Micromodels of Software: Modelling & Analysis with Alloy

Software Design Group

MIT Lab for Computer Science

First draft / November 2001

© Daniel Jackson, 2001

Manual

Daniel Jackson

Case Studies

Language Design

Daniel Jackson
Ilya Shlyakhter
Manu Sridharan

Tool Design

Daniel Jackson
Brian Lin
Ilya Shlyakhter
Manu Sridharan

Tool Implementation

Joe Cohen
Parser

Dan Kokotov
SATLab engine

Brian Lin
Parser and type checker
Visualization

Jesse Pavel
Graphical user interface

Ilya Shlyakhter
Translation to SAT
SATLab engine

Manu Sridharan
Parser and type checker
Static semantic checks
Transformations

Chapter 1: Introduction

1.1 What Is a Micromodel?

A *model* is an analyzable representation of a system. To be useful, it must be simpler than the system itself, but faithful to it; there's not much point analyzing a model if it doesn't tell you something about the system itself. The key to a good model is abstraction: capturing the aspects that matter, and ignoring the rest. Models are built to explain and evaluate existing systems, and to explore the design of systems yet to be built.

Many different languages and approaches for modelling software exist. Why another? Our models have some distinctive features:

- They are *micromodels*: usually orders of magnitude smaller than the systems they model. Ten or twenty lines may be enough to say something useful about a system whose implementation has thousands of lines of code. The modelling language is itself also unusually small. It is easy to learn and easy to use, but powerful and flexible enough for complex applications.
- They are *analyzable*. The language was designed hand-in-hand with a fully automatic tool that can simulate models and check their properties. When a property is found not to hold, the tool generates a counterexample – an instance of a structure of a behaviour illustrating the failure. Simulation generates instances of structures and behaviours, without making you provide sample inputs or test cases.
- They are *declarative*. A declarative model describes a system's state and behaviour by listing properties or constraints. Unlike a program, which is *operational*, it doesn't explain how states are constructed, or how a new state is obtained from an old state during execution. Instead, it gives constraints that define a well-formed state, and that say how a new state should be related to an old one. When you build a program, you ask yourself 'how would I make *X* happen?'. But when you build a declarative model, you ask 'how would I recognize that *X* has happened?'.
- They are *structural*. Software systems, roughly speaking, are complex in two ways. There's complexity due to event sequencing, and complexity in the structure of the state itself – this is the kind we focus on. Sometimes one of these dominates: in a microwave oven or antilock braking system, it's the event sequences you need to worry about, but in a word processor, it's the structure of the state. Complex systems tend to have both kinds of complexity, and as embedded devices get more powerful,

even they tend to have complex states.

None of these features is new in itself. Martin Fowler's 'analysis patterns' [] are small; formal specifications in Z [] are declarative and structural; and model checking languages such as SMV [] and Promela [] are analyzable. What is new is the combination of features, especially of declarative modelling and analyzability.

Researchers used to think that a model can't be executable and declarative at the same time. Some said that models should be 'operational', like programs, and argued the importance of the immediate and concrete feedback you get from execution. Others said they should be declarative, because then they could be more succinct, and express properties more directly. Of course, both of these viewpoints are right: we want succinct declarative models, but we want to be able to run them. With the right technology, this can be done.

A declarative model cannot, admittedly, be executed like an imperative program, so it can't be used directly in an implementation (at least right now). But its execution can be simulated. Viewed as a form of analysis, traditional execution is rather inflexible: you have to provide the initial state, and the model can only be run forwards. Our analysis mechanism can generate before and after states; it can simulate execution backwards; and it can generate sample states from invariants alone.

1.2 Why Micromodels?

You won't really know why micromodelling is useful until you try it.

The basic motivation for all modelling is that it's a good way to end up with a high quality system. I have to admit upfront that 'good' here is a value judgment. If you're not building something that has to be very reliable, and can tolerate bugs, and if you're willing to spend inordinate amounts of time reworking and retesting your code, then modelling might not be good for you.

This is not an entirely frivolous observation. I have noticed, in my teaching, that software engineers are divided, very roughly, into two categories. There are the *thinkers* who get pleasure from getting it right; who like to feel that they've nailed the problem; who want to discuss and solve design problems. And there are *tinkers* who like to build mechanisms: they enjoy the thrill of the first successful run more than the satisfaction that (may) come a bit later that the system really works. Thinkers like to design abstract interfaces; tinkers like to devise low-level hacks. Both are important in a software project, but only thinkers tend to make good modellers. That said, our tool's automation brings to modelling the same concreteness and responsiveness that makes programming fun and compulsive, and we have seen it bring out the thinker in many a tinker.

1.2.1 Why Micro?

It should be obvious why a model must be small. After all, if it's as big as the code itself, you'll double the effort, and that will usually break the budget. There are models from which code can be generated generated, but if you look at them carefully, you'll see that they have two parts: a skeleton, which is the model proper, and the details, which are often written in a programming language anyway.

What's surprising is how much can be gained from a really tiny model. Instead of modelling all aspects of a system, you can focus on the risky ones: those that are most critical, most likely to fail, most expensive to implement, or whatever. By factoring out some aspect, you can build a model that allows you to solve an important problem that would be obscured in a larger and more comprehensive model.

If you've read this far, you probably aren't wondering why one would write a model at all. But the question is worth posing. The popularity of extreme programming [] has been fuelled by the heaviness of modelling languages like UML, and is giving respectability to the idea that well-documented code is enough. But this position is – well, extreme; just because there are bad models doesn't mean that models are bad. Without models, you can't talk about designs precisely until you have code. And even if you have code, there's still good reason to have models. Aspects of a system that can be crystallized in a few lines of model are often buried deep in the code, and dispersed throughout, tangled up with other, less relevant concerns. Code is our greatest asset – after all, it's the stuff that executes – and, at the same time, our greatest liability. If all you have is the code, you should expect to pay heavily whenever a new programmer is brought on board. And good luck to you if all your programmers leave…

Another objection to modelling comes from programming language researchers (such as Bob Harper []) who argue, more convincingly I think, that design representations are enormously important, but only worth maintaining if they are connected to the code, and kept in step with it. This is a laudable goal, but we shouldn't throw the baby out with the bath water. Just because we don't yet have means to guarantee conformance of an implementation to a model doesn't mean that we should abandon the model. The kinds of constraints that can be expressed in our modelling language are far more expressive than those any compiler can currently check. And often the leverage gained in writing a small model is due in large part to an abstraction, in which we take a view of structure far above that taken in the code.

1.2.2 Why Analysis?

Researchers in the field of formal methods have found, time and time again, that the simple act of writing a careful specification usually exposes misunderstandings and complexities, even if you do nothing with the specification afterwards. And critics have

often joked, with some justification, that formal specifications are 'write-only'. Even advocates of formal methods have downplayed the value of tools, and many published specifications have been developed with LaTeX as the only tool support.

Building a model incrementally with an analyzer, simulating and checking as you go along, is a very different experience from using pencil and paper alone. First, it's amazing how many simple mistakes even a quite competent modeller can make, writing down something whose doesn't mean what you though it meant. These tend to be exposed quickly. Second, the feedback you get from simulation is thought-provoking, and helps bring to mind important properties. Third, you can check whether the model has some properties you intended but didn't state explicitly. Even rather mundane consequences can expose subtle problems in a design. And finally, using a tool is fun: successfully checking properties and generating plausible simulations gives you a sense of progress, and confidence in the model.

Analysis tends to make models not only more correct but also more succinct. We often rework parts of a model, and ask the analyzer to check that a reformulation of some part doesn't change its meaning.

1.2.3 Why Declarative?

Declarative languages fit incremental modelling. The more you say, the more constrained the system description, and the fewer behaviours are possible. In contrast, with a programming language, saying more causes more to happen. So if you want to start with as few commitments as possible, a declarative language allows you to start by saying very little. An operational language would require you to list the possibilities.

Also, because a declarative model is built up by conjunction, often all you need to do to say more, and to make your model fit the system more closely, is to add a property. That is, the model can grow steadily. Operational languages make you undo options that you've previously listed.

Partiality – describing behaviour only in part and not completely – isn't a quality only of half-baked models. Most models are partial in their final form. The more partial the description of the system to be implemented, the more freedom is left to the implementer. And the more partial the description of a system's environment, the more confidence we can have that the system doesn't rely on some environmental property that may turn out not to hold.

If the language is a language of properties, like ours, then of course there is no need for a separate property language for expressing checks. The claim that the system as described has some property is just an implication: that the property describing the system implies the property being checked.

Masking exploits the fact that properties look no different when they are part of the

system description. Suppose we check that our system as modelled has some property *P*, and the analyzer finds a counterexample. We might try and fix the problem, but perhaps we want to carry on, and check a property *Q*. If *Q* fails too, we might wonder whether fixing *P* would fix *Q* also. To check this, all we need to do is make *P* a property of the system, masking any errors that arise from *P* not holding.

