Software Abstractions

Logic, Language, and Analysis

REVISED EDITION

Daniel Jackson

The MIT Press
Cambridge, Massachusetts
London, England

© 2012 Daniel Jackson

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotion use. For information, please email *special_sales@mitpress.mit.edu* or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Adobe Warnock and ITC Officina Sans, by the author, using Adobe Indesign and his own software, on Apple computers. Diagrams were drawn with OmniGraffle Pro. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Jackson, Daniel, 1963–
Software abstractions : logic, language, and analysis / Daniel Jackson.—Rev. ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-01715-2 (hardcover : alk. paper) 1. Computer software—Development. I. Title.

QA76.76.D47J29 2012 005.1—dc23 2011024317

10 9 8 7 6 5 4 3 2 1

3: Logic

At the core of every modeling language is a *logic* that provides the fundamental concepts. It must be small, simple, and expressive. A "working logic," designed for expressing abstractions, unlike a logic designed for theoretical investigations, cannot be completely minimal, but must be flexible enough to allow the same idea to be expressed in different ways.

This chapter introduces a *relational logic* that combines the quantifiers of first-order logic with the operators of the relational calculus. It's easy to learn—especially if you're familiar with basic set theory, or with relational query languages—and surprisingly powerful.

Although designed for software abstractions, the logic has been kept free of any notions that would tie it to a particular programming language or execution model. Its key characteristic, which distinguishes it from traditional logics, is a generalization of the notion of relational join. As in a relational database, a relation is a set of tuples. Sets are represented as relations with a single column, and scalars as singleton sets. Consequently, the same join operator can be applied to scalars, sets, and relations, and changing the "multiplicity" of a relation (that is, whether it maps an element to a scalar or a set) in its declaration does not require a change to the constraints in which it appears. Dispensing with the distinction between sets and scalars also makes constraints more uniform and easier to write, and eliminates the problem of partial function application, so there's no need for special "undefined" values. There are a few other novelties too, such as the ability to nest multiplicities in declarations.

3.1 Three Logics in One

Our logic supports three different styles, which can be mixed and varied at will. In the *predicate calculus* style, there are only two kinds of expression: relation names, which are used as predicates, and tuples formed from quantified variables.

In this style, the constraint that an address book, represented by a relation *address* from names to addresses, maps each name to at most one address might be written

```
all n: Name, d, d′: Address |
   n->d in address and n->d′ in address implies d = d′
```

In the *navigation expression* style, expressions denote sets, which are formed by "navigating" from quantified variables along relations. In this style, the same constraint becomes

```
all n: Name | lone n.address
```

In the *relational calculus* style, expressions denote relations, and there are no quantifiers at all. Using operators we'll define shortly, the constraint can be written

```
no ~address.address - iden
```

The predicate calculus style is usually too verbose, and the relational calculus is often too cryptic. The most common style is therefore the navigational one, with occasional uses of the other styles when appropriate.

Discussion

Which choice would you actually make for this constraint?

None of these. Multiplicity constraints of this kind are so common that the Alloy logic has some special syntax for them. In this case, you could say that each name is mapped to at most one address by writing

```
address in Name->lone Address
```

Where is the predicate calculus style used?

A common use is in comprehension expressions, which allow you to construct a set or relation from a constraint. For example, if you have a relation *r* that relates three elements from sets *A*, *B* and *C*, and you want the columns instead in the order *B*, *A*, *C*, you can define a new relation by comprehension:

```
r′ = {b: B, a: A, c: C | a->b->c in r}
```

The predicate calculus style can also be appealing when writing a very subtle constraint, because it's so concrete and straightforward, and the quantifications often match a formulation of the constraint in natural language.

Where is the relational calculus style used?

Experienced modelers find it useful for some commonly recurring constraints that can be expressed more concisely this way, writing, for example, *no ^r & iden* to say that the relation *r* is acyclic. Also, you might write a constraint in the navigation style and notice that a quantified variable can be "cancelled out." For example, the constraint

```
all p: Person | p.uncle = p.parent.brother
```

can be written more concisely as

```
uncle = parent.brother
```

(so long as *uncle* and *parent* only map members of the set *Person*).

Do the styles have equivalent expressive power?

No. The navigational style is the most expressive. Predicate calculus lacks transitive closure, so reachability properties can't be expressed. The relational calculus has no quantifiers, and not all occurrences of the quantifiers of predicate calculus can be expressed purely relationally.

Does the style have an impact on the performance of the analysis?

Not in general. Basic modeling decisions about how many relations to use, and how many columns each relation has, have a far bigger impact.

3.2 Atoms and Relations

All structures in our models will be built from *atoms* and *relations*, corresponding to the basic entities and the relationships between them.

3.2.1 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; this is a modeling abstraction. So what do you do if you want to model something that *is* divisible, or mutable, or interpreted? You just introduce relations to capture these properties as additional structure.

3.2.2 Relations

A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. You can think of a relation as a table, in which each entry is an atom. The order of the columns matters, but not the order of the rows. Each row must have an entry in every column.

A relation can have any number of rows, called its *size*. Any size is possible, including zero. The number of columns in a relation is called its *arity*, and must be one or more. Relations with arity one, two, and three are said to be *unary*, *binary*, and *ternary*. A relation with arity of three or more is a *multirelation*.

A unary relation corresponds to a table with one column; it represents a *set* of atoms. A unary relation with only one tuple, corresponding to a table with a single entry, represents a *scalar*.

Example. A set of names, a set of addresses, each of size 3, and a set of address books of size 2:

```
Name = {(N0), (N1), (N2)}
Addr = {(D0), (D1), (D2)}
Book = {(B0), (B1)}
```

Example. Some scalars:

```
myName = {(N0)}
yourName = {(N1)}
myBook = {(B0)}
```

Example. A binary relation from names to addresses, for modeling a world in which there is only one address book (and therefore no need to model address books explicitly), with size 2:

```
address = {(N0, D0), (N1, D1)}
```

Example. A ternary relation (as used in chapter 2) from books to names to addresses, for modeling a world in which there are multiple address books, each with its own name to address mapping:

```
addr = {(B0, N0, D0), (B0, N1, D1), (B1, N1, D2), (B1, N2, D2)}
```

Book `B0` maps name `N0` to address `D0`, and name `N1` to address `D1`; book `B1` maps name `N1` and name `N2` to address `D2`. Fig. 3.1 shows this relation as a table.

Book	*Name*	*Addr*
B0	N0	D0
B0	N1	D1
B1	N1	D2
B1	N2	D2

size = 4

arity = 3

FIG. 3.1 A ternary relation viewed as a table.

A relation with no tuples is *empty*. A unary relation with at most one tuple—that is, a relation that is either a scalar or empty—is called an *option*.

Example. An email application might store the user's email address, and, optionally, a distinct address used for the "reply-to" field of messages. The former might be modelled as a scalar *userAddress*, and the latter as an option *replyAddress*, which either contains an address or is empty.

In the Alloy logic, all values are relations, so a tuple will be represented by the relation containing it, in the same way that a scalar is represented by a singleton set. We'll therefore use the term *tuple* to describe a singleton relation—a relation containing exactly one tuple.

Example. Two scalars, and the tuple that associates them:

```
myName = {(N0)}
myAddress = {(A1)}
myLink = {(N0, A1)}
```

3.2.3 Expressing Structure with Relations

With relations, you can express structures in space and time, overcoming the apparent limitations of atoms as a modeling construct.

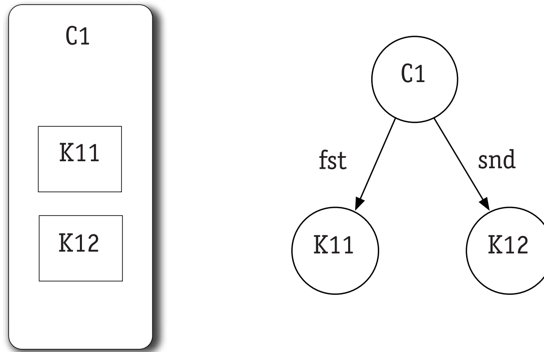

FIG. 3.2 A key card containing two keys (left) represented with atoms and relations (right).

Although the only objects in the logic are indivisible atoms, you can model a composite object with atoms for the components and a relation to bind them together.

Example. To say that directories can contain files, you could introduce a relation *contents* that maps directories to the files they contain, which would include the tuples *(D0, F0)* and *(D0, F1)* when directory *D0* contains the files *F0* and *F1*.

Example. Hotel key cards, each holding two cryptographic keys, can be modeled as a set *Card* of cards, a set *Key* of keys, and two relations *fst* and *snd* from *Card* to *Key*. If a card *C1* has *K11* and *K12* as its first and second keys respectively, and a card *C2* has *K21* and *K22*, the relations would have these values:

```
fst = {(C1, K11), (C2, K21)}
snd = {(C1, K12), (C2, K22)}
```

This is illustrated, for card *C1*, in fig. 3.2.

When the content of an object is itself a relation, a multirelation is used to model containment.

Example. The relation *addr* mentioned above (and shown in fig. 3.1) associates address books, names and addresses. Each address book can be viewed as containing a name/address table.

Although atoms are immutable, you can model mutation, in which the value of an object changes over time, by separating the identity of the

object and its value into separate atoms, and relating identities, values and times.

Example. The history of values of some stocks might be represented by a relation *value* that includes the tuple *(S0, V0, T0)* if stock *S0* has value *V0* at time *T0*, and *(S0, V1, T1)* if it has value *V1* at time *T1*.

Example. An address book's changing contents could be modeled with a relation *addrT* on names, addresses and times, with a value such as

```
addrT = {(N0, D0, T0), (N0, D1, T0), (N2, D2, T1)}
```

if the book maps name *N0* to addresses *D0* and *D1* at time *T0*, and *N2* to *D2* at time *T1*.

When a model concerns only a single object and its changing value, a set of atoms can be used for that object to represent its value at different times.

Example. The *addr* relation of chapter 2 associated books, names and addresses. It could be used to model a static world in which there are several address books, each containing its own name/address table. In fact, however, it was used to represent the changing value of a single address book, with the book atoms playing the same role as the time atoms of *addrT*.

