Titanium Language Reference Manual, version 2.19

Paul N. Hilfinger
Dan Oscar Bonachea
Kaushik Datta
David Gay
Susan L. Graham
Benjamin Robert Liblit
Geoffrey Pike
Jimmy Zhigang Su
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-15
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-15.html

November 17, 2005

Acknowledgement

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract F30602-95-C-0136, the National Science Foundation under grant ACI-9619020, the Department of Energy under contracts W-7405-ENG-48 and DE-AC03-765F00098, and donations from Sun Microsystems and Microsoft. The views expressed herein are not necessarily those of the Government, Sun Microsystems, or Microsoft.

Titanium Language Reference Manual

Version 2.19

P. N. Hilfinger (editor), Dan Bonachea, Kaushik Datta,
David Gay, Susan Graham, Ben Liblit, Geoff Pike, Jimmy Su,
and Katherine Yelick

November, 2005

Abstract

The Titanium language is a Java dialect for high-performance parallel scientific computing. Titanium's differences from Java include multi-dimensional arrays, an explicitly parallel SPMD model of computation with a global address space, a form of value class, and zone-based memory management. This reference manual describes the differences between Titanium and Java.

Contents

Preface

This document informally describes our current design for the Titanium language. It is in the form of a set of changes to Java. Unless otherwise indicated, the reader may assume the syntax and semantics of Java 2, version 1.4. Currently, however, the standard Java packages provided in Titanium officially comprise a subset of the standard version 1.0 Java library.

Acknowledgments. Many people have contributed to the Titanium project over the years, both as implementors and commentators. We especially wish to thank Peter McCorquodale, Phillip Colella, and Tong Wen (of the Lawrence Berkeley National Laboratory) for their many comments. Johnathon Jamison has also contributed numerous comments in the process of implementing an alternative front end for Titanium as part of the Harmonia project. Amir Kamil from the Titanium group has also commented extensively. Former members of the Titanium group who have contributed to this report include Alexander Aiken, Andrew Begel, Joe Darcy, and Luigi Semenzato.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract F30602-95-C-0136, the National Science Foundation under grant ACI-9619020, and the Department of Energy under contracts W-7405-ENG-48 and DE-AC03-765F00098. We would also like to thank Sun Microsystems and Microsoft for their support of related parallel-compiler work. The information presented here does not necessarily reflect the views of Sun Microsystems, Inc., Microsoft, Inc., or the position or policy of the Government, and no official endorsement should be inferred.

Chapter 1

Lexical Structure

Titanium adds the following new keywords to Java:

```
broadcast   foreach     immutable   inline      local
nonshared   op          overlap     partition   polyshared
sglobal     single      template
```

Chapter 2

Program Structure

Types introduced by Titanium are contained in the package `ti.lang` ('Ti' being the standard chemical symbol for Titanium). There is an implicit declaration

```
    import ti.lang.*;
```

at the beginning of each Titanium program.

The main procedure of a Titanium program must have signature

```
public single static void main(String single [] single args) {
  ...
}
```

or any variation thereof formed by removing one or more **single** qualifiers.

Chapter 3

New Standard Types and Constructors

3.1 Points

The immutable class `Point<`N`>`, for N a compile-time positive integer constant, is a tuple of N int's. `Point<`N`>`s are used as indices into N-dimensional arrays.

3.1.1 Operations on Points

In the following definitions, p and p_i are of type `Point<`N`>`; x, k, and k_i are integers;

1. p`[`i`]`, $1 \leq i \leq N$, is component i of p. That's right: the numbering starts at 1. It is an error for i to be out of bounds.

2. `[`$k_1, \ldots, k_n$`]` is a point whose component i is k_i.

3. `Point<`N`>.all(`x`)` is the `Point<`N`>` each of whose components is x.

4. `Point<`N`>.direction(`d, x`)` for $1 \leq |d| \leq N$, is the `Point<`N`>` whose component $|d|$ is $x \cdot \text{sign}(d)$, and whose other components are 0. x defaults to 1.

5. The arithmetic operators `+`, `-`, `*`, `/`, applied to two `Point<`N`>`s produce `Point<`N`>`s by componentwise operations. They are also defined between `Point<`N`>`s and (scalar) integers: for p a `Point<`N`>`, s a scalar, and $\oplus$ an arithmetic operator, $p \oplus s =$ $p \oplus$ `Point<`N`>.all(`s`)` and $s \oplus p =$ `Point<`N`>.all(`s`)` $\oplus$ p. The `/` operator, in contrast to its meaning on two scalar integer operands, rounds toward $-\infty$, rather than toward 0. It is an error to divide by 0.

6. Unary negation on a `Point<`N`>`, $-p$, produces the componentwise negation of p.

7. The assignment operators `+=`, `-=`, `*=`, `/=`, applied to a `Point<`N`>`s on the left and another `Point<`N`>` or a scalar on the right act like the assignment operators on scalars in standard Java. That is,

   p $\oplus$`=` E

   where E is a `Point<`N`>` or integer expression assigns the value of $p \oplus E$ to p, evaluating the location of p exactly once.

8. If R is any of `<`, `>`, `<=`, `>=`, or `==`, then p_0 R p_1 if p_0`[`i`]` R p_1`[`i`]` for all $1 \leq i \leq N$. The expression p_0`!=`p_1 is equivalent to `!(`p_0`==`p_1`)`. The member function `equals` on `Point<`N`>`s is the same as `==`

9. The expressions p_0`.lowerBound (`p_1`)` and p_0`.upperBound (`p_1`)` yield p such that p`[`i`]` is the minimum (respectively, maximum) of p_0`[`i`]` and p_1`[`i`]`.

10. The expression p_0`.permute (`p_1`)`, is the `Point<`N`>`, p, for which $p[p_1[i]] = p_0[i]$, where p_1 is a permutation of $1, \ldots, N$.

11. The expression p`.replace (`k, v`)` returns a `Point<`N`>` that is identical to p except that $p[k] = v$.

12. p`.arity` $= N$, and is a compile-time constant.

13. The expression p`.toString()` yields a text representation of p.

3.2 Domains and RectDomains

The type `Domain<`N`>`, for N a compile-time positive integer constant, is an arbitrary set of `Point<`N`>`s. The type `RectDomain<`N`>` is a "rectangular" set of `Point<`N`>`s: that is, a set

$$\{p \mid p_0 \leq p \leq p_1, \text{ and for some } x, \; p = p_0 + S{*}x\}$$

where all quantities here are `Point<`N`>`s. S here (a `Point<`N`>`) is called a *stride,* and is always greater than or equal to the *unit stride*, `Point<`N`>.all(1)`. The point p_0 is the *origin,* and p_1 the *maximum* of the set. `RectDomain<`N`>`s are used as the index sets (bounds) of N-dimensional arrays.

A `Domain<`N`>` is an Object, having a reference type, whereas the `RectDomain<`N`>` types are immutable classes (§5). As a result, a `Domain<`N`>` resides in the demesne (§6.1) in which it was created[1]. Library methods that yield `Domain<`N`>`s are not guaranteed to produce new values (i.e., values that do not compare `==` to previously created `Domain<`N`>`s). That is, the implementation is free to take advantage of the fact that there are no methods that mutate `Domain<`N`>`s, and to re-use any `Domain<`N`>` with the properties otherwise required of the specified result.

3.2.1 Operations on Domains and RectDomains

In the following descriptions, D is a `Domain<`N`>`, R is a `RectDomain<`N`>`, and RD may be either a `Domain<`N`>` or a `RectDomain<`N`>` (N is a positive integer.)

1. There is a standard (implicit) coercion from `RectDomain<`N`>` to `Domain<`N`>`.

2. RD`.isRectangular()` is true for all `RectDomain`s and for any `Domain` that is rectangular.

3. A `Domain<`N`>` may be explicitly converted to a `RectDomain<`N`>` using the usual conversion syntax: `(RectDomain<`N`>)` D. It is a run-time error if D is not rectangular.

4. If p_0 and p_1 are `Point<`N`>`s, then `[` p_0 : p_1 `]` is the `RectDomain<`N`>` with stride `Point<`N`>.all(1)`, origin p_0, and upper bound p_1. If s is also a `Point<`N`>`, $s \geq$ `Point<`N`>.all(1)`, then `[` $p_0 : p_1 : s$ `]` is the `RectDomain<`N`>` with origin p_0, upper bound p_1, and stride s. Because of the definitions of origin and upper bound at the beginning of this section, it follows that if not $p_0 \leq p_1$, then both `[` $p_0 : p_1$ `]` and `[` $p_0 : p_1 : s$ `]` are empty.

   So, for example, to get the set of all `Point<2>`s of the form $[i, j]$, with $0 \leq i < M$ and $0 \leq j < N$, we may use

   ```
   [ [0, 0] :  [M-1, N-1] ]
   ```

   and to get the subset of this set in which all coordinates are even we may write

   ```
   [ [0, 0] :  [M-1, N-1] :  [2, 2] ]
   ```

5. If i_j, k_j, and s_j, $1 \leq j \leq N$, are **int**s, with $s_j > 0$, then

   `[` $i_1 : k_1 : s_1, \ldots, i_N : k_N : s_N$`]`

   is the same as

[1] As a result, applying methods to a `Domain<`N`>` can carry unexpected performance penalties if the programmer does not pay attention to locality. Hence, recommended practice is to make a local copy of a `Domain<`N`>` using its copy constructor before invoking a sequence of operations upon it.

$$\texttt{[[} i_1, \ldots, i_N \texttt{] : [} k_1, \ldots, k_N \texttt{] : [} s_1, \ldots, s_N \texttt{]].}$$

and

$$\texttt{[} i_1 : k_1, \ldots, i_N : k_N \texttt{]}$$

is the same as

$$\texttt{[[} i_1, \ldots, i_N \texttt{] : [} k_1, \ldots, k_N \texttt{]].}$$

6. *RD*.`arity` $= N$, and is a compile-time constant.

7. The expression *RD*`.min()` yields a `Point<`N`>` such that *RD*`.min()[`k`]` is the minimum over all p in *RD* of p`[`k`]`. Likewise for *RD*`.max()`. For empty domains, *RD*`.min()` and *RD*`.max()` are not defined.

8. The expression *RD*`.lwb()` is a synonym for *RD*`.min()`. *RD*`.upb()` is defined as *RD*`.max()+`*RD*`.all(1)`, so that *RD*`.max()<`*RD*`.upb()`.

9. *RD*`.boundingBox()` yields the smallest `RectDomain<`N`>` that contains all the points of *RD* and is either empty or unit-strided. For `RectDomains` R with unit stride, R`.boundingBox()=`R.

10. R.`stride()` yields the minimal `Point<`N`>`, $s \geq$ `Point<`N`>`.all(1) such that R `=` `[`$p_0 : p_1 : s$`]` for some p_0 and p_1. (This need not be the same as the stride used to construct R in the first place: for example,

    ```
    ([ 0:1:2, 0:1:2 ]).stride()
    ```

    is `[1,1]`, not `[2,2]`.) It is undefined on an empty `RectDomain`.

11. *RD*.`size()` is the cardinality of *RD* (the number of Points it contains). The predicate *RD*.`isEmpty()` is true iff *RD*.`size()` $= 0$.

12. *RD*.`permute(`p_1`)` is the domain consisting of all points p.`permute(`p_1`)` for p in *RD*. It is a `RectDomain<`N`>` if *RD* is a `RectDomain<`N`>`. Otherwise it is a `Domain<`N`>`.

13. The operations `+`, `-`, and `*` are defined between any combination of `RectDomain<`N`>`s and `Domain<`N`>`s. They stand for union, difference, and intersection of the sets of elements in the operand domains. The intersection of two `RectDomain<`N`>`s yields a `RectDomain<`N`>`. All other operations and operands yield `Domain<`N`>`s.

14. For a `Point<`N`>`, p, the expressions RD`+`p, RD`-`p, RD`*`p, and RD`/`p compute the domains

$$\{d | d = d' \oplus p, \text{ for some } d' \in RD\}$$

where $\oplus$ = `+`, `-`, `*`, or integer division rounded toward $-\infty$ (unlike ordinary integer division, which rounds toward 0). Divisions require that each $p[i]$ be non-zero, and, if RD is a `RectDomain`, that each RD`.stride()`$[i]$ either be divisible by $|p[i]|$ or less than $|p[i]|$; it is an error otherwise. If RD is a `RectDomain`, so is the result.

15. The assignment expressions

R_1 `*=` R_2`;` D `*=` RD`;` D `+=` RD`;` D `-=` RD`;`

RD `+=` p`;` RD `-=` p`;` RD `*=` p`;` RD `/=` p`;`

where p is a `Point<`N`>`-valued expression, act like the assignment operators on scalars in standard Java. That is,

RD $\oplus$`=` V

assigns the value of $RD \oplus V$ to RD, evaluating the location of RD exactly once. Only the particular combinations shown are legal, since others would involve type errors.

16. The operations `<`, `>`, `<=`, and `>=` are defined between `RectDomains` and between `Domains` and represent set comparisons (`<` is strict subset, etc.).

The function `equals` is defined as set equality on all combinations of `Domain<`N`>` and `RectDomain<`N`>`. When the arities do not match, the result will always be **false**, even when the operands denote empty sets.

Between `RectDomains`, R_1`==`R_0 and R_1`!=`R_2 are synonyms for R_1`.equals(`R_2`)` and `!`R_1`.equals(`R_2`)`. The expression RD`.equals(`x`)` is true iff x is a `Domain<`N`>` that contains the same set of points as RD; it is false when x is any other reference value. As a result, its value is **false** if x is **null**, or does not denote a `Domain<`N`>` (so that again, even if RD and x both denote empty domains, they are not equal if their arities do not match).

17. The expression RD`.contains(`p`)`, for `Point<`N`>` p, is true iff $p \in RD$.

18. The expression R`.accrete(`k`,` dir`,` s`)` gives the result of adding elements to R on the side indicated by direction dir, expanding it by $k \geq 0$ strides of size $s > 0$ (all arguments but R are integers). That is, it computes

$R + (R + $`Point<`$N$`>.direction`$(dir, s)) + \ldots + (R + $`Point<`$N$`>.direction`$(dir, s \cdot k))$

and returns the result (always rectangular) as a `RectDomain`. It is an error if R could not have been constructed with a stride of s in direction *dir*, so that the resulting set of elements would not be a `RectDomain`. It follows that `accrete` is the identity on empty `RectDomain<`N`>`s. The argument s may be omitted; it defaults to 1. Requires that $1 \leq |dir| \leq N$.

19. The expression R`.accrete(`k`, `S`)` gives the result of expanding R on all sides by $k \geq 0$, as for the value V computed by the sequence

    ```
    V = R.accrete(k, 1, S[1]); V = V.accrete(k, -1, S[1]);
    V = V.accrete(k, 2, S[2])....
    ```

    The argument S is a `Point` with the same arity as R. It may be omitted, in which case it defaults to `all(1)`.

20. The expression R`.shrink(`k`, `*dir*`)` gives the result of shrinking R on the side indicated by direction *dir* by $k \geq 0$ strides of size R`.stride()[`$|dir|$`]` That is, it computes

    R`*`(R+`Point<`N`>.direction`(dir, s))`*`$\ldots$`*`(R+`Point<`N`>.direction`$(dir, -k \cdot s)$).

    where s is R`.stride()[`$|dir|$`]`. Requires that $1 \leq |dir| \leq N$.