The term 'declarative' has been used in the functional languages world to mean free of side-effects. Of course, our language is declarative in this sense too.

1.2.4 Why Structural?

Complex states, involving elaborate configurations of simple elements, are all around us. Think of our highway systems, postal routes and telephone switches; of company organizations, family trees and political connections; of shopping lists, library catalogues and address books. Structure is there in the problem domain of most computing systems. And when it's not there, we introduce it, in the form of hierarchy or naming schemes – for example, by filing our mail into folders and by giving nicknames to our friends so we don't need to remember their email addresses.

The problems of dealing with concurrency have kept us occupied so intensively that we have neglected structural complexity. Many problems that seem at first glance to be about events and concurrency turn out, when viewed more broadly, to have important structural aspects. Designing a single traffic light may best be viewed in terms of events and transitions between simple states. But designing a system for a network of roads raises considerations of topology – how the roads may be connected, and how the placement of lights affects global flow.

Structural complexity is also increasing in importance, for two reasons. First, the growth in computational power makes it more feasible to store and manipulate structures, even in small embedded devices. Second, systems are more and more often implemented in a style in which they self-assemble, forming complex structures to fulfill particular functions. Look how many design patterns [] achieve compile-time decoupling by putting objects together at runtime.

1.3 Risk-Driven Modelling

It's dogmatic and unrealistic to insist on building a complete model of your system. Any system has parts with elaborate behaviours that are easy to implement and best left expressed in code alone. A graphical user interface, for example, is often best built by experimenting with an implementation, tweaking it in response to feedback from users. The key to improved design is judicious modelling: focusing your modelling

effort where it's most likely to bring benefit.

How do you decide what to model? You have to identify those aspects of the system in which the greatest risks lie. Some parts of the system may be more crucial than others; in a word processor, for example, a latent fault in the code that manipulates the document is more worrisome than a latent fault in the user interface. Modelling need not respect module boundaries. Sometimes the aspect of the system that warrants attention crosses module boundaries. In an email client, for example, a message may be sent to the wrong address because of a fault in the address book, in the reply mechanism, or in the code that structures messages.

You may pick some aspect because your intuition tells you that you're likely to get it wrong. Other aspects may be no less important, but you're more confident that you can implement them correctly. A fault in the code that organizes messages into folders is probably less likely than a fault in the code that sychronizes your local folders with a remote server.

Skeptics may argue that it's impossible to know in advance everything that might go wrong. And of course software, unlike other engineering artifacts, is prone to failure from unexpected couplings. It would be foolish to conclude from this that you shouldn't try to identify and prioritize risks. For a safety-critical system, it makes sense to construct models even of mundane aspects. But for most systems, there's a diminishing benefit as you move down the list of modelling priorities.

Focusing on risk early on in a development demands a new mindset. The tinker asks 'how do I make that happen?' and dwells on small aspects that will make the program faster or more featureful. The thinker asks 'how might I get it wrong?' and dwells on small aspects that might undo all the work of the project, in failures at runtime, or in design flaws that require massive reworking of the code.

A modelling language must bring clarity to the trickiest aspects of a system. This is why it's important that the definition of the language itself be crystal clear. The last thing the modeller needs is to struggle with obscurities and complications that have nothing to do with the artifact being modelled. For any language that's expressive enough to capture complex properties, there's only one practical way to achieve this: by giving it a mathematical semantics. Some critics of formal methods have mistakenly equated mathematics with complexity. But the purpose of formalization is exactly the opposite. Saying precisely what a language construct means, in terms of basic set theory for example, is so difficult for any but the simplest constructs, that it rules out complexity from the start.

A classic example of this misconception is the view that informal notations are simple and formal specification languages are complicated. If you look behind the surface syntax, and consider notations of comparable expressiveness, you'll discover that the informal notations are usually far more complicated. A popular constraint language

that was touted originally as an improvement on Z has added so many complications – implicit flattenings and coercions, reflection mechanisms, dynamic type casting, a mixture of subtype and parametric polymorphism, and more – that its meaning has become a full-time occupation for a raft of researchers.

A risk-driven approach suggests that a modelling language should be measured by the tricky cases – after all, these are the ones to which it will be applied in practice. The informal notations mostly fail this test. One of the best known object modelling notations, for example, makes it painful to express multiple and dynamic classifications because, in place of simple mathematical classification notions (sets and subsets), it uses implementation concepts, such as subclassing. And yet the same notation cannot easily express tricky aspects of code, such as closures and views.

Chapter 2: Basic Notions

This chapter introduces the mathematical building blocks that underlie Alloy, our modelling language. In comparison to a standard presentation of first-order logic and basic set theory, it's notable more for what's missing than for what's present. Look out for the treatment of scalars and sets, and the definitions of the dot and arrow operators, which are generalized forms of relational join and cross-product.

We'll need a little bit of meta-notation to talk about types and the universe of atoms. To make it clear when our notation isn't Alloy, we'll use mathematical symbols (such as angle brackets) rather than their ASCII counterparts, and we'll set identifiers in all caps.

2.1 Atoms and Relations

The structures in our models will be built from relations and atoms. An atom is a primitive entity that is

- *indivisible*: it can't be broken down into smaller parts;
- *immutable*: its properties don't change over time; and
- *uninterpreted*: it doesn't have any built-in properties, the way numbers do, for example.

Elementary particles aside, very few things in the real world are atomic. But that won't stop us from modelling them as atomic. In fact, our modelling approach has no built-in notion of composites at all. To model a part `a` that consists of parts `b` and `c`, we'll treat `a` as atomic, along with `b` and `c`, and represent the containment by an explicit relation that maps `a` to `b` and to `c`. Containment is just one example of a structural relationship, and there's little reason to single it out for special treatment.

A relation is a structure that relates atoms. Mathematically, it's a set of tuples, each tuple consisting of a sequence of atoms. You can think of a relation as a table, in which each entry is an atom. The order in which the columns appear is important, but the order of the rows is irrelevant. Each row must have an entry in every column. Relations are first-order; entries are always atoms, and never relations.

In any model, we'll partition the universe of atoms into disjoint basic types. Relations are typed, so that in any given column, it will only be permissible to place atoms of a certain type. Suppose we have basic types `S` and `T`, for example. Then a relation that is

declared to have the type ⟨S,T⟩ must have atoms of type S in the first column and atoms of type T in the second column.

A relation can have only column, with a type like ⟨S⟩, and it can have more than two columns, with a type like ⟨S,T,U⟩. But it must have at least one column. Let's call a relation with *k* columns a *k-relation*; *k* is usually called the relation's *arity*. 1-relations, 2-relations and 3-relations are said to be *unary*, *binary* and *ternary*.

A relation can be *empty* – that is, having no rows – or *non-empty*. We will only consider *finite* relations, namely those with a finite number of rows. If a relation has entries of only one type – that is all columns have the same type – it's said to be *homogeneous*, otherwise it's *heterogeneous*.

In Alloy, every expression denotes a relation. So there won't be any sets of atoms; they'll be represented by unary relations. And there won't be any scalars; they'll be represented by singleton unary relations – that is, relations with one column and one row. So, where a conventional language would distinguish a (a scalar), {a} (a singleton set containing a scalar), (a) (a tuple) and {(a)} (a relation), we'll treat them all as the same, and represent them as {(a)}.

This is odd at first, but very handy in practice. It makes the semantics of the language simple and uniform. It side-steps the partial function problem (since without scalars, there's no way to apply a function at all!). And it allow models to be written more succinctly. When using the language, you don't really have to think of sets as unary relations; that's more of a trick to give meaning to the built-in operators without having to overload them. But you will have to bear in mind the representation of scalars as singletons. From now on, when we say 'set' we mean a unary relation, and when we say 'scalar', we mean a singleton unary relation. And when used informally, the term 'relation' will mean a relation that isn't a set – that is, a relation of arity 2 or more.

Example. Given two basic types, PERSON and DATE:

```
type     PERSON
atoms    AKIVA, BECCA, CLAUDIA, DANIEL, EMILY, …

type     DATE
atoms    JAN-1, JAN-2, …, FEB-1, FEB-2, …, DEC-31
```

here's a 1-relation, 2-relation and 3-relation:

```
relation akiva
type     ⟨PERSON⟩
value    {(AKIVA)}

relation parents
type     ⟨PERSON,PERSON⟩
value
```

```
{(AKIVA,CLAUDIA),(AKIVA,DANIEL),(BECCA,CLAUDIA),(BECCA,DANIEL)}
```

```
relation birthdayBook
type     ⟨PERSON,PERSON,DATE⟩
value    {(AKIVA,DANIEL,MAY-7),(AKIVA,BECCA,FEB-11),(DANIEL,CLAU-
DIA,SEP-28)}
```

The relation `akiva` is a unary singleton – that is, a scalar. The relation `parents` is binary, and maps each person to his or her parents. The relation `birthdayBook` is ternary; it relates *p*, *q* and *d*, if date *d* is the recorded birthday of person *q* in the birthday book of person *p*. In this case, `AKIVA` has entries in his birthday book for `DANIEL` and `BECCA`, and `DANIEL` has an entry for `CLAUDIA`.

