formed by the use of the reflexive transitive closure operator.

```
UnionType closureResolveChild(UnionType $parentType$, Expr $child$)
1 let $childType$ = type($child$)
2 let $G = \langle V, E\rangle$ be a directed graph with a set of vertices $V$ and a
  set of edges $E$
3 $V = \{b$ : BasicType | $\exists b' \in$ BasicType for which $[b \to b'] \in$
  $parentType \vee [b' \to b] \in parentType \vee [b \to b'] \in childType \vee [b' \to b]$
  $\in childType\}$
4 $\forall b, b' \in V$, augment $E$ with an edge from $b$ to $b'$ iff $[b \to b'] \in$
  $parentType \vee (b \neq b' \wedge intersect(b, b') \neq \{\})$
5 let $childType' = \{[b_1 \to b_2] \in childType \mid \exists [b_1' \to b_2'] \in parentType$
  such that there is a path from $b_1'$ to $b_1$ in $G$ and a path from $b_2$
  to $b_2'$ in $G\}$
6 return $childType'$
```

Figure 4-9 Type Resolution Algorithm for Closure Expressions

In such a case, the Visitor reports a redundancy error.

Note that the algorithm in Figure 4-9 does not fully resolve the type of a closure expression's child. For example, assuming the type hierarchy in Figure 4-2, the relevance type of the closure expression in `^(Animal->Cat) & Cat->Cat` is {[*Cat*→*Cat*]}. Hence, the relevance type of `(Animal->Cat)` should be {[*Cat*→*Cat*]} as well. However, the algorithm in Figure 4-9 will resolve the type of `(Animal->Cat)` to {[*Animal*→*Cat*]}. This "incomplete" resolution does not cause problems in practice, but it can be fixed by replacing line 5 in Figure 4-9 with the following code snippet:

```
5-a   let childType′ = {}
5-b   ∀[$b_1$→$b_2$] ∈ childType do
5-c      let leftUnion = {}
5-d      let rightUnion = {}
5-e      ∀[$b_1'$→$b_2'$] ∈ parentType such that there is a path from $b_1'$ to
         $b_1$ in G and a path from $b_2$ to $b_2'$ in G do
5-f         for all paths [$b_1'$, $b_{11}$, $b_{12}$, …, $b_{1n}$, $b_1$] in G do
5-g            leftUnion = union(leftUnion, {[intersect($b_{1n}$, $b_1$)]})
5-h         for all paths [$b_2$, $b_{21}$, $b_{22}$, …, $b_{2n}$, $b_2'$] in G do
5-i            rightUnion = rightUnion + {[intersect($b_{21}$, $b_2$)]}
5-j      childType′ = childType′ + product(leftUnion, rightUnion)
```

The type of a VariableExpr is already resolved by the time the resolution Visitor examines it. But, if a given VariableExpr references a field named *f*, the Visitor still needs to establish that the resolved type of VariableExpr intersects with the type of exactly one field named *f*. If the resolved type intersects with the types of more than one *f*, the Visitor reports an overloading error.

4.3.3 Normalization of UnionTypes

As stated in section 4.2.3, instances of UnionType are normalized upon creation. The pseudo code for the normalization algorithm is shown in Figure 4-10. Note that the algorithm takes a UnionType *u* and a RelationType *r* as arguments and creates a new

```
UnionType makeUnion(UnionType u, RelationType r = [$B_0 \rightarrow B_1 \rightarrow … \rightarrow B_{k-1}$])
1     let u′ = u
2     if u′ does not contain a RelationType of arity k then
3        return u′ = u′ + {r}
4     if $\exists$ r′ $\in$ u′ such that isSubsetOf(r, r′) then
5        return u′
6     u′ = u′ – {r′ $\in$ u′ | isSubsetOf(r′, r)}
7     let rNormal = r
8     for (i = 0; i < k; i++) do
9        if BasicType rNormal[i] has an abstract super type then
10          let b = abstract supertype of rNormal[i]
11          if u′ contains $r_1$, …, $r_n$ such that ($\forall j$ $1 \le j \le n$, $\forall m$ $0 \le m < k$,
            $r_j$[m] is a subset of b when m = i and $r_j$[m] =
            rNormal[m] otherwise) and ($r_1$[i] + … + $r_n$[i] +
            rNormal[i] = b) then
12             rNormal = rNormal[1]$\rightarrow … \rightarrow$rNormal[i-1]$\rightarrow$b$\rightarrow … \rightarrow$rNormal[k]
13             u′ = u′ – {$r_1$,…, $r_n$}
14             goto 9
15    return u′ = u′ + {rNormal}
```