Finally, although atoms are uninterpreted, you can give them properties by introducing relations between them.

Example. The sequence numbers in a network protocol might be represented by atoms of a set *SeqNumber = {(N0), (N1), …}*, and ordered by a relation *precedes*, which contains the tuple *(N0, N1)* when sequence number *N0* comes before sequence number *N1*.

Example. The book atoms in the model of section 2.4 were ordered *Book0, Book1, …* with *Book0* representing the value of the book initially, *Book1* the value after one step, and so on. The ordering was imposed by importing a library module that includes a relation *next* mapping *Book0* to *Book1*, *Book1* to *Book2*, etc.

Example. Image-editing programs such as Adobe Photoshop allow you to apply color transformations to images. To explore the particular properties of one transformation, one would need a detailed model of colors and transformation functions. But to explore the abstractions underlying such a scheme, application of transformations to partial image selections, combining transfor-

mations with layers, undoing and redoing transformations, and so on, it may be sufficient to take a more abstract view, in which an image is just a mapping from pixel locations to RGB values, and a color transformation is a function from RGB values to RGB values. A relation

```
transform = {(RGB0, RGB1), (RGB1, RGB0)}
```

might model the transformation that exchanges the RGB values *RGB0* and *RGB1*.

Discussion

Are the names of the atoms significant?

No. Atom names never appear in models; they're only used to describe instances produced by simulation or checking. The Alloy Analyzer lets you assign your own names to the atoms of each set, but by default uses the full name of the set. So the atoms of *Book* will be *Book0*, *Book1*, and so on, rather than *B0*, *B1*, and so on.

What does an expression such as `{(N0, D0), (N1, D1)}` *mean?*

It's an expression in the language of traditional mathematics. In this case, it denotes the set consisting of two tuples, the first tuple having *N0* as its first element and *D0* as its second element, and the second tuple having *N1* as its first element and *D1* as its second element. The terms *N0*, *N1*, *D0*, and *D1* are names for atoms. None of this belongs to the logic itself; I'm using it just to explain the meaning of the fundamental notions. In Alloy, you can't refer to atoms explicitly at all. You could, however, declare scalar variables *N0*, *N1*, *D0*, and *D1*, and then, as we shall see, this relation could be denoted by the expression *N0->D0 + N1->D1*.

Why the extra parentheses in a set expression such as `{(N0)}`*?*

In Alloy, all structures are relations, and a set is simply a relation all of whose tuples contain only one element. The set *{N0}* would be modeled as this relation in Alloy; it cannot be represented directly. Because this kind of expression never appears in a model, the extra syntax of the parentheses is not inconvenient. In fact, on the contrary, the unification of sets and relations makes the syntax simpler, since there is no need to convert between sets and relations, or between scalars and sets.

Can relations contain relations?

No. Our relations are flat, or *first order*, meaning that entries are always atoms, and never themselves relations. Take the relation *addr*, from the example of section 3.2.2, which we used to model the idea that address books contain name/address mappings. In our flat representation, the relation's value was a ternary relation associating books, names and addresses:

```
addr = {(B0, N0, D0), (B0, N1, D1), (B1, N1, D2), (B1, N2, D2)}
```

More conventionally, this might be represented as a function from address books to a function from names to sets of addresses:

```
addrC = {
   (B0, {(N0, {D0}), (N1, {D1})}),
   (B1, {(N1, {D2}), (N2, {D2})})
   }
```

The relation *addrC* is not directly representable in Alloy. We'll see later in this chapter (in subsection 3.4.3) that the name/address mapping for book *b*, which would conventionally be written *addrC(b)*, can be written *b.addr* in Alloy.

Why not admit higher-order relations?

The restriction to flat relations makes the logic more tractable for analysis. Flat relations, as the relational database community has discovered, are expressive enough for almost all applications, and their simplicity and uniformity is appealing. The lack of symmetry in *addrC* (above), for example, means that it cannot be accessed from the right as easily as from the left. The expression *addr.a* denotes the mapping from address books to the names they use for address *a*, and *addr.Addr.n* denotes the set of address books that have an entry for name *n*; for *addrC*, both would require a more complex construction.

Is there a loss of expressive power in the restriction to flat relations?

Yes, there is, but it can usually be worked around. Almost always, a situation that seems to call for a higher-order relation can be reformulated without one. Suppose we're modeling the prerequisite structure of a university course catalog, in which each course has a set of prerequisites, and, for admission to a course, a student is required to have taken all the courses in at least one of the course's prerequisites. In a higher-order setting, this structure could be represented as a relation from courses to sets of courses. For example, the relation

```
prereqC = {(C3, {(C0), (C1)}), (C3, {(C0), (C2)})}
```

would indicate that a student wanting to take course *C3* must have taken either *C0* and *C1*, or *C0* and *C2*. Simply flattening this relation to

```
prereqBad = {(C3, C0), (C3, C1), (C3, C2)}
```

won't work, because it loses the grouping of the prerequisites. The solution is to introduce a new set of atoms to model prerequisites, along with a relation mapping prerequisites to their constituent courses:

```
prereq = {(C3, P0), (C3, P1)}
courses = {(P0, C0), (P0, C1), (P1, C0), (P1, C2)}
```

This retains the essential structure: course *C3* now has two possible prerequisites, *P0*, consisting of course *C0* and *C1*, and *P1*, consisting of course *C0* and *C2*.

There is another respect, by the way, in which a higher-order relation is more expressive than a flat relation. A function that maps atoms drawn from a set *A* to sets of atoms drawn from a set *B* can include a mapping from an atom to the empty set, thus distinguishing an atom being mapped to nothing and an atom not being mapped at all. A binary relation from atoms in *A* to atoms in *B* cannot make such a distinction. Instead, you'd declare an additional set: the address books with empty mappings, for example, would belong to the set *Book* but would not be mapped by *addr*.

Why not include composite objects as a language construct?

Traditional specification languages such as VDM and Z allow you to model composite objects directly with composite mathematical objects. For example, an address book might be represented not as an atom, but as a relation from names to addresses. A relationship between an address book and another object would then be expressible only with a higher-order relation. The reasons for excluding composite objects in their own right are thus the reasons we've already given for preferring flat relations. Also, mathematical objects have no identity distinct from their value; if you want to talk about the values of different address books at different times, you need to introduce atoms representing the identities of the books anyway.

Can you really work without interpreted atoms such as the integers?

Yes, almost all the time. And it turns out that on most occasions that you might think you need integers, it's cleaner and more abstract to use

atoms of an uninterpreted type with some relations to give whatever interpretation is needed. Alloy does actually support integers, albeit in a restricted way (due to the limitation of finite bounds). The treatment of integers is explained in sections 3.7 and 4.8.

Can relations have infinite size and arity?

Nothing in our logic precludes relations of infinite size, but for all the models we'll look at, it's sufficient to consider only finite instantiations. A relation must have a finite arity, though.

Are multirelations useful in practice?

Yes, because relations are flat rather than nested, arities greater than two are very common. To model execution traces of a system whose state involves relationships will require ternary relations: two columns for the relationship at a given time, and an additional column for the time. Relations of arity four are less common; as an example, the states of a network routing table might relate the host at which the table resides (1), a second host that is the desired destination of an incoming packet (2), the port to be used in forwarding the message (3), and the time at which this table entry is present (4). Arities of five or greater are rare.

Why don't the columns in a relation have names?

If you're more familiar with relational databases than relational logic, you may find it odd that the columns of a relation are identified by their position rather than by name. In modeling, relations tend to have much smaller arities than relations in a database; it's rare for a relation to have more than four columns. Moreover, in the constraints of a model, joins tend to be applied to a relation on particular columns, in a particular order. By arranging the columns carefully, almost all joins can be made to be on the first or last column of a relation. Consequently, treating columns positionally rather than by name is more convenient, and results in more succinct and natural expressions.

If the order of columns matters, how do you represent an unordered relationship?

An unordered relationship can be represented in different ways. The simplest way is to use a relation *r* (ordered, as always), and add a constraint *r* = ~*r* that makes it symmetric—the same forward and backward. For example, *spouse* = ~*spouse* says that if you're my spouse, I'm your

spouse. This trick may be philosophically dubious, but in practice it's fine, and much easier than introducing additional constructs.

Is the idea of treating scalars and sets as relations new?

No. It goes back to Tarski's foundational work on the relational calculus [72]. All of Tarski's relations were binary, however, so his encoding was a bit less natural: a set was a relation that mapped each atom in the set to every possible atom. Rick Hehner's "bunches" [29] have a similar flavor, but unify scalars and sets in a new kind of algebraic structure.

Isn't it confusing to treat scalars as sets?

On first encountering this idea, some people are disturbed. After all, isn't the distinction between a set and its elements the very foundation of set theory? In a first-order logic, however, in which sets of sets are never used, no confusion arises. And in practice, breaking down the distinction between sets and scalars brings a nice uniformity. When writing a navigation expression, you don't have to worry about whether an expression represents a set or a scalar. The grandfathers of person *p*, for example, can be written *p.parents.father*, in which the dot operator is applied to a scalar such as *p* in exactly the same way it is applied to a set such as *p.parents*.

Combined with the treatment of partiality, this allows us to write *p*'s mother-in-law as *p.wife.mother + p.husband.mother* (or equivalently as *p.(wife + husband).mother*), without worrying that if *p* has no wife the expression *p.wife* may be undefined.

Which terms are Alloy specific, and which are standard in logic and set theory?

All the terms introduced so far are standard, with the exception of *multirelation* (for a relation with more than two columns) and *option* (for a set that is empty or singleton).

Is Alloy's option like the option of the ML programming language?

Rather than treating options as singleton or empty sets, most modeling and programming languages use a union type. ML's option is such a union: a tagged value that is either a scalar or some special null value. For modeling, this is less convenient, because the tagging wraps the value and changes its type. Consider, for example, a model of an email application with a scalar *userAddress* representing the user's address, and an option *replyAddress* representing a separate address to be used in the

"reply-to" field of messages. In Alloy, these variables have the same type, and can be combined and compared with set operators;

```
userAddress = replyAddress
```

for example, is true if *replyAddress* is defined and equal to *userAddress*. In the traditional approach, the two variables have distinct types, and cannot be compared without projecting *replyAddress* first.

