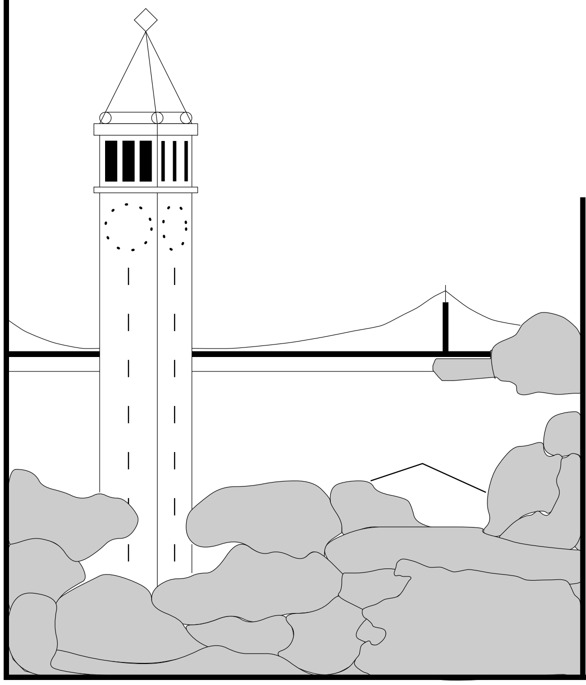
The INFIDEL Virtual Machine

Luigi Semenzato



Report No. UCB/CSD 93-761

25 July 1993

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

INFIDEL is an abstract machine that operates on grids. It is written in L as an extension of Basil. It has been designed as a target for the FIDIL compiler, but it can be programmed directly in L. The machine implements the abstract types 'grid' and 'domain.' Domains represent sets of points with integer coordinates. Grids are an extension of arrays for finite-difference algorithms. These types correspond closely to the FIDIL types 'map' and 'domain.' INFIDEL serves three purposes. First, it is proposed as an intermediate step in the compilation of FIDIL programs. Second, it defines the level at which FIDIL programs and foreign code can be linked together. Third, part of the interface is not only available for direct use in application programs, but is also usable in yet-to-be-written system code that will implement INFIDEL on new architectures.

The INFIDEL Virtual Machine

Luigi Semenzato

25 July 1993

Research supported at UC Berkeley by DARPA and the National Science Foundation under grant DMS-8919074.

Contents

Chapter 1

Overview

INFIDEL is an abstract machine that operates on grids. It is written in L as an extension of Basil. It has been designed as a target for the FIDIL compiler, but it can be programmed directly in L[Sem93].

The machine implements the abstract types *grid* and *domain*. Domains represent sets of points with integer coordinates. Grids are an extension of arrays for finite-difference algorithms. These types correspond closely to the FIDIL types *map* and *domain*.

INFIDEL serves three purposes. First, it is proposed as an intermediate step in the compilation of FIDIL programs. Second, it defines the level at which FIDIL programs and foreign code can be linked together. Third, part of the interface is not only available for direct use in application programs, but is also usable in yet-to-be-written system code that will implement INFIDEL on new architectures.

The next two chapters (2 and 3) describe data structures and algorithms for respectively domains and grids. Chapter 4 is a reference manual for INFIDEL.

Chapter 2

Domains

A domain is a set of indices. More precisely, it is a finite set of n-dimensional points with integer coordinates. Its intended primary use is to describe index sets for operations on grids. As such, its representation is optimized for the types of grids that, in our experience, are most likely to occur in modern algorithms for partial differential equations. Our goal is twofold: the design aims to obtain efficient operations not only on domains, but also on grids with those domains, exposing opportunities for vectorization and parallelization on both regular and irregular shapes.

According to these principles, we have identified several domain representations: *manhattan*, *bitmap*, *thin*, *thick*, *tiled*, *collage*. These representations are supported by INFIDEL. Any of them can represent an arbitrary set of points, but each is optimized for a set of points with specific properties.

Good abstraction dictates that the programmer should not be concerned with the particular representation used, but always think of a domain just as a set of indices. We support this up to a point. Functions that operate on domains (union, intersection, shift, etc.) take domain arguments with arbitrary representations; they make a choice of representation for the result, and compute it. This choice is not always optimal; moreover, it is easy to construct an index set that fares poorly in any of the available representations. However, we believe that what we provide is adequate for a large number of existing modern PDE algorithms.

Here we present briefly each domain representation. A full discussion of their properties, and of the algorithms used to operate on them, is in section 2.2.

2.1 Domain types

2.1.1 Manhattan domains

The basic index set, the one used by arrays in most conventional languages, is an n-dimensional box of points, defined by its lower and upper bounds in each dimension. Its immediate extension is a union of boxes. A box is a convenient object for vectorization, and it is easy to partition for parallelization. A union of boxes maintains these properties. The shape arises frequently—for instance, the boundary region of a box can be described as a union of boxes, as shown in fig. 2.1.

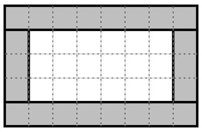


Figure 2.1: The boundary of a box as a union of boxes

A manhattan domain is a union of disjoint boxes in *canonical form*. This form does not minimize the number of boxes necessary to describe the domain, but keeps its number small. The representation is unique for a given set of indices.

2.1.2 Thin and thick domains

The thin representation is used for irregular, sparse index sets. The thin descriptor is an ordered list of points. The thick domain is a manhattan with thin holes. The descriptor is an ordered pair (m, τ), where m is a manhattan descriptor, and τ a thin descriptor, with $\tau \subset m$, and it represents the set difference of m and τ. The manhattan component m of a thick domain is called its *base*. Figure 2.2 shows examples of thin and thick domains.

2.1.3 Bitmap domains

For irregular, medium density situations, the bitmap domain is more appropriate than the thick or the thin. The bitmap descriptor is a boolean grid (see section 3.1) with a manhattan domain, called its *base*. The grid values (true or false) indicate which of the base points belong to the domain. Figure 2.3 shows a bitmap domain.

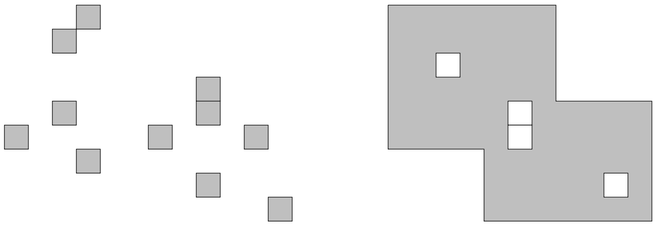


Figure 2.2: A thin domain and a thick domain

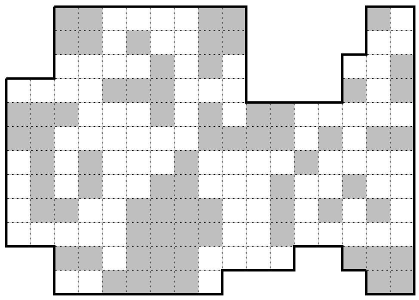


Figure 2.3: A bitmap domain

2.1.4 Tiled domains

A tiled descriptor represents concisely certain regular index sets; roughly speaking, those produced by iterating with non-unit stride over manhattan domains. Such domains are encoded as the intersection of some infinite, regular tiling of the n-dimensional space, and a manhattan domain.

The tiling is obtained by covering the space with identical, n-dimensional tiles. The representative tile is a boolean array. The elements of a tile are called *tesserae*. A true-valued tessera, also called *in-tessera*, means that the point is in the domain; a false-valued tessera (an *out-tessera*) means the opposite. The tiles are laid down so that the lower bound of the first tile is at the origin.

A tiled domain descriptor is the ordered pair (b, t), where b is a manhattan descriptor, called the *base*, and t is a tile. Fig. 2.4 shows an example.