21. The expression R`.shrink(`k`)` gives the result of shrinking R on all sides by $k \geq 0$ strides, as for the value V computed by the sequence

    ```
    V = R.shrink(k, 1).shrink(k, -1).shrink(k, 2)....
    ```

22. The expression R`.border(`k`, `*dir*`, `*shift*`)` is equivalent to

    R.accrete(k, *dir*) - R.boundingBox() + `Point<`N`>`.direction(*dir*, *shift*-k),

    where all arguments are of type `int` with $k \geq R$`.stride()[|`*dir*`|]` and $0 < |dir| \leq N$. When R has unit stride in direction *dir*, this consists of the k-thick layer of index positions on the side of R indicated by *dir*, shifted by *shift* positions in direction *dir*. Thus,

    ```
    R.border(1, dir, 0)
    ```

    is the layer of cells on the face of and interior to R in direction *dir*, while

    ```
    R.border(1, dir, 1)
    ```

    is the layer of cells just over that face (outside the interior of R). As shorthand,

R`.border(`k`, `dir`) = `R`.border(`k`, `dir`, 1)`

and

R`.border(`dir`) = `R`.border(1, `dir`, 1).`

23. The expression R`.slice(`k`)`, where $0 < k \leq N$, and $N > 1$ is a `RectDomain<`$N-1$`>` consisting of the set

$$\{[p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_N] \mid p = [p_1, \ldots, p_n] \in R\}.$$

24. The expression D`.RectDomainList()` returns a one-dimensional array (see §4.3) of type `RectDomain<`N`>[1d]` containing zero or more disjoint, non-null domains whose union (treated as `Domain<`N`>`s) is D.

25. The expression `Domain<`N`>.toDomain(`X`)`, where X is of type `RectDomain<`N`>[1d]` and has disjoint members, yields a `Domain<`N`>` consisting of the union of the elements of X.

26. The expression D.PointList() returns an array of type `Point<`N`>[1d]` of distinct points consisting of all the members of D.

27. The expression `Domain<`N`>.toDomain(`X`)`, where X is of type `Point<`N`>[1d]` and has distinct members, yields a `Domain<`N`>` whose members are the elements of X.

28. The expression RD`.toString()` yields a text representation of RD.

29. The expression `Domain<`N`>.setRegion(`reg`)`, where reg is a `Region`, is described in §6.3.2.

30. The type `Domain<`N`>` has a copy constructor: `new Domain<`N`>(`D`)` creates a new local copy of D. Likewise, `new (`W`) Domain<`N`>(`D`)` will create a local copy of D in region W.

3.2.2 'Foreach' Statements

The construct

`foreach (`p` in `RD`) `S

or equivalently

```
foreach (Point<n> p in RD) S
```

for RD any kind of n-dimensional domain, executes S repeatedly, binding p to the points in RD in some unspecified order. The control variable p is a local variable whose scope is confined to S, and it is constant (**final**) within its scope. As for other local variables in Java, you may not use a control variable name that shadows another local variable (including formal parameters and other control variables). The control constructs **break** and **continue** function in **foreach** loops analogously to other loops.

Chapter 4

New Type Constructors

4.1 Syntax

Titanium modifies Java syntax to allow for *grid types* (§4.3), and the qualifiers **local** (§6.1), **nonshared** (§6.2), **polyshared** (§6.2) and **single** (§9.4.1).

Type:
    QualifiedBaseType ArraySpecifiers$_{opt}$

QualifiedBaseType:
    BaseType Qualifiers$_{opt}$

ArraySpecifiers:
    ArraySpecifiers ArraySpecifier
    ArraySpecifier

ArraySpecifier:
    `[]` *Qualifiers*$_{opt}$
    `[` *IntegerConstantExpression* `d ]` *Qualifiers*$_{opt}$

Qualifiers:
    Qualifier Qualifiers
    Qualifier

Qualifier: **single** | **local** | **nonshared** | **polyshared**

BaseType: PrimitiveType | *ClassOrInterfaceType* where *PrimitiveType* and *ClassOrInterfaceType* are from the standard Java syntax. The grouping of qualifiers with the types they modify is as suggested by the syntax: In *QualifiedBaseType,* the qualifiers modify the base type. Array specifiers apply to their *QualifiedBaseType* from right to left: that is, '`T [1d][2d]`' is "1-D array of (2-D arrays of T)." The qualifiers in the array specifiers apply to the type resulting from the immediately preceding array specification. That is, for qualifiers Q_1, Q_2, Q_3,

`Object` Q_3 `[]` Q_1 `[]` Q_2 `V;`

defines `V` to be a Q_1 array of Q_2 arrays of Q_3 `Objects`. We call Q_1 the *top-level qualifier,* and Q_3 the *base qualifier.*

The qualifier **single** is outwardly contagious. That is, the type "array of single T" (`T single [...]`) is equivalent to "single array of single T" (`T single [...] single`), and the type "array of single array of array of T" (`T [...] [...] single [...]`) is equivalent to "single array of single array of array of T" (`T [...] single [...] single [...]`).

4.2 Atomic types

An *atomic type* is either a primitive type (**int**, **double**, etc.), or an immutable class (see §5) whose non-static fields all have atomic type.

4.3 Arrays

A Java array (object) of length N—hereafter called a *standard array*—is an injective mapping from the interval of non-negative integers $[0, N)$ to a set of variables, called the *elements* of the array. Titanium extends this notion to *grid arrays* or *grids;* an N-dimensional grid is an injective mapping from a `RectDomain<`N`>` to a set of variables. As in Java, the only way to refer to any kind of array object—standard or grid—is through a pointer to that object. Each element of a standard array resides in (is mapped to from) precisely one array[1]. The elements of a grid, by contrast, may be shared among any number of distinct grids. That is, even if the array pointers `A` and `B` are distinct, `A[p]` and `B[p]` may still be the same object. We say then that `A` and `B` *share* that element.

It is illegal to cast grid values to the type `Object`. However, as for other normal objects, the value **null** is a legal value for a grid variable, and it is legal to compare grid quantities of the same arity using `==` or `!=`. If A and B are two grids of the same arity, then A`==`B

[1]This does *not* mean that there is only one name for each element. If `X` and `Y` are arrays (of any kind), then after '`X=Y;`', `X[p]` and `Y[p]` denote the same variable. That is an immediate consequence of the fact that arrays are always referred to through pointers.

iff A`.domain()==`B`.domain()` and for every p in their common domain, A`[`p`]` and B`[`p`]` are the same variable. Similarly for `!=`.

For a non-grid type T, the type "N-dimensional grid with element type T" is denoted

T`[`N`d]`

where N is a positive compile-time integer constant. When that constant is an identifier, it must be separated from the 'd' by a space[2]. To declare a variable that may reference two-dimensional grids of **double**s, and initialize it to a grid indexed by the `RectDomain<2>` D, one might write

```
double[2d] A = new double[D];
```

For the type "N_0-dimensional grid whose elements are N_1-dimensional grids. . . whose elements are of type T," we write

T`[`N_0`d][`N_1`d]`$\cdots$.

There is a reason for this irregularity in the syntax (where one might have expected a more compositional form, in which the dimensionalities are listed in the opposite order). The idea is to make the order of indices consistent between type designations, allocation expressions, and array indexing. Thus, given

```
double[1d][2d] A = new double[D1][D2];
```

we access an element of `A` with

```
A[p1][p2]
```

where `D1` is a `RectDomain<1>`, `D2` a `RectDomain<2>`, `p1` a `Point<1>`, and `p2` a `Point<2>`[3].

Two grid types are assignment compatible only if they are identical, aside from narrowing and widening conversions on top-level qualifiers (see §6.1 and §6.2.4). For Java arrays, assignment conversion of one array-of-reference type to another is also allowed if the element types are assignment convertible and have the same local, sharing, and single qualification.

Grid types are (at least for now) not quite complete Java Objects, in that they may not be coerced to type Object. This restriction aside, they are otherwise reference types, and are subject to reference qualification (see §4).

[2]It is true that when N is an integer literal, N`d` is syntactically indistinguishable from a floating-point constant in Java. However, in this context, such a constant would be illegal.

[3]Take these examples with a grain of salt. In general, it is preferable to use a `RectDomain<3>` index set in preference to the array-of-arrays seen here, if the algorithm justifies it, because compilers are apt to do better with the former.

4.3.1 Array Operations

In the following, take $p, p_0, \ldots$ to be of type `Point<`N`>`, A to be a grid of some type T `[`N`d]`, and D to be a `RectDomain<`N`>`.

1. A`.domain()` is the domain (index set) of A. It is a `RectDomain<`N`>`.

2. A`[`p`]` denotes the element of A indexed by p, assuming that p is in A`.domain()`. It is an lvalue—an assignable quantity—unless A is a global array of local references, in which case it is unassignable.

3. The expression A`.copy(`B`)` copies the contents of the elements of B with indices in `A.domain()*B.domain()` into the elements of A that have the same index. It is illegal if A and B do not have the same grid type. It is also illegal if A is a global array whose element type has local qualification (it is easy to construct instances in which such a copy would be unsound). Finally, it is illegal if B resides in a private region, A resides in a shared region (see §6.3), and the elements of B have a non-atomic type (see §4.2). (This last provision is a conservative restriction to prevent situations where objects allocated in shared regions contain references to objects in private regions.) It *is* legal for A and B to overlap, in which case the semantics are as if B were first copied to a (new) temporary before being copied to A. See also the operations for sparse-array copying described in §12.15, and the operations for non-blocking array copying described in §12.16.

4. `A.arity` is the value of A`.domain().arity` when `A` is non-null, and is a compile-time constant.

5. If $i_1, \ldots, i_N$ are integers, then A`[`$i_1, \ldots, i_N$`]` is equivalent to A`[ [`$i_1, \ldots, i_N$`] ]`. (Syntactic note: this makes `[...]` similar to function parameters; applications of the comma operator must be parenthesized, unlike C/C++)

6. The following operations provide remappings of arrays.

   (a) A`.translate(`p`)` produces a grid, B, whose domain is `A.domain()+`p, such that B`[`p`+`x`]` aliases A`[`x`]`.

   (b) A`.restrict(`R`)` produces the restriction of A to index set R. R must be a `RectDomain<`N`>`.

   (c) A`.inject(`p`)` produces a grid, B, whose domain is `A.domain()*`p, such that B`[`p`*`x`]` aliases A`[`x`]`.

   (d) A`.inject(`p`).project(`p`)` produces A. For other arguments, `project` is undefined.

(e) A`.slice(`k, j`)` produces the grid, B, with domain A`.domain().slice(`k`)` such that

$$B\texttt{[[}p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n\texttt{]] = }A\texttt{[[}p_1, \ldots, p_{k-1}, j, p_{k+1}, \ldots, p_n\texttt{]]}.$$

It is an error if any of the index points used to index A are not in its domain, or if A`.arity`≤ 1.

(f) `A.permute(`p`)`, where p is a permutation of $1, \ldots, N$, produces a grid, B, where A`[`$i_1, \ldots, i_N$`]` aliases B`[`$i_{p[1]}, \ldots, i_{p[N]}$`]`.

7. `A.set(`v`)`, where v is an expression of type T, sets all elements of A to v. This operation is illegal if A is global and T has local qualification. It is also illegal if A resides in a shared region, v resides in a private region, and T is a non-atomic type (see §4.2).

8. `A.exchange(`v`)`, where v is an expression of type T, is a barrier operation that fills in A with the same values on all processes, with each process contributing its value of v as the value of one element. See §9.2.1.

9. `A.isContiguous()` is true iff all elements of A are adjacent to each other in virtual memory. This property has no semantic significance, but can be important for tuning performance. Arrays are created fully contiguous and in row-major order[4]. Operations that produce views of subsets of the data can result in non-contiguous results.

10. Because of the close relationship between grids and their domains, a number of common idioms involve operations on both. As a convenience, several methods capture these idioms[5]:

(a) `A.size()` is equivalent to `A.domain().size()`.

(b) `A.isEmpty()` is equivalent to `A.domain().isEmpty()`.

(c) `A.shrink(`k, dir`)` is equivalent to `A.restrict(A.domain ().shrink(`k, dir`))`.

(d) `A.shrink(`k`)` is equivalent to `A.restrict(A.domain ().shrink(`k`))`.

(e) `A.border(`$k, dir, shift$`)` is equivalent to
`A.restrict(A.domain().border(`$k, dir, shift$`))`.

(f) `A.border(`k, dir`)` is equivalent to `A.restrict(A.domain().border(`k, dir`))`.

(g) `A.border(`dir`)` is equivalent to `A.restrict(A.domain().border(`dir`))`.

[4]However, `A.isContiguous()` does not imply that `A` is in row-major order. The `permute` method, for example, preserves contiguity but not row-major order.

[5]As of version 3.110 of the compiler.

11. The I/O operations described in §12.17.1.
12. The sparse-copying operations described in §12.15.
13. The non-blocking copy operations described in §12.16.

4.3.2 Overlapping Arrays

As for any reference type in Java, two grid variables can contain pointers to the same grid. In addition, several of the operations above produce grids that reference the same elements. The possibility of such *overlap* does not sit well with certain code-generation strategies for loops over grids. Using only intraprocedural information, a compiler can sometimes, in principle, determine that two grid variables do not overlap, but the problem becomes complicated in the presence of arbitrary data structures containing grid pointers, and when one or more grid variable is a formal parameter.

For these reasons, Titanium has a few additional rules concerning grid parameters:

1. Formal grid parameters to a method or constructor (including the implicit **this** in the case of methods defined on grids) may not overlap unless otherwise specified—that is, given two formals F_1 and F_2, none of the variables $F_1[p_1]$ may be the same as $F_2[p_2]$. It is an error otherwise.

2. The qualifier '`overlap(`F_1`,`F_2`)`' immediately following a method header, where F_1 and F_2 name formal parameters of that method, means that the restriction does not apply to F_1 and F_2. (There is no restriction that F_1 and F_2 have the same type or even that they be grids. When they are not grids with the same element type, however, the qualifier has no effect.)

4.3.3 Restrictions on Standard Arrays

The possibilities of local qualification and of residence in private regions require additional restrictions on the standard array method `System.arraycopy`, analogous to those on the `.copy` operation for grids. For standard arrays A and B, the call

```
System.arraycopy (A, k0, B, k1, len)
```

is illegal if A is a global array whose element type has local qualification. It is also illegal if A resides in a private region, B resides in a shared region (see §6.3), and the elements of A have a non-atomic type (see §4.2).

Chapter 5

Immutable Classes

Immutable classes (or *immutable types*) extend the notion of Java primitive type to classes. An immutable class is a final class that is not a subclass of Object and whose non-static fields are final. It is declared with the usual class syntax and an extra modifier

```
    immutable class C { ... }
```

They differ from ordinary classes in the following ways:

1. Non-static fields are implicitly final, but need not be explicitly declared as final. Because all immutable classes are final and are not subtypes of `Object`, they may never have an **extends** or **implements** clause.

2. Because immutable classes are not subclasses of any other class, their constructors may not call `super()` explicitly, and, contrary to the usual rule for Java classes other than Object, do not do so implicitly either.

3. There are no type coercions to or from any immutable class, aside from those defined on standard types in the Titanium library (see §3.2). The standard classes `Point<`N`>` and `RectDomain<`N`>` are immutable.

4. The value **null** may not be assigned to a variable of an immutable class. Each class variable, instance variable, or array component with immutable type T is initialized to the *default value* for that type. For each process, this default value initially consists of the value in which all the instance fields are set to their respective default values. The language inserts a static initializer at the end of the definition of T whose effect is to evaluate `new` T`()` and then set the default value of T for the process to the result. Initialization of the fields in objects of these types otherwise follows the rules of Java 1.4.