So there aren't really any scalars in Alloy?

Not in the standard sense. Whereas a conventional language would distinguish *a* (a scalar), *{a}* (a singleton set containing a scalar), *(a)* (a tuple), and *{(a)}* (a relation), Alloy treats them all as the same, and represents them as *{(a)}*.

Why is the term "option" useful? Isn't an option either a scalar or empty?

The term is used to describe a *variable* whose value is unknown, rather than a particular value, in the same way that you might refer to a "vehicle" without knowing whether it's a car or a truck. By definition, every scalar is also an option; both are sets; and every set is a relation. Typically you want to use the term that tells you most about a relation, so you don't call it a "relation" if you know it's a set, or a "set" if you know it's a scalar.

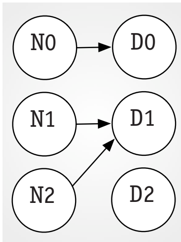

FIG. 3.3 Functional but not injective.

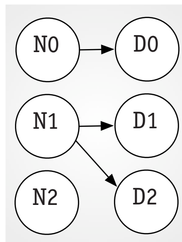

FIG. 3.4 Injective but not functional.

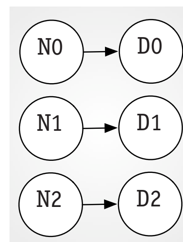

FIG. 3.5 Functional and injective.

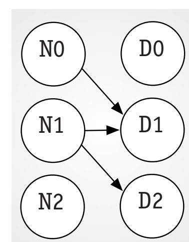

FIG. 3.6 Neither functional nor injective.

3.2.4 Functional and Injective Relations

A binary relation that maps each atom to at most one other atom is said to be *functional*, and is called a *function*. A binary relation that maps at most one atom to each atom is *injective*.

Example. Here are four possible values of a relation mapping names to addresses, illustrated in figs. 3.3–3.6:

```
address1 = {(N0, D0), (N1, D1), (N2, D1)}
address2 = {(N0, D0), (N1, D1), (N1, D2)}
address3 = {(N0, D0), (N1, D1), (N2, D2)}
address4 = {(N0, D1), (N1, D1), (N1, D2)}
```

The first is functional but not injective; the second is injective but not functional; the third is both functional and injective; and the fourth is neither. An empty binary relation is functional and injective.

Discussion

Where does the idea of treating functions as relations come from?

The idea of treating functions as relations has been pioneered in modeling by the specification language Z [67]. Its use goes back at least to Zermelo and Fraenkel's set theory (hence the "Z" in Z). Alloy is actually more minimalist than Z. Although Z doesn't distinguish functions and relations, it does distinguish scalars, sets, and tuples from each other. In Alloy, everything's a relation.

Is it standard to treat functions as relations?

No. Most other modeling languages distinguish functions from other relations. In UML's constraint language OCL [57], for example, navigating through an association can either produce an empty set or an undefined value, depending on the multiplicity of the association.

Is an injective relation an injection?

The term "injection" is traditionally applied only to a relation that is both functional and injective, so I try to avoid using it. Unfortunately, there isn't a common name for an injective relation.

3.2.5 Domain and Range

The *domain* of a relation is the set of atoms in its first column; the *range* is the set in the last column.

Example. A relation with its domain and range:

```
address = {(N0, D0), (N1, D1), (N2, D1)}
domain (address) = {(N0), (N1), (N2)}
range (address) = {(D0), (D1)}
```

A relation of higher arity has a domain and range too.

Example.

```
addr = {(B0, N0, D0), (B0, N1, D1), (B1, N2, D2)}
domain (addr) = {(B0), (B1)}
range (addr) = {(D0), (D1), (D2)}
```

Discussion

Are domain and range special operators?

No, but they are predefined for binary relations as functions in the Alloy library. They're easily expressed with the other operators (introduced later): the domain and range of a binary relation *r* are *r.univ* and *univ.r* respectively.

Are the domain and range functions commonly used?

They are used less frequently than in languages such as Z, because of Alloy's rich declaration syntax (see section 3.6), which encourages you to introduce sets that explicitly represent a relation's domain and range.

What about total and partial functions?

The term "domain" is often used to refer to the set of atoms that *might* be mapped by a relation or function. In that case, a total function is one that maps every member of its domain. This notion requires that a set be associated implicitly with a relation. (Alternatively, a total function relation might be one that maps every atom in the universe, but this is a very rare case.) Our logic is simpler than this: the relation is just its tuples, and the domain and range of the relation are determined by this set of tuples. I do occasionally use the terms "total" and "partial" informally, referring to whether a relation is total or partial over the set that appears in its declaration.

3.3 Snapshots

Particular values of sets and binary relations can be shown graphically in a *snapshot*. You create a node for each atom, and draw an arc for each tuple connecting the nodes corresponding to the first and second atoms in the tuple. To show several relations, you label each tuple arc with the relation it belongs to. Sets can be shown in two ways: either by an extra label in a node naming a set it belongs to, or by drawing a labelled contour around some nodes.

> *Example*. A multilevel address book modeled by a relation *address* mapping names to targets, where targets are names or addresses, and names are aliases or groups, might be represented textually by
>
> ```
> address = {(G0, A0), (G0, A1), (A0, D0), (A1, D1)}
> Target = {(G0), (A0), (A1), (D0), (D1), (D2)}
> Name = {(G0), (A0), (A1)}
> Alias = {(A0), (A1)}
> Group = {(G0)}
> Addr = {(D0), (D1), (D2)}
> ```
>
> or graphically by the snapshot of fig. 3.7.

Multirelations can be shown as graphs by *projecting* out one or more columns. Projection takes two steps. Suppose just one column is being projected out. In the first step, the column is moved to the front, so that it becomes the first column of the relation; each tuple is permuted accordingly. In the second step, the relation is split into an indexed collection of relations. For each atom that appears in the first column, we associate the relation consisting of all those tuples that begin with that atom, but with the atom removed. For an atom a and relation r, this new

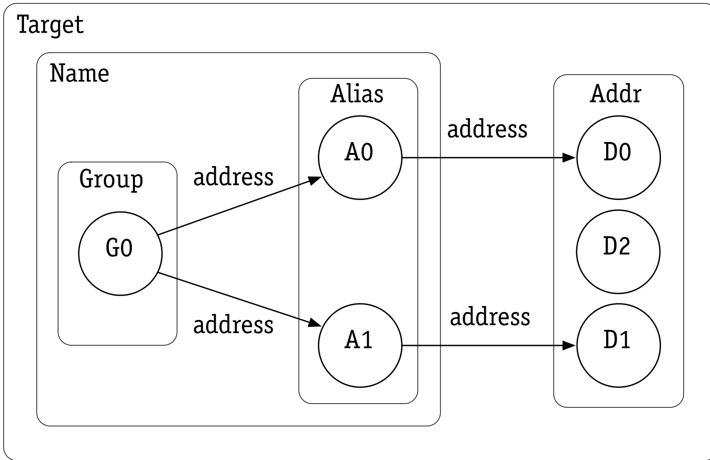

FIG. 3.7 A sample snapshot.

relation is given by the expression *a.r* (using the join operator defined in subsection 3.4.3).

Example. A world of several multilevel address books modeled by the relation *addr* mapping books to names to targets, where targets are names or addresses, and names are aliases or groups, might be represented textually by

```
addr = {(B0, G0, A0), (B0, G0, A1), (B0, A0, D0), (B0, A1, D1),
   (B1, A0, D1)}
Book = {(B0), (B1)}
```

(and with appropriate assignments to the other sets as in the previous example). Its projection onto the first column gives

```
B0.addr = {(G0, A0), (G0, A1), (A0, D0), (A1, D1)}
B1.addr = {(A0, D1)}
```

which could be shown visually as two graphs, the one for *B0* being that of fig. 3.7 (but with *addr* for *address*).

Examples. All of the diagrams generated by the Alloy Analyzer in chapter 2 are snapshots. The analyzer lets you customize how instances are displayed; you can select a set and project all relations in the instance onto the columns associated with that set. In this case, a projection using the set *Book* was chosen. Under projection, a binary relation becomes a set; this is why, for example, the rela-

tion *names* from books to the names they map appears as a label in fig. 2.13. The analyzer can show sets only by labeling nodes; it can't currently draw contours.

3.4 Operators

The language of arithmetic consists of constants (such as *0*, *1*, *2* ...) and operators (such as +, -, ×). Likewise, the language of relations has its own constants and operators.

Operators fall into two categories. For the *set operators*, the tuple structure of a relation is irrelevant; the tuples might as well be regarded as atoms. For the *relational operators*, the tuple structure is essential: these are the operators that make relations powerful.

3.4.1 Constants

There are three constants:

none	empty set
univ	universal set
iden	identity

Note that *none* and *univ*, representing the set containing no atom and every atom respectively, are unary. To denote the empty binary relation, you write *none*->*none*, and for the universal relation that maps every atom to every atom, *univ*->*univ* (using the arrow operator defined in subsection 3.4.3). The identity relation is binary, and contains a tuple relating every atom to itself.

Example. For a model in which there are two sets

```
Name = {(N0), (N1), (N2)}
Addr = {(D0), (D1)}
```

the constants have the values

```
none = {}
univ = {(N0), (N1), (N2), (D0), (D1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (D0, D0), (D1, D1)}
```

Note that *iden* relates all the atoms of the universe to themselves, not just the atoms of some subset.

Discussion

Are these constants implicitly parameterized by type?

No. In some modeling languages, these constants are actually indexed collections of constants, and the appropriate instance must be selected by some means, either implicit or explicit. In Z, for example, the identity relation takes an explicit type parameter, and the empty relation is polymorphic. In Alloy, these constants are just three simple constants, with the values of *iden* and *univ* determined by the values of all the declared sets. Consequently, it's rare to use *iden* and *univ* without qualification; you'll usually write *s <: iden*, for example, to give the identity relation on the set *s* (using the restriction operator defined in subsection 3.4.3.6). If you forget to do this, you may get some surprises. For example, *iden in r* not only says that the relation *r* is reflexive but also that it maps every atom in the universe, which is likely to be inconsistent with *r*'s declaration.