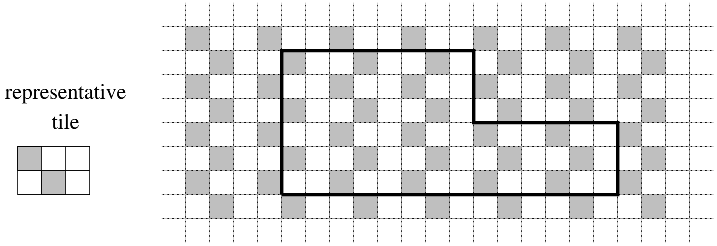


Figure 2.4: A tiled domain

2.1.5 Collage domains

The set of tiled domain is not closed under the union operation, and cases of domains that are almost tiled, but not quite, may arise, as shown in Fig. 2.5. To represent these domains we use the collage descriptor. This descriptor encodes a domain as the union of tiled domains (b_i, t_i) with certain properties, among which: the b_i's are all disjoint, and the t_i's have the same side lengths. This representation captures well union, intersection, and difference of tiled domains, and simplifies gracefully.

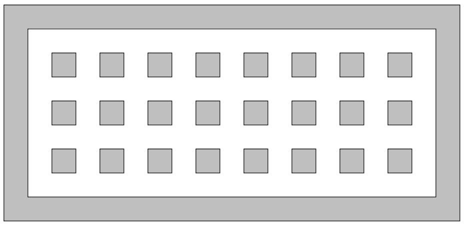


Figure 2.5: Union of two tiled domains that is not tiled

2.2 Domain Algorithms

We present efficient algorithms for operating on each domain type. The operations are of two types: set-theoretic (union, intersection, difference) and geometric (shift, transpose, contract, expand, inject, project). The formal definition of these operators is in appendix B.

In the following sections, we often talk about *descriptors* and the sets of points they represent. For instance, a box descriptors $b = (l, u)$ represents the points in box B with lower bound l and upper bound u. The $\wp$ operator transforms a descriptor into a set of points. For instance: $\wp b = \wp(l, u) = \mathrm{B}$.

2.2.1 Manhattan algorithms

Formal definition of manhattan domain

We give the formal definition of *canonical form* for a manhattan domain.

A box B is the set of points p in $\mathbf{Z}^n$ whose coordinates fall between two points l and u, called the lower bound and the upper bound of B. More precisely:

$$p = \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix},\ l = \begin{bmatrix} l_1 \\ \vdots \\ l_n \end{bmatrix},\ u = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix};$$

$$\mathrm{B} = \{p \in \mathbf{Z}^n \mid l_i \le p_i \le u_i, i = 1, \ldots, n\}.$$

We define an ordering relation for points in $\mathbf{Z}^n$:

$$p < q \ \textbf{iff} \ \exists j \ \text{ such that } \ p_j < q_j \text{ and } p_k = q_k,\ k = j+1, \ldots, n$$

This is just lexicographic ordering, if we assume the coordinate with lowest index to be the the least significant.

A similar relation for boxes is introduced. If box B^α has bounds l^α, u^α and box B^β l^β, u^β, then we have:

$$\mathrm{B}^\alpha < \mathrm{B}^\beta \ \textbf{iff} \ l^\alpha < l^\beta.$$

The *unit vectors* v_i, $i = 1 \ldots n$, are vectors with a 1 in the i-th position and 0 elsewhere. For instance:

$$v_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

A *canonical form* for a set of points $\mathcal{P}$ in $\mathbf{Z}^n$ is a sequence of boxes $\mathcal{C} = (\mathrm{B}^1, \ldots, \mathrm{B}^m)$, ordered by the $<$ relation above, forming a partition of $\mathcal{P}$ with the property of *prioritized maximal extension*:

$$\forall p, q \in \mathcal{P} \ \text{ with } \ p = q + v_i$$

$$\left.\begin{array}{l} p \in \mathrm{B}^{\alpha}, \\ q \in \mathrm{B}^{\beta}, \\ \alpha \neq \beta \end{array}\right\} \Longrightarrow \begin{bmatrix} l_1^{\alpha} \\ \vdots \\ l_{i-1}^{\alpha} \end{bmatrix} \neq \begin{bmatrix} l_1^{\beta} \\ \vdots \\ l_{i-1}^{\beta} \end{bmatrix} \vee \begin{bmatrix} u_1^{\alpha} \\ \vdots \\ u_{i-1}^{\alpha} \end{bmatrix} \neq \begin{bmatrix} u_1^{\beta} \\ \vdots \\ u_{i-1}^{\beta} \end{bmatrix}.$$

In the case $i = 1$, the above relation is equivalent to:

$$\forall p, q \in \mathcal{P} \text{ with } p = q + v_1 \;\; \exists \alpha \text{ such that } p, q \in \mathrm{B}^{\alpha}.$$

This property can be intuitively explained with this rule: a box should extend as much as possible along dimension i, as long as that does not preclude maximum extension along dimensions $i - 1, \ldots, 1$.

Breaking and coalescing

To reason about boxes as sets of points, we associate a unit box to each point; that is we associate point $p = [p_1 \ldots p_n]$ to the n-dimensional volume in $\mathbf{R}^n$ with lower bound p and upper bound $p + [1 \ldots 1]$. We represent a box as the union of the unit boxes of its points. The d-faces of the box are the d-dimensional rectangles in $\mathbf{R}^n$, with $d \leq n$, that define its boundary. For instance, when $n = 3$, the box is a parallelepiped, the 2-faces are the faces of the parallelepiped, the 1-faces its edges, the 0-faces its vertices. The n-face is the box itself.

Most of the manhattan algorithms utilize two procedures: BREAK and COALESCE. BREAK takes an arbitrary list of boxes and produces a list of boxes in *sufficiently fragmented form* representing the same set of points. This form has the property that the $2n$ planes defining a box B do not cut any box adjacent to or overlapping B. In other words, any two boxes B_i and B_j from a list in sufficiently fragmented form are in one of three possible situations with respect to each other:

1. they are identical;
2. they are non-adjacent, that is $\forall p \in \mathrm{B}_i \forall q \in \mathrm{B}_j, ||p - q|| \geq 2$;
3. they share a d-face, $0 \leq d \leq n$ (the special case $d = n$ is the same as situation 1, i.e. they are identical).

COALESCE takes a list of disjoint boxes in sufficiently fragmented form and returns a list of boxes in canonical forms representing the same set of points. It proceeds by joining boxes as much as possible along dimension 0, then joining the boxes of the result as much as possible along dimension 1, and so on.

Union, intersection, difference

The union of manhattan domains x and y is computed as follows:

1. compute the union of the boxes in the canonical form of x and y;
2. compute a fragmented form using BREAK;
3. find pairs of identical boxes;
4. eliminate one box from each pair, keep unpaired boxes;
5. compute the canonical form using COALESCE.

For intersection and symmetric difference, only step 4 changes:

4. (**intersection**) keep one box from each pair, discard unpaired boxes.
4. (**symmetric difference**) discard all pairs, keep unpaired boxes.

The difference $x - y$ is computed as $x \cap (x \otimes y)$, where $\otimes$ is the symmetric difference operator.

Transpose, project

The transposition (or projection) of a box B_x is also a box B_y, whose bounds can be computed with simple operations on the bounds of B_x. However, performing these operations on a list of boxes in canonical form does not produce, in general, a canonical form. The resulting list of boxes is still disjoint, and it suffices to break and coalesce it to compute the result. These are the steps for evaluating the transposition (or contraction) of a manhattan domain m:

1. transpose (or contract) each box of m;
2. compute a fragmented form using BREAK;
3. compute the canonical form using COALESCE.

Contract

Contracting requires an extra step, because the contracted boxes may overlap. Between the BREAK and the COALESCE step, redundant boxes must be eliminated, similarly to the union algorithm.