5. Circular chains of immutable types are disallowed. That is, no immutable type may contain a non-static subcomponent of its own type. Here, a *non-static subcomponent* is a non-static field, or a non-static subcomponent of a field, where the field has an immutable type.

6. By default, the operators `==` and `!=` for a type `T` are defined by comparison of the non-static fields of the operands using `==` and `!=`, as if defined

   ```
   public boolean single op==(T single x) {
      return this.F1 == x.F1 && this.F2 == x.F2 && ...
             && this.Fk == x.Fk;
   }
   public boolean single op!=(T single x) {
      return this.F1 != x.F1 || this.F2 != x.F2 || ...
             || this.Fk != x.Fk;
   }
   ```

   where the Fk are the non-static fields of `T` in declaration order. (This implies that if `==` or `!=` has been overridden on the type of a field, it is the new, overriding definition that is used for the implicit comparison.) These default definitions of `==` and `!=` are not accessible via calls to `op==` or `op!=`. Any `void finalize()` method supplied with an immutable object is ignored.

7. As a consequence of these rules, it is impossible, except in certain pathological programs, to distinguish two objects of an immutable class that contain equal fields. The compiler is free to ignore the possibility of pathology and unbox immutable objects.

8. Nested immutable classes must be static, and may not be local. That is, whereas it is legal to have an immutable class that is nested immediately inside a class, it is not legal to nest an immutable class in a block.

9. The '+' operator (concatenation) on type `String` is extended so that if S is a string-valued expression and X has an immutable type, then S+X is equivalent to S+X`.toString()` and X+S is equivalent to X`.toString()`+S. These expressions are therefore illegal if there is no `toString` method defined for X, or if that method does not return a `String` or `String local`.

10. An immutable class must have a zero-argument constructor with at least as much access as the class itself. If no such constructor is provided by the user, the compiler automatically generates a publically accessible zero-argument constructor with an empty method body.

Chapter 6

Pointers and Storage Allocation

6.1 Demesnes: Local vs. Global Pointers

In Titanium, the set of all memory is the union of a set of local memories, called *demesnes* here to give them a veneer of abstraction. Each object resides in one demesne. Each *process* (§9) is associated with one demesne—called simply the *demesne of the process.* A local variable resides in the demesne of the process that allocates that variable. An object created by **new** and the fields within it reside in the demesne of the process that evaluates the **new** expression.

The intent is that on a uniprocessor implementation, there will be a single demesne, and likewise on a shared-memory multiprocessor. On a pure distributed-memory multiprocessor, there would be a single demesne corresponding to each processor. Finally, on a distributed cluster of shared-memory multiprocessors, there would be one demesne for the processes on each shared-memory node.

The static types ascribed to variables (locals, fields, and parameters) containing reference values and to the return values of functions that return reference values are either *global* or *local.* A variable having a local type will contain only null or pointers whose demesnes are the same as that of the variable. A pointer contained in a variable with global type may have any demesne. A locally qualified variable may be assigned only a value whose type is (statically) local. The purpose of the distinction between local and global references is to improve performance. Dereferencing a global reference requires a test to see whether the referenced datum is accessible with a native pointer; communication is needed if it is not. Local references are for those data known to be accessible with a native pointer. Global references can be used anywhere but local references don't travel well.

Standard Java type designators for reference types denote global types. The modifier keyword **local** indicates a local reference. For examples:

```
Integer i1;                    /* i is a global reference */
```

```
    Integer local i2;            /* i is a local reference */
    int local i3;                /* illegal: int not a reference type */
```

For grids, there is an additional degree of freedom:

```
    /* A1 is a global pointer to an array of variables that may reside
     * remotely (i.e., in a different demesne from the variable A1). */
    int[1d] A1;
    /* A2 is a local pointer to an array of variables that reside
     * locally (in the same demesne as A2). */
    int[1d] local A2;

    /* A3 is a local pointer to an array of variables that reside
     * locally and contain pointers to objects that may reside
     * remotely. */
    Integer [1d] local A3;
    /* A4 is a local pointer to an array of variables that reside
     * locally and contain pointers to objects that reside locally */
    Integer local [1d] local A4;
    /* A5 is a global pointer to an array of variables that may
     * reside remotely and contain pointers to objects that
     * may reside remotely */
    Integer [1d] A5;
    /* A6 is a global pointer to an array of variables that may
     * reside remotely and contain pointers to objects that reside
     * in the same demesne as that array of variables. */
    Integer local [1d] A6;
```

We refer to a reference type apart from its local/global attribute as its *object type*.

Coercions between reference types are legal if, first, their object types obey the usual Java restrictions on conversion (plus Titanium's more stringent rules on arrays). Second, a reference value may only be coerced to have a local type (by means of a cast) if it is null or denotes an object residing in the demesne of the process performing the coercion. It is an error to execute such a cast otherwise. Coercions from local to corresponding global types are implicit (they extend assignment and method invocation conversion). Global to local coercions must be expressed by explicit casts.

If a field selection `r.a` or `r[a]` yields a reference value and `r` has static global type, then so does the result of the field selection.

All reference classes support the following operations:

`r.creator()` Returns the identity of the lowest-numbered process whose demesne contains the object pointed to by `r`. NullPointerException if `r` is null. This number may differ from the identity of the process that actually allocated the object when multiple processes share a demesne.

`r.isLocal()` Returns true iff `r` may be coerced to a local reference. This returns the same value as `r instanceof T local`, assuming `r` to have object type `T`. On some (but not all) platforms `r.isLocal()` is equivalent to `r.creator() == Ti.thisProc()`.

`r.regionOf()` See §6.3.2.

`r.clone()` Is as in Java, but with local references inside the object referenced by `r` set to null. The result is a global pointer to the newly created clone, which resides in the same region (see §6.3.2) as the object referenced by `r`. This operation may be overridden.

`r.localClone()` Is the default `clone()` function of Java (a shallow copy), except that `r` must be local. The result is local and in the same region (see §6.3.2) as the object referenced by `r`.

To indicate that the special variable **this** in a method body is to be a local pointer, label the method local by adding the `local` keyword to the method qualifiers:

```
public local int length () {...}
```

Otherwise **this** is global. If necessary, the value for which a method is invoked is coerced implicitly to be global. A static method may not be local.

6.2 Data Sharing

In Titanium, the type of any reference includes information about how the referent is shared among multiple processes. The **nonshared** and **polyshared** qualifiers encode this information. By default, most data is assumed to be shared; there is no explicit **shared** qualifier. A **nonshared** qualifier declares that the reference addresses data that is not shared: it is only accessible by processes within the same demesne as the data itself. A **polyshared** qualifier indicates that the reference addresses data that may be shared or may be non-shared.

6.2.1 Basic Syntax

The **nonshared** and **polyshared** qualifiers can modify any reference type, including named reference classes, named interfaces, Java arrays, and Titanium arrays. Sharing qualifiers on arrays may be set independently for elements and for the array as a whole. Thus:

```
  /* a shared object */
  Object x;

  /* a non-shared object */
  Object nonshared y;

  /* either a shared or a non-shared array of non-shared objects */ \\
 Object nonshared [] polyshared z;
```

Sharing qualifiers may modify types in any context where types are expected, including field declarations, variable declarations, formal parameter declarations, method return type declarations, catch clauses, cast expressions, **instanceof** expressions, and template parameters. The **nonshared** qualifier can be used in allocation expressions to allocate non-shared data rather than the default shared data. However, the **polyshared** qualifier may not be used to directly allocate data with unknown sharing; this is a compile-time error: allocated data must be either shared or non-shared at the topmost level. Thus:

```
  /* ok: a shared object */
  Object a = new Integer(6);

  /* ok: a non-shared object */
  Object nonshared b = new Integer nonshared (6);

  /* error: cannot allocate polyshared data */
  Object polyshared z = new Integer polyshared (6);

  /* ok: a shared array of polyshared objects */
  Object polyshared [] c = new Object polyshared [10];

  /* ok: a non-shared array of polyshared objects */
  Object polyshared [] nonshared d = new Object polyshared [10] nonshared;

  /* error: cannot allocate polyshared data */
  Object polyshared [] polyshared e = new Object polyshared [10] polyshared;
```

Reference types can have zero or one sharing qualifier: it is an error to qualify any type as both **nonshared** and **polyshared**. Primitive and immutable types may not be qualified as either **nonshared** or **polyshared**.

6.2.2 Methods

Within a reference class, a sharing qualifier may also appear among the qualifiers for a non-static method or constructor, in which case it applies to the implicit **this** parameter within the method or constructor body. It is an error to add any sharing qualifier to a static method, immutable method, or immutable constructor. It is an error to add more than one sharing qualifier to a reference class method.

Two methods that vary in the sharing qualifiers on their formal parameters or on the implicit **this** parameter are always considered distinct types for purposes of method overloading and overriding. A method with return type R_1 may override one with return type R_2 only if:

- If R_1 is void then R_2 is void.
- If R_1 is a primitive or immutable type, then R_1 is identical either to R_2 or to R_2 `single`.
- Otherwise, R_1 is assignment compatible with R_2.

6.2.3 Implicit Sharing Qualifiers

In several instances, the sharing status of entities is implicit:

1. Within field initialization expressions of reference classes, **this** is assumed to be polyshared.
2. If the compiler provides a default constructor (Java spec §8.8.7), then this constructor is assumed to be polyshared.
3. String literals are assumed to be shared.
4. The null type can be converted to any shared, non-shared, or polyshared reference type, so the sharing qualification of the `null` literal is unspecified and irrelevant.

In all other contexts, unqualified types are assumed to be shared.

6.2.4 Conversion

Standard Java conversion rules are modified as follows.

All widening reference conversions (Java spec §5.1.4) are restricted to apply only when at least one of the following conditions hold:

1. the source type and destination type have identical top-level sharing qualifiers (that is, the sharing qualifiers that apply to the types as a whole—as opposed to the types of components—are identical).
2. the source type is the null type

3. the destination type's top-level sharing qualifier is **polyshared**

Widening reference conversion of array types is further constrained to apply only when the top-level sharing qualifiers of the element types are identical. For example:

```
/* ok: (top-level) destination type is polyshared */
Object polyshared o1 = new Integer nonshared (7);

/* ok: top-level destination type is polyshared, element types
 * match.  */
Object nonshared [] polyshared o2 =
   new Object nonshared [10] nonshared;

Object nonshared [] polyshared [] nonshared o3 =
   new Object nonshared [10] nonshared [10] nonshared;

/* error: top-level qualifiers on elements do not match:
 * (polyshared vs. non-shared) */
Object polyshared [] o4 = new Integer nonshared [];

Object nonshared [] nonshared [] polyshared o3 =
   new Object nonshared [10] nonshared [10] nonshared;
```

All narrowing reference conversions (Java spec §5.1.5) are restricted to apply only when the source type and destination type have identical top-level sharing qualifiers. Narrowing reference conversion of array types is further constrained to apply only when the top-level sharing qualifiers of the element types are identical. Informally, one cannot convert polyshared data back to shared or non-shared data, even using a run time test.

6.2.5 Restricted Operations

Non-shared data must only be used by processes in the same demesne. Local references carry no special restrictions, as the data to which they refer is already known to reside in the appropriate demesne. Any reference to shared data is similarly unconstrained. However, a global reference to non-shared or polyshared data is restricted. If `p` is a global reference to non-shared or polyshared data, then the following operations are forbidden:

1. reading the value of any field of `p`;
2. reading the value of any element of `p`, if `p` is an array;
3. assigning into any field of `p`;

4. assigning into any element of `p`, if `p` is an array;
5. calling any non-static method of `p`;
6. evaluating `p instanceof T` for any type `T`;
7. performing a narrowing reference conversion of `p` to a global type;
8. using a **synchronized** statement to acquire the mutual exclusion lock of `p`;
9. using `p` within a string concatenation expression.

The restriction on narrowing reference conversion does allow casting to a local type. It merely forbids checked casts that do *not* simultaneously recover localness.

6.2.6 Early Enforcement

Titanium contains two alternative definitions of sharing. Late enforcement is the more relaxed of the two, and entails only those restrictions already listed above. Early enforcement entails the following additional restrictions:

1. Any type that is non-shared or polyshared must also be local; global pointers may only address shared data.
2. If any use of a named class is (implicitly) qualified as shared, then all fields embedded within that class must be qualified as shared. "Embedded" here is defined as:
   (a) If class C defines field $C.f$, then field $C.f$ is embedded within class C.
   (b) If class C has superclass D, then all fields embedded within class D are also embedded within class C.
   (c) If class C has embedded field $B.f$, and $B.f$ has immutable type I, then fields embedded within immutable class I are also embedded within class C.
3. If any use of an array type is (implicitly) qualified as shared, then the elements of the array must be shared as well.

As a result of these additional restrictions, nonshared data is only accessible within the process that creates it, whereas under late enforcement, nonshared data is available to any process in its demesne.

The relative merits of early and late enforcement were not originally clear. Late enforcement was originally the implemented policy, but the preferred policy is early enforcement. At some point this will become the default and the compiler will provide late enforcement as an option.

6.3 Region-Based Memory Allocation

Java uses garbage collection to reclaim unreachable storage. Titanium retains this mechanism but also includes a more explicit (but still safe) form of memory management: *region-based* memory allocation.

In a region-based memory allocation scheme, each allocated object is placed in a program-specified *region*. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. A simple example is shown in Figure 6.3. Each iteration of the loop allocates a small array. The call `r.delete()` frees all arrays.

```
class A {
  void f()
  {
    PrivateRegion r = new PrivateRegion();

    for (int i = 0; i < 10; i++) {
      int[] x = new (r) int[i + 1];
      work(i, x);
    }
    try {
        r.delete();
    }
    catch (RegionInUse oops) {
        System.out.println("oops - failed to delete region");
    }
  }

  void work(int i, int[] x) { }
}
```

Figure 6.1: An example of region-based allocation in Titanium.

A region r can be deleted only if there are no *external* references to objects in r (a reference external to r is any pointer not stored within r). A call to `r.delete()` throws an exception when this condition is violated.

6.3.1 Shared and Private Regions

There are two kinds of regions: *shared* regions and *private* regions. Objects created in a shared region are called *shared-region objects*; all other objects are called *private objects*.

Garbage-collectible objects are taken to reside in an anonymous shared region. It is an error to store a reference to a private object in a shared-region object. It is also an error to broadcast or exchange a private object. As a consequence, it is impossible to obtain a private pointer created by another process.

All processes must cooperate to create and delete a shared region, each getting a copy of the region that represents the same, shared, pool of space. The copy of the shared region object created by a process p is called the *representative* of that region in process p (see the `Object.regionOf` method below). Creating and deleting shared regions thus behaves like a barrier synchronization and is an operation with global effects (see §9.4.1).

A region is said to be *externally referenced* if there is a reference to an object allocated in it that resides in

1. A live local variable;
2. A static field;
3. A field of an object in another region.

The process of attempting to delete a region r proceeds as follows:

1. If r is externally referenced, throw a `ti.lang.RegionInUse` exception.
2. Run the `finalize` methods of all objects in r for which it has not been run.
3. If r is now externally referenced, throw a `ti.lang.RegionInUse` exception.
4. Free all the objects in r and delete r.

Garbage-collected objects behave as in Java. In particular, deleting such objects differs from the description above in that finalization does not wait for an explicit region deletion.

6.3.2 Details of Region-Based Allocation Constructs

Shared regions are represented as objects of the `ti.lang.SharedRegion` type, private regions as objects of the `ti.lang.PrivateRegion` type. The signature of the types is as shown in Figure 6.2.