Are these constants useful?

The identity relation is essential to the relational calculus style. For example, the constraint *no ^r & iden* says that the relation *r* is acyclic. A common use for the empty relation is for instantiating predicates (see subsection 4.5.2) that take sets as arguments, as in the frame conditions of section 6.2.

Aside from these cases, the constants are rarely used. To say a relation is empty or non-empty, it's easier to use the expression quantifiers (explained in subsection 3.5.2) than the constant *none*, writing *no r*, for example, rather than *r = none->none*. Universal relations are usually limited to particular sets, so instead, you'd write *Name->Addr*, for example, for the relation that maps all names to all addresses.

Do the constants add any expressive power?

A subtle point for those interested in language design issues: You might think that these constants could be omitted, and defined instead in a library module. But the universal set can't be defined in this way, since all quantifiers and comprehensions require explicit bounds. You could define the universal set explicitly as the union of all the free set variables (the top-level signatures; see subsection 4.2.1), but then you'd have to change the definition whenever a new set is introduced.

The other two constants can in fact be defined. The identity relation, for example, can be expressed as the comprehension *{x, y: univ | x = y}*.

But they were included because it's more convenient to use constants than library functions, and because the analyzer can exploit their special properties more readily this way.

3.4.2 Set Operators

The set operators are

+	union
&	intersection
-	difference
in	subset
=	equality

and here is what they mean:

- a tuple is in *p* + *q* when it is in *p or* in *q* (or both);
- a tuple is in *p* & *q* when it is in *p and* in *q*;
- a tuple is in *p* - *q* when it is in *p* but *not* in *q*;
- *p in q* is true when every tuple of *p* is also a tuple of *q*;
- *p* = *q* is true when *p* and *q* have the same tuples.

These operators can be applied to any pair of relations so long as they have the same arity. Because scalars are just singleton sets, the braces used to form sets from scalars in traditional mathematical notation aren't needed. For scalars *a* and *b*, for example, the expression *a* + *b* denotes the set containing both *a* and *b*.

Examples. Given the following sets

```
Name = {(G0), (A0), (A1)}
Alias = {(A0), (A1)}
Group = {(G0)}
RecentlyUsed = {(G0), (A1)}
```

- `Alias + Group = {(G0), (A0), (A1)}`
  gives the set of atoms that are aliases or groups;
- `Alias & RecentlyUsed = {(A1)}`
  gives the set of atoms that are aliases and have been recently used;
- `Name - RecentlyUsed = {(A0)}`
  gives the set of atoms that are names but have not been recently used;

- `RecentlyUsed` **in** `Alias`
  says that every thing that has been recently used is an alias, and is false, because of the tuple *{(G0)}*, which is recently used but not an alias;

- `RecentlyUsed` **in** `Name`
  says that every thing that has been recently used is a name, and is true;

- `Name = Group + Alias`
  says that every name is a group or an alias, and conversely every group or alias is a name, and is true.

Examples. Given the following relations, representing portions of an address book cached in memory and stored on disk,

```
cacheAddr = {(A0, D0), (A1, D1)}
diskAddr = {(A0, D0), (A1, D2)}
```

- `cacheAddr + diskAddr = {(A0, D0), (A1, D1), (A1, D2)}`
  is the relation that maps a name to an address if it's mapped in the cache *or* on disk;

- `cacheAddr & diskAddr = {(A0, D0)}`
  is the relation that maps a name to an address if it's mapped in the cache *and* on disk;

- `cacheAddr - diskAddr = {(A1, D1)}`
  is the relation that maps a name to an address if it's mapped in cache but *not* on disk;

- `cacheAddr = diskAddr`
  says that the mappings in the cache are the same as those on disk, and is false, because of the tuple *(A1, D1)* in *cacheAddr* and *(A1, D2)* in *diskAddr*.

Examples. Given the following scalars,

```
myName = {(N0)}
yourName = {(N1)}
```

- `myName + yourName = {(N0), (N1)}`
  is the set of atoms that are either my name or your name;

- `myName = yourName`
  says that my name is the same as your name, and is false;

- `yourName` **in none**
  says that there is no name that is your name, and is false also.

Discussion

Is the set operator equals sign the one you used before?

No. Statements like *cacheAddr = {(A0, D0), (A1, D1)}* are used in this chapter alone to explain the meaning of the logic, and always have an Alloy expression on the left, and a description of a relation (in conventional mathematical notation) on the right. In this case, the equals sign is a special definitional symbol, and is not symmetric: it would make no sense to write *{(A0, D0), (A1, D1)} = cacheAddr*.

A statement like *Name = Group + Alias*, on the other hand, is a constraint in the Alloy logic, and the equals sign is the set operator defined in this section. This equality notion is symmetric, and the statement is equivalent to *Group + Alias = Name*. I could have used a different symbol for the definitional equals, but that seemed a bit pedantic.

Is equality structural equality or reference equality?

A relation has no identity distinct from its value, so this distinction, based on programming notions, doesn't make sense here. If two relations have the same set of tuples, they aren't two relations: they're just one and the same relation. An atom is nothing but its identity; two atoms are equal when they are the same atom. If you have a set of atoms that represent composite objects (using some relations to map the atoms to their contents), you can define any notion of structural equality you want explicitly, by introducing a new relation. (And for those C++ programmers out there: no, you can't redefine the equals symbol in Alloy.)

Aren't there type constraints on these operators?

Not the conventional ones. In some simply typed languages, such as Z, the two arguments to a set operator must have the same type. So an expression such as *Book + Addr*, representing the union of the set of address books and the set of addresses, would be illegal. In Alloy, such expressions are not in general illegal, and can be put to good use. In modeling the value of a Java variable *v* of type *C*, for example, you might introduce a singleton set *Null* containing the null reference, and then declare

```
v: C + Null
```

to say that *v* is either null or a reference in the set *C*. In type systems that don't allow unions of this form, it can be hard to express this constraint with a declaration, and it may be necessary to weaken it to allow a reference to any class, or to distinguish null values of different types.

Alloy does impose some constraints, though. The arities of the arguments must match, so an expression like *addr + Name* is illegal. And if it can be shown, from declarations of variables alone, that an expression can be replaced by an empty relation without affecting the value of the constraint in which it appears, that expression is deemed to be ill-formed, even though its meaning is clear. For example, both *Name & Book* and *Name & (Alias + Book)* would be ill-typed because the occurrences of *Book* in both (and also *Name* in the first) could be replaced by *none* without affecting their meaning.

Why the keyword in*?*

The keyword *in* was carefully chosen for its ambiguity. Because scalars are represented as singleton sets, *in* will sometimes denote *membership* (between a scalar and a set, or a tuple and a relation), conventionally written $\in$, and sometimes *subset* (between two sets or two relations), conventionally written $\subseteq$.

3.4.3 Relational Operators

The relational operators are

->	arrow (product)
.	dot (join)
[]	box (join)
~	transpose
^	transitive closure
*	reflexive-transitive closure
<:	domain restriction
:>	range restriction
++	override

3.4.3.1 Arrow Product

The *arrow product* (or just *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.

When *p* and *q* are sets, *p->q* is a binary relation. If one of *p* or *q* has arity of two or more, then *p->q* will be a multirelation.

When *p* and *q* are tuples, *p->q* will also be a tuple. In particular, when *p* and *q* are scalars, *p->q* is a pair.

Example. Given the following names, addresses, and address book mapping

```
n = {(N0)}
n′ = {(N1)}
d = {(D0)}
d′ = {(D1)}
address = {(N0, D0), (N1, D1)}
```

we have

- `n -> d = {(N0, D0)}`
  is the tuple mapping name *n* to address *d*;
- `address = n -> d + n′ -> d′`
  says that *address* maps *n* to *d* and *n′* to *d′* (and maps nothing else), and is true.

Example. Given the following sets of names, addresses, and address books

```
Name = {(N0), (N1)}
Addr = {(D0), (D1)}
Book = {(B0)}
```

we have

- `Name -> Addr = {(N0, D0), (N0, D1), (N1, D0), (N1, D1)}`
  is the relation mapping all names to all addresses;
- `Book -> Name -> Addr =`
  `{(B0, N0, D0), (B0, N0, D1), (B0, N1, D0), (B0, N1, D1)}`
  is the relation associating books, names and addresses in all possible ways.

Example. Given the following address book mappings and address books

```
address = {(N0, D0), (N1, D1)}
address′ = {(N2, D2)}
b = {(B0)}
b′ = {(B1)}
```

`b -> address + b′ -> address′ = {(B0, N0, D0), (B0, N1, D1), (B1, N2, D2)}`
is the relation that associates book *b* with the name-address mapping *address*, and *b′* with *address′*.

3.4.3.2 Dot Join

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

s_1 -> ... -> s_m
t_1 -> ... -> t_n

you first check whether the last atom of the first tuple (that is, s_m) matches the first atom of the second tuple (that is, t_1). If not, the result is empty—there is no join. 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 ->... -> s_{m-1} -> t_2 -> ... -> t_n

Examples. Here are some example of joins of tuples:

```
{(N0, A0)} . {(A0, D0)} = {(N0, D0)}
{(N0, D0)} . {(N0, D0)} = {}
{(N0, D0)} . {(D1)} = {}
{(N0)} . {(N0, D0)} = {(D0)}
{(N0, D0)} . {(D0)} = {(N0)}
{(B0)} . {(B0, N0, D0)} = {(N0, D0)}
```

The *dot join* (or just *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 zero arity).

When *p* and *q* are binary relations, *p.q* is their standard relational composition.

Example. Given a relation *to* that maps a message to the names it's intended to be sent to, and a relation *address* that maps names to addresses

```
to = {(M0, N0), (M0, N2), (M1, N2), (M2, N3)}
address = {(N0, D0), (N0, D1), (N1, D1), (N1, D2), (N2, D3), (N4, D3)}
```

the relation *to.address* maps a message to the addresses it should be sent to:

```
to.address = {(M0, D0), (M0, D1), (M0, D3), (M1, D3)}
```

and is illustrated in fig. 3.8 overleaf.