Shift, expand

These are the easiest operations, as shifting or expanding each box in a manhattan domain preserves the canonical form. No breaking and coalescing is necessary.

Inject

This is the only operation on a manhattan domain that does not produce another manhattan domain (except in trivial cases). The result of inject(m, S) is the tiled domain (b, t), where the base is given by $b = \text{expand}(m, S)$, the tile has sides with dimensions S, and its only in-tessera is the one in the lower left corner.

2.2.2 Tiled and collage algorithms

The tiled representation is somewhat more *ad hoc* than the manhattan representation. The canonical form is not unique: there can be many different tiled representations of the same domain. However, once the tile size is chosen, the form is unique. Choosing the most convenient tile size appears to be a hard problem, and we do it only for some specific cases.

Tiles that produce the same tiling of the space are *equivalent*. The smallest tile of an equivalence set is an *irreducible* tile. All tiles in the equivalence set are obtained by replicating the irreducible tile in the set along one or more directions. The *replication function* R takes a tile t and a replication factor $S = [s_1 \ldots s_n]$ and replicates t s_i times along direction i.

A domain b is *regularly tiled* by a tile t with side lengths ε if

$$b = \text{expand}(\text{contract}(b, \varepsilon), \varepsilon).$$

In this case the boundary of b follows tile boundaries when the space is tiled by t, and never cuts any tile. A tiled domain (b, t) is *regular* if its base b is regularly tiled by t.

A collage domain c is encoded as a set of regular tiled domains with disjoint bases, called the *components* of c:

$$\wp c = \bigcup_{(b_i, t_i) \in c} \wp(b_i, t_i)$$

where b_i is a manhattan domain, t_i a tile with side lengths ε_i, and the following additional properties hold:

1. $\varepsilon_i = \varepsilon_j$ for all valid i, j (all tiles have the same dimensions);

2. $\forall i, j$ such that $i \neq j$, $t_i \neq t_j$ (the tiles are all different);

3. there is no replication factor S (except for the trivial replication factor $[1 \ldots 1]$) such that $t_i = R(u_i, S)$ for all i, with adequate u_i (the tiles are mutually irreducible).

4. t_i must have at least one in-tessera for all i (there are no empty components).

5. t_i may not be the unit tile (if it were, the collage would have been promoted to a manhattan).

This representation allows the run-time system to reconstruct manhattan domains from unions of tiled domains in certain cases. More specifically, if the result of an operation on collage domains has a single component with a full tile, a manhattan descriptor is returned.

In some cases a collage domain with more than one component could be represented more concisely by a manhattan descriptor, but the system does not recognize it. Figure 2.6 is an example.

Figure 2.6: A collage domain that should be a manhattan

In general, there is no guarantee that a "minimal tile"[1] is always used. Set operations on domains with differently-sized tiles may yield a result with a larger tile size. This has a negative impact not so much on the cost of domain operations, as much on the efficiency of vectorizing operations on grids with such domains. We expect this not to be a problem in practice, because of the relatively regular patterns of expansion and contraction found in most algorithms.

[1]This is just an intuitive notion. A tiled domain can always be encoded by a unitary tile and a manhattan base with lots of small boxes.

Kernel descriptors

It is easier to perform certain operations on an alternative representation for a collage domain, the *kernel descriptor*. This representation is similar to the collage descriptor but has different properties. A kernel descriptor is a set of tiled descriptors with regularly-tiled bases and exactly one in-tessera in each tile. The bases may overlap, and the tiles are all different but have all the same size.

The implementation of certain operations produces sets of tiled descriptors with irregularly-tiled bases as intermediate results. When these sets satisfy all other properties of a collage or a kernel descriptor, they are called, respectively, *quasi-collage* and *quasi-kernel*.

Union

We show only the union algorithm; intersection and symmetric difference share the same general structure and differ only in obvious details.

We first consider the case of two collage domains with the same tile size. Similarly to the manhattan situation, the union algorithm for domain descriptors x and y proceeds by *mutually decomposing* the bases of both operands. Every tiled component (b_i^x, t_i^x) of x is partitioned into a set of tiled components $\{(b_{ij}^x, t_i^x)\}$:

$$\wp(b_i^x, t_i^x) = \bigcup_j \wp(b_{ij}^x, t_i^x)$$

with the following property: given (i, j), either $b_{ij}^x \cap b_k^y = \emptyset$ for all k, or $b_{ij}^x \subseteq b_k^y$ for some k. All of this holds when exchanging x and y. The result is that every decomposed base of x is either identical to a decomposed base of y, or disjoint from all of them—and vice versa. The procedure DECOMPOSE is used to compute these subdivisions. After subdividing, the union is performed component by component on the decomposed domains. Then components with identical tiles are merged, and property 3 and 5 are imposed in order. Property 4 is preserved by the union algorithm, but must be imposed in the intersection and symmetric difference, since the algorihm may create empty components that must be removed. The following procedure takes domains x and y and computes the result $x \cup y$ in d_u:

1. set $d_x \leftarrow$ DECOMPOSE(x, y), $d_y \leftarrow$ DECOMPOSE(y, x);
2. set $d \leftarrow d_x \cup d_y$.
3. set $d_u \leftarrow \emptyset$;

4. repeat until d is empty:

   (a) take a tiled component (b, t) out of d;

   (b) if there is another tiled component (b, t') (same base, different tile) in d, take that out as well and set u to the descriptor of the union of (b, t) and (b, t'). This is equal to $(b, t \wedge t')$, where the $\wedge$ operator is the logical-or of the operands' tesserae.

   (c) if there is no other tiled component with the same base in d, set $u \leftarrow (b, t)$;

   (d) set $d_u \leftarrow d_u \cup u$;

5. replace all components of d_u with the same tile with a single component, taking the union of their bases;

6. if all the tiles can be simplified using the same replication factor, do so;

7. if d_u has a single component with a unitary tile, convert it to a manhattan descriptor.

When the tiles of x and y do not have the same size, we construct two descriptors, x' and y', that encode the same index sets as x and y respectively, and have equally-sized tiles. This is done by computing the least common multiple (LCM) of the two side lengths (one from x, one from y) in each direction. The vector thus produced is called the *LCM-size*. Each tile t_i^x of x and t_i^y of y is replicated by an adequate factor to produce a tile whose size is the LCM-size. Since these tiles produce the same tiling of the space, it is not necessary to change the bases. The descriptors x' and y' are quasi-collage; the algorithm proceeds as given, but at the end a regularization step is necessary, as described in the next section.

Regularization

Given a quasi-collage descriptor c', this procedure computes a collage descriptor c representing the same set of points:

1. compute a quasi-kernel descriptor k' from c'. To do so, break every component of c' with N in-tesserae into N components with 1 in-tessera, and merge components with identical tiles.

2. Compute a kernel descriptor k from k' by regularizing the bases. Given an irregular tiled component (b_{irr}, t), with a tile size vector ε, and coordinate vector σ for the single in-tessera of t, the base for an equivalent regularly-tiled descriptor is given by

$$b_{reg} = \text{expand}(\text{project}(\text{shift}(b_{\text{irr}}, -\sigma), \varepsilon), \varepsilon).$$

3. Compute c from k by transforming the components of k into collage descriptors, and taking their union.

Shift

A collage domain is shifted by shifting each component, and then regularizing the result. To shift a component by a shift amount S, the base is shifted by S, and the tile is *rotated* by S. Rotating a tile is equivalent to shifting it with wrap-around, as if the tile was closed onto itself in an n-dimensional toroidal shape.

Transpose, expand, inject

Transposition (expansion) of a collage domain is done component-wise, by transposing (expanding) the base and the tile of each component. Injection is achieved by expanding the base and injecting the tile of each component. It is easy to verify that the collage properties are preserved by these transformations. No regularization is needed.