Don't confuse the name of a relation (such as `akiva`), which can appear in Alloy, from the name of an atom (such as `AKIVA`), which cannot. Of course, in practice, we'll use the variables to refer to the atoms: since `akiva` has the value `{(AKIVA)}`, `becca` has the value `(BECCA)`, and so on, we'll be able to describe relation values without referring to atoms directly. With the operators defined later, we'll see how to specify a particular value of the `parents` relation like this:

```
parents = akiva->claudia + akiva->daniel + becca->claudia + becca-
>daniel
```

or more succinctly, like this:

```
parents = (akiva + becca) ->(claudia + daniel)
```

Binary relations are classified by properties that apply only to them. If a relation has a row containing the entries *a* and *b*, we'll say it 'relates' *a* to *b* or 'maps' *a* to *b*. A homogeneous binary relation is *reflexive* if it relates every atom to itself – that is, every atom that belongs to the type of the columns. It's *symmetric* if whenever it relates *a* to *b*, it also *b* to *a*, and *anti-symmetric* if it can only relate *a* to *b* and *b* to *a* when *a* and *b* are the same atom. It's *transitive* if it relates whenever it relates *a* to *b*, and *b* to *c*, it also relates *a* to *c*. Finally, it's an *equivalence* if it's reflexive, anti-symmetric and transitive.

We'll call the types of a binary relation it's *left* and *right* types, and, correspondingly, the sets of atoms in the first and second columns the left and right sets. We won't use the standard terms 'domain' and 'range', because they clash with other terminology, and because their meaning isn't clear for non-binary relations. For a unary relation, the left and right types are the same; for a tenary relation, the left and right types are the first and third. The left set is not necessarily the set of atoms in the left type. If it is, the relation is *total*; otherwise it's *partial*. If the right set is the set of atoms in the right type, the relation is *onto* or *surjective*.

A *function* is just a binary relation with the property that each atom in the left set is mapped to at most one atom in the right set. An *injection* is a binary relation whose

transpose (mirror image) is a function: that is, it maps at most one atom in the left set to an atom in the right set. Conventionally, an injection must also be a function – what we'd call an injective function. A relation is *total* if it maps every atom in its left type.

Two relations are *equal* if they contain the same rows. That is, two relations p and q are equal if whenever p has some row (a, b, c, …), the same row of atoms appears in q, and vice versa. Two atoms are equal if they are the same atom – that is, not two atoms at all, but one.

Relations are used to model many different structural features:

- Containment. To express containment, you can define a binary relation that relates a to b when a contains b. The relation would be injective for a tree structure, in which no component is shared between containers, and homogeneous for a recursive hierarchy.
- Labelling. To express labelling of entities with attributes, you can use a heterogeneous function from the entity to its attribute. Naming is a common form of labelling; a function of type $\langle X, \mathit{name} \rangle$ attaches names of type *Name* to atoms of type X. The function is total if every entity is named, and injective if names are unique.
- Grouping. To form a collection of atoms into a single group, you can use a set: that is, put the atoms into a one-column table. To form a group of groups, you can use a binary relation that relates two atoms when they are in the same group. If the grouping forms a partition – that is, each atom is in exactly one group – relation will be an equivalence. A labelling function implicitly defines a grouping equivalence: the relation that relates two atoms when they have the same label. This suggests another way to define a grouping: introduce a type of atoms that represent the groups, and a function that maps each atom to the group it belongs to.
- Linking. Often, we want to capture a relationship between peers in which neither plays a superior role. The relation we use is typically less constrained. Topological structure – of a communications network, train track layout, call forwarding database, for example – can be modelled as a homgeneous relation, often unconstrained, or constrained only by a global property, such as lack of cycles, or connectedness. Undirected links are expressed with symmetric relations.

Example. Here are relations illustrating each of the structural features:

```
relation over
type     ⟨WINDOW,WINDOW⟩

relation addr
type     ⟨MACHINE,IP⟩

relation sameAddr
type     ⟨MACHINE,MACHINE⟩
```

```
relation links
type     ⟨URL,URL⟩
```

The relation `over` maps one window in a user interface to another when the first occludes the second. The relation `addr` maps a machine to its IP address, and implicitly defines the relation `sameAddr` that relates machines with the same address. The relation `links` relates two URL's when the document at the first contains a link the second.

Just asking what properties a relation should have is often productive. The choice of properties for `over`, for example, will be fundamental to the design of the user interface's window manager.

2.2 Operators

Expressions in Alloy are constructed from variables and operators. We'll use the standard operators on sets – union, intersection and difference – but every set we apply them to will actually be a relation, since every expression denotes a relation. The union of two relations, for example, is just the union of their tuple sets, and contains any tuple that is in one or both of the relations. We'll write the operators in ASCII as `+` (union), `&` (intersection), and `->` (difference).

The quintessential relational operator is *composition*, or *join*. Let's see how to combine tuples before we combine relations. The *join* of two tuples

$(s_1, \ldots, s_m)$
$(t_1, \ldots, t_n)$

exists if the last atom of the first (s_m) matches the first atom of the second (t_1) If so, it's the tuple that starts with the atoms of the first tuple, and finishes with the atoms of the second, omitting just the matching atom:

$(s_1, \ldots, s_{m-1}, t_2, \ldots, t_n)$

The join `p.q` of relations `p` and `q` is the relation you get by taking every combination of a tuple in `p` and a tuple in `q`, and including their join, if it exists. The relations p and q may have any arity, so long as they aren't both unary (since that would result in a relation with no columns at all!).

This generalized definition of join may look unfamiliar, but in its specific applications, it reduces to well-known operators. For a set `s` and a binary relation `r`, the join `s.r` gives the set of atoms that `r` maps atoms in `s` to – the relational image of `s` under `r`. The reverse application, `r.s`, gives the set of atoms that are mapped by `r` to atoms of `s`. For binary relations `p` and `q`, the join `p.q` is the standard relational composition of `p` and `q`.

The *transpose* `~r` of a relation `r` takes the mirror image of a relation, forming a new relation by reversing the order of each tuple. It has no effect on a relation that is unary or symmetric.

Example. Given the relations:

```
relation akiva
type     ⟨PERSON⟩
value    {(AKIVA)}

relation daniel
type     ⟨PERSON⟩
value    {(DANIEL)}

relation feb11
type     ⟨DATE⟩
value    {(FEB-11)}

relation parents
type     ⟨PERSON,PERSON⟩
value
{(AKIVA,CLAUDIA),(AKIVA,DANIEL),(BECCA,CLAUDIA),(BECCA,DANIEL)}

relation birthdayBook
type     ⟨PERSON,PERSON,DATE⟩
value    {(AKIVA,DANIEL,MAY-7),(AKIVA,BECCA,FEB-11),(DANIEL,CLAU-
DIA,SEP-28)}
```

Here are some expressions and their values:

```
expr    akiva.parents
meaning akiva's parents
value   {(DANIEL),(CLAUDIA)}

expr    akiva.parents.~parents
meaning akiva's parents' children
value   {(AKIVA),(BECCA)}

expr    akiva.birthdayBook
meaning akiva's birthday book
value   {(AKIVA,DANIEL,MAY-7),(AKIVA,BECCA,FEB-11)}

expr    (akiva.birthdayBook).feb11
meaning the person who's birthday is feb-11 in akiva's birthday
book
value   {(BECCA)}
```

The *transitive closure* `^r` of a binary relation `r` is the smallest relation that contains `r`

and is transitive. Closure can be defined in terms of join; it's the limit of the series

```
r + r.r + r.r.r + …
```

The *reflexive transitive closure* `*r` is the smallest relation that contains `r` and is both transitive and reflexive, so it's just like `^r` but includes additionally a mapping from every atom to itself.

The *product* `p->q` of two relations `p` and `q` is the relation you get by taking every combination of a tuple from `p` and a tuple from `q` and concatenating them. In other words, it's like join, but without the matching and dropping of the intermediate atom. For two sets `s` and `t` – that is, unary relations – the product `s->t` is just the standard cartesian product.

Finally, let's introduce operators for comparing relations. The formula `p in q` is true when every tuple of `p` is also a tuple of `q`. In other words, viewed as sets of tuples, `p` is a subset of `q`. Equality is just containment in both directions; `p=q` is true when both `p in q` and `q in p` are true.