Figure 4-10 Normalization Algorithm for UnionTypes

UnionType *u′* which is a normalized union of RelationTypes in *u* and *r*. It is easy to see how *makeUnion* can be used to incrementally build an instance of UnionType out of a set of RelationTypes.

The algorithm terminates on all well-formed instances of UnionType and RelationType. The outer **for** loop executes a finite number of times, and each iteration is completed in finite time. Specifically, the execution of the inner loop (lines 9 through 13) depends on whether the current BasicType in *r* has an abstract supertype. Since the type hierarchy is finite, the test on line 9 will become false in at most $d_{\max}$ repetitions.

A loose upper bound on the running time of the algorithm is $O(|u| * |r|^2 * d_{\max})$ where $|r|$ is arity of the input RelationType *r*. There will be $|r|$ iterations of the **for** loop. In the worst case, every BasicType in r will be a leaf type in an abstract type hierarchy and u will contain groups of RelationTypes which, when combined with r, will lend themselves to repeated compression into a single RelationType. Hence, each iteration of the **for** loop will result in $O(d_{\text{max}})$ repetitions of lines 9 through 14. Since the cost of executing the containment test on line 11 is $O(|u| * |r|)$, we get the bound of $|r| * O(d_{\text{max}}) * O(|u| * |r|) = O(|u| * |r|^2 * d_{\text{max}})$.

The proof of the algorithm's correctness rests on the observation that lines 2-6 establish the first condition of the UnionType invariant and lines 8-13 establish the second. Lines 4-5 ensure that u' will not contain r if r is a subset of a RelationType already in u. The statement on line 6 rids u' of all RelationTypes that are subsets of r. Lines 8-13 guarantee that $u' + \{rNormal\}$ contains no RelationTypes $r_1, r_2, \ldots, r_n$ such that there exists a RelationType $r \equiv r_1 + r_2 + \ldots + r_n$. This can be proved by induction using the following loop invariant: if the BasicType $rNormal[i]$ has an abstract supertype and u' contains a subset of RelationTypes u'' such that $u'' + rNormal$ is semantically equivalent to a RelationType r' whose i^{th} element is a supertype of $rNormal[i]$ and $r'[j] = rNormal[j]$ for $j \neq i$, then $rNormal = r'$ and $u' = u' - u''$ at the end of the i^{th} iteration of the **for** loop.

5 Related Work

This chapter presents an overview of several type systems that are either theoretically related to Alloy's system (sections 5.1 and 5.2) or were implemented in the context of formal specification languages (sections 5.3 and 5.4). The reviewed systems present unsatisfying solutions to the problem of typing specification languages. For example, soft typing may be unable to determine a unique type for a given expression; complete typing needs special constructs to handle intersection checking; predicate subtypes of PVS are undecidable; and the type system of Z does not support subtypes. Details of these and other drawbacks are discussed and contrasted with Alloy's handling of the same issues in the following sections.

5.1 Soft Typing

Cartwright and Fagan introduced union types in [1] as a means of facilitating *soft typing* of functional languages. The idea behind soft typing is to offer the programmer the advantages of *static typing* while retaining the expressiveness of *dynamic typing*. A soft type system never rejects a program that a static type checker would deem "ill-typed". Instead, it inserts explicit run-time checks around the arguments of potentially erroneous applications, thus making "ill-typed" programs type-correct.

According to Cartwright et al, a soft type system must satisfy two criteria: (1) the *minimal text principle* which states that no program phrases or operations should need to have their types explicitly declared, and (2) the *minimal failure principle* which requires that most program components be unchanged unless they can cause run-time type errors. To satisfy these criteria, a soft typechecker must be able to perform type reconstruction and support parametric polymorphism. In fact, it should also be capable of typing non-uniform expressions such as

```
λx. if x then 1 else nil
```

which cannot be handled by parametric polymorphism alone. (The typing if-then-else as

$\forall\alpha$ `bool`$\rightarrow\alpha\rightarrow\alpha\rightarrow\alpha$ would cause the above function to fail to type check since 1 and `nil` are of different types.)