Contract, project

Because division is always harder than multiplication, contraction (projection) requires one extra step. If the contraction (projection) factor in each direction is a multiple of the tile size in that direction, then it is sufficient to contract (project) each tile and each base. If not, the tile must first be replicated until its size is a multiple of the contraction factor. Then the domain must be regularized; then each base and each tile is contracted (projected).

2.2.3 Bitmap algorithms

Bitmap domains are represented as logical grids with manhattan domains. Operations on bitmap domains have corresponding operations on grids. With no exception, the algorithms are quite easy to derive and express in terms of the INFIDEL grid primitives, and we don't discuss them here.

Operations on bitmap domains always produce bitmap domains. The INFIDEL function `simplify-bitmap-domain` returns a manhattan descriptor when its argument is a completely full bitmap domain (all the elements of the grid are true), and the null domain descriptor when its argument is a completely empty bitmap. This operation is somewhat expensive, so it is not done automatically after each bitmap operation that might take advantage of it.

2.2.4 Thin algorithms

A thin domain descriptor is a list of points in lexicographic order. The algorithms for operating on these descriptors are also quite obvious. The sorted list representation makes the set operations (union, intersection, difference) much faster than if a manhattan descriptor had been used, because adjacent points in a manhattan descriptor are merged into a single box.

When one operand of certain set operation is thin, and the other is not, it may be hard to determine the best representation for the result. The system uses parameterizable heuristics to decide if, for instance, the union of a thin and a manhattan should produce a thin or a manhattan descriptor. In certain cases, it is clear what the result should be: for instance, the difference of a manhattan and a thin produces a thick descriptor if they are not disjoint, and a manhattan otherwise.

2.2.5 Thick algorithms

A thick descriptor k is the pair (m, τ), where m is a manhattan descriptor and τ a thin descriptor, with $\tau \subset m$. The set of points described by k is $\wp(m - \tau)$. Set operations on thick descriptors are easily described (and implemented) in terms of operations on manhattan and thin descriptors, using set algebra. For instance, if $k_x = (m_x, \tau_x)$ and $k_y = (m_y, \tau_y)$, then we have:

$$\begin{aligned} k_x \cup k_y &= (m_x - \tau_x) \cup (m_y - \tau_y) \\ &= (m_x \cup m_y, (\tau_x - s_y) \cup (\tau_y - s_x) \cup (\tau_x \cap \tau_y)). \end{aligned}$$

Geometric operations are also straightforward.

2.3 INFIDEL Domain Interface

2.3.1 The domain type

The type of a domain expression in INFIDEL is (domain n), where n is its dimensionality. After transformation, variables of this type acquire type `t`. Domain objects are garbage-collected and never need to be explicitly freed, although it is possible to free them passing them to `free-gc`. Most domain descriptors are small, with one exception: bitmap domains can occupy as much space as the grids they describe, and it may be desirable to free them explicitly.

Type-predicate operators allow determining what descriptor is used to encode a given domain. The structure names for domain descriptors are `manhattan`, `collaged`, `thind`, `thickd`, `bitmapd`. INFIDEL also defines the `tiled` structure, which is only used as a subcomponent of a collage domain. The type predicate is obtained by prepending `-p` to the structure name (`manhattan-p`, `collaged-p`, etc.).

2.3.2 Domain constructors

INFIDEL has three operators to construct domains. The operator `domcons` takes these arguments: l and u (the lower and upper bounds), and d (the dimensionality). It returns a manhattan descriptor consisting of a single box with the given bounds. The bounds l and u are passed as pointers to integers, and should point to integer arrays with length d.

The second operator, `domcons-t`, takes these arguments: p (the points in the domain), n (the number of points), and d (the dimensionality). It returns a thin descriptor with the given points, which are copied. The point array p is passed as a pointer to integer, which should point to an integer array of length nd.

The third operator, `to-domain`, takes a boolean grid and returns a bitmap domain. A copy of the grid is made for the descriptor (using the copy-on-demand mechanism described in section 3.2), so the grid can be safely freed.

These are the only constructors. The other types of descriptors are produced automatically as needed. Explicit conversion routines (`to-bitmap`, `to-manhattan`, `to-thin`) are available, but it is hoped that they will not be needed at the application level.

Chapter 3

Grids

3.1 Overview

Grids are n-dimensional arrays with arbitrary index sets. The INFIDEL virtual machine offers various types of operations on grids. We divide them in two main classes. *Elementwise* operations are those that access individual elements of one or more grids in a data-parallel fashion. *Remapping* operations are those that change the association between indices and elements in a grid. In INFIDEL, elementwise operations are eager, and remapping operations are lazy. An instance of a lazy operations can be made eager by adding an elementwise copy. The reason for the lazy semantics is precisely the avoidance of the extra copy.

In theory, grids could be used to represent a variety of data structures: sets, lists, vectors, matrices. In practice, grid operations are designed to work optimally for large grids with scalar elements, and index sets of the types described in section 2. These are the situations in which the target applications spend most of their execution time and memory resources. When conventional programming techniques are used in these situations, full exploitation of the features of the host computer may require considerable effort. The grid interface reduces this effort.

The interface is at a conveniently high level for the algorithm designer, and is meant to hide most architectural characteristics of the target computer. Although the design of the interface has taken into account future multiprocessor implementations, the only currently available implementation (the one we describe here) is for a uniprocessor with vector units. We consider this not only a useful tool by itself, for use on vector machines such as Cray computers, but also a building block for future parallel implementations.

As hinted, the interface is not completely machine-independent. However, the machine-dependent parts are not meant to provide a different functionality, and are included because they represent opportunities for optimization. It is up to the compiler writer, or the L programmer, to take advantage of them.

The INFIDEL run-time library performs several dynamic optimizations, including choice of optimal vectorizing direction in multidimensional grids, delayed allocation and early release of grid memory. Most of these optimizations are automatic: some require that certain high-level hints be passed to grid operations.

3.2 Chunks

The run-time system maintains three components for each grid: domain, data descriptor, and data. The data is divided in *chunks*. In the uniprocessor implementation, this division allows several dynamic memory optimizations. Chunks have a maximum size, to reduce heap fragmentation. They are reference-counted, and they can be shared among grids. The sharing is transparent. A copy-on-demand mechanism guarantees correct update semantics for shared chunks. Chunks are also allocated on demand: `grid-alloc` returns a grid without allocating its chunks. The first time an element of a chunk is set, that chunk is allocated. Correspondingly, an *early-release* mechanism is provided. The mechanism deallocates chunks immediately after their last use in an elementwise operation.

Chunkification affects almost every aspect of the interface design and system implementation; therefore we postpone a full discussion of its properties to the sections describing individual operations.

3.3 The grid type

Grid expressions have type `(grid` n e`)`, where n is the dimensionality of the grid, and e is the element type. Just like domains, grids are L boxed structures, and they are stored in locations of type `t`. Grids, therefore, can and will be garbage-collected. However, the garbage collection algorithm cannot guarantee that grids that are no longer used will be timely freed (or even freed at all). Especially when grids are large, `free-gc` should be used to reclaim the space as soon as possible. Optionally, a grid may be freed during its last use, as described in section 3.5.

3.4 Grid allocation

The arguments to the INFIDEL operator `grid-alloc` are: domain, element type, and optional hints. The return value is a grid with the given domain and element type.

The hints are passed in a structure of type `grid-alloc-hints` that provides various information about the grid, among which how the grid's data should be partitioned. Partitioning hints may be given at different levels of abstraction: from a complete specification of the partitioning, to hints about the preferred vectorization direction. When no partitioning hint is given, the system chooses an initial partition automatically, based on the grid's domain and element type.

A grid's chunks are allocated lazily. A chunk is allocated when one or more of its elements are set for the first time. Attempting to use a value from an unallocated chunk results in a run-time error.