Example. These formulas are true if the variables have the values given above:

```
akiva.parents = claudia + daniel
(akiva->daniel) in parents
(akiva->becca->feb11) in birthdayBook
```

These formulas are true under the conventional interpretation of their variable names:

```
parents = mother + father
grandparents = parents.parents
ancestor = ^parents
siblings = parents.~parents - (Person->Person)
```

2.3 Types

So far, we've used types only in passing to show the shape of a relation. Improving readability is one important use of types, but there are more powerful ones.

Alloy's type checker will rule out certain models. The typing rules have two functions. the first is to rule out expressions that have no meaning. For example, you can't form the union of two relations with different arities, and you can't join two unary relations.

The second is more controversial: to catch errors by ruling out expressions that might actually be intended. For example, the rule for join says that *p.q* is well-typed if the right type of *p* matches the left type of *q*. If this isn't true, the join may still be perfectly well defined: so long as both aren't unary, it will be the empty relation. But this is

almost certainly an error, and little is gained by allowing it.

The type rules also prevent us from taking the union of two relations whose column types don't match one-by-one; `links+parents`, for example, would be ruled out because `links` maps `URL` to `URL`, and `parents` maps `PERSON` to `PERSON`. It's not at all obvious why this should be ruled out. But making the type system more expressive, so it rules out fewer plausible expressions, makes it more complicated and harder to understand. Our very simply type system works reasonably well in practice.

Sometimes it's useful to be able to construct values from basic types. The basic types are actually implicit in Alloy, so there are no names for them. The expression

```
univ [e]
```

constructs the universal relation whose type is the type of the expression `e`. For example, `univ[akiva]` gives the set of all persons; `univ[akiva->akiva]` gives the binary relation that relates every person to every person.

Similarly,

```
iden [e]
```

constructs the identity relation for the basic type of the expression `e`, which in this case must be a set, and

```
none [e]
```

constructs the empty relation whose type is the type of the expression `e`.

2.4 Formulas

We've seen how to make a formula from two expressions: using *in* to compare two relations. Larger formulas are made from smaller formulas by combining them with the standard logical operators:

```
!F      // negation: not F
F && G  // conjunction: F and G
F || G  // disjunction: F or G
```

It's useful to have a variety of implication operators:

```
F => G  // implication: F implies G; same as !F || G
F <=> G // biimplication: F when G; same as F =>G && G => F
F => G,H // if F then G else H; same as F => G && !F => H
```

Intuitively, the quantified formula `all x:e | F` is true when `F` holds for every binding of `x` to a scalar drawn from the set denoted by the expression `e`. Since, as we explained above, sets are represented by unary relations, and scalars by singleton sets, each value that `x` is bound to is actually a relation – a relation like `{(a)}` for an atom `a` such that `(a)` is a tuple in `e`. Quantified variables are always bounded by an expression; you can't quantify over a type.

The various quantifiers in Alloy are:

```
all x: e | F  // universal: F is true for every x in e
some x: e | F // existential: F is true for some x in e
no x: e | F   // F is true for no x in e
sole x: e | F // F is true for at most one x in e
one x: e | F  // F is true for exactly one x in e
```

Several variables can be quantified over at once. For example,

```
one x:e, y:f | F
```

means that there is one way to pick `x` and `y` that makes `F` true. The keyword `disj` before the declaration marks the variables as disjoint, so

```
all disjoint x,y: e | F
```

says that `F` holds whenever `x` and `y` are given different values drawn from `e`.

Quantifiers can be applied to expressions too:

```
all e   // e contains every tuple; same as e = univ[e]
some e  // e contains some tuple; e is non-empty
no e    // e contains no tuple; it's empty
sole e  // e contains at most one tuple
one e   // e contains exactly one tuple
```

So far, we've assumed that the bounding expression in a quantified formula denotes a set (is unary, that is). It may actually denote any relation. But if it denotes a non-unary relation, the quantifier is interpreted as second-order, and the value of the quantified variable is not restricted to singletons. Even with a set as the bounding expression, you can make the quantifier second-order by inserting the keyword `set` in the declaration:

```
all x: set e | F  // F is true for every binding of x to a subset
of e
```

Comprehensions construct values directly from properties. The expression `{x: e | F}` denotes the set of atoms `x` from `e` for which `F` holds, or more precisely, it denotes the relation that includes the tuple `(a)` if `F` holds when `x` has the value `{(a)}`. More than

one variable can be declared; `{x: e, y: f | F}`, for example, creates a binary relation.

Example. For relations `spouse` and `parents` that map a person to his or her spouse and parents, and sets `Person` and `Woman` representing the set of all persons and women, the following formulas encode various properties of human relationships:

```
// no polygamy
all p: Person | sole p.spouse

// a married person is his or her spouse's spouse
all p: Person | some p.spouse => p.spouse.spouse = p

// no incest
no p: Person | some (p.spouse.parents & p.parents)

// a person's siblings are those persons with the same parents
all p: Person | p.siblings = {q: Person | q.parents = p.parents} -
p

// everybody has one mother
all p: Person | one p.parents & Woman

// somebody is everybody's ancestor
some x: Person | all p: Person | x in p.*parent
```

Here's an illustration of a higher-order quantifier – a formula that states the associativity of join for binary relations:

```
all p, q, r: A->B | (p.q).r = p.(q.r)
```

A list of formulas given within curly braces is implicitly conjoined. So

```
all x: e {F G H}
```

for example, is equivalent to

```
all x: e | F && G && H
```

Chapter 3: Language

3.1 General Structure

Here's a model inspired by Paul Simon's 1973 song 'One man's ceiling is another man's floor'. It includes all the gross structural elements of an Alloy model:

```
1      -- first Alloy example
1.1    module CeilingsAndFloors
1.2    sig Platform {}
1.3    sig Man {ceiling, floor: Platform}
1.4    fact {all m: Man | some n: Man | Above (n,m)}
1.5    fun Above (m, n: Man) {m.floor = n.ceiling}
1.6    assert BelowToo {all m: Man | some n: Man | Above (m,n)}
1.7    run Above for 2
1.8    check BelowToo for 2
```

Line 1 is a comment. Line 1.1 declares the name of the module. Alloy has a simple module system that allows a model to be composed of text from different files; in this case, the module is self-contained.

Lines 1.2 declares a *signature*: a set, `Platform`, and with it, implicitly, a basic type, `PLATFORM` say. Line 1.3 declares another signature, a set `Man`, also with its basic type, `MAN` say. This signature declaration also introduces two relations, `ceiling` and `floor`, each of type ⟨`MAN`,`PLATFORM`⟩.

Line 1.4 introduces a *fact*: a formula that constrains the values of the sets and relations. Line 1.5 declares a *function*: a parameterizable formula intended to be used elsewhere. Note that the body of the fact uses the function. The *assertion* in line 1.6 specifies a formula that is intended to be valid: in other words, it's a consequence that's supposed to follow from the facts. The last two lines are *commands*: line 1.7 instructs the tool to find an instance of the `Above` function using 2 atoms in each basic type, and line 1.8 instructs it to search for a counterexample of the assertion.

The function `Above` relates a man `m` to a man `n` when `m` is above `n`, by virtue of `m`'s floor being `n`'s ceiling. The fact formalizes the statement 'One man's ceiling is another man's floor', on the assumption that Paul Simon didn't mean that there is exactly one man whose ceiling is another's floor, but rather than every man's ceiling is another man's floor. The assertion claims that the converse holds: that one man's floor is another man's ceiling. In fact, it's not valid, and the tool finds a counterexample.

Let's now delve into some detail to understand the elements of a model.

3.2 Signatures and Fields

A declaration of the form

```
sig S {…}
```

introduces a signature `S`, consisting of a basic type and a set of atoms drawn from that type. The basic type can't be referred to explicitly; types of expressions are inferred in Alloy. So the identifier `S` always refers to the set.

A signature may include declarations of *fields*. Each field declaration introduces a relation whose left type is the signature type. So the field `f` in the signature `S` here

```
sig S {f: T}
sig T {}
```

is a relation from the type of `S` to the type of `T`. The signature creates a local namespace, so that a different field with the same name could be declared in another signature and would represent a different relation.

3.2.1 Implicit Facts

There are some implicit facts in this declaration that limit the possible values of the relation. Declaring `f` in `S` not only makes the type of `S` the left type of `f`, but also constrains the left set of `f` to be a subset of the set `S`. The mention of `T` on the right-hand side of the declaration likewise constrains the right set of `f` to be a subset of `T`. In other words, `f` maps atoms in `S` to atoms in `T` – not very surprising.

A final implicit fact is that `f` is constrained to be a total function: it maps each atom in `S` to exactly one atom in `T`. To weaken this constraint, you can insert a keyword after the colon: `option` to say that each atom of `S` is mapped to at most one atom of `T`, or `set` to eliminate the constraint entirely. These rules may seem funny, but they have an intuitive effect: with the declaration as shown, `s.f` will be a scalar in `T` when `s` is a scalar in `S`, a scalar or empty when `option` is included, and a set when `set` is included.