The authors handle the typing of non-uniform expressions by introducing *union types*. Like an Alloy union type, a soft union type $\alpha \cup \beta$ is the union of disjoint types α and β. A union type is considered to be the supertype (or superset) of its individual components in both Alloy and soft type systems (i.e. $\alpha \subseteq \alpha \cup \beta$). However, unlike in Alloy, the interaction between union types, subtypes, and polymorphism has some unexpected consequences. One such consequence is that a program expression may not have the best, or *principal*, type. For instance, if $f_1 : (\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha$ and $f_2 : a + b \rightarrow \alpha$, then ($f_1$ f_2) could be typed as both $a\rightarrow a$ and $a + b \rightarrow a + b$. We have avoided dealing with similar problems by eliminating parametric polymorphism from Alloy 3.0 (which was present in Alloy 2.0) and replacing it with a simple template mechanism similar to that of C++.

5.2 Complete Typing

Widera noted in [21] that, while sound type checking is too stringent for dynamically typed languages, soft typing is too forgiving in many instances. Specifically, soft typing fails to reject *provable* type errors (i.e. "function calls that provably cannot succeed"), and sound typing rejects type errors that *cannot be proven* to cause run-time failures.

For example, consider the following Scheme programs:

```
// program A                    // program B
(define (square x) (* x x))    (define (double x) (* 2 x))
(define A (square 'a))         (define (get-b y) (if y 'b 5))
                               (define B (double (get-b #f)))
```

Program A is provably wrong, since the function call `(square 'a)` is guaranteed to fail at run-time due to a type error. The program will be rejected by a sound type checker but not by a soft one. Program B, on the other hand, would execute correctly but would be rejected by a static type checker; the type checker would infer that `double` requires its argument to be of type `num` but that `(get-b #f)` in `(double (get-b #f))` has the type (`num` $\cup$ `symbol`) which is not a subtype of `num`. However, (`num` $\cup$ `symbol`) implies that the run-time type of `(get-b #f)` could be either a number or a symbol, so that the type checker *cannot prove* that the call to `double` will fail due to a type error.

The complete typing solution to these problems is to use *intersection checking*. A complete type checker will reject the function application ($f\ a$) if and only if $t \cap s = \varnothing$, where t and s are the inferred types of a and $dom(f)$ respectively. Hence, a complete type checker would reject program A but not B.

Like a complete type system, the Alloy type system also uses intersection checking to check whether an application of a function, predicate, or a built-in operator is meaningful. However, unlike a complete type checker, Alloy's type checker does not need to introduce any special constructs to implement intersection checking, since the types of Alloy leaf expressions are explicit rather than inferred. Specifically, in order for intersection checking to be sound, the type checker must be able to ascertain that $dom(f) \subseteq type(dom(f))$. The type of f is explicitly declared in Alloy, so asserting this fact is trivial. But, in a dynamically typed language, the type checker must infer the type of $dom(f)$, which guarantees that the inferred type is a subset of $dom(f)$ but not vice versa.

5.3 PVS

The Prototype Verification System (PVS) [17] language interprets types as sets of values and subtypes as subsets: type X' is a subtype of type X if the set of values associated with X' is a subset of the set associated with X. In addition to simple types [3], PVS supports dependent types and *predicate subtypes* [18]. A predicate subtype is induced by a predicate over its domain type. For example, the predicate "less than zero" ranging over the integers denotes a subtype of the integers—namely, the negative integers:

```
   lessThanZero?(i : int) : bool = i < 0
   negativeInt : TYPE = (lessThanZero?).
```

The typechecking mechanism for predicate subtypes relies on the general theorem proving capabilities of the PVS theorem prover, since there are circumstances in which the typechecker cannot algorithmically determine the correctness of a specification that uses predicate subtypes. For example, PVS makes no assumptions about the cardinality of the sets associated with its types. However, it requires that the types of uninterpreted constants be non-empty, thus making it impossible for the typechecker to determine whether the following constant declaration is valid:

```
   c: negativeInt.
```

The typechecker deals with this problem by having the theorem prover generate the *proof obligation*

```
   c_TCC1: OBLIGATION ∃(x: negativeInt): TRUE,
```

which requires the theorem prover (or the user) to establish that the type `negativeInt` is non-empty.