Once all chunks of a grid are allocated, their total size is at least as large as the number of elements in the grid multiplied by the element size. It can be larger in the following cases:

- when the grid has a thick or bitmap domain D, the layout of the data in the chunks is the same as for a grid whose domain is the base of D (its manhattan component). This layout strategy sacrifices space to regularity. The storage locations corresponding to a **false** value in the boolean grid of the bitmap domain, or the thin holes of the thick domain, are not normally accessible.
- A value $\mu > 0$ may be passed in the `allowable-memory-overhead` field of the optimization hints. This hint informs the system that it is acceptable to allocate μ times more space than strictly necessary for a manhattan domain, if doing so helps in avoiding bad strides along one or more dimensions.

3.5 Elementwise operations

INFIDEL offers essentially a single data-parallel elementwise operation, namely `map-grid`. The arguments to `map-grid` are: an operator f, a domain D called the *restriction domain*, and one or more grids $G_1, \ldots, G_g$. The domain D is intersected with the domain of each grid G_i to obtain the *computation domain* D', that is $D' = D \cap \left(\bigcap_{i=1}^{g} \delta(G_i)\right)$, where $\delta(G)$ denotes the domain of grid G. The operator f takes g arguments. f should have no side effects other than storing values into its arguments. `Map-grid` applies f to $G_1[p], \ldots, G_g[p]$ at each point

$p \in D'$, in parallel. The square brackets denote the indexing operation: $G[p]$ is the location at which element p of grid G is stored. When used in a value context, the same notation stands for the value itself. Examples:

```
(map-grid D setf :grids (A B))
```

stores elements of grid `B` into grid `A`, restricted to domain `D`.

```
(map-grid D
          (slambda (x y z) (setf x (+ y z)))
          :grids (A B C))
```

computes the elementwise sum of `B` and `C` and stores the result in `A`, again restricted to domain `D`.

A variation of `map-grid` is `map-grid*`, which takes the extra argument p. A variable named p is bound to each index during execution, and it may be used in the operator body (which should be an `slambda`).

3.5.1 Arguments to elementwise operations

The grid arguments to `map-grid` must either be variables, or constant-factor remapping expressions of variables. Example:

```
(map-grid D
          (slambda (x y z) (setf x (+ y x)))
          :grids ((A)
                  ((grid-shift B [0 1]))
                  ((grid-shift B [0 -1]))))
```

The main reason for allowing remapping expressions as arguments is that it exposes opportunities for an important vector optimizations: improved vector-register allocation through strip-mining. This optimization requires knowing, at compile time, that two of the arguments are remappings of the same grid (`B` in the example); and it also requires knowing the remapping parameters (in this case, the shift amount).

In general, instead of using remapping expressions in `map-grid`, one can obtain the same effect by storing the (lazily) remapped grids in temporary variables.

3.5.2 Optimization directives in elementwise operations

Additional information can be passed to `map-grid` for the purpose of optimization. The information can be specific to a grid argument, or to the restriction domain. Each item of information is called, respectively, *grid directive* or *domain directive*. Grid directives are passed as keyword arguments associated with grid parameters. Domain directives are passed as additional keyword arguments to `map-grid`.

Here we present the directives and explain their meaning. In section 3.7 we discuss in more detail the way in which they operate.

The `:free` grid directive

Grids are large objects, and memory is one of the critical resources in the target applications The `:free` directive, when applied to grid G, informs `map-grid` that this is the last use of G and the early-release mechanism should be applied to its chunks. For instance, if this were the last use of `B` and `C`, one might write:

```
(let ((((grid 2 float) A) (grid-alloc D float)))
   (map-grid D (slambda (x y z) (setf x (+ y z)))
               :grids ((A) (B :free t) (C :free t)))
   ...
```

Using the `:free` directive causes each chunk to be freed immediately after its last use. Since chunks are also allocated on demand, the total amount of memory used in the above example is likely to exceed the amount of memory used by `B` and `C` only by a few chunks, and by just a single chunk in an optimal situation.

The `:partition` grid directive

In certain cases, through compile-time analysis in the front end, or knowledge of the algorithm, the data descriptor of a grid (also called *partition descriptor* can be computed at compile time. When the data descriptor of all grids in a `map-grid` operation is a constant, it can be passed to `map-grid` via the `:partition` grid directive. If the restriction domain is also a constant, several simplifications can be performed. The resulting code has less run-time overhead and is more suitable for small grids.

The `:mask` domain directive

This directive affects only the size of the generated code, with no significant impact on its speed. It indicates that the computation domain could be a bitmap or thick domain, and its default value is conservatively `t`. In the default case the compilation of `map-grid` produces two versions of the statement(s) that operate on individual grid elements: one for the bitmap and thick domains, and one for all others. When the `:mask` directive is `nil`, the bitmap/thick version is not produced.

3.5.3 Unsafety in elementwise operations

Elementwise operations have *unsafe* semantics. When the computation domain D is of type thick or bitmap, the *effective computation domain* D_e is the entire base of the thick or bitmap domain; that is, its manhattan component. This means that f is applied to grid elements over D_e instead of D. This choice allows good vectorization at the expense of operating on more elements than strictly needed. The operator f is modified to prevent side effects for those points in $D_e - D$. However, some architectures make it impossible, or difficult at any rate, to prevent all side effects. In particular, on several members of the Cray family it is not possible to selectively disable execution of arithmetic operations that may cause floating point exceptions.

We consider the issue in more detail. On the Cray X-MP and Y-MP the effect of storing a vector element can be conditionally nullified by using the Conditional Vector Merge instruction. Given a boolean grid b with domain D_e such that $b[p] = p \in D$, then the assignment

$$x[p] \leftarrow \mathcal{E}$$

(where $\mathcal{E}$ is some side-effect-free expression) is rewritten as

$$x[p] \leftarrow \textbf{if } b[p] \textbf{ then } \mathcal{E} \textbf{ else } x[p].$$

This almost obtains the desired result. Unfortunately, the execution of $\mathcal{E}$ can produce floating point exceptions on points in $D_e - D$. Such exceptions are completely meaningless and should be ignored. Most processor architectures (including the Cray) do not allow taking the exception conditionally. Also, the Cray does not support IEEE Floating Point-style exceptional values; therefore turning off exceptions is generally not desirable, as errors can go undetected.

A possible way to avoid unwanted exceptions is to conditionally replace the operands of every floating-point operation with "safe" values. This may be too expensive if intermediate results need to be replaced as well. The expression $\mathcal{E}$ can be analyzed to determine safe input values, that would guarantee all intermediate results to be safe. Even so, the cost of the conditional replacement may be too high and we have not implemented this solution.

The INFIDEL programmer must deal with these situations directly. As a palliative, the INFIDEL `set-hidden-elements` operator takes a grid and a value, and it stores that value in all "unreachable" elements of the grid: those whose indices are in $D_e - D$. The programmer should choose a "safe" value for those points, that is, one that will not produce exceptions in subsequent elementwise operations. The programmer's understanding of the algorithm enables her to use `set-hidden-elements` sparingly.

3.6 Remapping Operations

The remapping operations change the association between indices and elements of a grid, and possibly add or remove elements. They are: shift, transpose, inject, project, restrict (the FIDIL **on** operator), and merge (the FIDIL disjoint union operator). They all have *lazy* semantics. The result of these operations does not require allocation of additional data memory, but only a typically small amount of descriptor memory. This is achieved by sharing the chunks of the result with the chunks of the operand (or operands). The operations however return a conceptually new object, not just a different view of the same object.

The lazy semantics do not always optimize the resulting program in terms of speed and memory usage. The alternative is to use *eager* versions of the same routines, which are obtained by combining the lazy versions with an elementwise copy on a newly allocated grid. The optimal choice between eager and lazy semantics in a specific situation depends on many factors, among which: the parameters of the operation, the target architecture, and the subsequent reference pattern of the grids involved.