Using analogy with object-oriented languages, we'll say that the expression `s.f` *dereferences* `s` with the field `f`.

Example. Given this declaration

```
sig Person {spouse: option Person, parents: set Person}
```

the expression `p.spouse` for a scalar `p` in `Person` will be empty or a scalar, and `p.parents` will be any set.

Example. These declarations and fact introduce 2 sets and 3 relations:

```
sig Person {name: Name, pets: set Pet}
sig Pet {name: Name}
fact {all p: Person | no p.name & p.pets.name}
```

In the body of the fact, the first occurrence of the identifier `name` refers to the `name` relation of `Person`, and the second refers to the `name` relation of `Pet`. It says that persons don't share names with their pets.

3.2.2 Resolving Field Overloading

Sometimes there isn't enough context to disambiguate field names, and in that case we write `S$f` to refer to the field `f` of signature `S`. Suppose we want to say that the `spouse` relation is symmetric: that if `p` is the spouse of `q`, then `q` is the spouse of `p`. We can write it like this:

```
fact {all p, q | p = q.spouse => q = p.spouse}
```

or more succinctly like this:

```
fact {Person$spouse = ~Person$spouse}
```

In the second formulation, without the explicit mention of`Person`, it would not be possible to determine which signature the field `spouse` belongs to.

3.2.3 Ternary Fields

So far, our fields have corresponded only to binary relations. The field `f` in this declaration:

```
sig S {f: T - U}
sig T {}
sig U {}
```

is a ternary relation whose column types are the types of `S`, `T` and `U` respectively. Given a scalar `s` from the set `S`, the expression `s.f` will denote a binary relation whose left set is a subset of `T` and whose right set is a subset of `U`. In this case, there is no implicit fact constraining `s.f` to be a singleton.

Perhaps you've noticed that what appears on the righthand side of a field declaration is (so far at least) just an expression, albeit with the possible addition of the keywords

`option` and `set` when the expression denotes a set. You can put just about any expression there; to declare a field that's a 4-relation, for example, you might write

```
sig S {f: T -> U -> V}
```

3.2.4 Multiplicity Markings

When a field maps an atom to a relation, it's often convenient to impose some basic cardinality properties on the relation in the declaration itself. We'll use the *multiplicity markings* ? (zero or one), ! (exactly one) and + (one or more). Imagine that `m` and `n` in the declaration of `f` are replaced by multiplicity markings:

```
sig S {f: T m-> U}
```

These markings say that `s.f` maps `m` atoms in `T` to each atom in `U`, and maps each atom in `T` to `n` atoms in `U`. So to make the relation `s.f` a function, for all scalars `s` in `S`, we'd choose ? for `m`. For a total function, we'd choose !; and for an injection, we'd choose ? for `n`.

Example. The multiplicity markings in this declaration:

```
sig Person {birthdayBook: Person ->?  Date}
```

constrain `p.birthdayBook` to be a partial function: that is, a person's birthday book has at most one entry for each person.

Despite the similarity in appearance between dereferences in Alloy and in Java, remember that the join operator is defined for arbitrary sets (and indeed relations). In particular, if `s` denotes a set of atoms, `s.f` can be thought of as the union of the sets or relations `x.f` for each atom `x` in `s`.

Example. Here are some funky uses of the dot operator:

```
sig Person {birthdayBook: Person ->?  Date}
fact {all p: Person | some q: p.birthdayBook.Date | p in
q.birthdayBook.Date}
fact {all p: Person | p in p.birthdayBook.Date.birthdayBook.Date}
```

Both facts say that same thing: that every person appears in the birthday book of at least one of the persons whose birthday is listed in his or her book. The first says it more conventionally, albeit forming the join of the expression `p.birthdayBook` to extract its left set. The second takes full advantage of the join operator. The expression `p.birthdayBook.Date` denotes the set of persons in the birthday book of `p`; dereferencing this with `birthdayBook` gives the relation that includes the mappings of all those persons' birthday books. Finally, dereferencing with `Date` gives the set of persons with birthday in any of those books.

We'll see more examples of multiplicity markings in Section 3.4.

3.3 Signature Extension

A declaration like

```
sig T extends S {…}
```

introduces a *subsignature* `T` of the signature `S`. Declaring a subsignature doesn't introduce any new types. The new subsignature (here `T`) is just a set, constrained to be a subset of its supersignature (here `S`), and its type is the same as the type of the supersignature.

Subsignatures may themselves have subsignatures:

```
sig S {}
sig T extends S {…}
sig U extends T {…}
```

Taking the supersignature of a signature repeatedly, you end up eventually at a signature without a supersignature – the *root* signature. Here, `S` is the root signature of `T` and `U`. The type of a signature is determined by the type of its root.

Declaring a field in a subsignature is like declaring it in the root signature, with an implicit fact constraining the left set of the field to be a subset of the subsignature. In other words, the field maps only atoms in the subsignature. This declaration of `f`

```
sig X {}
sig S {}
sig T extends S {f: X}
```

constrains `f` to map only atoms in the set `T`, exactly as if we'd recorded a fact

```
fact {all s: T-S | no s.f}
```

or more succinctly

```
fact {S$f.X in T}
```

Note that the explicit name of the field is `S$f` and not `T$f`: fields are disambiguated by type, and only root signatures have distinct types. You can't declare a field with the same name in two subsignatures of the same signature, since there is only one namespace between them, and it would not be possible to resolve the names.

Moreover, subsignatures are not necessarily disjoint. To say that two subsignatures are

mutually disjoint, the keyword `disjoint` is attached to each of their declarations.

Example. To classify persons into disjoint sets of men and women, to give each person a mother and father, and to have a man be his wife's husband and vice versa, we might write:

```
sig Person {mother: Woman, father: Man}
disj sig Man extends Person {wife: option Woman}
disj sig Woman extends Person {husband: option Man}
fact {Person$wife = ~Person$husband}
```

Since a field declared in a subsignature belongs, according to its type, to the root signature, we don't have to worry about applying it only to atoms of the subsignature. The expression `p.wife`, for example, will just denote the empty set when `p` is in the set `Woman`. To say that nobody is married to him or herself, for example, we can write

```
fact {no p: Person | p in p.(wife+husband)}
```

without the kinds of downcast that would be needed were our signatures classes in an object-oriented language and the fields instance variables.

To indicate that a signature contains exactly one atom, mark it as *static*. You can declare several subsignatures at once if they don't declare fields.

Example. To declare a set of colours, including red, green and blue, we could write:

```
sig Colour {}
static disj sig Red, Green, Blue extends Colour {}
```

3.4 Implicit Dereferencing

Expressions involving several dereferences – that is, applications of the join operator – can be hard to read. To make them easier on the eye, there are two additional operators, `[]` and `::`, which mean the same as the dot operator – `q[p]` is equal to `p.q` and `p::q` – but bind differently. Double colon binds tighter than dot, and square brackets bind looser.

Example. A person's birthday book always includes his or her own birthday:

```
sig Date {}
sig Person {birthdayBook: Person ->?  Date}
fact {all p: Person | some p.p::birthdayBook}
```

If `d.next` is the date that follows the date `d` in the calendar, then this says that a person's birthday book only includes persons born later:

```
sig Date {next: Date}
sig Person {birthdayBook: Person ->?  Date}
fact {all p: Person | p.birthdayBook[Person] in
p.birthdayBook[p].*next}
```

Without the new operators, we could write the last fact with parentheses instead:

```
fact {all p: Person | Person.(p.birthdayBook) in
p.(p.birthdayBook).*next}
```

Some formulas repeatedly dereference the same expression. Taking an idea from Pascal, we can make these more succinct by factoring out the expression being dereferenced. The formula

```
with e | F
```

is like `F`, but treats isolated field names as dereferences of `e`. To determine whether a field name is isolated, we first consider only fields that might be appropriate dereferences for `e` (by looking at the type of `e`, and obtaining the fields of its signature). Then we check that the specifier hasn't explicitly indicated that the field isn't isolated, by binding it to an expression with a double colon to its left.

Example. The fact above is about the birthday book of person `p`. In this reformulation, the identifier `birthdayBook` is short for `p.birthdayBook`:

```
sig Date {next: Date}
sig Person {birthdayBook: Person ->?  Date}
fact {all p: Person | with p | birthdayBook[Person] in
birthdayBook[p].*next}
```

Sometimes `with` is handy, but like any implicit mechanism, it can make the text more, not less, inscrutable. You can bind several expressions at once using `with`, and you can nest `with` statements. But you can't bind two expressions with the same type, since there would be no way to choose one over the other. So use `with` judiciously. The most natural usage is when recording properties for the atoms of a signature with many fields.