The ability to specify subtypes using arbitrary predicates lends great expressive power and functionality to PVS. Predicate subtypes can be effectively used to discover errors in specifications, automate proofs, enforce invariants, and prevent application of partial functions outside of their domain, as demonstrated in [18]. However, the advantages of using predicate subtypes come at the price of a serious loss in tractability, a decrease in usability, and a blurring of the phase distinction between type checking and deeper analysis.

Alloy subtypes are, of course, less expressive than predicate subtypes. Nevertheless, almost all constraints expressible in terms of predicate subtypes can also be efficiently formulated in Alloy as constraints over relations. Additionally, the Alloy Analyzer cleanly separates the typechecking and analysis stages, and all user-tool interaction is driven by the user rather than the tool.

5.4 Z and Object-Z

The formal specification language Z [20] has a simple type system which consists of *basic types* and *composite types*. Basic types are the *given sets* of a specification. Composite types are built out of basic types by (recursive) application of Z type constructors.

There are three kinds of composite types in Z: *set types*, *Cartesian product types*, and *schema types*. The set type $\mathbb{P}\ t$ is assigned to a set of objects that are members of the same type t. For example, the set of integers {-1, 0, 1} would be typed as $\mathbb{P}\ Z$. The Cartesian product type $t_1 \times \ldots \times t_n$ designates the type of the ordered n-tuple $(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are objects of types $t_1, \ldots, t_n$ respectively. The schema type

$$\langle\!| p_1 : t_1; \ldots p_n : t_n |\!\rangle$$

represents the type of a *signature* which declares the distinct identifiers $p_1, \ldots, p_n$ to be of types $t_1, \ldots, t_n$ respectively.

A signature together with a *property* over that signature forms a *schema*, the fundamental unit of specification in Z. Schemas can be used to incrementally specify the *initial state*, *state space*, and *state changes* of a system. Although Z provides no inheritance mechanism for re-use of schemas, it provides a set of logical combination operators for composing several existing schemas into a new schema. For example, we could apply the logical "and" operator to schemas *Sphere* and *Chocolate*

Sphere

$radius : R$

$radius > 0$

Chocolate

$cocoa, cocoaButter : 0..100$

$cocoa + cocoaButter = 100$

to produce the schema $Truffle \triangleq Chocolate \wedge Sphere$, which would be implicitly expanded as follows:

Truffle

$radius : R$
$cocoa, cocoaButter : 0..100$

$radius > 0 \wedge cocoa + cocoaButter = 100$

Z signatures closely resemble record structures in functional languages, and their associated schema types are analogous to record types. However, unlike record types, Z schema types do not support the notion of subtyping. Consequently, a Z typechecker would reject certain function applications which one would intuitively expect to work. For example, it is impossible to write an operation *taste* that accepts an instance of *Chocolate* and apply it to an instance of *Truffle*, even though a *Truffle* is necessarily a *Chocolate*.

Object-Z [6] is an extension of Z developed to facilitate specification in an object-oriented style. As such, it augments Z with the notions of *class*, *class type*, and *class inheritance* [4]. An Object-Z class is a state schema bundled together with all of its associated operations (which would be separate schemas in Z). A class introduces a class type of the same name and provides a template for objects that are instances of the class. Class definitions in an Object-Z specification may be related by (possibly multiple) inheritance.

Object-Z supports two kinds of reference polymorphism: one based on inheritance and the other on the class union construct. The use of inheritance based polymorphism is denoted by the symbol "↓"; declaration "$c : \downarrow C$" says that c refers to an object of class C or any derivative of C [4]. The polymorphism based on the class union construct is strongly reminiscent of Java's interface mechanism. In particular, if $C_1, \ldots, C_k$ are (possibly unrelated) classes such that all of them contain operations (or attributes) named $f_1, \ldots, f_n$ with the same signatures, then $\{C_1, \ldots, C_k\}$ form a new class type whose *polymorphic core* consists of $f_1, \ldots, f_n$ [3]. A reference c declared to be of type $\{C_1, \ldots, C_k\}$ can only be used to access the operations or attributes in the polymorphic core of $\{C_1, \ldots, C_k\}$.