A lazy remapping of a sufficiently large grid is cheap, when compared to the eager version, as no data is copied. The remapping affects the cost of operations that access the remapped grid's elements (that is, `map-grid`). For the vector processor implementation, using a lazily-remapped grid in an elementwise operation corresponds to a simple index translation that in most cases does not affect the vectorization efficiency. In these cases the access overhead is almost nil.

The availability of lazy remapping semantics is useful even on a distributed-memory multiprocessor (DMMP); for instance, when two or more remappings are applied to a grid before any of the elements are used. On a DMMP, however, accessing a lazily-remapped grid may involve moving data across the interconnection network, and is much more expensive. To avoid duplications of transfers, the semantics should be those of *latched evaluation* [Sku90]. In this scheme, when a lazily-remapped grid is used for the first time, new chunks are permanently allocated for it; and further references to that grid use the new chunks.

3.7 Grid Algorithms

In this section we present the algorithms used in the vector processor implementation of elementwise and remapping operations.

A grid G is a triplet (d, P, C), where d is the domain descriptor, P the *data descriptor*, and C the *chunk vector*. The domain descriptor encodes the domain of the grid as one of the objects presented in section 2. The data descriptor specifies how points in the domain map into locations in the chunks. The chunk vector is an array of chunks. A chunk is a one-dimensional array where G's elements are stored. Fig. 3.1 gives a visual representation of G.

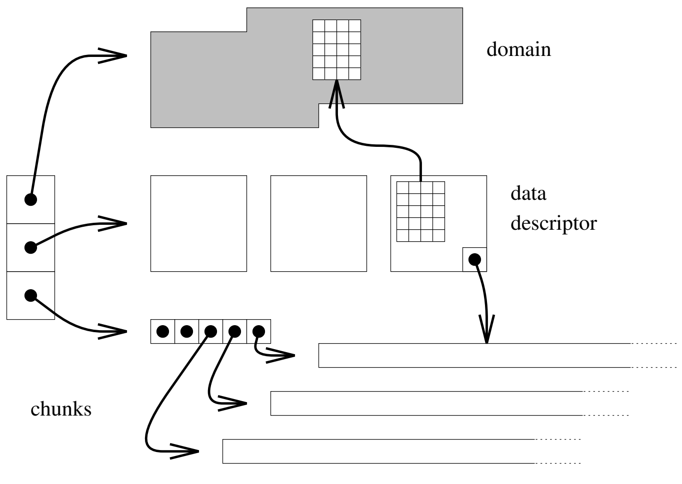


Figure 3.1: A grid.

3.7.1 Section descriptors

Given a grid $G = (d, P, C)$, the grid element $G[p]$, with $p \in d$, is stored in $C[c][I]$, where $c = c(p, P)$ and $I = I(p, P)$. We call c the *chunk index* and I the *linear index*. This section explains how c and I are computed from p and P.

The data descriptor P of G is a set of *section descriptors*. Each of these encodes the layout of a subset of G's elements that can be accessed in a regular and efficient way. There are two types of section descriptors. One, the *thin section descriptor*, is used when the domain descriptor d is thin; the other, the *tiled section descriptor*, in all other cases. We discuss the tiled descriptor first.

A tiled section descriptor S is the tuple $(b_S, \varepsilon_S, \sigma_S, c_S, z_S, \phi_S)$ where:

- b_S is a box descriptor;
- ε_S (an injection factor) and σ_S (a shift factor) are integer vectors of length n;
- c_S (the *chunk index*) is an integer;
- z_S (the *zero index*) is also an integer;
- ϕ_S (the *stride vector*) is an integer n-vector.

The box, injection factor, and shift factor represent the section domain δ_S, by the following relation:

$$\delta_S = \text{inject}(\wp b_S, \varepsilon_S) \ll \sigma_S$$

(recall that $\ll$ is the shift operator). This is equivalent to a tiled domain with a single in-tessera, or a kernel domain with a single component.

The chunk index, zero index, and stride vector encode the mapping between an index and a position in a chunk. For $p \in \delta_S$, the following formula defines the relation between p, c, and I:

$$G[p] \equiv C[c_S][z_S + p_S \cdot \phi_S] \tag{3.1}$$

with

$$p_S = \frac{p - \sigma_S}{\varepsilon_S}. \tag{3.2}$$

and $x \cdot y$ is the inner product of x and y. We call p_S the *section index*. Using p_S instead of p in 3.1 may seem confusing, but is essential to guarantee the correct functioning of lazy remappings.

A thin section descriptor consists of a thin domain descriptor τ, a point range $\rho = (i_l, i_u)$, a chunk index c and a linear index z. The point range specifies which points of τ are included in the section, by giving the indices of the first and last point in τ's list of points. The location of the first element of the section is $C[c][z]$. Following points map into subsequent elements.

3.7.2 Non-thin elementwise operations

We describe the implementation of an elementwise operation with domain D on grids $G_1, \ldots, G_g$, with $D \subseteq \delta(G_i)$), for the cases in which the descriptor of D is not thin.

Since D may be fairly irregular, and the grids are partitioned, one important issue is computing efficiently the locations of the grids' elements. We do this by decomposing the operation into many elementwise operations on disjoint subdomains whose union is D. For each of these subdomains the linear index generation is regular and allows vectorization.

A *computation partition* P^C is a set of disjoint section domains δ_j^C, such that $\bigcup_j \delta_j^C = D$, and given a grid G_i, there exist a k such that $\delta_j^C \subseteq \delta(S_k^{G_i})$; that is, each computation section domain is a subset of some section domain of each grid.

Every computation section domain δ^C is described by the familiar tuple $(b_C, \sigma_C, \varepsilon_C)$, with:

$$\delta^C = \text{inject}(\wp b_C, \varepsilon_C) \ll \sigma_C.$$

The indices of the section are obtained by

$$p = p_C \varepsilon_C + \sigma_C, \quad p_C \in b_C.$$

The section index p_S is obtained from equation 3.2:

$$p_S = \frac{p_C \varepsilon_C + \sigma_C - \sigma_S}{\varepsilon_S}.$$

The linear index I then is given by:

$$I = z_S + \frac{\sigma_C - \sigma_S}{\varepsilon_S} \cdot \phi_S + p_C \cdot \frac{\varepsilon_C}{\varepsilon_S} \phi_S.$$

which we can rewrite as

$$I = z_{CS} + p_C \cdot \phi_{CS}$$

with $z_{CS} = z_S + \frac{\sigma_C - \sigma_S}{\varepsilon_S} \cdot \phi_S$ and $\phi_{CS} = \frac{\varepsilon_C}{\varepsilon_S} \phi_S$. Showing that the integer divisions in the last two formulas are exact is left as an act of faith to the reader.

This shows that in order to compute the linear index from p_C it is sufficient to use z_{CS} instead of z_S, and ϕ_{CS} instead of ϕ_S. This is convenient because p_C is determined only by the computation section domain and not (directly) by any of the grids involved.

Let f be the operator of the elementwise operation. The steps for executing the operation are:

1. obtain P^C by splitting D and all the $\delta(S_k^{G_i})$;
2. for each $\delta^C \in P^C$ do the following:
   (a) find the section S of each grid such that $\delta^C \subseteq \delta(S)$; we do this with a linear search through the grid's partition descriptor;
   (b) compute the zero index z_{CS} and the stride vector ϕ_{CS} for each grid;
   (c) generate all indices $p_C \in b_C$ and evaluate f on the chunk elements indexed by $z_{CS} + p_C \cdot \phi_{CS}$.

This description omits the memory management actions that release dying chunks after their last use. The scheme assigns a reference count to each chunk. In the preamble of an elementwise operation, the counts of dying chunks (those belonging to grids that are dead after the operation) are increased to reflect the number of times each chunk will be referenced during the operation. Then the grid is freed; but those chunks that have further uses remain allocated. After operating on each computation section, the count of each dying chunk used in that section is decremented, and the chunk is freed when the count reaches zero.