Example. A file system whose objects are either files or directories, with a root directory and a mapping from each directory to its contents:

```
sig Object {}
sig FileSystem {
 disj files, dirs: set Object,
 root: dirs,
 contains: dirs -> (files + dirs)
 }
```

```
fact {all f: FileSystem | with f | dirs + files in root.*contains}
```

This idiom is so common that we allow the fact to be attached to the signature directly, without the quantifier or the `with`:

```
sig Object {}
sig FileSystem {
 disj files, dirs: set Object,
 root: dirs,
 contains: dirs -> (files + dirs)
 }{dirs + files in root.*contains}
```

The implicitly quantified variable is named `this`, so it can be mentioned explicitly if necessary. Here again, as an attached fact, is the constraint that a person's birthday book lists his or her own birthday:

```
sig Date {}
sig Person {
 birthdayBook: Person ->? Date
 }{some this.birthdayBook}
```

Now that we have `with`, we can explain field declarations more precisely. Declaring a field `f` with an expression `E` like this

```
sig S {…, f: E, …}
```

generates an implicit fact

```
all this: S | with this | f in E
```

To make it easy to infer the type of the field, the expression `E` can only mention the names of signatures, and fields of `S` that are declared before `f`. So the declaration of `contains` in `FileSystem` is short for the fact

```
all this: FileSystem | with this | contains in dirs -> (files +
dirs)
```

which is equivalent to

```
all fs: FileSystem | fs.contains in fs.dirs -> (fs.files +
fs.dirs)
```

3.5 Functions

A *function* is a parameterized formula that can be applied by instantiating the param-

eters with expressions whose types match the declared parameter types.

Example. Here's a model of LISP-like lists, in which a non-empty list has a pointer to the rest of the list (itself a list), and to an element, with a fact ruling out cyclic lists:

```
sig Elt {}
sig List {}
sig NonEmptyList extends List {next: List, elt: Elt}
fact {all p: List | p !in p.^next}
```

We can define two functions that correspond to adding an element to the front of a list (`cons`) and extracting the first element of a list (`car`):

```
fun cons (before, after: List, e: Elt) {
 after.next = before
 after.elt = e
 }
fun car (before: List, e: Elt) {
 e = before.elt
 }
```

and now we can apply the functions in an assertion that claims that if you add an element to a list, then extract an element, you get back the element you started with:

```
assert GetBack {all p,q: List, e,f: Elt | cons (p,q,e) && car
(q,f) = f = e}
```

The uses of the functions `cons` and `car` aren't really 'applications' here; they're predicates relating the inputs and outputs. Alloy lets you write them more naturally as function applications, in which the inputs to a function are presented to arguments, and the application expression as a whole is given the value of the output.

Without changing the declarations of the functions, we can apply them like this:

```
assert GetBack {all p: List, e: Elt | car (cons (p,e)) = e}
```

To understand this, you need to know that there is a convention that the *second* argument in a function declaration is treated as the function's result. The reason for choosing the second (rather than the last, for example) is that often the type of the first argument and the result are the same (as in `cons`), and placing them one after the other allows them to share a type declaration.

The application of `car` poses no complications, since, from the form of the body of the function's declaration, you can see that `car (p)` where `p` is any expression can be replaced by `p.elt`.

The application of `cons` is much trickier. First, its definition is implicit, so an applica-

tion can't be inlined by a simple textual replacement. Second, it's a *non-deterministic* function: it can return any list whose `next` and `elt` fields are set appropriately.

Alloy therefore employs the following rule for handling function applications. Suppose we have a function application expression, `E` say, of the form

$$\texttt{f(}a_1\texttt{, }a_2\texttt{, …, }a_n\texttt{)}$$

whose smallest enclosing formula is some formula `F`. Then the declaration of the function `f` must list n+1 arguments, of which the *second* will be treated as the result. The entire enclosing formula is taken to be short for

$$\texttt{all result: D | f (}a_1\texttt{, result, }a_2\texttt{, …, }a_n\texttt{) => F [result/e]}$$

where `D` is the right-hand side of the declaration of the (second) missing argument, and `F[result/E]` is `F` with the fresh variable `result` substituted for the application expression `E`. The application of `f` in this elaborated formula is now a formula, and is treated simply as an inlining of the formula of `f`.

So in this case, the smallest enclosing formula for the application of `cons` is

```
car (cons (p,e)) = e
```

Desugaring, we get:

```
all result: List | cons(p,result,e) => car (result) = e
```

Note the effect of the quantification: our replacement formula says that taking the `car` of the result of the `cons` application gives `e` for *every* possible result of the `cons`. In other words, a formula involving a non-deterministic function is true when it holds for every possible result the function may yield.

Now we apply the same rule to desugar the application of `car`, and we get:

```
all result: List | cons(p,result,e) =>
  (all result2: Elt | car (result, result2) => result2 = e
```

which, with a bit of rearranging (moving the innermost quantifier out), is the same as the formula that we wrote initially.

This approach may seem a bit complicated, but it has a nice uniformity. Whether the function is explicit (equating the result to an expression) or implicit, whether it's deterministic or not, the treatment is the same. In the simpler cases, one can show that the treatment reduces to what you'd expect. Inlining the body of `car`, for example, we get

```
all result: List | cons(p,result,e) =>
  (all result2: Elt | result2 = result.elt => result2 = e
```

which, using a rule of logic (a variant of the 'one point rule') can be transformed to

```
all result: List | cons(p,result,e) => result.elt = e
```

just as if we'd replaced the application of `car` by the righthand side of the equality in its body.

Our scheme seems to give as intuitive a meaning as one could expect from applying a non-determinstic function, but it does have (at least) one foul outcome: the formula

```
f(x) = f(x)
```

is not generally true for non-deterministic functions. It says that two applications of `f` to `x` must give the same result – but they needn't. So it's wise to be careful when applying non-deterministic functions. Sometimes, a function can be made deterministic. Just adding the fact

```
fact {all p,q: List | p.elt = q.elt && p.next = q.next => p = q}
```

to our list model, for example, has the effect of 'canonicalizing' lists: it says that there is only one list with a given `elt` and `next` field. The function `cons` is now deterministic.

Functions are often grouped around the type of the first argument, like the methods of a class in an object-oriented language. You can declare such a 'receiver' argument, and also a result argument, with the following shorthand:

```
fun S::f (…):T {…}
```

The declaration is equivalent to

```
fun f (this: S, result: T, …) {…}
```

where `this` and `result` are keywords that name the arguments declared anonymously, and the ellipsis in the new argument list is the old argument list. (Actually, this expanded declaration is not legal: you can't declare arguments with these names explicitly.) Either receiver or result can be omitted.

You can also apply a function with its first argument presented in prefix position; for example,

```
s..f (a,b,c)
```

is short for

```
f (s,a,b,c)
```

These conventions are just trivial shorthands, but they can make a model more read-

able. The declaration shorthand is independent of the application shorthand; you can write the first argument as a prefix in an application whether or not it was declared receiver-style in the function's declaration.

Example. Our list functions can be declared like this:

```
fun List::cons (e: Elt): List {
 result.next = this
 result.elt = e
 }
fun List::car (): Elt {
 result = this.elt
 }
```

and applied like this:

```
assert GetBack {all p: List, e: Elt | p..cons(e)..car () = e}
```

3.6 Polymorphism

Some models are independent of the meaning of one or more signatures. These signatures don't have fields and their atoms aren't constrained by facts. They're just placeholders into which any signature could be inserted. When modelling a linked list, for example, we're not concerned about what kind of element is stored in the list; the properties of the list only involve the signature of the atoms that form the backbone of the list. Such models are generic: their properties hold (or fail to hold!) whatever the properties of the irrelevant signatures may be.

No special language support is needed for such a model it it stands alone. Signatures are, after all, uninterpreted. But sometimes we want to build another model from such a model, in which the generic signature is replaced by a signature whose properties do matter.

To support this, Alloy allows the elements of a model to be parameterized by one or more signature placeholders. Uses of these elements can then instantiate the parameters by binding concrete signatures to the placeholders.

Example. Our list model can be recast generically:

```
sig List[t] {}
sig NonEmptyList[t] extends List[t] {next: List[t], elt: t}
fact [t] {all p: List[t] | p !in p.^next}
fun cons[t] (before, after: List[t], e: t) {
 after.next = before
```

```
 after.elt = e
 }
fun car[t] (before: List[t], e: t) {
 e = before.elt
 }
assert GetBack[t] {all p: List[t], e: t | car (cons (p,e)) = e}
```

The identifier `t` that appears throughout the model is a `signature variable` that can be instantiated with a signature. The generic model can be analyzed in its own right, however; when the assertion is checked, for example, the analyzer will just substitute a dummy, concrete signature for the variable.