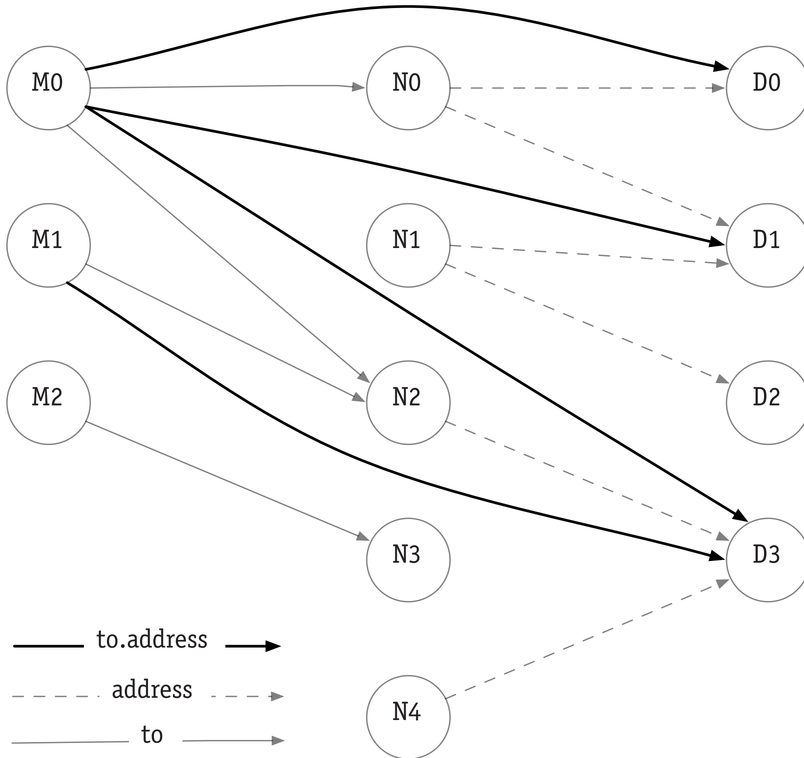

FIG. 3.8 A snapshot illustrating a dot join of two relations: feint arcs for the relation *to*; dashed arcs for *address*; and solid arcs for their join *to.address*.

If *p* and *q* are functions, *p.q* will be a function too, and in this case dot is equivalent to functional composition.

Examples. Given a function *address* mapping names to addresses, a function *user* mapping an address to its username portion, and a function *host* mapping an address to its hostname portion

address = {(N0, D0), (N1, D0), (N2, D2)}
user = {(D0, U0), (D1, U1), (D2, U2)}
host = {(D0, H0), (D1, H1), (D2, H2)}

the expressions *address.user* and *address.host* are the functions that map a name to the corresponding user and host respectively:

```
address.user = {(N0, U0), (N1, U0), (N2, U2)}
address.host = {(N0, H0), (N1, H0), (N2, H2)}
```

When *s* is a set, and *r* is a binary relation, *s.r* is the image of the set *s* under the relation *r*; this image is the set you get if you follow the relation *r* for each member of *s*, and collect together in a single set all the sets that result. This is perhaps the most common use of dot, and is called *navigation* in object modeling parlance.

When *x* is a scalar, and *r* is a binary relation, *x.r* is the set of atoms that *x* maps to. For a function *f* and a scalar *x* in its domain, *x.f* is the scalar that *f* maps *x* to. So in this case, join is like function application, but note that *x.f* will be the empty set when *x* is not in the domain of *f*. Traditionally, a function applied outside its domain gives no result at all, and an expression involving such an application may therefore be undefined. In our logic, there are no undefined expressions.

You can navigate in both directions; *s.r* is the image of the set *s* going forward through *r*, and *r.s* is the image going backward.

Example. Given a multilevel address book represented by a relation *address*, and sets of aliases, groups, and addresses

```
address = {(G0, A0), (G0, A1), (A0, D0), (A1, D1)}
Alias = {(A0), (A1)}
Group = {(G0)}
Addr = {(D0), (D1), (D2)}
```

we have the following expressions:

- `Alias.address = {(D0), (D1)}`
  the set of results obtained by looking up any alias in the address book;
- `Group.address = {(A0), (A1)}`
  the set of results obtained by looking up any group in the address book;
- `address.Group = {}`
  the set of names that when looked up in the address book yield groups;
- `address.Alias = {(G0)}`
  the set of names that when looked up in the address book yield aliases.

Joins of relations of higher arity are common too, especially the forms *x.q* and *q.x*, where *x* is a scalar, and *q* is a multirelation.

Example. Given a particular address book *b*, and a ternary relation *addr* associating books, names, and addresses

```
b = {(B0)}
addr = {(B0, N0, D0), (B0, N1, D1), (B1, N2, D2)}
```

the expression *b.addr* is the name-address mapping for book *b*:

```
b.addr = {(N0, D0), (N1, D1)}
```

Example. Given a time *t*, and a ternary relation *addr* that contains the triple *n->a->t* when name *n* maps to address *a* at time *t*

```
t = {(T1)}
addr = {(N0, D0, T0), (N0, D1, T1), (N1, D2, T0), (N1, D2, T1)}
```

the expression *addr.t* is the name-address mapping at time *t*:

```
addr.t = {(N0, D1), (N1, D2)}
```

Example. Given a relation *addr* of arity four that contains the tuple *b->n->a->t* when book *b* maps name *n* to address *a* at time *t*, and a book *b* and a time *t*

```
addr = {(B0, N0, D0, T0), (B0, N0, D1, T1), (B0, N1, D2, T0),
   (B0, N1, D2, T1), (B1, N2, D3, T0), (B1, N2, D4, T1)}
t = {(T1)}
b = {(B0)}
```

the expression *b.addr.t* is the name-address mapping of book *b* at time *t*:

```
b.addr.t = {(N0, D1), (N1, D2)}
```

Note that *b.addr.t* doesn't need parentheses to indicate the order in which the joins are applied. The expressions *b.(addr.t)* and *(b.addr).t* are equivalent: you can project onto a particular book, and then onto a particular time, or you can first select the time, and then the book.

Discussion

Is dot join associative?

Not in general. If the arguments are binary relations, it is. But in general, the expressions *(a.b).c* and *a.(b.c)* are not equivalent. Moreover, one

may be ill-formed and the other well-formed. Because of the dropped column, the arity of a join is always one less than the sum of the arities of its arguments. If *s* and *t* are unary, and *r* is ternary, for example, the expression *t.r* will be binary, and *s.(t.r)* will be unary. The expression *s.t*, however, would have zero arity, and is thus illegal, so *(s.t).r* is likewise illegal, and is certainly not equivalent to *s.(t.r)*.

Is dot join the same as a database join?

Not quite. In relational database query languages, the join operator matches columns by name rather than position, and the matching column is not dropped. You can define a more database-like join as follows. Let *id3* be the ternary identity relation

```
id3 = {a, b, c: univ | a = b and b = c}
```

and define

```
p ⊙ q = p.id3.q
```

Then *p* ⊙ *q* concatenates matching tuples like dot join, but retains the matching elements like database join. It also provides a nice shorthand for restrictions (introduced in section 3.4.3.6): *s <: r* and *r :> s* can be written *s* ⊙ *r* and *r* ⊙ *s*. (Thanks to Butler Lampson for this insight.)

3.4.3.3 Box Join

The box operator *[]* is semantically identical to join, but takes its arguments in a different order, and has different precedence. The expression

```
e1 [e2]
```

has the same meaning as

```
e2.e1
```

> *Example*. Given a relation *address* from names to addresses, and a scalar *n* representing a name, the expression *address[n]* is equivalent to *n.address*, and denotes the set of addresses that *n* is mapped to.

Dot binds more tightly than box, however, so

```
a.b.c [d]
```

is short for

```
d.(a.b.c)
```

The rationale for this operator is that it allows a syntactic distinction to be made between dereferencing a field of a composite object (with dot join) and performing an indexed lookup (with box join), even though there is no semantic distinction between the two.

> *Example.* Given a ternary relation *addr* associating books, names, and addresses, the expression *b.addr[n]* denotes the set of addresses associated with name *n* in book *b*, and is equivalent to *n.(b.addr)*.

The choice of the box is motivated by analogy to array notation.

> *Example.* In a model of a class *C* that has an array-valued field *f*, the result of dereferencing *x* with field *f*, and then retrieving the object at index *i* can be denoted *x.f[i]*, just as in Java, or equivalently as *i.(x.f)*.

3.4.3.4 Transpose

The *transpose* *~r* of a binary relation *r* takes its mirror image, forming a new relation by reversing the order of atoms in each tuple.

> *Example.* Given a relation representing an address book that maps names to the addresses they stand for
>
> ```
> address = {(N0, D0), (N1, D0), (N2, D2)}
> ```
>
> its transpose is the relation that maps each address to the names that stand for it:
>
> ```
> ~address = {(D0, N0), (D0, N1), (D2, N2)}
> ```

A binary relation *r* is *symmetric* if, whenever it contains the tuple *a->b*, it also contains the tuple *b->a*, or more succinctly as a relational constraint:

```
~r in r
```

Taking the transpose of a symmetric relation has no effect. The *symmetric closure* of *r* is the smallest relation that contains *r* and is symmetric, and is equal to *r + ~r*.

> *Examples.* A relation *connects* mapping hosts to the neighbors they are connected to in a network would be symmetric if the connections were bidirectional. The transpose of a relation *wife* mapping men to their wives is the relation *husband* mapping women to their husbands, and its symmetric closure is the relation *spouse* mapping each person to his or her spouse.

Some useful facts about transpose:

- *s.~r* is equal to *r.s*, and is the image of the set *s* navigating backward through the relation *r*;
- *r.~r* is the relation that associates two atoms in the domain of the relation *r* when they map to a common element; when *r* is a function, *r.~r* is the equivalence relation that equates atoms with the same image.
- *r.~r in iden* therefore says that *r* is injective, and *~r.r in iden* says that *r* is functional.