3.7.3 The universal transducer

The *universal transducer* T is a function that encodes arbitrary sequences of applications of inject, project, shift, and transpose, on either domains or grids. We use the universal transducer in the implementation of the `map-grid` operator, and in the evaluation and simplification of remapping expressions.

The first argument to T is a domain, or a grid; the second argument is a *transducer factor*, the tuple $(\sigma, \kappa, \varepsilon, \sigma', \omega)$. We define

$$T(x, \tau) \equiv \text{transpose}(\text{inject}(\text{project}(x \ll \sigma, \kappa), \varepsilon) \ll \sigma', \omega)$$

with $\tau = (\sigma, \kappa, \varepsilon, \sigma', \omega)$. We overload T to operate on points as well, with the following definition:

$$T(p, \tau) = \left(\frac{p + \sigma}{\kappa} \varepsilon + \sigma' \right) \otimes \omega$$

where $\otimes$ is the *permutation* operator, thus defined: $(x \otimes y)[i] = x[y[i]]$. The elements of y are restricted to be a permutation of $[1, 2, \ldots, n]$. Every permutation factor y has an inverse, y^{-1}, defined by:

$$(x \otimes y) \otimes y^{-1} = (x \otimes y^{-1}) \otimes y = x \quad \text{for all } x.$$

Also note that $\otimes$ is associative: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$.

With this definition of $T(p, \tau)$, given a grid G, the following identity occurs:

$$G[p] = T(G, \tau)[T(p, \tau)] \tag{3.3}$$

when $p \in \text{domainOf}(G)$ and $(p - \sigma) \bmod \kappa = 0$. The latter condition on p is necessary because the project-by-κ operation causes elements of G to be lost in the remapping: precisely those whose index p is such that $(p - \sigma) \bmod \kappa \neq 0$.

The usefulness of T comes from the fact that given two transducer factors τ_1 and τ_2, it is possible to compute a factor $\tau_{21} = \tau_2 \circ \tau_1$ that combines their effect: that is,

$$T(T(X, \tau_1), \tau_2) = T(X, \tau_{21}) \quad \text{for all } X.$$

For this to be true in all cases, the *null factor* τ_0 must be introduced. This factor does not have a corresponding tuple, but is defined by the following identity:

$$T(X, \tau_0) = \text{null grid or null domain, for all } X.$$

Here is an example of a remapping sequence that produces a null grid:

$$\text{contract}(\text{expand}(\text{contract}(G, 2), 2) \ll 1, 2).$$

The rest of this section shows transducer factors corresponding to remapping operations, how transducer factors are composed, and how to put a transducer factor in *normal form*.

Transducers for domain operators

Let $\overline{1} = [1, \ldots, 1], \overline{0} = [0, \ldots, 0], \omega_{\text{id}} = [1, 2, \ldots, n]$. The following equivalences exists:

$$\begin{aligned}
\text{shift}(X, S) &= T(X, (S, \overline{1}, \overline{1}, \overline{0}, \omega_{\text{id}})) \\
\text{project}(X, S) &= T(X, (\overline{0}, S, \overline{1}, \overline{0}, \omega_{\text{id}})) \\
\text{inject}(X, S) &= T(X, (\overline{0}, \overline{1}, S, \overline{0}, \omega_{\text{id}})) \\
\text{transpose}(X, S) &= T(X, (\overline{0}, \overline{1}, \overline{1}, \overline{0}, S))
\end{aligned}$$

The transducer for the shift operator can also be defined by:

$$\text{shift}(X, S) = T(X, (\overline{0}, \overline{1}, \overline{1}, S, \omega_{\text{id}})).$$

Composition of transducer factors

We show how to compute a transducer factor τ_{21} such that

$$T(X, \tau_{21}) = T(T(X, \tau_1), \tau_2).$$

The composition of τ_1 and τ_2 is encoded by the following equations:

$$T(T(p, \tau_1), \tau_2) = \left[\frac{\left(\frac{p + \sigma_1}{\kappa_1} \varepsilon_1 + \sigma_1' \right) \otimes \omega_1 + \sigma_2}{\kappa_2} \varepsilon_2 + \sigma_2' \right] \otimes \omega_2 \tag{3.4}$$

with the restrictions

$$(p + \sigma_1) \bmod \kappa_1 = 0 \tag{3.5}$$

$$\left[\left((p + \sigma_1) \frac{\varepsilon_1}{\kappa_1} + \sigma_1' \right) \otimes \omega_1 + \sigma_2 \right] \bmod \kappa_2 = 0 \tag{3.6}$$

Equation 3.5 defines which grid elements "survive" the contraction by κ_1, and 3.6 the contraction by κ_2. We define:

$$\begin{aligned} \hat{\sigma}_2 &= \sigma_2 \otimes \omega_1^{-1} \\ \hat{\kappa}_2 &= \kappa_2 \otimes \omega_1^{-1} \\ \hat{\varepsilon}_2 &= \varepsilon_2 \otimes \omega_1^{-1} \\ \hat{\sigma}_2' &= \sigma_2' \otimes \omega_1^{-1}. \end{aligned}$$

We can then rewrite 3.6 as:

$$\left[(p + \sigma_1) \frac{\varepsilon_1}{\kappa_1} + \sigma_1' + \hat{\sigma}_2 \right] \bmod \hat{\kappa}_2 = 0$$

or equivalently:

$$[\varepsilon_1 p + \varepsilon_1 \sigma_1 + (\sigma_1' + \hat{\sigma}_2)\kappa_1] \bmod \kappa_1 \hat{\kappa}_2 = 0. \tag{3.7}$$

To derive a single transducer factor whose effect combines those of τ_1 and τ_2, we must solve equations 3.5 and 3.7 simultaneously. If the system has no solutions, then $\tau_{21} = \tau_0$, the null factor. Otherwise, the solutions satisfy the equation

$$(p + \alpha) \bmod \beta = 0$$

where α and β can be computed from the coefficients of the system, as shown in appendix A. This is the same condition imposed by a shift by α and a contraction by β. Therefore we have:

$$\tau_{21} = (\alpha, \beta, \varepsilon_{21}, \sigma'_{21}, \omega_{21}).$$

We now show how to compute ε_{21}, σ'_{21}, and ω_{21}. We can rewrite equation 3.4 as:

$$T(p, \tau_{21}) = \left[(p + \sigma_1)\frac{\varepsilon_1 \hat{\varepsilon}_2}{\kappa_1 \hat{\kappa}_2} + (\sigma'_1 + \hat{\sigma}_2)\frac{\hat{\varepsilon}_2}{\hat{\kappa}_2} + \hat{\sigma}'_2\right] \otimes \omega_1 \otimes \omega_2. \qquad (3.8)$$

For convenience, we define:

$$\begin{aligned}
\bar{\sigma} &= \sigma_1 \\
\bar{\varepsilon} &= \varepsilon_1 \hat{\varepsilon}_2 \\
\bar{\kappa} &= \kappa_1 \hat{\kappa}_2 \\
\bar{\sigma}' &= (\sigma'_1 + \hat{\sigma}_2)\frac{\hat{\varepsilon}_2}{\hat{\kappa}_2} + \hat{\sigma}'_2 \\
\bar{\omega} &= \omega_1 \otimes \omega_2.
\end{aligned}$$

The value of $\bar{\sigma}'$ is not necessarily integer. In an implementation it is more convenient to compute $\bar{\kappa}\bar{\sigma}'$:

$$\bar{\kappa}\bar{\sigma}' = (\sigma'_1 + \hat{\sigma}_2)\hat{\varepsilon}_2 \kappa_1 + \hat{\sigma}'_2 \kappa_1 \hat{\kappa}_2.$$