The Java syntax for `new` is redefined as follows, where `T` is a type that is distinct from `ti.lang.PrivateRegion` and `ti.lang.SharedRegion`:

1. `new ti.lang.PrivateRegion()` or `new ti.lang.SharedRegion()` creates a region containing only the object representing the region itself.
2. `new T...` allocates a garbage-collected object, as in Java.

```
package ti.lang;

final public class PrivateRegion extends Region
{
  public PrivateRegion() { }
  /** Run finalization on all unfinalized objects in THIS.
   *  Frees all resources used to represent objects in THIS.
   *  Throws RegionInUse if THIS is externally referenced
   *  before or after finalization. */
  public void delete() throws RegionInUse;
};

final public class SharedRegion extends Region
{
  public single SharedRegion() { }
  /** See PrivateRegion.delete, above. */
  public single void delete() throws RegionInUse single;
};

abstract public class Region
{
};
```

Figure 6.2: Library definitions related to regions.

3. `new (expression) T...` creates an object in the region specified by `expression`. The static type of `expression` must be assignable to `ti.lang.Region`. At runtime the value v of `expression` is evaluated. If v is:
   (a) `null`: allocates a garbage-collected object, as in Java.
   (b) an object of type `ti.lang.PrivateRegion` or `ti.lang.SharedRegion`: allocates an object in region v.
   (c) In all other cases a runtime error occurs.

The class `java.lang.Object` is extended with the following method:

```
    public final ti.lang.Region local regionOf();
```

This returns the region of the object, or `null` for garbage-collected objects. For shared-region objects, the local representative of the shared region is returned.

The expression `Domain<`N`>.setRegion(`*reg*`)`, where *reg* is a `Region`, dynamically makes *reg* the `Region` used for allocating all internal pointer structures used in representing domains (until the next call of `setRegion`). A null value for `reg` causes subsequent allocations to come from garbage-collected storage. Returns the previous value passed to `setRegion` (initially **null**). The value of `N` is irrelevant; all general domains are allocated in the same region.

Chapter 7

Templates

Titanium uses a "Templates Light" semantics, in which template instantiation is a somewhat augmented macro expansion, with name capture and access rules modified as described below.

7.1 Instantiation Denotations

Define

TemplateInstantiation:
    Name `"<"` *TemplateActual* { `","` *TemplateActual* }* `">"`
TemplateActual:
    Type | *AdditiveExpression*

where the AdditiveExpression is a ConstantExpression. Resolution of Name is as for type names. [Note: We use AdditiveExpression to prevent non-LALRness with `<` and `>` operators.]

7.2 Template Definition

Template definitions occur at the outermost level; they are never nested inside other class, interface, or template definitions.

CompilationUnit:
    PackageDeclaration$_{opt}${ *ImportDeclaration* }* { *OuterDeclaration* }*

OuterDeclaration:
  TypeDeclaration
  TemplateDeclaration
TemplateDeclaration:
  TemplateHeader ClassDeclaration
  TemplateHeader InterfaceDeclaration
TemplateHeader:
  template `"<"` *TemplateFormal* { `","` *TemplateFormal* }`*` `">"`
TemplateFormal:
  class *Identifier*
  PrimitiveType Identifier

The first form of TemplateFormal allows any type as argument; the second allows ConstantExpressions of the indicated type.

Templates introduce new opportunities for creating illegal circular dependencies among types. In Java, it is a compile-time error if a class depends on itself. In Titanium, the definition of *directly depends* is extended as follows:

> A class C directly depends on a type T if T is mentioned in the **extends** or **implements** clause of C either as a superclass or superinterface, as a qualifier within a superclass or superinterface name, or as a qualifier of a parameter of a template superclass or superinterface.

Otherwise, the definition of *depends* is as in Java: A class C depends on a reference type T if any of the following conditions hold:

1. C directly depends on T.
2. C directly depends on an interface I that depends on T.
3. C directly depends on a class D that depends on T (using this definition recursively).

7.3 Names in Templates

A template belongs to a particular package, as for classes and interfaces. Access rules for templates themselves are as for similarly modified classes and interfaces.

Template instantiations belong to the same package as the template from which they are instantiated. As a result, it is essentially useless to instantiate a template from a different package except with public classes and interfaces[1].

Names other than template parameters in a template are captured at the point of the template definition. Names in a TemplateActual (and at the point it is substituted for in a template instantiation) are resolved at the point of instantiation.

7.4 Template Instantiation

References to TemplateInstantiation are allowed as Types. Instantiations that cause an infinite expansion or a loop are compile-time errors. Inside a template, one may refer to the "current instantiation" by its full name with template parameters; or by the template's simple name. The constructor is called by the simple name.

```
template<class T> class List {
    ...
    List (T head, List tail) {...}

    List tail() { ... }
}
```

or

```
template<class T> class List {
    ...
    List (T head, List<T> tail) {...}

    List<T> tail() { ... }
}
```

7.5 Name Equivalence

For purposes of type comparison, all simple type names and template names that appear as TemplateActuals are replaced by their fully-qualified versions. With this substitution, two type names are equivalent if their QualifiedName parts are identical and their TemplateActuals are identical (identical types or equal values of the same type).

[1]An alternative rule (not currently implemented) states that template instantiations have package-level access to all classes and interfaces mentioned in the template actuals, so that the expansion of a template might be illegal according to the usual rules (e.g., the template expansion could reside in package `P` and yet contain fields whose types were non-public members of package `Q`). The rationale for this alternative is that otherwise, a package of utility templates would be useless unless one was willing to make public all classes used as template parameters.

Chapter 8

Operator Overloading

Titanium allows overloading of the following Java operators as instance methods of classes and interfaces.

Unary prefix	`- ! ~`
Binary	`< > <= >=`
	`+ - * / & | ^ % << >> >>>`
Matchfix	`[] []=`
Assignment	`+= -= *= /= &= |= ^= %= <<= >>= >>>=`

On immutable types, furthermore, one may also overload the binary operators `==` and `!=`. They must have exactly the following signatures:

```
public boolean single op==(T single I);
public boolean single op!=(T single I);
```

where `T` is the enclosing class.

If $\oplus$ is one of these operators, then declaring `op`$\oplus$ produces a new overloading of that operator, replacing the corresponding default definition of the operator in the case of `==` and `!=` on immutable types (so when `op==` is overloaded, the default definition of `==` is unavailable). No space is allowed between the keyword **op** and the operator name. These new methods can be called like normal methods, e.g. `a.op+(3)`. There are no restrictions on the number of parameters, parameter types or result type of operator methods.

Suppose that E, E_0, E_1 are expressions, T_0 is the static type of E_0, and $\oplus$ is a unary prefix or binary operator. Then the following re-writings apply to syntactically valid expressions when the re-written expression is also legal according to other language rules:

1. $E_0 \oplus E_1 \quad\Longrightarrow\quad (E_0).\texttt{op}\oplus(E_1)$
2. $E_0 \oplus E_1 \quad\Longrightarrow\quad (E_1).\texttt{op}\oplus(E_0, E_1)$, if T_0 is primitive.
3. $\oplus E_0 \quad\Longrightarrow\quad E_0.\texttt{op}\oplus()$
4. $E_0\texttt{[}E_1, \ldots, E_n\texttt{]} \texttt{ = } E \quad\Longrightarrow\quad (E_0)\texttt{.op[]=}(E_1, \ldots, E_n, E)$
5. $E_0\texttt{[}E_1, \ldots, E_n\texttt{]} \quad\Longrightarrow\quad E_0\texttt{.op[]}(E_1, \ldots, E_n)$, in other contexts

When $\oplus$= is an assignment operator:

6. $E_0 \;\oplus\texttt{=}\; E_1 \quad\Longrightarrow\quad E_0 \texttt{ = (}T_0\texttt{)}(E_0).\texttt{op}\oplus\texttt{=}(E_1)$
7. $E_0 \;\oplus\texttt{=}\; E_1 \quad\Longrightarrow\quad E_0 \texttt{ = (}T_0\texttt{) } (E_1).\texttt{op}\oplus\texttt{=}(E_0, E_1)$, if T_0 is primitive.

In all cases, however, the expressions E_0, E_1, and E are evaluated in the order they appear in the original expression, and each is evaluated only once. It is not possible to redefine plain assignment (`=`). As a consequence of the legality requirements, E_0 must be an lvalue in cases (6) and (7), and T_0 may be a primitive type only in cases (2) and (7).

There is a conflict between the description above and the standard Java description of how the '`+`' operator works when the right argument is of type String, which arises whenever the programmer defines `op+` with an argument of type String. We resolve this by analogy with ordinary Java overloading of static functions. If class `A` defines `op+` on String, then `anA + aString` resolves to the user-defined '+', as if resolving a match between an overloaded two-argument function whose first argument is of type `Object` and one whose first argument is of type `A`.

Chapter 9

Processes

A *process* is essentially a thread of control[1]. Processes may either correspond to virtual or physical processors—this is implementation dependent. Each process has a demesne (an area of memory heap; see §6) that may be accessed by other processes through pointers. Each process has a distinct non-negative integer *index*, returned by the function call `Ti.thisProc()`. The indices of all processes form a contiguous interval of integers beginning at 0.

9.1 Process teams

At any given time, each process belongs to a *process team*. The function `Ti.myTeam()` returns an `int[1d]` containing the indices of all members of the team in ascending order. Teams are the sets of processes to which broadcasts, exchanges, and barriers apply. Initially, all processes are on the same team, and all execute the main program from the beginning. This team has the index set `[0:Ti.numProcs()-1]`.

In the current version of Titanium, there is only one process team—the initial one. The process-team machinery in accordingly a bit heavier than needed. It will become useful should we decide to introduce the partition construct (see §11.1).

9.2 Interprocess Communication

Titanium processes may communicate with each other through variables and data structures they share. However, the programmer must be careful that appropriate barriers occur before attempts by one process to access variables set in another, and (usually) that

[1]We have chosen to use the term "process" here, but acknowledge that this choice is a potential source of confusion. Processes in Titanium differ in some crucial details from Java's threads. At the same time, they do not necessarily have any one-to-one relationship with whatever the operating system calls a "process."

multiple processes avoid attempting to modify the same variable without an intervening barrier. Titanium also provides specialized constructs to facilitate communication for some common cases, as detailed in the following sections.

9.2.1 Exchange

The member function `exchange`, defined on Titanium array types, allows all processes in a team to exchange data values with each other. Given E is of some type T and A is a grid of T with an index set that is a superset of Ti.myTeam().`domain()`, the call

A`.exchange(`E`)`

acts as a barrier (see §9.3) among the processes with indices in `Ti.myTeam()`. It is an error if the processes reach different textual instances of `exchange`. The collective effect of `exchange` is as if it collected all the values of E and then copied them into all the arrays A, so that after proceeding from the barrier, the value of $A[i]$ for the A supplied by each process will be the value of E supplied by the process indexed by `Ti.myTeam()[`i`]`, for each i in `Ti.myTeam()`'s domain. It is illegal to exchange arrays of local pointers (that is arrays of a type qualified '**local**'). If T is non-atomic and A resides in a shared region on any processor, then it is also illegal to pass an argument that resides in a private region.

Thus, the code

```
double [1d] single [2d] x = new double[Ti.myTeam().domain()][2d];
x.exchange(new double [D]);
```

creates a vector of pointers to arrays, each on a separate processor, and distributes this vector to all processors.

Access to the arrays A is restricted in order to give implementations some leeway to implement `exchange` efficiently. If the notation A_k denotes the array reference calculated by a process k in an exchange operation, the value of a read or effect of a write to $A_k[j]$ by process $i \neq k$ is undefined between the last dynamically encountered barrier before the call to `exchange` and the next dynamically encountered barrier after returning from the call. As for other method calls, the actual "call to `exchange`" is an event that occurs after evaluation of A and v. As a result of these restrictions, the implementation need not actually complete the copy of data to all the arrays A before any given process continues from its call to `exchange`, nor need it wait for all processes to reach the barrier before it begins setting elements of A for the processes that have reached the barrier.

9.2.2 Broadcast

The `broadcast` expression allows one process in a team to broadcast a value to all others. Specifically, in

```
broadcast E from p
```

—where E is an arbitrary value and p (a UnaryExpression) evaluates to the index of a process on the current process team—process p (only) evaluates E to yield a value V, and all other processes in the team wait at the `broadcast` (if necessary) until p performs the evaluation and then all return the value V. Processes may proceed even if some processes have not yet reached the broadcast and received the broadcast value. It is an error if the processes reach a different sequence of textual instances of the call. It is an error if the processes do not agree on the value p—in fact, p must be a single-valued expression (see §9.4.1), whose value is non-negative and less than `Ti.numProcs()`. It is an error for the evaluation of E on processor p to throw an exception (which, informally, would keep one process from reaching the barrier).

9.3 Barriers

The call

```
Ti.barrier();
```

causes the process executing it to wait until all processes in its team have executed the same textual instance of the barrier call. It is an error for a process to attempt to execute a different sequence of textual instances of barrier calls than another in its team. Each textually distinct occurrence of `partition` and each textually distinct call on `exchange` waits on a distinct anonymous barrier.

9.4 Checking Global Synchronization

In SPMD programs, some portions of the data and control flow are identical across all processes. In particular, the sequences of global synchronizations (barriers, broadcasts, etc.) in each process must be identical for the program to be correct.

With the aid of programmer declarations, the Titanium compiler performs a (conservative) check for a stronger correctness condition statically. This stronger condition requires that a supersequence of the global synchronizations—the statements with *global effects*—is identical across all processes. To be able to perform this check, the compiler must know that the evaluations of certain expressions in a program—those whose values control which synchronization operations are performed—produce *coherent sequences* of values across all processes in the current process team. We call the sequences of values produced by evaluations of one expression by two different processes "coherent" if corresponding pairs of values are identical in the case of primitive types, or, in the case of reference types, point to replicated objects that are of the same dynamic type and that, in the case of arrays, have

identical bounds. To ensure the coherence of such expressions, the compiler must ensure that the sequences of values assigned to certain storage locations by the processes in a given team are themselves coherent. The programmer must mark such storage locations with the type qualifier **single**.
The following properties are important in checking these coherence constraints:

1. An expression e is *single-valued* if it is known statically to produce coherent sequences of values in all processes that start evaluating e. Variables (storage locations) may also be single-valued, meaning that they are single-valued whenever used as expressions (such variables must be marked by being given single-qualified types, and there are restrictions on assigning to them, described below in §9.4.2).

2. A termination T (exception of type t, a break, continue or return) of a statement S is *universal* if either all processes P that start S terminate abruptly with termination T or no process P that starts S terminates abruptly with termination T—in addition, the sequences of exception values produced by terminations that are exceptions must be coherent across all processes in P. A **catch** clause in a **try** statement and the **throws** list of a method or constructor declaration indicate that an exception is universal by qualifying the exception type with **single**.

3. A statement has *global effects* if it or any of its substatements either assigns to any storage (variable, field or array element) whose type is t **single**, *may call* a method or constructor that has global effects, or is a **broadcast** expression.

4. A method or constructor *has global effects* if it is qualified with **single** (which is a new *MethodModifier* that can modify a method or constructor declaration). No other method or constructor has global effects, except as required by the following rules:

   (a) A method has global effects if any statement of its body has global effects; however, assignments to **single** local variables do not count as global effects.

   (b) A constructor has global effects if any statement of its body has global effects. However, an assignment to a **single** local variable does not count for this purpose, nor does an assignment to a non-static **single** field of the object being constructed if it occurs within the constructor body and the destination field is specified as a simple identifier, possibly qualified by **this**.