  Example. If *mother* is the relation that maps a child to its mother, the expression *mother.~mother* is the sibling relation that maps a child to its siblings (and also to itself).

Discussion

Why did you write ~r in r *to say that* r *is symmetric?*

You might have expected *~r = r* instead. The two conditions are equivalent, but I prefer the first because (1) it matches the informal statement more closely; (2) it follows the pattern of the conditions for reflexivity and transitivity; and (3) it's a good habit from an analysis perspective to write constraints in their weakest form. Admittedly, this is a bit pedantic, and it's not unreasonable to expect a definition of symmetry to be symmetric.

3.4.3.5 Transitive Closure

A binary relation is *transitive* if, whenever it contains the tuples *a->b* and *b->c*, it also contains *a->c*, or more succinctly as a relational constraint:

```
r.r in r
```

The *transitive closure* *^r* of a binary relation *r*, or just the *closure* for short, is the smallest relation that contains *r* and is transitive. You can compute the closure by taking the relation, adding the join of the relation with itself, then adding the join of the relation with that, and so on:

```
^r = r + r.r + r.r.r + …
```

Example. A relation *address* representing an address book with multiple levels (which maps aliases and groups to groups, aliases, and addresses), and its transitive closure:

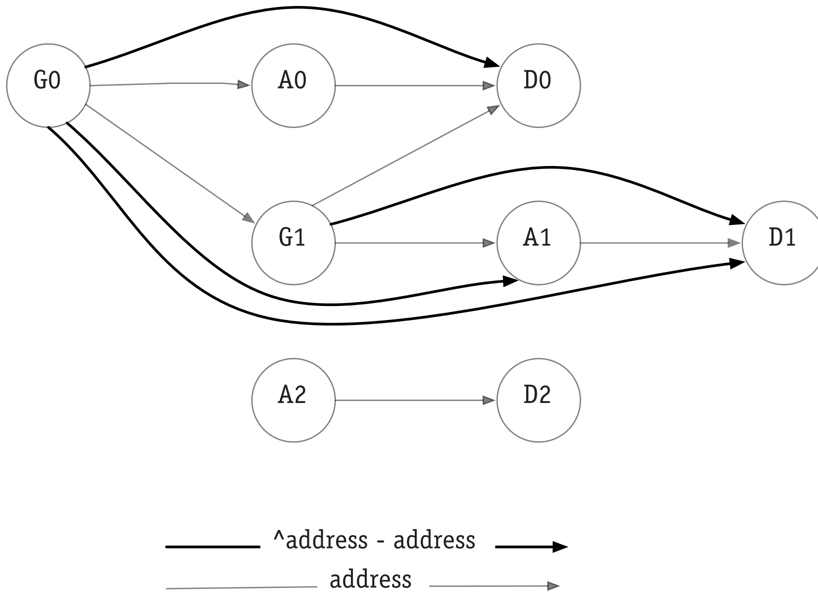

FIG. 3.9 A snapshot illustrating transitive closure of a relation: the feint arcs represent the relation *address*; the solid arcs are those that are added to it to form its closure ^*address*.

```
 address =
   {(G0, A0), (G0, G1), (A0, D0), (G1, D0), (G1, A1), (A1, D1), (A2, D2)}
^address =
    {(G0, A0), (G0, G1), (A0, D0), (G1, D0), (G1, A1), (A1, D1), (A2, D2),
    (G0, D0), (G0, A1), (G1, D1),
    (G0, D1)}
```

I've broken the transitive closure into lines to indicate the contribution from the relation itself (on the first line), from its square *address.address* (on the second), and from its cube *address.address.address* (on the third). Fig. 3.9 shows the closure graphically.

Viewing a relation as a graph, the transitive closure represents reachability. Since the relation itself represents the paths that are one step long, its square the paths that are two steps long, and so on, the closure relates one atom to another when they are connected by a path of any length (except for zero).

A binary relation *r* is *reflexive* if it contains the tuple *a* -> *a* for every atom *a*, or as a relational constraint,

```
iden in r
```

The *reflexive-transitive closure* **r* is the smallest relation that contains *r* and is both transitive and reflexive, and is obtained by adding the identity relation to the transitive closure:

```
*r = ^r + iden
```

From the graphical viewpoint, the reflexive-transitive closure relates one atom to another when they are connected by a path of any length, including zero.

Because *iden* relates every atom in the universe to itself (as explained in section 3.4.1 and the discussion that follows it), the reflexive-transitive closure will do so as well.

Discussion

Why does the reflexive-transitive closure associate "irrelevant" atoms?

Suppose a model has a set *Book* of books, a set *Name* of names, a set *Addr* of addresses, a book *b*, and a relation *addr* mapping books to their contents, with the following values:

```
Book = {(B0), (B1)}
Name = {(N0), (N1)}
Addr = {(D0), (D1)}
b = {(B0)}
addr = {(B0, N0, N1), (B0, N1, D0), (B1, N1, D1)}
```

Then the universe will contain all the atoms

```
univ = {(B0), (B1), (N0), (N1), (D0), (D1)}
```

and the identity relation will map each to itself:

```
iden = {(B0, B0), (B1, B1), (N0, N0), (N1, N1), (D0, D0), (D1, D1)}
```

The expression *^(b.addr)*, denoting the direct and indirect mapping of names in book *b* to the names and addresses reachable, will map names to names and addresses:

```
^(b.addr) = {(N0, N1), (N1, D0), (N0, D0)}
```

The expression **(b.addr)* will include the tuples of both these relations. In addition to tuples such as *(N0, N0)*, which are expected, it will also includes tuples such as *(B0, B0)*.

Although this seems odd, it follows naturally from the definition of reflexive-transitive closure and the identity relation. The alternative would be to have sets implicitly associated with each relation that represent the possible members of its domain and range, which would complicate the logic.

In practice this is not a problem. Closures often appear in navigation expressions, and the irrelevant self-tuples disappear in the join. For example, the names and addresses reachable in zero or more steps from a set of names *friends* would be denoted *friends.*(b.addr)*, and would not include any books, because *friends* and *Book* would be disjoint. If you need to remove the extra tuples explicitly, you can always write *s <: *r* to restrict the closure to map only atoms in the set *s*.

How many iterations can it take to form the closure of a relation?

For a finite universe, transitive closure needs only a finite unwinding, limited by the length of the longest path in the graph. For some relations, the transitive closure requires very few unwindings even if the universe is large. Stanley Milgram's famous experiment in which he had residents of Kansas attempt to get letters to residents of Boston via acquaintances showed that it took on average only six steps for a letter to arrive [54]. If six steps were really enough to connect any two people, it would mean that the closure of the *knows* relation is the universal relation, and that it can be obtained in six unwindings.

3.4.3.6 Domain and Range Restrictions

The restriction operators are used to filter relations to a given domain or range. The expression *s <: r*, formed from a set *s* and a relation *r*, contains those tuples of *r* that start with an element in *s*. Similarly, *r :> s* contains the tuples of *r* that end with an element in *s*.

Restrictions can be applied to relations of any arity of two or more, but are most often applied to binary relations.

Examples. Given a relation representing a multilevel address book and sets representing the aliases, groups, and addresses

```
address = {(G0, A0), (G0, G1), (A0, D0),
               (G1, D0), (G1, A1), (A1, D1), (A2, D2)}
Alias = {(A0), (A1), (A2)}
Group = {(G0), (G1)}
Addr = {(D0), (D1), (D2)}
```

- `address :> Addr = {(A0, D0), (G1, D0), (A1, D1), (A2, D2)}`
  contains the entries that map names to addresses (and not to other names);
- `address :> Alias = {(G0, A0), (G1, A1)}`
  contains the entries that map names to aliases;
- `Group <: address = {(G0, A0), (G0, G1), (G1, D0), (G1, A1)}`
  contains the entries that map groups.

Applying a restriction to a binary relation is like taking the image of a set, but without dropping the matching elements. Put more formally, if *r* is a binary relation, and *s* is a set, then

```
range (s <: r) = s.r
domain (r :> s) = r.s
```

The identity relation maps every atom in the universe to itself. Often, what we want instead is a relation that maps every atom in some set *s* to itself, which can be written *s <: iden*.

3.4.3.7 Override

The *override p ++ q* of relation *p* by relation *q* is like the union, except that the tuples of *q* can replace the tuples of *p* rather than just augmenting them. Any tuple in *p* that matches a tuple in *q* by starting with the same element is dropped. The relations *p* and *q* can have any matching arity of two or more.

> *Example.* An address book might be represented by two relations, *homeAddress* and *workAddress*, mapping an alias to email addresses at home and at work:
>
> ```
> homeAddress = {(A0, D1), (A1, D2), (A2, D3)}
> workAddress = {(A0, D0), (A1, D2)}
> ```
>
> The preferred address for an alias, which is the work address if it exists, and otherwise the home address, is given by
>
> ```
> homeAddress ++ workAddress = {(A0, D0), (A1, D2), (A2, D3)}
> ```

Override can be defined in terms of simpler operators. Taking the override of *p* by *q* is equivalent to taking the union of *q* and what's left of *p* after removing the tuples that start with an element in the domain of *q*:

```
p ++ q = p - (domain (q) <: p) + q
```

Override is useful for modeling insertions into map datatypes, and assignment-like statements in programs.

Example. Insertion of a key *k* with value *v* into a hashmap can be modeled by representing the value of the map before and after as two relations *m* and *m′* from keys to values, satisfying

```
m′ = m ++ k->v
```

Example. The environment *e* of an executing Java program can be viewed (simplistically) as a relation mapping variables to object references. The effect of a Java assignment

```
x = y
```

with a variable on both sides can be modeled in Alloy as

```
e′ = e ++ x->y.e
```

where *e* and *e′* are the values of the environment before and after execution. The state of the heap at any point can be represented by one relation for each field (that is, instance variable) of each class. A setter statement such as

```
x.f = y
```

in which *x* and *y* are variables and *f* is a field can thus be described by

```
f′ = f ++ x.e->y.e
```

where *f* and *f′* represent the values of the field *f* before and after execution.