A generic fact, function, or assertion must be explicitly parameterized by any signature variables mentioned in its body. The variables are like arguments; the name doesn't matter, so long as it's used consistently. We could have declared `car` like this, for example:

```
fun car[s] (before: List[s], e: s) {
 e = before.elt
 }
```

To use a generic model as part of another model, you instantiate the signature variables with signature names. There's no need to instantiate the signature variables when applying functions; this is done automatically based on the types of the arguments.

Example. A diary containing a mapping from dates to lists of appointments, each of which holds its time:

```
sig Time {} sig Date {}
sig Diary {appts: Date -> List[Appt]}
sig Appt {at: Time}
fun Diary::getStart (d: Date): Time {
 result = car(this.appts[d]).at
 }
```

The function `getStart` takes a diary and a date and returns the time of the first apointment on that date. Note that, in its body, the application of `car` doesn't require the signature variable to be explicitly instantiated. The variable is determined to be bound to `Appt` from the type of its first argument.

3.7 Modules

Alloy models are divided into *modules*. You can, if you like, put your entire model in a

single module. All the examples in this chapter will compile if you simply add to the top of the file a module declaration such as:

```
module Main
```

Some models or model fragments are used repeatedly in other models. To make reuse convenient, and to allow structuring of large models, Alloy allows a model to incorporate the contents of other modules. For example, the generic lists described above might be declared in a module `Lists`; many other models could then use lists without having to cut-and-paste the text explicitly.

Alloy's module system is a simplified version of Java's package system. Modules can be arranged in a tree, and are given pathnames from the root. The names of the files containing the modules, and their locations in the directory hierarchy, must match the module names.

Each element in a module – that is, signature, fact, function or assertion – has a *qualified name* obtained by prefixing the element's name with the module name. To refer to an element in the same module, you can use the qualified name, or just the element's declared name. To refer to an element in a different module, you must use the qualified name.

To be able to make reference at all to an element from a different module, you must import the module explicitly at the top of the file. An import declaration with the keyword `use` makes the elements of the module available; an import declaration with the keyword `open` makes the elements available, and additionally, allows them to be referred to by their unqualified names.

Example. The generic list elements might be declared in a module `library/Lists`:

```
module library/Lists
sig List[t] {}
…
```

The diary model might then be in a module `Diary` that imports the module:

```
module Diary
open library/Lists
sig Time {} …
fun Diary::getStart (d: Date): Time {
  result = car(this.appts[d]).at
  }
```

Note that the reference to `car` doesn't need to be qualified because the imported module was opened. With a `uses` import instead, a qualified name would be needed:

```
module Diary
```

```
uses library/Lists
sig Time {} …
fun Diary::getStart (d: Date): Time {
  result = library/Lists/car(this.appts[d]).at
  }
```

3.8 Commands

There are two forms of commands. The `check` command is used to check an assertion; its result is a counterexample, if found. The `run` command is used to find an instance of a function. Both commands take the name of the assertion or function, and an indication of the scope in which the analysis is to be performed. For example,

```
check A for 3
```

would cause the assertion `A` to be checked for all configurations in which each basic type is constrained to have no more than 3 atoms. Individual types can be constrained; for example

```
check A for 3 but 2 S, 5 T
```

says that all types should be constrained to 3 atoms, except for the types associated with signatures `S` and `T`, which are constrained to contain at most 2 and 5 atoms respectively.

3.9 To be added

- Evaluation commands.
- Multiplicity markings in context of implicit dereferencing.
- Argument decl constraints for functions.
- Let expressions. If statements and expressions.
- Explaining decls in polymorphic sigs.
- Explain det fun keyword.
- Instantiation with subsignatures.
- Explain file names for modules. Class path?

Chapter 4: Using the Analyzer

An outline of how to use the analyzer with the graphical user interface is given here. The analyzer is also available in a command-line version, and as a programmatic interface (API) for use as a component in other tools.

4.1 What The Analyzer Does

The analyzer's basic function is very simple. Given a constraint, it attempts to find a solution. If it finds one, it displays the solution as a tree of values assigned to variables, and optionally as a graph of nodes and edges. It can also show the evaluation of each constituent formula and expression of the constraint. The analyzer includes a rudimentary editor.

4.2 Screen Layout

The screen is divided into 4 panes, each of which can be minimized so that only selected panes are showing. They are: the *editor pane*, showing the model being analyzed; the *solution pane*, showing an expandable tree that represents the solution to a command that has been executed; the *syntax pane*, which shows the abstract syntax tree of the constraint that has just been executed, annotated with the values of each node under the assignment reported in the solution pane; and the *console pane*, in which output from the backend SAT solver is reported.

Compilation errors and the diagrammatic visualization appear in pop-up windows. There is a status bar at the bottom that reports progress and indicates the position of the cursor in the buffer.

4.3 Basic Usage Mode

Here's how the tool is typically used:

- Start by loading a model, either by opening an existing file (with File|Open), or by creating a new one (with `File|New`), writing some text, and then saving it (with `File|Save`).

- Compile the file (with `Tools|Build`). If there's a syntax or type error, a window will pop-up with a list of errors. Click on an error to highlight its location in the source.
- Select a command (from the `Command` menu). By default, the last command in a file will be selected, unless you have previously chosen a command, in which case it will be the default.
- Instruct the analyzer to execute the command (with `Tools|Execute`). An indication that a search for a solution is in progress will appear in the status bar. If a solution is found, it will appear in the solution pane. You can then examine the solution by expanding nodes in the solution tree. You can also visualize the solution (with `Tools|Visualize`).

Note that the analyzer works on *files* and not *buffers*: it will refuse to analyze a new model that you haven't yet saved, and if you make changes without saving them, it will remind you to save them if you issue an analysis command, since it will be the file contents and not the buffer that gets analyzed.

4.4 Starting up

To start up the tool, type

```
java -jar alloy.jar
```

Depending on your platform, you may be able instead to double-click on the jar (Java archive) file, or execute a batch file included in the distribution. You can change the look and feel of the graphical user interface by appending to the command line one of the following:

```
-style windows
-style metal
```

4.5 Commands

Many of the commands have keyboard shortcuts, shown in the menus to the right of the command. You can cut and paste into the editor from other applications, but be careful when pasting from an application that does not create plain ASCII text. For example, if you paste from a Word document, you might have trouble with different kinds of linebreaks, smart quotes, etc.

4.5.1 File Menu

New

Clears the editor pane for the creation of a new model.

Open...

Clears the editor pane, and replaces its contents by the contents of the specified file.

Reload

Replaces the contents of the editor pane with the contents of the file on disk. This is designed to make it easy to use a separate text editor. Just open the file in the analyzer, edit it in the separate editor, and to perform an analysis, just reload, build and execute.

Save

Saves the contents of the editor pane as a file on disk. If a name for the file has not been given, you will be prompted for one.

Save as...

Saves the contents of the editor pane as a file on disk with a given name. The name will be remembered, so that subsequent saves will be to the file of that name.

Save a copy as...

Saves the contents of the editor pane as a file on disk with a given name. The name will not be remembered. This allows you to save intermediate versions of your model as you work on it.

Quit

Closes down the analyzer.

4.5.2 Edit Menu

Cut

Deletes the selected text in the editor buffer and copies it into the paste-buffer.

Copy

Copies the selected text into the the paste-buffer, without deleting it from the editor buffer.

Paste

Pastes the contents of the paste-buffer at the point of the cursor.

Delete

Deletes the selected text without affecting the paste-buffer.

Undo

Undoes the last editing command.

Redo

Replays the last editing command whose effect was undone.

4.5.3 Tools Menu

Build

Compiles the file that was last loaded or saved. If errors were found, a pop-up window will list the errors and their location in the source text. Click on an error to highlight its location in the editor pane.

Execute

Executes the command selected in the `Command` menu.

Edit instance

Displays a dialog that allows you to manually enter a candidate solution to the selected command. When you close the dialog, the tool will evaluate the selected command for the assignment of valeus to variables specified. You can then use the syntax pane to examine the values of formulas and expressions.

Visualize

Displays a visualization of the last solution found. The visualization will stay open

unless closed, but will not be automatically refreshed when new solutions are found. The visualization can be customized (see below).

Next solution

Attempts to find an additional solution for the selected command.

Cancel

Cancels the execution of a command that is underway.

4.5.4 Commands Menu

4.5.5 Options Menu

Solve using Chaff

Selects Chaff as the SAT solver to be used for executing commands.

Solve using RelSat

Selects RelSat as the SAT solver to be used for executing commands.

Multiple Solutions

If checked, includes a symmetry-breaking formula in the command so that if multiple solutions are requested, a solution is omitted if isomorphic to one previously shown.

Detect shared subformulas

Enables an optimization. As far as we know, it always improves performance.