5. A method call $e_1.m_1(\ldots)$ *may call* a method m_2 if m_2 is m_1, or if m_2 overrides (possibly indirectly) m_1.

9.4.1 Single-valued Expressions

In the following, the e_i are expressions, v is a non-static member, vs a static member, T is a type, and X is a type or expression. All instances of operators refer to built-in definitions; user-defined operators are governed by the same rules as function calls. The following expressions are single-valued:

1. compile-time constant expressions;
2. **this**;
3. variables whose type is t **single**;
4. $e_1.v$ if e_1 is single-valued and v is declared single;
5. e_1.`length`, if e_1 has type T [] `single`.
6. $X.vs$ if vs is declared **single**.
7. $e_1[e_2]$ if e_1 and e_2 are single-valued, e_1 is an array (standard or grid), and the type of the elements of e_1 is single;
8. $e_1[e_2]$ if e_1 and e_2 are single-valued and e_1 is a Point.
9. $e_1 \oplus e_2$, for e_1 and e_2 single-valued and $\oplus$ a built-in binary operator (likewise for unary prefix and postfix operators);
10. $(T)e_1$ if T is single;
11. e_1 **instanceof** T if e_1 is single-valued;
12. $e_1 = e_2$ if e_2 is single-valued;
13. $e_1 \oplus = e_2$ if e_1 and e_2 are single-valued and $\oplus =$ a built-in assignment operator;
14. $e_1 ? e_2 : e_3$ if e_1, e_2 and e_3 are single-valued;
15. $e_0.v(e_1, \ldots, e_n)$ if:
    (a) e_0 is single-valued
    (b) e_i is single-valued if the i'th argument of v is declared single;
    (c) the result of v is declared single;
16. $X.vs(e_1, \ldots, e_n)$ if:

   (a) e_i is single-valued wherever the i'th argument of vs is declared single;

   (b) the result of vs is declared single;

17. **new** $T(e_1, \ldots, e_n)$ if e_i is single-valued wherever the i'th argument of the appropriate constructor for T is declared single;

18. **new** $T[e_1] \ldots [e_n] \underbrace{[] \ldots []}_{k}$, if e_1 is single-valued, $n \geq 1$, and $k \geq 0$. In this case, the type is T $[I_1]$ `single` $[I_2] \ldots [I_{k+n}]$, where the I_j are the appropriate index-type denotations[2]. (If T refers to an array type $T'[\ldots]$, then **new** $T[e_1]$ is taken to be equivalent to **new** $T'[e_1][\ldots]$.)

19. Array initializers in declarations and array creation expressions produce single-valued expressions. Thus, **new** $T[\cdots]\{e_1, \ldots, e_n\}$ produces a result of type $T[\cdots]$ `single`.

20. $[e_1, \ldots, e_n]$ if $e_1, \ldots, e_n$ are single-valued;

21. $[e_1 : e_2 : e_3]$ if e_1, e_2, e_3 are single-valued (and similarly for the other domain literal syntaxes);

22. `broadcast` e_1 `from` e_2, if e_1 has atomic type.

The rule for **broadcast** excludes reference values from single expressions, which may seem odd since the recipients of the broadcast will manifestly get the same value. This merely illustrates, however, the subtle point that "single" and "equal" are not synonyms. Consider a loop such as

```
x = broadcast E from 0;
while (x.N > 0) {
   ...
   Ti.barrier ();
   x.N -= 1;
}
```

(where '`x`' and the '`N`' field of its type are declared **single**). If we were to allow this, it is clearly easy to get different processes to hit the barrier different numbers of times, which is what the rules concerning **single** are supposed to prevent.

[2]As of version 3.86 of the compiler, however, all of the e_i must be single-valued before the allocator's result is single-valued. This is rectified in later versions.

9.4.2 Restrictions on Statements with Global Effects

The following restrictions on individual statements and expressions are enforced by the compiler:

1. Any assignment to a local variable, method or constructor argument, field or array element whose type is single must be from a single-valued expression.

2. If a method call $e_0.v(e_1, \ldots, e_n)$ may call a method m_1 that has global effects then:

   (a) e_0 must be single-valued;

   (b) e_i must single-valued if the i'th argument of v is declared single.

3. In a method call $X.vs(e_1, \ldots, e_n)$ if vs has global effects then e_i must be single-valued if the i'th argument of vs is declared single.

4. In an object allocation **new** $T(e_1, \ldots, e_n)$, if the constructor has global effects then e_i must be single-valued if the i'th argument of the constructor is declared single.

5. In an array allocation `new` $T[e_1] \cdots [e_n]$, if T is an immutable type and the zero-argument constructor for T has global effects then the e_i must be single-valued.

6. In `broadcast` e_1`from` e_2, e_2 must be single-valued.

9.4.3 Restrictions on Control Flow

The following restriction is imposed on the program's control flow: the execution of all statements and expressions with global effects must be controlled exclusively by single-valued expressions. The following rules ensure this:

1. An **if** (or `?` operator) whose condition is not single-valued cannot have statements (expressions) with global effects as its 'then' or 'else' branch.

2. A **switch** statement whose expression is not single-valued cannot contain statements with global effects.

3. A **while**, **do/while** or **for** loop whose exit condition is not single-valued, and a **foreach** loop whose iteration domain is not single-valued cannot contain statements or expressions with global effects.

4. If the main statement of a **try** has non-universal terminations then its **catch** clauses cannot specify any universal exceptions.

5. Associated with every statement or expression that causes a termination t is a set of statements S from the current method or constructor that will be skipped if termination t occurs. If the termination is not universal and the construct being terminated is not a static initializer, then S must not contain any statements or expressions with global effects. (When a static initializer is terminated by an exception, the program as a whole terminates, making restrictions on control flow unnecessary).

The rules for determining whether a termination is universal are essentially identical to the restrictions on statements with global effects: any termination raised in a statement that cannot have global effects is not universal. In addition:

6. In `throw` e, the exception thrown is not universal if e is not single-valued.

7. In a call to method or constructor v declared to throw exceptions of types $t_1, \ldots, t_n$, exception t_i is universal only if type t_i is **single** and the following conditions are met:

   (a) For a normal method call $e_0.v(e_1, \ldots, e_n)$: e_0 is single-valued and e_i is single-valued when the i'th argument of v is declared single;

   (b) For a static method call $X.vs(e_1, \ldots, e_n)$: e_i is single-valued when the i'th argument of vs is declared single;

   (c) For an object allocation `new` $T(e_1, \ldots, e_n)$: e_i is single-valued when the i'th argument of the appropriate constructor for T is declared single.

9.4.4 Restrictions on Methods and Constructors

The following additional restrictions are imposed on methods and constructors:

1. If a method is declared to return a single result, then the expression in all its return statements must be single-valued. The return terminations must all be universal.

2. If a method or constructor signature includes `throws` t **single**, then the body of that method or constructor must not terminate with a non-universal exception assignable to t.

3. A method that overrides a corresponding method in a supertype must preserve the singleness of the method argument types.

4. A method may not have global effects if it overrides or implements a method that does not have global effects. The reverse is not true—an overriding method need not have global effects if the method it overrides or implements does.

9.5 Consistency of Shared Data

The consistency model defines the order in which memory operations issued by one processor are observed by other processors to take effect. Although memory in Titanium is partitioned into demesnes, the union of those demesnes defines the same notion of shared memory that exists in Java. Titanium semantics are consistent with Java semantics in the following sense: the operational semantics given in the Java Language Specification (Chapter 17) correctly implements the behavioral specification given below. We use the behavioral specification here for conciseness and to avoid constraints (or the appearance of constraints) on implementations.

As Titanium processes execute, they perform a sequence of actions on memory. In Java terminology, they may *use* the value of a variable or *assign* a new value to a variable. Given a variable V, we write $\mathit{use}(V, A)$ for a use of V that produces value A and $\mathit{assign}(V, A)$ for an assignment to V with value A. A Titanium program specifies a sequence of memory events, as described in the Java specification:

> [An implementation may perform] a *use* or *assign* by [thread] T of [variable] V ...only when dictated by execution by T of the Java program according to the standard Java execution model. For example, an occurrence of V as an operand of the + operator requires that a single *use* operation occur on V; an occurrence of V as the left-hand operand of the assignment operator (=) requires that a single *assign* operation occur.

Thus, a Titanium program defines a total order on memory events for each process. This corresponds to the order that a naive compiler and processor would exhibit, i.e., without reorderings. The union of these process orders forms a partial order called the *program order*. We write $P(a, b)$ if event a happens before event b in the program order.

During an actual execution, some of the events visible in the Titanium source may be reordered, modified, or eliminated by the compiler or hardware. However, the processes see the events in some total order that is related to the program order. For each execution there exists a total order, E, of memory events from P such that:[3]:

1. $P(a, b) \Rightarrow E(a, b)$ if a and b access the same variable, and at least one of them is an *assign*.

2. $P(l, a) \Rightarrow E(l, a)$, if l is a lock or barrier statement.

3. $P(a, u) \Rightarrow E(a, u)$, if u is an unlock or barrier statement.

4. $P(l_1, l_2) \Rightarrow E(l_1, l_2)$, if l_1 and l_2 are locks, unlocks, or barriers.

[3]See author's notes 1 and 2.

5. E is a correct serial execution, i.e.,
   (a) If $E(assign(V, A), use(V, B))$ and there is no intervening *write* to V, then $A = B$.
   (b) If there were n processes in P, then there are n consecutive barrier statements in E for each instance of a barrier.
   (c) A process T may contain an *unlock* operation l, only if the number of preceding *locks* by T (according to E) on the object locked by l is is strictly greater than the number of *unlocks* by T.
   (d) A process T may contain a *lock* operation l, only if the number of preceding *locks* by other processes (according to E) is equal to the number of preceding *unlocks.*
6. $P(a, b) \Rightarrow E(a, b)$ if a and b both operate on volatile variables.

Less formally, rule 1 says that dependences in the program will be observed. Rules 2 and 3 say that reads and writes performed in a critical demesne must appear to execute inside that demesne. Rule 4 says that synchronization operations must not be reordered. (Titanium extends the Java rules for synchronization, which include only lock and unlock statements to explicitly include barrier.) Rule 5 says that the usual semantics of memory and synchronization constructs are observed. Rule 6 says that operations on volatile variables must execute in order.

As in Java, the indivisible unit for assignment and use is a 32-bit value. In other words, the *assign* and *use* operations in the program order are on at most 32-bit quantities; assignment to double or long variables in Titanium source code corresponds to two separate operations in this semantics. Thus, a program with unsynchronized reads and writes of double or long values may observe undefined values from the reads[4]. It is a weakness of current implementation, unfortunately, that global pointers are not indivisible in this sense.

9.6 Static Storage

In standard Java, a static field refers to the same (one) variable at all points in the program. Unless it is a final, compile-time constant, a static fields is created and initialized at the time its containing class is initialized. Titanium, by contrast, replicates static fields, so that each declaration of such a field denotes one variable for each process. Constant expressions of type `String` are interned separately in each process, regardless of whether they are final

[4]See authors' note 3.

(see §3.10.5 of *The Java Language Specification, Third Edition*). String literals themselves have type `String` **local**.

The static initializers of each class execute once for each process. More specifically, the static initializers of a class—including implicit initializers of fields that are not compile-time constant and the computation of the default value for immutable types (§5)—are concatenated into a single block forming the body of a method. This method then is called for each class on each process to perform the static initializations for that class. It is subject to the usual rules concerning global effects, except that it is legal for any unchecked exception to terminate static initialization; in such cases, the exception also terminates the program as a whole. All static initializers in the program are called in the same order in every process.

Chapter 10

Odds and Ends

10.1 Inlining

The **inline** qualifier on a method declaration acts as in C++. The semantics of calls to such methods is identical to that for ordinary methods. The qualifier is simply advice to the compiler. In particular, you should probably expect it to be ignored when the dynamic type of the target object in a method call cannot be uniquely determined at compilation time.

The qualifier may also be applied to loops:

```
foreach (p in Directions) inline {
    ...
}
```

This is intended to mean that the loop is to be unrolled completely. It is ignored if `Directions` is not a compile-time constant.

10.2 Macro Preprocessing

Titanium supports the C preprocessor language, as described in *International Standard ISO/IEC 9899: Programming Languages—C*. That is, all Titanium sources are first independently subjected to macro preprocessing, which recognizes the set of directives described in that Standard:

`#` Does nothing.

`#define` Defines a textual macro, possibly with parameters.

`#error` Prints a selected error message and cause an error.

`#if...#elif...#else...#endif` For conditional compilation.

`#ifdef` *NAME* Short for `#if defined(`*NAME*`)`.

`#ifndef` *NAME* Short for `#if !defined(`*NAME*`)`.

`#include` Inserts text files verbatim.

`#line` Specifies a source file and line number to ascribe to the current source text for purposes of generating messages.

`#pragma` Modifies the behavior of the program in various implementation-dependent ways.

`#undef` Undefines names introduced by previous `#define` directives.

This syntax includes a small set of macro operators for use inside definitions

`#`P The textual macro parameter P in string quotes.

P `##` Q The tokens P and Q (after textual parameter substitution) concatenated into a single token.

Since each Titanium source is preprocessed independently, only `#define`s that appear in a given file or those files it explicitly `#include`s will be available. One consequence of these semantics is that class declarations should not appear in a file that is `#include`d by more than one loaded file, because that would lead to a multiply-defined class.

The preprocessor predefines certain names. Due to its legacy, these include the standard names defined in the C language specification, in addition to those that are specific to Titanium. The names of likely interest to Titanium programmers are as follows:

`__FILE__` Name of the current source file (a string literal).

`__LINE__` Line number on which the directive appears within the current source file (an integer).

`__DATE__` Current date in "MMM DD YYYY" form (a string literal).

`__TIME__` Current local time in "HH:MM:SS" format on a 24-hour clock (a string literal).

`__TITANIUMC__` The major version number for the compiler translating this source file (an integer).

`__TITANIUMC_MINOR__` The minor version number for compiler translating this source file (an integer).

For full details of all these features, see any current reference work on the C programming language.

Chapter 11

Features Under Consideration

This section discusses features of Titanium whose implementation we have deferred indefinitely until we can evaluate the need for them.

11.1 Partition

The constructs

partition `{` C_0 `=>` S_0`;` C_1 `=>` S_1`;` `...;` C_{n-1} `=>` S_{n-1}`;` `}`

and

partition V `{` C_0 `=>` S_0`;` C_1 `=>` S_1`;` `...;` C_{n-1} `=>` S_{n-1}`;` `}`

divide a team into one or more teams without changing the total number of processes.

The construct begins and ends with implicit calls to `Ti.barrier()`. When all processes in a team reach the initial barrier, the system divides the team into n teams (some possibly empty). All those for which C_0 is true execute S_0. Of the remaining, all for which C_1 is true execute S_1, and so forth. All processes (including those satisfying none of the C_i) wait at the barrier at the end of the construct until all have reached the barrier.

Since the construct partitions the team, it also changes the value of `Ti.myTeam()`, (but not `Ti.thisProc()`) for all processes for the duration of the construct. If supplied, the variable name V is bound to an integer value such that `Ti.thisProc`() = `Ti.myTeam`()[V]. Its scope is the text of all the S_i.