Discussion

What are the operator precedences?

Operators have a standard precedence ranking so that constraints aren't marred by masses of parentheses. The ranking follows the usual conventions: unary operators (closure, transpose) precede binary operators; product operators (such as dot and arrow) precede sum operators (plus, minus, intersect). The details are given in appendix B. All operators associate to the left.

3.5 Constraints

We've seen how to make a constraint from two expressions using the comparison operators *in* and =. Larger constraints are made from smaller constraints by combining them with the standard logical operators, and by quantifying constraints that contain free variables over bindings.

3.5.1 Logical Operators

There are two forms of each logical operator: a shorthand and a verbose form (similar to the operators used in boolean expressions in programming languages):

- **not** ! negation
- **and** && conjunction
- **or** || disjunction
- **implies** => implication
- **iff** <=> bi-implication

The negation symbol can be combined with comparison operators, so *a != b* is equivalent to *not a = b*, for example. The shorthand and the verbose forms are completely interchangeable, so you can write *a not = b* as well.

There is an *else* keyword that can be used with the implication operator;

```
F implies G else H
```

is equivalent to

```
(F and G) or ((not F) and H)
```

Implications are often nested. The common idiom

```
C1 implies F1
else C2 implies F2
else C3 implies F3
```

says that under condition *C1*, *F1* holds, and if not, then under condition *C2*, *F2* holds, and if not, under condition *C3*, *F3* holds.

Conjunctions of constraints are so common that we'll often omit the *and* operator, and wrap the entire collection of constraints in braces. So *{F G H}* is equivalent to *F and G and H*.

Sometimes, it's more natural to use a conditional expression than a conditional formula. This takes the form

```
C implies E1 else E2
```

or

```
C => E1 else E2
```

where *C* is a constraint, and *E1* and *E2* are expressions, and has the value of *E1* when *C* is true, and the value of *E2* otherwise.

Examples. Suppose an address book is modeled with three relations: *homeAddress* and *workAddress* mapping an alias to email addresses at home and at work, and *address* mapping an alias to the preferred address. To say that the preferred address for an alias *a* is the work address if it exists, otherwise the home address, we can write

```
some a.workAddress =>
  a.address = a.workAddress
  else a.address = a.homeAddress
```

or, using a conditional expression

```
a.address =
    some a.workAddress => a.workAddress else a.homeAddress
```

3.5.2 Quantification

A quantified constraint takes the form

```
Q x: e | F
```

where *F* is a constraint that contains the variable *x*, *e* is an expression bounding *x*, and *Q* is a quantifier.

The forms of quantification in Alloy are

- **all** *x: e | F* — *F* holds for every *x* in *e*;
- **some** *x: e | F* — *F* holds for some *x* in *e*;
- **no** *x: e | F* — *F* holds for no *x* in *e*;
- **lone** *x: e | F* — *F* holds for at most one *x* in *e*;
- **one** *x: e | F* — *F* holds for exactly one *x* in *e*.

To remember what *lone* means, it might help to think of it as being short for "less than or equal to one."

Several variables can be bound in the same quantifier;

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

for example, says that there is exactly one combination of values for *x* and *y* that makes *F* true. Variables with the same bounding expression can share a declaration, so this constraint can also be written

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

By using the keyword *disj* before the declaration, you can restrict the bindings only to include ones in which the bound variables are disjoint from one another, so

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

means that *F* is true for any distinct combination of values for *x* and *y*. (See subsection 3.5.3 for cases in which *x* and *y* are not scalars.)

Examples. Given a set *Address* of email addresses, *Name* of names, and a relation *address* representing a multilevel address book mapping names to names and addresses,

- `some n: Name, a: Address | a in n.address`
  says that some name maps to some address (that is, the address book is not empty);
- `no n: Name | n in n.^address`
  says that no name can be reached by lookups from itself (that is, there are no cycles in the address book);
- `all n: Name | lone d: Address | d in n.address`
  says that every name maps to at most one address;
- `all n: Name | no disj d, d': Address | d + d' in n.address`
  says the same thing, but slightly differently: that for every name, there is no pair of distinct addresses that are among the results obtained by looking up the name.

Quantifiers can be applied to expressions too:

- **some** *e* — *e* has some tuples;
- **no** *e* — *e* has no tuples;
- **lone** *e* — *e* has at most one tuple;
- **one** *e* — *e* has exactly one tuple.

Note that *some e* and *no e* could be written *e != none* and *e = none* respectively, but using the quantifiers makes the constraints more readable.

Examples. Using the sets and relation from the previous example,

- `some Name`
  says that the set of names is not empty;
- `some address`
  says that the address book is not empty: there is some pair mapping a name to an address or to a name;
- `no (address.Addr - Name)`
  says that nothing is mapped to addresses except for names;
- `all n: Name | lone (n.address & Address)`

says that every name maps to at most one address (more succinctly than in the previous example);

- **all** n: Name | **one** n.address **or no** n.address
  says the same thing.

3.5.3 Higher-Order Quantification

Quantified variables don't have to be scalars; they can be sets, or even multirelations. A logic that allows this is no longer "first order" and becomes "higher order." Alloy includes such quantifications, but they cannot always be analyzed (see subsection 5.2.2).

Examples. Higher-order quantifications are often useful for stating properties about operators:

- **all** s, t: **set univ** | s + t = t + s
  the union operator on sets is commutative;
- **all** p, q: **univ lone**->**lone univ** | p.q **in univ lone**->**lone univ**
  the join of two functions is a function too.

Discussion

Does Alloy allow freestanding declarations?

The declaration forms described in this section can be used for quantified variables, and for fields, and they can be used as constraints (using *in* rather than the colon). Top-level relation declarations are not supported, as they are unnecessary (as explained in the discussion following section 4.2.2).

When can higher-order quantifications be analyzed?

Generally, the Alloy Analyzer cannot handle formulas that involve higher-order quantifications, so their use is discouraged. But in some useful cases, higher-order quantifiers can be eliminated by a scheme known as "Skolemization," which turns a quantified variable into a free variable whose value can then be found by constraint solving. See subsection 5.2.2 for more details.

What's the difference between quantifiers and multiplicities?

Novices are sometimes confused by the difference between these quantifications:

```
lone p: some X | F
some p: lone X | F
```

What makes these confusing is the use of the same keywords for the quantifier and the bounding expression's multiplicity. In the first case, the quantifier is *lone* (at most one), and the multiplicity is *some* (one or more), so *p* is constrained to be drawn from the nonempty subsets of *X*, and the constraint says that *F* holds for at most one such subset *p*. In the second case, the quantifier is *some*, and the multiplicity is *lone*, so the constraint says that *F* holds for some option *p*, and is equivalent to

```
(some p: X | F) or (let p = none | F)
```

and is thus not really a higher-order quantification at all. I've never come across a need for the first, but the second is occasionally useful.

Does anyone really confuse quantifiers and multiplicities?

Lest you think the last question was of purely academic interest, consider the following common (but surprisingly subtle) modeling mistake. Suppose you want to model the flushing of a line from a cache defined as a set of lines. You might be tempted to write something like

```
one x: Line | cache′ = cache - x
```

reading it as "pick one line so that the cache after is the cache before with that line removed." But this is not what the Alloy constraint actually means. Rather, it says there is exactly one line for which the old and new cache are related by that constraint. Note that, in particular, if the cache is empty before and after, the constraint will be false, because there will be multiple values of *x*—in fact, any value—for which the body holds. What you meant to write instead was

```
some x: Line | cache′ = cache - x
```

namely that there is *some* line that can be removed from the cache before to obtain the cache after. This constraint allows for the cache to be empty before and after. If this were not desired, analysis would find the error, and you could then elaborate the formula, for example to

```
some x: Line | x in cache and cache′ = cache - x
```

3.5.4 Let Expressions and Constraints

When an expression appears repeatedly, or is a subexpression of a larger, complicated expression, you can factor it out. The form

```
let x = e | A
```

is short for *A* with each occurrence of the variable *x* replaced by the expression *e*. The body of the let, *A*, and thus the form as a whole, can be a constraint or an expression.

Example. Revisiting the version of the address book with three relations—*homeAddress* and *workAddress* mapping an alias to email addresses at home and at work, and *address* mapping an alias to the preferred address—we can say that the preferred address for an alias *a* is the work address if it exists, otherwise the home address, by writing

```
all a: Alias |
    let w = a.workAddress |
        a.address = some w implies w else a.homeAddress
```

or

```
all a: Alias |
  a.address =
        let w = a.workAddress |
            some w implies w else a.homeAddress
```

Discussion

Can let *bindings be recursive?*

No. They only provide a convenient shorthand, and don't allow recursive definitions. A variable introduced by a let on the left-hand side of a binding cannot appear on the right-hand side of the same binding, or one that precedes it in the same *let* construct.

3.5.5 Comprehensions

Comprehensions make relations from properties. The comprehension expression

$$\{x_1: e_1, x_2: e_2, \ldots, x_n: e_n \mid F\}$$

makes a relation with all tuples of the form x_1->x_2->...->x_n for which the constraint *F* holds, and where the value of x_i is drawn from the value of the bounding set expression e_i. Each expression e_i must denote a set, and not a relation of higher arity.

Examples. In a multilevel address book represented by a relation *address* mapping names in the set *Name* to names and also to addresses in the set *Addr*,

- `{n: Name | no n.^address & Addr}`
  is the set of names that don't resolve to any actual addresses;
- `{n: Name, a: Addr | n->a in ^address}`
  is a relation mapping names to addresses that corresponds to the multilevel lookup.

3.6 Declarations and Multiplicity Constraints

A declaration introduces a relation name. We've just seen how declarations are used in quantified constraints and comprehensions. Freestanding declarations of relation names make sense too, although we'll see in chapter 4 how, in the full Alloy language, these would instead be declared within "signatures."

The notion of multiplicity is closely tied to the notion of declaration. It's not essential in a logic, but I've included it in this chapter because it's so useful, and can be explained independently of the structuring mechanisms of Alloy.