Although Object-Z supports class inheritance and inheritance-based polymorphism, its type system does not have a subtyping mechanism. Hence, the class type of a class A is not a subtype of the class type of B when A is a subclass of B. As a result, dereferencing an object as if it belonged to a subclass may result in an undefined expression since there is no way to statically determine that the dereference is invalid.

6 References

[1] J. P. Bowen, “Formal Methods in Safety-Critical Standards,” In *Proc. Software Engineering Standards Symposium (SESS ’93)*, Birghton, UK: IEEE CS Press, 1993, pp. 168-177.

[2] R. Cartwright and M. Fagan, “Soft typing,” In *Proc. SIGPLAN '91 Conf. Programming Language Design and Implementation*, Toronto, 1991, pp. 278-292.

[3] J. Chen, “Class types as sets of classes in object-oriented formal specification languages,” In *Technology of Object-Oriented Languages and Systems 15*, Melbourne: Prentice Hall, 1994, pp. 205-215.

[4] J. Chen and B. Durnota, “Type checking classes in object-Z to promote quality of specifications” in *Software quality and productivity: theory, practice, education and training*, M. Lee, B. S. Barta and P. Juliff, Eds. London: Chapman and Hall, 1994, pp. 205-215.

[5] A. Church, “A formulation of the simple theory of types,” *Journal of Symbolic Logic*, vol. 5, pp. 56-68, 1940.

[6] R. Duke, G. Rose, and G. Smith, *Object-Z: A Specification Language Advocated for the Description of Standards*, Software Verification Research Centre, The University of Queensland, Queensland, Australia, Tech. Rep. 94-45, Dec. 1994.

[7] J. Edwards, D. Jackson and E. Torlak, “A Type System for Object Models,” , [Online document], Apr. 2004, [cited 2004 May 6], Available HTTP: http://sdg.lcs.mit.edu/pubs/TR/subtypes.pdf.

[8] J. Edwards, D. Jackson, E. Torlak and V. Yeung, "Faster Constraint Solving with Subtypes," *International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, MA, 2004.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

[10] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodolgy (TOSEM)*, vol. 11, no. 2, Apr., pp. 256-290, 2002.

[11] D. Jackson, I. Shlyakhter, and M. Sridharan, “A Micromodularity Mechanism,” In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, Vienna, 2001.

[12] D. Jackson, "Automating First-Order Relational Logic," In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, CA, 2000.

[13] D. Jackson, I. Schechter and I. Shylakhter, "Alcoa: The Alloy Constraint Analyzer," In *Proc. International Conference on Software Engineering*, Limerick, Ireland, 2000.

[14] D. Jackson and J. Wing, "Lightweight Formal Methods," *An Invitation to Formal Methods*, H. Saiedian, ed., *IEEE Computer*, vol. 29, no. 4, Apr., pp. 16-30, 1996.

[15] L. Lamport and L. C. Paulson, "Should Your Specification Language be Typed?," ACM Transactions on Programming Languages and Systems, vol. 21, no. 3, pp. 502-526, 1999.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT solver," 39th Design Automation Conference, Las Vegas, Jun. 2001.

[17] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Language Reference*, [Online document], Dec. 2001 (Rev. 2.4), [cited 2004 Feb 9], Available HTTP: http://pvs.csl.sri.com/doc/pvs-language-reference.pdf

[18] J. Rushby, S. Owre, N. Shankar, "Subtypes for Specifications: Predicate Subtyping in PVS," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, Sep., pp. 709-720, 1998.

[19] I. Shlyakhter, "Generating Effective Symmetry-Breaking Predicates for Search Problems," Workshop on Satisfiability (SAT '01), 2001. *Electronic Notes in Discrete Mathematics*, vol. 9, June 2001.

[20] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed., Prentice Hall, 1992.

[21] M. Widera, "A Sketch of Complete Type Inference for Functional Programming," In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, Kiel, Germany, 2001.

[22] A. Wong and M. Chechik, "Formal Modelling in a Commercial Setting: A Case Study," *arXiv: CS.SE*, vol. 1, June 1999.