Chapter 12

Additions and Changes to the Standard Library

12.1 Grids

The syntax below is not, of course, legal (and grid types are not classes); we use it simply as a convenient notation. For each type T and positive integer n:

```
final class T [n d] {
  public static final int single arity = n;
  public static int single arity () { return n; }
  public RectDomain<n> single domain();

  public T [n d] single translate(Point<n> single p);
  public local T [n d] local single translate(Point<n> single p);

  public T [n d] single restrict(RectDomain<n> single d);
  public local T [n d] local single restrict(RectDomain<n> single d);

  public T [n d] single inject(Point<n> single p);
  public local T [n d] local single inject(Point<n> single p);

  public T [n d] single project(Point<n> single p);
  public local T [n d] local single project(Point<n> single p);

  public T [n d] single permute(Point<n> single p);
  public local T [n d] local single permute(Point<n> single p);

  public T [(n - 1)d] single slice(int single k, int single j);
  /* only if n > 1 */

public local T[(n - 1)d] local single slice(int single k, int single j);

public int single size();
public boolean single isEmpty();

public T[n d] single shrink(int single k, int single dir);
public local T[n d] local single shrink(int single k, int single dir);
public T[n d] single shrink(int single k);
public local T[n d] local single shrink(int single k);

public T[n d] single border(int single k, int single dir,
                            int single shift);
public local T[n d] local single border(int single k, int single dir,
                                        int single shift);

public T[n d] single border(int single k, int single dir);
public local T[n d] local single border(int single k, int single dir);

public T[n d] single border(int single dir);
public local T[n d] local single border(int single dir);

public void set(T value);

final public boolean isLocal ();
public boolean isContiguous ();
final public int creator ();
public ti.lang.Region local regionOf ();

public void copy(T [n d] x) overlap(this, x);

public void copyNBI(T [n d] x) overlap(this, x);
public ti.lang.Handle copyNB(T [n d] x) overlap(this, x);

public void copy(T [n d] x, Domain<n> d) overlap(this, x);
public void copy(T [n d] x, Point<n> [1d] p) overlap(this, x);

public void gather(T [1 d] x, Point<n> [1d] p);
public void scatter(T [1 d] x, Point<n> [1d] p);

public single void exchange(T myValue);
```

```
      public void readFrom(java.io.RandomAccessFile file)
         throws java.io.IOException;
      public void readFrom(java.io.DataInputStream str)
         throws java.io.IOException;
      public void writeTo(java.io.RandomAccessFile file)
         throws java.io.IOException;
      public void writeTo(java.io.DataOutputStream str)
         throws java.io.IOException;
   }
```

12.2 Points

```
template<int n> public immutable class Point {
      public static Point<n> single all(int single x);
      public static Point<n> single direction(int single k, int single x);
      public static Point<n> single direction(int single k);
      public Point<n> single op+(Point<n> single p);
      public Point<n> single op+=(Point<n> single p);
      public Point<n> single op-();
      public Point<n> single op-(Point<n> single p);
      public Point<n> single op-=(Point<n> single p);
      public Point<n> single op*(Point<n> single p);
      public Point<n> single op*=(Point<n> single p);
      public Point<n> single op/(Point<n> single p);
      public Point<n> single op/=(Point<n> single p);
      public Point<n> single op*(int single n);
      public Point<n> single op*(int single n, Point<n> single p);
      public Point<n> single op*=(int single n);
      public Point<n> single op/(int single n);
      public Point<n> single op/(int single n, Point<n> single p);
      public Point<n> single op/=(int single n);
      public boolean single equals(Point<n> single p);
      public boolean single op==(Point<n> single p);
      public boolean single op!=(Point<n> single p);
      public boolean single op<(Point<n> single p);
      public boolean single op<=(Point<n> single p);
      public boolean single op>(Point<n> single p);
```

```
    public boolean single op>=(Point<n> single p);
    public int single op[] (int single x);

    public Point<n> single lowerBound (Point<n> single p);
    public Point<n> single upperBound (Point<n> single p);
    public Point<n> single replace (int single k, int single v);
    public static final int single arity = n;
    public static int single arity () { return n; }
    public Point<n> single permute (Point<n> single p);

    public String single local toString ();
  }
```

12.3 Domains

A `Domain<`n`>` is an Object, and therefore differs in subtle ways from a `RectDomain<`n`>`. Specifically, `==` and `!=` on Domains denote pointer (in)equality, as for other reference types.

```
  template<int n> public final class Domain {

    public Domain<n> (Domain<n> single d);

    public static Domain<n> single toDomain(Point<n> single [1d] p);
    public static Domain<n> single toDomain(RectDomain<n> single [1d] rd);
    public static Domain<n> toDomain(Point<n> [1d] p);
    public static Domain<n> toDomain(RectDomain<n> [1d] rd);

    public Domain<n> single op+(RectDomain<n> single d);
    public Domain<n> single op+(Domain<n> single d);
    public Domain<n> single op+=(Domain<n> single d);
    public Domain<n> single op+=(RectDomain<n> single d);
    public Domain<n> single op-(Domain<n> single d);
    public Domain<n> single op-(RectDomain<n> single d);
    public Domain<n> single op-=(Domain<n> single d);
    public Domain<n> single op-=(RectDomain<n> single d);
    public Domain<n> single op*(RectDomain<n> single d);
    public Domain<n> single op*(Domain<n> single d);
    public Domain<n> single op*=(RectDomain<n> single d);
    public Domain<n> single op*=(Domain<n> single d);
    public Domain<n> single permute(Point<n> single p);
    public RectDomain<n> [1d] single RectDomainList();

    public boolean equals(Object d);
    public boolean single equals(Domain<n> single d);
    public boolean single equals(RectDomain<n> single d);
    public boolean single op<(RectDomain<n> single d);
    public boolean single op<(Domain<n> single d);
    public boolean single op<=(Domain<n> single d);
    public boolean single op<=(RectDomain<n> single d);
    public boolean single op>(RectDomain<n> single d);
    public boolean single op>(Domain<n> single d);
    public boolean single op>=(Domain<n> single d);
    public boolean single op>=(RectDomain<n> single d);

    public Domain<n> single op+(Point<n> single p);
    public Domain<n> single op+=(Point<n> single p);
    public Domain<n> single op-(Point<n> single p);
    public Domain<n> single op-=(Point<n> single p);
    public Domain<n> single op*(Point<n> single p);
    public Domain<n> single op*=(Point<n> single p);
    public Domain<n> single op/(Point<n> single p);
    public Domain<n> single op/=(Point<n> single p);
    public Point<n> [1d] single PointList();

    public static final int single arity = n;
    public static int single arity() { return n; }
    public Point<n> single lwb();
    public Point<n> single upb();
    public Point<n> single min();
    public Point<n> single max();
    public int single size();
    public boolean single contains(Point<n> single p);
    public RectDomain<n> single boundingBox();
    public boolean single isEmpty();
    public boolean single isRectangular();

    public String toString();
    public int hashCode ();
    public static Region setRegion(Region r);
  }
```

12.4 RectDomains

```
template<int n> public immutable class RectDomain {
  public boolean isRectangular();

  public Domain<n> single op+(RectDomain<n> single d);
  public Domain<n> single op-(RectDomain<n> single d);
  public RectDomain<n> single op*(RectDomain<n> single d);
  public RectDomain<n> single op*(Domain<n> single d);
  public RectDomain<n> single op*=(RectDomain<n> single d);

  public RectDomain<n> single op+(Point<n> single p);
  public RectDomain<n> single op+=(Point<n> single p);
  public RectDomain<n> single op-(Point<n> single p);
  public RectDomain<n> single op-=(Point<n> single p);
  public RectDomain<n> single op*(Point<n> single p);
  public RectDomain<n> single op*=(Point<n> single p);
  public RectDomain<n> single op/(Point<n> single p);
  public RectDomain<n> single op/=(Point<n> single p);

  public boolean single equals(RectDomain<n> single d);
  public boolean single op==(RectDomain<n> single d);
  public boolean single op!=(RectDomain<n> single d);
  public boolean single op<(RectDomain<n> single d);
  public boolean single op<=(RectDomain<n> single d);
  public boolean single op>(RectDomain<n> single d);
  public boolean single op>=(RectDomain<n> single d);

  public boolean equals(Object d);
  public boolean single equals(Domain<n> single d);
  public Domain<n> single op+(Domain<n> single d);
  public Domain<n> single op-(Domain<n> single d);
  public boolean single op<(Domain<n> single d);
  public boolean single op<=(Domain<n> single d);
  public boolean single op>(Domain<n> single d);
  public boolean single op>=(Domain<n> single d);

  public final static int single arity = n;
  public static int single arity () { return n; }
  public Point<n> single lwb();
  public Point<n> single upb();
```

```
      public Point<n> single min();
      public Point<n> single max();
      public Point<n> single stride();
      public int single size();
      public boolean single isEmpty();
      public RectDomain<n> single accrete(int single k, int single dir,
                                          int single s);
      public RectDomain<n> single accrete(int single k, int single dir);
      public RectDomain<n> single accrete(int single k, Point<n> single S);
      public RectDomain<n> single accrete(int single k); /* S = all(1) */
      public RectDomain<n> single shrink(int single k, int single dir);
      public RectDomain<n> single shrink(int single k);
      public RectDomain<n> single permute(Point<n> single p);
      public RectDomain<n> single border(int single k, int single dir,
                                         int single shift);
      public RectDomain<n> single border(int single k, int single dir);
      public RectDomain<n> single border(int single dir);
      public boolean single contains(Point<n> single p);
      public RectDomain<n> single boundingBox();
         /* only if n > 1 */
      public RectDomain<n - 1> single slice(int single k);

      public String single local toString();
    }
```

12.5 Reduction operators

Each process in a team must execute the same sequence of textual instances of executions of reduction operations, barriers, exchanges, and broadcasts.

```
package ti.lang;
public final class Reduce
{
  /* The result of Reduce.F (E) is the result of applying the binary
   * operator F to all processes' values of E in some unspecified
   * grouping.  The result of Reduce.F (E, k) is 0 or false for all
   * processes other than K, and the result of applying F to all the
   * values of E for processor K.
   *
   * Here, F can be 'add' (+), 'mult' (*),  'max' (maximum),
 * 'min' (minimum), 'and' (&), 'or' (|), or 'xor' (^).
 */

public static single int add(int n, int single to);
public static single long add(long n, int single to);
public static single double add(double n, int single to);
public static single int single add(int n);
public static single long single add(long n);
public static single double single add(double n);

public static single int mult(int n, int single to);
public static single long mult(long n, int single to);
public static single double mult(double n, int single to);
public static single int single mult(int n);
public static single long single mult(long n);
public static single double single mult(double n);

public static single int max(int n, int single to);
public static single long max(long n, int single to);
public static single double max(double n, int single to);
public static single int single max(int n);
public static single long single max(long n);
public static single double single max(double n);

public static single int min(int n, int single to);
public static single long min(long n, int single to);
public static single double min(double n, int single to);
public static single int single min(int n);
public static single long single min(long n);
public static single double single min(double n);

public static single int or(int n, int single to);
public static single long or(long n, int single to);
public static single boolean or(boolean n, int single to);
public static single int single or(int n);
public static single long single or(long n);
public static single boolean single or(boolean n);

public static single int xor(int n, int single to);
public static single long xor(long n, int single to);
public static single boolean xor(boolean n, int single to);
public static single int single xor(int n);
```

```
  public static single long single xor(long n);
  public static single boolean single xor(boolean n);

  public static single int and(int n, int single to);
  public static single long and(long n, int single to);
  public static single boolean and(boolean n, int single to);
  public static single int single and(int n);
  public static single long single and(long n);
  public static single boolean single and(boolean n);

  /* These reductions use OPER.eval as the binary operator, and are
   * otherwise like the reductions above. */

  public static single Object gen(ObjectOp oper, Object o, int single to);
  public static single Object gen(ObjectOp oper, Object o);

  public static single int gen(IntOp oper, int n, int single to);
  public static single int single gen(IntOp oper, int n);

  public static single long gen(LongOp op, long n, int single to);
  public static single long single gen(LongOp oper, long n);

  public static single double gen(DoubleOp oper, double n, int single to);
  public static single double single gen(DoubleOp oper, double n);
}
```

```
package ti.lang;
public final class Scan
{
  /* Scan.F (E) produces the result of applying the operation F to all
   * values of E for this and lower-numbered processes, according to
   * some unspecified grouping. */

  public static single int add(int n);
  public static single long add(long n);
  public static single double add(double n);

  public static single int mult(int n);
  public static single long mult(long n);
  public static single double mult(double n);

  public static single int max(int n);
  public static single long max(long n);
  public static single double max(double n);

  public static single int min(int n);
  public static single long min(long n);
  public static single double min(double n);

  public static single int or(int n);
  public static single long or(long n);
  public static single boolean or(boolean n);

  public static single int xor(int n);
  public static single long xor(long n);
  public static single boolean xor(boolean n);

  public static single int and(int n);
  public static single long and(long n);
  public static single boolean and(boolean n);

  /* As for the preceding scans, but with F being oper.eval. */
  public static single Object gen(ObjectOp oper, Object o);
  public static single int gen(IntOp oper, int n);
  public static single long gen(LongOp oper, long n);
  public static single double gen(DoubleOp oper, double n);
}
```

```
interface IntOp {
  int eval(int x, int y);
}

interface LongOp {
  long eval(long x, long y);
}

interface DoubleOp {
  double eval(double x, double y);
}

interface ObjectOp {
  Object eval(Object arg0, Object arg1);
}
```

12.6 Complex Numbers

The immutable class `ti.lang.Complex` provides the standard basic operations on complex numbers familiar to users of Fortran and C. The real and imaginary parts of these numbers come from the set $\Re \cup \{-0\}$, where -0 is the IEEE negative 0.

```
package ti.lang;

/** IEEE-extended complex numbers, rectangular representation */
public immutable class Complex {
    public double re, im;

    /** Zero */
    public Complex();
    public Complex(double re, double im);

    /* A string in the form <num>, <num>i, <num>+<num>i,
       or <num>-<num>i */
    public String toString();

    /** The Complex value denoted by S.  S must have the form <num>,
     *  <num><i>, <num><op><num><i>, or (<num>,<num>),
     *  where each <num> is parsable as a double, <i> is i or I,
     *  and <op> is + or -.  The parenthesized form is Fortran format:
```

```
 *  (real part, imag part).  The <nums> may contain
 *  no embedded whitespace; otherwise whitespace is ignored. */
public static Complex parseComplex(String s) throws NumberFormatException;

/* Arithmetic and Comparison */

public Complex op-();
public Complex op+(double d1);
public Complex op+(Complex c1);
public Complex op-(double d1);
public Complex op-(Complex c1);
public Complex op*(double d1);
public Complex op*(Complex c1);
public Complex op/(double d1);
public Complex op/(Complex c1);
public boolean op==(double d1);
public boolean op==(Complex c1);
public boolean op!=(double d1);
public boolean op!=(Complex c1);

public Complex op+(double d1, Complex c1);
public Complex op-(double d1, Complex c1);
public Complex op*(double d1, Complex c1);
public Complex op/(double d1, Complex c1);
public Complex op^(double d1, Complex c1);
public Complex op==(double d1, Complex c1);
public Complex op!=(double d1, Complex c1);

/** The complex conjugate of THIS. */
public Complex op~();
public Complex conj();

/** The value THIS raised to the D1 power. */
public Complex op^(double d1);
/** The value THIS raised to the C1 power. */
public Complex op^(Complex c1);

/* Assignment operations */

public Complex op+=(Complex c1);
public Complex op-=(Complex c1);
public Complex op*=(Complex c1);
```

```
    public Complex op/=(Complex c1);
    public Complex op^=(Complex c1);
    public Complex op+=(double d1);
    public Complex op-=(double d1);
    public Complex op*=(double d1);
    public Complex op/=(double d1);
    public Complex op^=(double d1);

    public Complex exp();
    public Complex log();
    public Complex sqrt();
    public Complex sin();
    public Complex cos();
    public Complex tan();
    public Complex asin();
    public Complex acos();
    public Complex atan();
    public Complex sinh();
    public Complex cosh();
    public Complex tanh();
    public Complex asinh();
    public Complex acosh();
    public Complex atanh();

    /** Absolute value (magnitude) */
    public double abs();
    /** The square of the magnitude. */
    public double absSquare();

    /** The argument (phase angle) in radians. The result will be
     *  between -pi and pi.  It is 0.0 when abs() is 0.0. */
    public double arg();
}
```