3.6.1 Declarations

A constraint of the form

```
relation-name : expression
```

is called a *declaration*, and says that the relation named on the left has a value that is a subset of the value of the *bounding expression* on the right. The bounding expression is usually formed with unary relations and the arrow operator, but any expression can be used. In fact, as we shall see in the next two sections, the bounding expression on the right can actually be a more general form of expression that uses multiplicity symbols.

Declarations in this form, with the colon as the operator, are used to declare bound variables in quantified formulas (section 3.5.2), and fields of signatures (section 4.2). But the same form can also constrain a previously declared variable, or an arbitrary expression (section 3.6.4), in which case the colon is replaced by the keyword *in*.

Examples. The *address* relation, representing a single address book, maps names to addresses:

```
address: Name->Addr
```

The *addr* relation, representing a collection of address books, maps books to names to addresses:

```
addr: Book->Name->Addr
```

A relation *address* representing a multilevel address book maps names to names and addresses:

```
address: Name->(Name + Addr)
```

The same relation can be declared in different ways, depending on how much information you want to put in the declaration.

Example. A declaration saying that a relation *address* maps aliases and groups to addresses and to aliases and groups

```
address: (Alias + Group)->(Addr + Alias + Group)
```

and a stronger declaration of the same relation, saying, in addition, that aliases, unlike groups, are always mapped directly to addresses:

```
address: (Alias->Addr) + (Group->(Addr + Alias + Group))
```

Relations, not just sets, can appear on the right-hand side of declarations too.

Example. An address book might be represented with three relations, representing the home, work, and preferred addresses:

```
workAddress, homeAddress: Alias->Addr
prefAddress: workAddress + homeAddress
```

3.6.2 Set Multiplicities

A declaration can include *multiplicity constraints*, which are sometimes implicit. Multiplicities are expressed with the *multiplicity keywords*:

- **set** any number
- **one** exactly one
- **lone** zero or one
- **some** one or more

Note that *one*, *lone*, and *some* are the same keywords used for quantification.

The meaning of a declaration depends on the arity of the bounding expression. If it denotes a set (that is, is unary), it can be prefixed by a multiplicity keyword like this

x: *m* e

which constrains the size of the set *x* according to *m*. For a set-valued bounding expression, omitting the keyword is the same as writing *one*. So if no keyword appears, the declaration makes the variable a scalar.

Examples

- `RecentlyUsed: set Name`
  says that *RecentlyUsed* is a subset of the set *Name*;
- `senderAddress: Addr`
  says that *senderAddress* is a scalar in the set *Addr*;
- `senderName: lone Name`
  says that *senderName* is an option: either a scalar in the set *Name*, or empty;
- `receiverAddresses: some Addr`
  says that *receiverAddresses* is a nonempty subset of *Addr*.

The declarations of variables in quantified constraints are declarations of exactly the same form, and follow the same rules. The only difference is that quantifiers introduce variables that are bound within the body of the quantified constraint; the other declarations we have seen introduce free variables.

Example. The quantification we saw above,

```
some n: Name, a: Address | a in n.address
```

has two declarations, binding the scalars *n* and *a*.

3.6.3 Relation Multiplicities

When the bounding expression is a relation (that is, a relation with arity greater than one), it may not be preceded by a multiplicity keyword. But if the bounding expression is constructed with the arrow operator, multiplicities can appear inside it. Suppose the declaration looks like this:

```
r: A m->n B
```

where *m* and *n* are multiplicity keywords (and where *A* and *B* are, for now, sets). Then the relation *r* is constrained to map each member of *A* to *n* members of *B*, and to map *m* members of *A* to each member of *B*.

Such a declaration can indicate the domain and range of the relation (see subsection 3.2.5), and whether or not it is functional or injective (see subsection 3.2.4):

- `r: A->one B`
  a function whose domain is *A*;

- r: A **one** -> B
  an injective relation whose range is *B*;
- r: A -> **lone** B
  a function that is partial over the domain *A*;
- r: A **one** -> **one** B
  an injective function with domain *A* and range *B*, also called a bijection from *A* to *B*;
- r: A **some** -> **some** B
  a relation with domain *A* and range *B*.

Examples. Some declarations and their meaning:

- workAddress: Alias -> **lone** Addr
  The relation *workAddress* is a function that maps each member of the set *Alias* to at most one member of the set *Addr*; each alias represents at most one work address.
- homeAddress: Alias -> **one** Addr
  Each alias represents exactly one home address.
- members: Group **lone** -> **some** Addr
  An address belongs to at most one group, and a group contains at least one address.

Multiplicities are just a shorthand, and can be replaced by standard constraints; the multiplicity constraint in

r: A *m* -> *n* B

can be written as

all a: A | *n* a.r
all b: B | *m* r.b

but multiplicities are preferable because they are terser and easier to read.

Example. The last declaration of the previous example

members: Group **lone** -> **some** Addr

can be replaced by

members: Group -> Addr

along with the constraints

all g: Group | **some** g.members

```
all a: Addr | lone members.a
```

The expressions *A* and *B* can be arbitrary expressions, and don't have to be relation names. They also don't have to represent unary relations. The rule is generalized simply by replacing "member" by "tuple." Thus

```
r: A m->n B
```

says that *r* maps *m* tuples in *A* to each tuple in *B*, and maps each tuple in *A* to *n* tuples in *B*.

Example. The declaration

```
addr: (Book->Name)->lone Addr
```

says that the relation *addr* associates at most one address with each address book and name pair.

3.6.4 Declaration Formulas

Declarations usually introduce new names, but the same form can also be used to impose constraints on relations that have already been declared, or on arbitrary expressions. In this case, the subset operator *in* is used in place of the colon, and the term "declaration formula" is used.

Example. For an address book represented by a relation *address* mapping groups and aliases to addresses

```
address in (Group + Alias)->Addr
```

a declaration formula might say that each alias maps to at most one address:

```
Alias <: address in Alias->lone Addr
```

A declaration formula is like any other formula, and can be combined with logical operators, placed inside the body of quantifications, and so on.

Example. Given a relation *addr* associating address books, names and addresses, the constraint that each address book is injective (that is, maps at most one name to an address) can be written

```
all b: Book | b.addr in Name lone->Addr
```

3.6.5 Nested Multiplicities

Multiplicities can be nested. Suppose you have a declaration of the form

```
r: A ->(B m->n C)
```

This means that, for each tuple in *A*, the corresponding tuples in *B*->*C* form a relation with the given multiplicity. In the case that *A* is a set, the multiplicity constraint is equivalent to

```
all a: A | a.r in B m->n C
```

Similarly,

```
r: (A m->n B)->C
```

will be equivalent to

```
all c: C | r.c in A m->n B
```

Examples. The declaration

```
addr: Book->(Name lone->Addr)
```

says that, for any book, each address is associated with at most one name, and is equivalent to

```
all b: Book | b.addr in Name lone->Addr
```

whereas

```
addr: (Book->Name) lone->Addr
```

says that each address is associated with at most one book/name combination. The first allows an address to have different names in different books; the second does not.

3.7 Cardinality and Integers

The operator # applied to a relation gives the number of tuples it contains, as an integer value. The following operators can be used to combine integers:

plus	addition
minus	subtraction
mul	multiplication
div	division
rem	remainder

and the following to compare them:

=	equals
<	less than
>	greater than
=<	less than or equal to
>=	greater than or equal to

Example. For a relation *address*

```
address: (Group + Alias) -> Addr
```

mapping groups and aliases to addresses, the constraint that every group has more than one address associated with it can be written

```
all g: Group | #g.address > 1
```

Example. Suppose an email program needs to break groups of addresses into smaller subgroups. Given a relation mapping groups to the addresses they contain,

```
address: Group -> Addr
```

a second relation

```
split: Group -> Group
```

might map a group to its subgroups under the constraint that no group is a subgroup of itself

```
no g: Group | g in g.split
```

that a group's subgroups contain all its addresses

```
all g: split.Group | g.address = g.split.address
```

and that the subgroups are disjoint

```
all g: Group, disj g1, g2: g.split | no g1.address & g2.address
```

The cardinality constraints on the division into subgroups might be that any group with more than 5 members is split up

```
all g: Group | #g.address > 5 implies some g.split
```

that no subgroup contains more than 5 members

```
all g: Group.split | #g.address =< 5
```

and that subgroups are of roughly equal size (differing from each other by at most one)

```
all g: Group, disj g1, g2: g.split |
      #g1.address < #g2.address
         implies #g2.address = #g1.address.plus[1]
```

The expression *e.sum* denotes the sum of a set of integers *e*, and the expression

```
sum x: e | ie
```

denotes the integer obtained by summing the values of the integer expression *ie* for all values of the scalar *x* drawn from the set *e*.

Example. The size of a group is the sum of the sizes of its subgroups:

```
all g: split.Group | #g.address = (sum g': g.split | #g'.address)
```

Discussion

Why not use the standard symbol + for addition?

An earlier version of Alloy used the same symbol for integer addition and relational union, but this complicated the language. The current version avoids any ambiguity by using distinct integer operators.

What kind of form is x.plus[1]?

You can think of *plus*, *minus*, etc. as predefined ternary relations representing the arithmetic operators. So this expression is just a standard join using dot and box, and could be written equivalently as *plus[x,1]*, for example.

Can integers be atoms in relations?

Yes, they can. But the arithmetic operators and comparisons cannot be applied to sets, so in a formula such as *S =< T*, if *S* and *T* are sets of integers, the operator will actually compare their sums, as if you'd written

```
S.sum =< T.sum
```

and similarly *S.plus[T]* will be short for *(S.sum).plus[(T.sum)]*.

The equality operator, however, retains its standard meaning, so *S = T* will be false when *S* and *T* are distinct sets of integers, even if their sums are equal. This has the somewhat disturbing consequence that

```
S =< T and T =< S implies S.sum = T.sum
```

will always be true, but

```
S =< T and T =< S implies S = T
```

may not be.

What kind of form is S.sum?

That's a good question. Unfortunately, it can't be interpreted as a standard join, since the sum takes the entire set and produces a single in-

teger (that is, a scalar) result. Think of *sum* as a special function, which, like equality, cannot be represented as a first-order relation.