Skolemize inside universal quantifiers

Enables a transformation in which existentially quantified variables that are within universal formulas are skolemized. Commands involving such formulas will not be analyzed if this option is not checked.

Remember tree modality

Remembers which nodes in the tree in the solution pane have been opened, so you can focus on some part of the solution, and examine it for repeated executions without

having to renavigate through the tree. Currently, the state of the tree is remembered by simple textual matching.

Change Text Font

Changes the font in the editor pane.

Change Tree Font

Changes the font used in the trees in the solution and syntax panes, and to label nodes and edges in the visualization.

Change Tab Width

Changes the width of tabs in the editor pane. Tabs are not converted to spaces, so this has only a visual effect: the saved file is unaffected.

Save Preferences

Saves the current preferences to a file in the user's home directory under the name `.alloyguirc`.

4.5.6 Help Menu

About...

Displays a splash panel. Help facility under development.

4.6 Solution Tree

The tree in the solution pane gives the values of variables in textual form. At the top-level, the nodes correspond to the signatures, the arguments of a function if the command was a 'run' command, and skolem constants that give witness values to quantified variables. Because quantified variables need not have unique names, the tool generates unique names for skolem constants. Currently the names are not very helpful, but in most cases, you should be able to guess how they correspond to the variables in the formula being analyzed.

Each of these nodes has a value that is a set of atoms; click on the node to expand it, and see the atoms as a subtree. An atom that belongs to a signature with fields can then clicked on to reveal the value of the atom's field. In general, this gives a subtree of

tuples; you can then examine the atoms in these tuples recursively. If a set is empty, no children are shown.

4.7 Syntax Tree

The syntax tree is under development. In its current form, it shows the formula that has been analyzed in the form in which it is represented internally. We plan to make it more readable.

4.8 Visualization

Selecting the visualization command (`Tools|Visualize`) opens the visualization window. The window has two panes. The *graph pane* shows a solution as a graph of nodes and edges. The *customization pane* consists of a collection of subpanes that allow you to customize the appearance of the graph. The customization commands are as follows.

4.8.1 Customize/General

View Unconnected Nodes

If checked, nodes that have no incoming or outgoing arrows will be shown; otherwise they will be hidden.

Save Dot Files

Causes the files generated during visualization to be saved in the format of AT&T's *dot* program, which the analyzer uses for graph layout.

4.8.2 Customization/Type

For each signature that corresponds to a basic type, you can determine:

- whether atoms of this type are shown at all; what shape and colour are used for displaying the atoms as nodes;
- whether the solution is to be projected onto the type;
- what string is to be used as a prefix for the names of atoms of the type; and

- whether when displayed as nodes, the atoms should be given the same rank by dot, so they are displayed at the same level.

Projection is used to enable the display of relations of arity greater than 2. Without projection selected, a field that corresponds to a relation of arity 3 or more will be displayed as a set of nodes corresponding to the tuples of the relation, with edges from the tuple nodes to their constituent atoms (with numbered edges to indicate the order or atoms in the tuple). With a given type projected, a separate graph is generated for each atom in the type. Tuples of higher arity relations are transformed to tuples in which the atoms of that type have been elided. For example, if a relation has the tuples

```
a1, b1, c1
a1, b1, c2
a2, b1, c3
```

then projection of the type `A` with atoms `a1, a2`, etc, will result in two graphs. When `a1` is chosen, the graph will include the edges

```
b1, c1
b1, c2
```

and when `a2` is chosen, the graph will include the edge

```
b1, c3
```

You can project on any number of types. There will then be a graph for each combination of atoms from the projected types.

4.8.3 Customization/Variable

For each set and relation variable, you can determine:

- whether the set or relation is displayed at all;
- what string is used to label the set or relation;
- for a set, whether nodes corresponding to atoms in the set should be given a string label;
- for a relation, whether tuples should be shown not as edges but instead as attribute lists inside nodes;
- the colour of a node of edge belonging to the set or relation;
- for a relation, whether edges are labelled.

4.8.4 Customization Commands

At the bottom of the customization pane, there are buttons for the following commands:

Generate Graph

Generates a graph from the last found solution using the current cutomization.

New Customization

Clears customization settings, resetting them to their default values.

Save Customization

Saves the current customization settings in a file.

Load Customization

Loads customization settings from a file previously saved.

4.9 Files

The analyzer reads and writes the following files:

- *Source files*, in whatever directory the user selects.
- *Visualization customization files*, in whatever directory the user selects.
- *Dot files*, containing solutions in the format of AT&T's *dot* program, in the directory of the associated source file.
- *A preference file* holding user interface preferences, under the name .alloyguirc in the user's home directory.
- *Temporary files* generated during solving, in the system's temporary directory. A subdirectory is created for each session of the tool's use, and is deleted on exit.

Chapter 5: Syntax

5.1 Lexical Issues

Non-alphanumeric characters and whitespace acts as separators. Presence of additional whitespace is irrelevant; linebreaks in particular are equivalent to tabs and spaces. Keywords and identifiers are case sensitive.

Characters between -- or // and the end of the line, and from /* to */, are treated as comments.

Identifiers may include any of the alphabetic characters, and, except as the first character, numbers, underscores and quote marks.

5.2 Grammar

Conventions:

- *x*… is a sequence of zero or more instances of *x*.
- *x*,… is a sequence of one or more comma-separated instances of *x*.
- [*x*] is zero or one instances of *x*.

```
module ::= moduleDecl import… paragraph…
moduleDecl ::= module [packageName /] moduleName
moduleName ::= id
packageName ::= dir…  id
dir ::= id /
import ::= (open | uses) packageSpecifier
packageSpecifier ::= packageName [/*]

paragraph ::= signature | fact | assertion | function | run |
check | eval

signature ::=
  [static] qualifier,… sig signame [typeparams] [extension] {
decl,… }
  [formulaseq]
```

```
typeparams ::= [ typeparam,… ]
extension ::= extends sig,…
fact ::= fact [paraname] [typeparams] formulaseq
assertion ::= assert [paraname]  [typeparams] formulaseq
function ::=
  [det] fun [thisarg] paraname  [typeparams]
  ( [decl,…] ) [: multexpr] formulaseq
thisarg ::= sig . | sig ::

run ::= run  paraname [scope] [excluded] [expect number]
check ::= check [paraname] [scope] [excluded] [expect number]
eval ::= eval paraname using paraname [scope] [excluded]
scope ::= for [number but] typescope,…
typescope ::= number (sig | int)
excluded ::= without global,…
global ::= facts | constraints | paraname

decl ::= var,… compop multexpr | qualifier… var,… : multexpr
qualifier ::= part | disj | exh
multexpr ::= [setmult] expr | expr [mult] -> [mult] expr
setmult ::= set | option | scalar
letdecl ::= var = expr
expr ::= var | sig $ var | unop expr
  | expr binop expr | expr [ expr ]
  | { decl,… [formulabody] }
   | ( expr )
  | invocation | this | result | sig
  | with expr,… | expr
  | let letdecl,… | expr
  | (none | univ | iden) [ expr ]
  | if formula then expr else expr
  | Int intexpr

invocation ::= [expr ..] paraname ( [expr,…] )

intexpr ::= number | # expr | sum expr | int expr
  | ( intexpr )
  | if formula then intexpr else intexpr
  | intexpr intop intexpr
  | sum decl,… | intexpr
  | with expr,… | intexpr
  | let letdecl,… | intexpr
```

```
intop ::= + | -

formulabody ::= formulaseq | | formula
formulaseq ::= { formula… }
formula ::= expr compop expr
  | intexpr intcompop intexpr
  | ( formula )
  | neg formula | formula logicop formula | formula thenOp formula
[elseOp formula]
  | formulaseq
  | quantifier decl,… formulabody
  | with expr,… formulabody
  | let letdecl,… formulabody
  | quantifier expr
  | invocation

thenOp ::= = | implies
elseOp ::= else | ,

neg ::= not | !
logicop ::= && | || | iff | <=> | and | or
quantifier ::= all | no | some | sole | one | two
binop ::= + | - | & | . | :: | ->
unop ::= ~ | * | ^
mult ::= ! | ? | +
compop ::= [neg] ( : | = | in )
intop ::= [neg] ( = | =< | >= | < | > )

signame ::= ([packageName /] id) | Int
sig ::= signame | paramsignature | typeparam
paramsignature ::= signame [ sig,… ]
paraname ::= id
var ::= id
name ::= id
typeparam ::= id
```

5.3 Precedence and Associativity

Precedence order for formula operators (tightest binding first):

```
!, &&, =>, <=>, ||
```

Precedence order for expression operators (tightest binding first):

```
::, ~/*/^, . , [], ->, &, +/-
```

Associativity:

```
-, & to the left
:: and . to the left (essential for type inference)
=> to the right
other operators are associative
```