12.7 Timer class

A Timer (type `ti.lang.Timer` provides a microsecond-granularity “stopwatch” that keeps track of the total time elapsed between start() and stop() method calls.

```
package ti.lang;
public class Timer {
  /**  The maximum possible value of secs (). Roughly 1.84e13. */
  public final double MAX_SECS;

  /** Creates a new Timer object for which secs () is initially 0. */
  public Timer ();

  /** Cause THIS to begin counting. */
  public void start();

  /** Add the time elapsed from the last call to start() to the value
   *  of secs (). */
  public void stop();

  /** Set secs () to 0. */
  public void reset();

  /** The count of THIS in units of seconds, with a maximum
   *  granularity of one microsecond. */
  public double secs();

  /** The count of THIS in units of milliseconds, with a maximum
   *  granularity of one microsecond. */
  public double millis();

  /** The count of THIS in units of microseconds, with a maximum
   *  granularity of one microsecond. */
  public double micros();
}
```

12.8 PAPI Counters

This class provides access to the Performance Application Programming Interface (PAPI) on platforms that support it. Creating and starting a `PAPICounter` provides an object that counts the events indicated by its constructor parameter.

```
package ti.lang;

/** PAPI counter library.  The available counters are platform dependent. */
```

```
public class PAPICounter {

    /** Constants denoting possible tracked events. */
    public static final int L1_DataCache_Misses;
    public static final int L1_ICache_Misses;
    public static final int L2_DataCache_Misses;
    public static final int L2_ICache_Misses;
    public static final int L3_DataCache_Misses;
    public static final int L3_ICache_Misses;
    public static final int L1_TotalCache_Misses;
    public static final int L2_TotalCache_Misses;
    public static final int L3_TotalCache_Misses;
    public static final int Branch_Unit_Idle;
    public static final int Integer_Unit_Idle;
    public static final int Float_Unit_Idle;
    public static final int LoadStore_Unit_Idle;
    public static final int DataTLB_Misses;
    public static final int ITLB_Misses;
    public static final int Total_TLB_Misses;
    public static final int L1_Load_Misses;
    public static final int L1_Store_Misses;
    public static final int L2_Load_Misses;
    public static final int L2_Store_Misses;
    public static final int Data_Prefetch_Cache_Misses;
    public static final int L3_DataCache_Hits;
    public static final int Memory_Access_Stall;
    public static final int Memory_Read_Stall;
    public static final int Memory_Write_Stall;
    public static final int No_Instruction_Issued;
    public static final int No_Instruction_Completed;
    public static final int Conditional_Branch_Instruction;
    public static final int Mispredicted_Conditional;
    public static final int Correctly_Predicted_Conditional;
    public static final int FMA_Instruction_Completed;
    public static final int Instruction_Issued;
    public static final int Instruction_Completed;
    public static final int Integer_Instruction;
    public static final int Floating_Point_Instruction;
    public static final int Load_Instruction;
    public static final int Store_Instruction;
    public static final int Branch_Instruction;
    public static final int Total_Cycles;
    public static final int Load_Store_Instruction;
    public static final int Synchronization_Instruction;
    public static final int L1_DataCache_Hits;
    public static final int L2_DataCache_Hits;
    public static final int L1_DataCache_Accesses;
    public static final int L2_DataCache_Accesses;
    public static final int L3_DataCache_Accesses;
    public static final int L1_DataCache_Reads;
    public static final int L2_DataCache_Reads;
    public static final int L3_DataCache_Reads;
    public static final int L1_DataCache_Writes;
    public static final int L2_DataCache_Writes;
    public static final int L3_DataCache_Writes;
    public static final int L1_ICache_Hits;
    public static final int L2_ICache_Hits;
    public static final int L3_ICache_Hits;
    public static final int L1_ICache_Accesses;
    public static final int L2_ICache_Accesses;
    public static final int L3_ICache_Accesses;
    public static final int L1_ICache_Reads;
    public static final int L2_ICache_Reads;
    public static final int L3_ICache_Reads;
    public static final int L1_ICache_Writes;
    public static final int L2_ICache_Writes;
    public static final int L3_ICache_Writes;
    public static final int L1_Cache_Hits;
    public static final int L2_Cache_Hits;
    public static final int L3_Cache_Hits;
    public static final int L1_Cache_Accesses;
    public static final int L2_Cache_Accesses;
    public static final int L3_Cache_Accesses;
    public static final int L1_Cache_Reads;
    public static final int L2_Cache_Reads;
    public static final int L3_Cache_Reads;
    public static final int L1_Cache_Writes;
    public static final int L2_Cache_Writes;
    public static final int L3_Cache_Writes;
    public static final int Float_Multiply_Instruction;
    public static final int Float_Add_Instruction;
    public static final int Float_Divide_Instruction;
    public static final int Float_SquareRoot_Instruction;
    public static final int Float_Inverse_Instruction;
```

```
    public static final int Float_Operations;

    /** A counter for the indicated EVENT. */
    public PAPICounter(int event);

    /** True iff EVENT can be monitored. */
    public static boolean checkEvent(int event);

    /** True iff the current counter is running (not idle).  Initially
     *  false. */
    public boolean single running();

    /** Starts the counter.  Requires that the counter be idle. */
    public void start();

    /** Stops (idles) the counter.  Requires that the counter be running. */
    public void stop();

    /** Resets the counter value to 0.  Requires that the counter be
     *  idle. */
    public void clear();

    /** Returns the current counter value.  Requires that the counter be
     *  idle. */
    public long getCounterValue();

    /** Returns the current counter value as a string. */
    public String toString();
}
```

12.9 ti.lang.Ti

This class provides miscellaneous static methods supporting SPMD programming and memory management in Titanium.

```
package ti.lang;

public class Ti {
    public static int thisProc();
    public static int single numProcs();
    public static single void barrier();
    public static int single [1d] single local myTeam();
    public static void poll();
    public static void suspend_autogc();
    public static void resume_autogc();
}
```

12.10 Additional properties

The values of the following properties are available, as in ordinary Java, through calls to `java.lang.System.getProperty`.

runtime.distributed Has the value `"true"` if and only if the Titanium program reading it has been compiled for a platform with a distributed memory architecture, and otherwise `"false"`.

runtime.shared Has the value `"true"` iff some processes may share a memory space. (The CLUMP backends make "shared" and "distributed" orthogonal, rather than mutually exclusive.)

runtime.model Indicates the specific platform that the Titanium program reading it has been compiled for.

java.version, java.vm.version The `tc` (Titanium compiler) version.

runtime.boundschecking Has the value `"true"` if bounds checking is on.

runtime.gc Has the value `"true"` if garbage collection is on.

compiler.flags Flags passed to `tc` to compile the application.

12.11 java.lang.Object

Titanium adds or modifies several methods in class `java.lang.Object`, as indicated in §6.1:

```
        final public int creator();
        final public boolean isLocal();
```

```
    protected Object clone() throws CloneNotSupportedException;
    protected local Object local localClone()
      throws CloneNotSupportedException;
    public final ti.lang.Region local regionOf();
```

1. `r.creator()` returns the identity of the lowest-numbered process whose demesne contains the object pointed to by `r`. NullPointerException if `r` is null. This number may differ from the identity of the process that actually allocated the object when multiple processes share a demesne.

2. `r.isLocal()` returns true iff `r` may be coerced to a local reference. This returns the same value as `r instanceof T local`, assuming `r` to have object type `T`. On some (but not all) platforms `r.isLocal()` is equivalent to `r.creator() == thisProc()`.

3. `r.clone()` is as in Java, but with local references inside the object referenced by `r` set to null. The result is a global pointer to the newly created clone, which resides in the same region (see §6.3.2) as the object referenced by `r`. This operation may be overridden.

4. `r.localClone()` is the default `clone()` function of Java (a shallow copy), except that `r` must be local. The result is local and in the same region (see §6.3.2) as the object referenced by `r`. See also §6.1.

5. `r.regionOf ()` See §6.3.2.

12.12 java.lang.Math

The static methods in `java.lang.Math`, with the exception of `java.lang.Math.random`, take `single` arguments and produce `single` results.

12.13 java.lang.Boolean, java.lang.Number and Subclasses

The signatures of certain constructors and methods in the standard wrapper classes of `java.lang`—`Boolean`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`—have added single qualification. We list below only the methods modified from Java.

```
public class Integer {
    public static int single parseInt(String single s, int single radix)
       throws NumberFormatException single;
    public static int single parseInt(String single s)
       throws NumberFormatException single;
    public static Integer single valueOf(String single s, int single radix)
       throws NumberFormatException single;
    public static Integer single valueOf(String single s)
       throws NumberFormatException single;
    public Integer(int single value);
    public Integer(String single s) throws NumberFormatException single;
    public int single intValue();
    public long single longValue();
    public float single floatValue();
    public double single doubleValue();
}

public class Short {
    public Short(short single value);
    public static Short single valueOf(String single s);
    public static Short single valueOf(String single s, int single radix);
    public static short single parseShort(String single s);
    public static short single parseShort(String single s, int single radix);
    public byte single byteValue();
    public short single shortValue();
    public int single intValue();
    public long single longValue();
    public float single floatValue();
    public double single doubleValue();
}

public class Byte {
    public Byte(byte single value);
    public static Byte single valueOf(String single s);
    public static Byte single valueOf(String single s, int single radix);
    public static byte single parseByte(String single s);
    public static byte single parseByte(String single s, int single radix);
    public byte single byteValue();
    public short single shortValue();
    public int single intValue();
    public long single longValue();
    public float single floatValue();
    public double single doubleValue();
}

public class Long {
    public static long single parseLong(String single s, int single radix)
    public static long single parseLong(String single s)
       throws NumberFormatException single;
    public static Long single valueOf(String single s, int single radix)
       throws NumberFormatException single;
    public static Long single valueOf(String single s)
       throws NumberFormatException single;
    public Long(long single value)
    public Long(String single s) throws NumberFormatException single;
    public int single intValue()
    public long single longValue()
    public float single floatValue()
    public double single doubleValue()
}

public class Double {
    public static Double single valueOf(String single s)
        throws NumberFormatException single;
    public boolean single isNaN(double single v)
    public boolean single isInfinite(double single v)
    public Double(double single value)
    public Double(String single s) throws NumberFormatException single;
    public boolean single isNaN()
    public boolean single isInfinite()
    public int single intValue()
    public long single longValue()
    public float single floatValue()
    public double single doubleValue()
    public static long single doubleToLongBits(double single value);
    public static double single longBitsToDouble(long single bits);
}

public class Boolean {
    public Boolean(boolean single value);
    public Boolean(String single s);
    public boolean single booleanValue();
    public static Boolean single valueOf(String single s);
}
```

12.14 Polling

The operation `Ti.poll()` services any outstanding network messages. This is needed in programs that have long, purely-local computations that can starve the NIC (e.g. a self-scheduled computation). It has no semantic content, and affects only performance (on some systems).

12.15 Sparse Titanium Array Copying

The `copy` method described in §4.3.1 applies to dense (rectangular) regions of grids specified by `RectDomains`. Here, we describe more general copying methods that take their index sets from general domains or from arrays of points.

The Titanium array operation `.copy()` takes an optional second parameter that can be a `Domain<`N`>` or a `Point<`N`> [1d]`, specifying the set of points to be copied (the types `Domain<`N`>` and `RectDomain<`N`>` have methods that make it easy to convert back and forth from `Point<`N`> [1d]`). For the purposes of modularity and performance, other sparse-array copying operations are divided into two lower-level components: a *gather* operation, which copies selected elements to contiguous points in an array, and its inverse, a *scatter* operation, which scatters values from a dense array using an index vector of points.

12.15.1 Copy

Assuming

```
    T [N d] dest, src;
    Domain<N> dom;
    Point<N> [1d] pts;
```

the calls

```
    dest.copy(src, dom)
    dest.copy(src, pts);
```

copy the specified points from `src` to `dest`.

Restrictions

1. `pts` must not overlap `dest` or `src`.
2. Each point element of `pts` or `dom` must be contained within the intersection of `.domain()`dest and `.domain()`src (it is a fatal error otherwise).
3. If `T` is a non-atomic type, then it may not be the case that `src` resides in a private region and `dest` resides in a shared region.
4. If `T` contains embedded local pointers then `src` and `dest` must both be local.
5. The contents of `src` (at the selected points) and `pts` may not be modified (i.e., by other processes) during the operation.

Effects

After the operation completes, the following conditions will hold:

1. For every `Point<`N`>` p in `dom` (or in `pts`), `dest[`p`] == src'[`p`]` (where `src'` denotes the contents of `src` prior to the operation).
2. All other values are unchanged.
3. `pts` is permitted to contain duplicate points, but by definition these will not affect the result.
4. `src` and `dest` are permitted to overlap, and if they do it will be as if the relevant values were first copied from `src` to an intermediate temporary array and then to `dest`.
5. During the operation, the contents of `dest` at affected points is undefined.

12.15.2 Gather

Assuming

```
    T [N d] src;
    Point<N> [1d] pts;
    T [1d] dest;
```

the call

```
    src.gather(dest, pts);
```

packs the values from `src` selected by `pts` into `dest`, maintaining ordering of the points and data, and preserving any duplicated points in the packed array.

Restrictions

1. `dest.domain().size() >= pts.domain().size()` (i.e. there is enough space to gather into). It is a fatal error otherwise.
2. Each point value in `pts` must be in `src.domain()`. It is a fatal error otherwise.
3. None of the arrays may overlap.
4. If `T` is a non-atomic type, then it may not be the case that `src` resides in a private region and `dest` resides in a shared region.
5. If `T` contains embedded local pointers then `src` and `dest` must both be local.
6. The contents of `src` (at the selected points) and `pts` may not be modified (i.e., by other processes) during the operation.

Effects

After the operation completes, the following conditions will hold:

1. `src` and `pts` will be unchanged
2. For all i such that $0 \leq i <$ `pts.size()`,
   `dest[[`i`] + dest.min()] == src[pts[[`i`] + pts.min()]]`.
3. The contents of `dest` are undefined while operation is in progress.
4. `pts` is permitted to contain duplicate points. If it does, the corresponding `dest` elements will contain the relevant data values once for each duplicate (as implied by the description of the effects).

12.15.3 Scatter

Assuming

```
T [1d] src;
Point<N> [1d] pts;
T [N d] dest;
```

the call

```
dest.scatter(src, pts);
```

unpacks the values from `src` into `dest` at the positions selected by `pts`.

Restrictions

1. `pts.domain().size() <= src.domain().size()` (i.e. there are enough values to be scattered in `src`). It is a fatal error otherwise.
2. Each point value in `pts` must be in `dest.domain()`. It is a fatal error otherwise.
3. None of the arrays may overlap.
4. If `T` is a non-atomic type, then it may not be the case that `src` resides in a private region and `dest` resides in a shared region.
5. If `T` contains embedded local pointers then `src` and `dest` must both be local.
6. The contents of `src` and `pts` may not be modified (i.e., by other processes) during the operation.

Effects

After the operation completes, the following conditions will hold:

1. `src` and `pts` will be unchanged.
2. For all i such that $0 \leq i <$ `pts.size()`,
   `dest[pts[[`i`] + pts.min()]] == src[[`i`] + src.min()]`.
3. `pts` is permitted to contain duplicate points. If it does, the relevant `destArray` element will contain the data value corresponding to the highest-indexed duplicate in `pts`.
4. The contents of `destArray` at the affected points are undefined while operation is in progress.

12.16 Non-blocking Array Copying

Non-blocking array copying exposes the initiation and synchronization of array copying to the user, allowing overlap of copying with other communication and computation without performing data-dependency and alias analyses to see that it is safe to do so. The motivation is that the programmer's application-level knowledge should enable significantly more aggressive overlapping than a general optimizer could ever hope to accomplish.

All non-blocking operations require an initiation (the copy call) and a subsequent synchronization on the completion of that operation before the result at the destination is defined. Between the initiation of the copy operation and the completion of the corresponding sync, any use of one of the target elements (by any process) produces an undefined value and any assignment to one of the target or source elements (by any process) has an undefined effect on the targets' final values.

There are two basic categories of non-blocking operations, defined by the synchronization mechanism used:

- An *explicit handle* (NB) operation returns a specific handle from the initiation (of type `ti.lang.Handle`), which the calling process must later use to synchronize that particular copy (through its `syncNB()` instance method).
- An *implicit handle* (NBI) operation does not return a handle from the initiation. Instead, the static routine `Handle.syncNBI()` must later be used to synchronize all outstanding NBI operations issued by the calling process.

Non-blocking operations proceed independently of inter-process synchronization operations (barriers, broadcasts, synchronized regions, etc.), and the 'sync' operations in `Handle` provide no inter-process synchronization. NB and NBI operations proceed independently of each other; `syncNBI` does not affect on-going NB operations.

Specifications In package `ti.lang`:

```
public immutable class Handle {

  /** Wait until the arraycopyNB or copyNB operation that created
   *  this Handle completes, and invalidate this Handle.
   *  Must be called by the same process that initiated the copy
   *  operation that returned this Handle.
   */
  public void syncNB();

  /** Wait until all arraycopyNBI and copyNBI operations
   *  issued by the calling process have completed.
   */
  public static void syncNBI();

}
```

The following methods in `java.lang.System` have the same parameter signatures as `arraycopy`:

```
    public static native Handle arraycopyNB(Object src, int src_position,
                                            Object dst, int dst_position,
                                            int length);

    public static native void arraycopyNBI(Object src, int src_position,
                                           Object dst, int dst_position,
                                           int length);
}
```

All Titanium array types define the following two methods:

```
    public void copyNBI(T [n d] x) overlap(this, x);
    public Handle copyNB(T [n d] x) overlap(this, x);
```

Examples of using explicit handles.

```
    int [1d] x,y;
    int [] ja,jb;
    ...
    Handle h = x.copyNB(y); // initiate a non-blocking copy from y -> x
                            // as for TiArray.copy()

    Handle h2 = System.arraycopyNB(ja,0,jb,0,jb.length);
                             // initiate a non-blocking copy from ja -> jb
                             // as for System.arraycopy

    h.syncNB();    // block until the first operation completes
    h2.syncNB();   // block until the second operation completes
```

Examples of using implicit handles.

```
    x.copyNBI(y);  // initiate a non-blocking copy from y -> x

    System.arraycopyNBI(ja,0,jb,0,jb.length);
                   // initiate a non-blocking copy from ja -> jb

    Handle.syncNBI(); // block until both operations complete
```

12.17 Bulk I/O

This library supports fast I/O operations on both Titanium arrays and Java arrays. These operations are synchronous (that is, they block the caller until the operation completes).

12.17.1 Bulk I/O for Titanium Arrays

Bulk I/O works through two methods on Titanium arrays: `.readFrom()` and `.writeTo()`. The arguments to the methods are various kinds of file—currently: `RandomAccessFile`, `DataInputStream`, `DataOutputStream`, in `java.io` and their corresponding subclasses, `BulkRandomAccessFile`, `BulkDataInputStream`, and `BulkDataOutputStream` from ti.io.

Consider a Titanium array type whose elements are of an atomic type (§4.2). The following methods are defined for this type:

```
/** Perform a bulk read of data into the elements of this
 *  array from INFILE.  The number of elements read will be
 *  equal to domain().size().  They are read sequentially in
 *  row-major order.  Throws java.io.IOException in the
 *  case end-of-file or an I/O error occurs before all
 *  data are read. */
void readFrom (java.io.RandomAccessFile infile)
     throws java.io.IOException;
void readFrom (java.io.DataInputStream infile)
     throws java.io.IOException;

/** Perform a bulk write of data from the elements of this array
 *  to OUTFILE.  The number of elements written will be equal
 *  to domain().size().  They are written sequentially in row-major
 *  order. Throws java.io.IOException in the case of disk full or
 *  other I/O errors. */
void writeTo (java.io.RandomAccessFile outfile)
    throws java.io.IOException;
void writeTo(java.io.DataOutputStream outfile)
    throws java.io.IOException;
```

I/O on Partial Arrays

To read or write a proper subset of the elements in a Ti array, first use the regular array-selection methods such as `.slice()` and `.restrict()` to select the desired elements, then make I/O calls on the resultant arrays (these operations are implemented very efficiently without performing a copy).

12.17.2 Bulk I/O for Java Arrays in Titanium

The BulkDataInputStream, BulkDataOutputStream, and BulkRandomAccessFile classes in the ti.io package implement bulk, synchronous I/O. They subclass the three classes in java.io that can be used for I/O on binary data (so you can still use all the familiar methods), but they add a few new methods that allow I/O to be performed on entire arrays in a single call, leading to significantly less overhead (in practice speedups of over 60x have been observed for Titanium code that performs a single large I/O using the readArray() and writeArray() methods, rather than many calls to a single-value-at-a-time method like DataInputStream.readDouble()). These classes only handle single-dimensional Java arrays whose elements have atomic types (see §4.2).

```
package ti.io;

public interface BulkDataInput extends java.io.DataInput {
  /** Perform bulk input into A[OFFSET] .. A[OFFSET+COUNT-1] from this
   *  stream.  A must be a Java array with atomic element type.
   *  Requires that all k, OFFSET <= k < OFFSET+COUNT be valid indices
   *  of A, and COUNT>=0  (or throws ArrayIndexOutOfBoundsException).
   *  Throws IllegalArgumentException if A is not an array of appropriate
   *  type.  Throws java.io.IOException if end-of-file or input error
   *  occurs before all data are read. */
  void readArray(Object A, int offset, int count)
      throws java.io.IOException;
  /** Equivalent to readArray (A, 0, N), where N is the length of
   *  A. */
  void readArray(Object primjavaarray)
      throws java.io.IOException;
}


public interface BulkDataOutput extends java.io.DataOutput {
  /** Perform bulk output from A[OFFSET] .. A[OFFSET+COUNT-1] to this
   *  stream.  A must be a Java array with atomic element type.
   *  Requires that all k, OFFSET <= k < OFFSET+COUNT be valid indices
   *  of A, and COUNT>=0  (or throws ArrayIndexOutOfBoundsException).
   *  Throws IllegalArgumentException if A is not an array of appropriate
   *  type.  Throws java.io.IOException if disk full or other output
   *  error occurs before all data are read. */
  void writeArray(Object primjavaarray, int arrayoffset, int count)
      throws java.io.IOException;
  /** Equivalent to writeArray (A, 0, N), where N is the length of
   *  array A. */
  void writeArray(Object primjavaarray)
      throws java.io.IOException;
}


public class BulkDataInputStream
    extends java.io.DataInputStream
    implements BulkDataInput
{
  /** A new stream reading from IN.  See documentation of superclass. */
  public BulkDataInputStream(java.io.InputStream in);

  public void readArray(Object A, int offset, int count)
       throws java.io.IOException;

  public void readArray(Object A)
       throws java.io.IOException;
};


public class BulkDataOutputStream
    extends java.io.DataOutputStream
    implements BulkDataOutput
{
  /** An output stream writing to OUT. See superclass documentation. */
  public BulkDataOutputStream(java.io.OutputStream out);

  public void writeArray(Object A, int offset, int count)
      throws java.io.IOException;

  public void writeArray(Object A)
      throws java.io.IOException;
};


public class BulkRandomAccessFile
    extends java.io.RandomAccessFile
    implements BulkDataInput, BulkDataOutput
{
  /** A file providing access to the external file NAME in mode
   *  MODE, as described in the documentation of the superclass. */
  public BulkRandomAccessFile(String name, String mode)
      throws java.io.IOException;
  /** A file providing access to the external file FILE in mode
   *  MODE, as described in the documentation of the superclass. */
  public BulkRandomAccessFile(java.io.File file, String mode)
      throws java.io.IOException;

  public void readArray(Object A, int offset, int count)
       throws java.io.IOException;
  public void readArray(Object A)
       throws java.io.IOException;

  public void writeArray(Object A, int offset, int count)
       throws java.io.IOException;
  public void writeArray(Object primjavaarray)
       throws java.io.IOException;
};
```

12.18 Controlling Garbage Collection

Two functions in `ti.lang.Ti` provide additional control over when garbage-collections may happen.

`suspend_autogc()` Temporarily suspends the use of automatic garbage collection on the calling process. While suspended, garbage collection will not run on the current process during allocation operations, except in response to explicit calls to System.gc(). In some configurations, suspending collection on one process may also prevent automatic collections from running on one or more other processes. Suspending collection for too long can lead to memory exhaustion.

`resume_autogc()` Resumes the use of automatic garbage collection on a particular process. Must be matched by a previous call to `suspend_autogc()` on this process. Pairs of calls to `suspend_autogc()` and `resume_autogc()` nest; automatic collection will only be enabled outside all such pairs of calls.

Chapter 13

Various Known Departures from Java

1. **Blank finals:** Currently the compiler does not prevent one from assigning to a blank final field multiple times. This minor pathology is sufficiently unimportant that is unlikely to be fixed, but it is best for programmers to adhere to Java's rules.

2. **Dynamic class loading:** Titanium does not implement `java.lang.ClassLoader` or the `java.lang.Class.forName` method.

3. **Thread creation:** SPMD processes are the only thread-like entities that may be created. Java classes that depend on thread creation, such as those in `java.awt` and `java.net`, are consequently not implemented.

4. **Strictfp:** The **strictfp** keyword is accepted, but has no effect.

5. **Covariant return types:** As in Java 1.5, but not 1.4, Titanium allows the return type of an overriding method to be a subtype of that of the overridden method.

6. **Static fields:** The compiler replicates static storage so that a single textual definition of a static field actually represents a distinct variable in each process. See §9.6.

Chapter 14

Handling of Errors

Technically, in those places that the language specifically says that "it is an error" for the program to perform some action, the result of further execution of the program is undefined. However, as a practical matter, compilers should comply with the following constraints, absent a compelling (and documented) implementation consideration. In general, a situation that "is an error" should halt the program (preferably with a helpful traceback or other message that locates the error). It is not required that the program halt immediately, as long as it does so eventually, and before any change to state that persists after execution of the program (specifically, to external files). Therefore, it is entirely possible that several erroneous situations might be simultaneously pending, and such considerations as which of them to report to the user are entirely implementation dependent.

14.1 Erroneous Exceptions

In addition to the erroneous conditions described in other sections of this manual, it is an error to perform any action that, according to the rules of standard Java, would cause the implementation to throw one of the following exceptions implicitly (that is, in the absence of an explicit 'throw' statement):

```
ArithmeticException  ArrayStoreException,
ClassCastException,
IndexOutOfBoundsException, NegativeArraySizeException,
NullPointerException,
ThreadDeath,
VirtualMachineError (and subclasses)
```

Appendix A

Notes

These are collected discussion notes on various topics. This section is not part of the reference manual.

A.1 On Consistency

Note 1. We debated whether there should be a single total order E for a given execution or one for every process in the execution. The latter seems to admit implementations that are not strictly cache-coherent, since processes may see writes happening in different orders. Our interpretation of the Java semantics is the stronger single serial order, so we have decided to use that in Titanium. This is subject to change if we find a significant performance advantage on some platform to the weaker semantics. Even with the single serial order, the semantics are quite weak, so it is unlikely that any program would rely on the difference. The following example is an execution that would be correct in the weaker semantics, but not in the stronger one – we are currently unable to find a motivating problem in which this execution would arise.

```
// initially X = Y = 0


        P1          P2              P3

        X = 2       X = Y           Y = X
        Y = 1

// Separating and labeling the accesses:

            P1          P2              P3
    (A)     Write X     Read Y  (C)     Read X  (E)
    (B)     Write Y     Write X (D)     Write Y (F)

// The following execution constitutes an incorrect behavior:

    Read Y (C) returns 1, Read X (E) returns 2,
    X = 2, Y = 1 at the end of execution.
```

We observe that:

1. Access (C) consumes the value produced by (B) since it returns 1. The only other candidate is (F). Let us assume that (F) indeed wrote the value 1. That would imply:
   (a) (D) hasn't taken place yet
   (b) (E) read 1
   (c) (A) must have written 1, which is false.
2. Similarly (E) consumes the value produced by (A).
3. According to P2, since the final value of X is 2:

   ```
   B < C < D < A
   ```

4. According to P3, since the final value of Y is 1:

   ```
   A < E < F < B
   ```

Note 2. The Titanium semantics as specified are weaker than Split-C's in that the default is weak consistency; sequential consistency (the default in Split-C) can be achieved through the use of volatile variables. However, this semantics is stronger than Split-C's *put* and *get* semantics, since Split-C does not require that dependences be observed. For example, a *put* followed by a *read* to the same variable is undefined in Split-C, unless there is an intervening *synch*. This stronger Titanium semantics is much nicer for the programmer, but may create a performance problem on some distributed memory platforms. In particular, if the network reorders messages between a single sender and receiver, which is likely if there are multiple paths through the networks, then two writes to the same variable can be reordered. On shared memory machines this will not be an issue. We felt that it was worth trying to satisfy dependences at some risk of performance degradation.

Note 3. The Java specification makes this qualification about divisibility only on non-volatile double and long values. It gives the (unstated) impression that accesses to 64-bit volatile values are indivisible. This seems to confuse two orthogonal issues: the size of an indivisible value and the relative order in which operations occur.

A.2 Advantages of Regions [David Gay]

1. The memory management costs are more explicit than with garbage collection: there is a predictable cost at region creation and deletion and on each field write. The costs of the reference counting for local variables should be negligible (at least according to the study we did for our PLDI paper, but I am planning a somewhat different implementation). With garbage collection, pauses occur in unpredictable places and for unpredictable durations.
2. Region-based memory management is safe.
3. I believe that this style of region-based memory management is more efficient than parallel garbage collection. Obviously this claim requires validation.
4. When reference-counting regions instead of individual objects two common problems with reference counting are ameliorated: minimal space is devoted to storing reference counts, and cyclic structures can be collected so long as they are allocated within a single region.

A.3 Disadvantages of Regions [David Gay]

1. Regions are obviously harder to use than garbage-collection.
2. As formulated above, regions will not mesh well with Java threads (which are currently not part of Titanium). You must stop everything to delete a shared region. Currently this is enforced by including a barrier in the shared region deletion operation, but with threads that is no longer sufficient. There are a number of possible solutions, but none of them seem very good:
   (a) Require `r.delete()` to be called from all threads. This would be painful for the programmers.
   (b) The implementation of `r.delete()` can just stop the other threads on the same process. However, to efficiently handle local variables containing references one needs to know all points where a process may be stopped (and obviously if these

points are “all points in the program” then efficiency is lost). So this solution doesn’t seem very good either.

Index