We now equate the right hand side of 3.8 (using the new definitions) and the definition of $T(p, \tau_{21})$:

$$\left(\frac{p + \bar{\sigma}}{\bar{\kappa}}\bar{\varepsilon} + \bar{\sigma}'\right) \otimes \bar{\omega} = \left(\frac{p + \alpha}{\beta}\varepsilon_{21} + \sigma'_{21}\right) \otimes \omega_{21}.$$

This must hold for all p. We now have enough conditions to determine the missing values, which are given by:

$$\begin{aligned}
\varepsilon_{21} &= \frac{\beta\bar{\varepsilon}}{\bar{\kappa}} \\
\sigma'_{21} &= \frac{\beta\bar{\varepsilon}\bar{\sigma} + \beta\bar{\kappa}\bar{\sigma}' - \bar{\kappa}\varepsilon_{21}\alpha}{\bar{\kappa}\beta} \\
\omega_{21} &= \bar{\omega}.
\end{aligned}$$

Normal form

If two transducer factors τ_1 and τ_2 are such that $T(x, \tau_1) = T(x, \tau_2)$ for all x, we say that τ_1 and τ_2 are equivalent and we denote it by $\tau_1 \equiv \tau_2$. It is easy to see that any factor has an infinite number of equivalent factors, because of the degree of freedom induced by the two independent shift factors, σ and σ'. We define the *normal form* of a transducer factor τ the factor τ_n such that $\tau_n \equiv \tau$ and $0 \leq \sigma'_n < \varepsilon_n$. Such form always exists and is unique, and can be computed as follows:

$$\begin{aligned} \sigma_n &= \sigma + \kappa\lfloor\sigma'/\varepsilon\rfloor \\ \kappa_n &= \kappa \\ \varepsilon_n &= \varepsilon \\ \sigma'_n &= \sigma' \bmod \varepsilon \end{aligned}$$

The proof is omitted.

3.7.4 Remapping operations

To obtain some remapping of a grid $G = (D, P, C)$, we compute separately the domain and the section descriptors of the result $\hat{G}$. As mentioned, the chunks of $\hat{G}$ are shared with G. The new domain is computed using domain operations; here we show how to compute the new section descriptors.

Remapping by transducer

First we consider the case in which the remapping can be encoded by a transducer factor τ. This includes the operations of project, inject, shift, and transpose. To obtain a new section descriptor, we start by deriving the new section domain $\hat{\delta}_S$ from the old one, $\delta_S = \text{inject}(b_S, \varepsilon_S) \ll \sigma_S$. We could construct a tiled descriptor to represent δ_S, and then use domain operators to transduce it; but it is possible to compute the result with arithmetic operations only. First notice that

$$\delta_S = T(b_S, \tau_S) \quad \text{with} \quad \tau_S = (\overline{0}, \overline{1}, \varepsilon_S, \sigma_S, \omega_{\text{id}})$$

and therefore

$$\hat{\delta}_S = T(\delta_S, \tau) = T(b_S, \tilde{\tau}) \quad \text{where} \quad \tilde{\tau} = \tau \circ \tau_S.$$

We want the result to be in the same form: $\hat{\delta}_S = \text{inject}(\hat{b}_S, \hat{\varepsilon}_S) + \hat{\sigma}_S$. Recall that

$$\hat{\delta}_S = T(b_S, \tilde{\tau}) = \text{transpose}(\text{inject}(\text{project}(b_S \ll \tilde{\sigma}, \tilde{\kappa}), \tilde{\varepsilon}) \ll \tilde{\sigma}', \tilde{\omega}).$$

Shifting and projecting a box produces another box (possibly the null box). By shuffling around the transpose operation, we can then write:

$$\begin{aligned}\hat{b}_S &= \text{project}(\text{transpose}(b_S \ll \tilde{\sigma}, \tilde{\omega}), \tilde{\kappa} \otimes \tilde{\omega}) \\ \hat{\varepsilon}_S &= \tilde{\varepsilon} \otimes \tilde{\omega} \\ \hat{\sigma}_S &= \tilde{\sigma}' \otimes \tilde{\omega}.\end{aligned}$$

If $\tilde{\tau}$ is in normal form, then $0 \le \hat{\sigma}_S < \hat{\varepsilon}_S$, as required to properly represent a tiled section domain. To compute

$$b' = \text{project}(b, \kappa)$$

where $b = (l, u), b' = (l', u')$, we use the following relations:

$$l' = \lceil l/\kappa \rceil, \quad u' = \lfloor u/\kappa \rfloor .$$

If $l'_i > u'_i$ for any dimension i, the result is the null box. In this case, the transduced section is null and it is not included in $\hat{P}$. Those chunks of G (if any) that are not used by any section of $\hat{G}$, are not included in the chunk vector $\hat{C}$. The chunk index $\hat{c}_S$ is computed from c_S taking into account deleted chunks.

As the last step, we compute the zero index $\hat{z}_S$ and the stride vector $\hat{\phi}_S$. Combining equations 3.1 and 3.2 we have:

$$G[p] = C[c_S][z_S + \frac{p - \sigma_S}{\varepsilon_S} \cdot \phi_S] \tag{3.9}$$

and from equation 3.3:

$$G[p] = T(G, \tau)[T(p, \tau)] = \hat{G}[T(p, \tau)] = \hat{C}[\hat{c}_S][\hat{z}_S + \frac{T(p, \tau) - \hat{\sigma}_S}{\hat{\varepsilon}_S} \cdot \hat{\phi}_S]. \tag{3.10}$$

Expanding $T(p, \tau)$ and equating the right hand sides of 3.9 and 3.10, we obtain:

$$\hat{\phi}_S = \left(\frac{\kappa}{\varepsilon \varepsilon_S} \otimes \omega\right) \hat{\varepsilon}_S(\phi_S \otimes \omega) \tag{3.11}$$

$$\hat{z}_S = z_S - \frac{\left(\frac{\sigma_S + \sigma}{\kappa}\varepsilon + \sigma'\right) \otimes \omega - \hat{\sigma}_S}{\hat{\varepsilon}_S} \cdot \hat{\phi}_S. \tag{3.12}$$

(Equation 3.11 is obtained by taking the limit for $p \to \infty$; equation 3.12 by setting $p = \sigma_S$.) By noting that $\tilde{\omega} = \omega$, we can write equivalent but more convenient expressions. It is particularly useful to rewrite 3.12 since some of its subexpressions do not necessarily have integer values:

$$\begin{aligned}\hat{\phi}_S &= \left(\frac{\kappa \tilde{\varepsilon}}{\varepsilon \varepsilon_S} \phi_S\right) \otimes \omega \\ \hat{z}_S &= z_S - \frac{(\sigma_S + \sigma)\varepsilon + (\sigma' - \tilde{\sigma}')\kappa}{\tilde{\varepsilon}\kappa} \otimes \omega \cdot \hat{\phi}_S.\end{aligned}$$

Remapping by restrict/merge

The restrict and merge operations do not change the association of indices and elements, but only add or remove elements. In the case of $\hat{G} = \text{merge}(G_x, G_y)$, the partition descriptor of $\hat{G}$ is the union of the section descriptors of G_x and a simple modification of the section descriptors of G_y, obtained by replacing the chunk index c_S with $c_S + N_x$, where N_x is the number of chunks in G_x. The chunk vector of the result, $\hat{C}$, is the concatenation of C_x and C_y.

For $\hat{G} = \text{restrict}(G, D_R)$, the partition of $\hat{G}$ is obtained by intersecting each section domain with D_R. If the result is the null domain, that section is not included in the result. The zero index and stride vector do not change. If some chunks of G are not used by any section of $\hat{G}$, they are not included in $\hat{C}$. The chunk index in the section descriptors is updated accordingly.

3.7.5 Reduction

To generate grid reduction code in the vector processor implementation, we add the remapping operator `grid-stretch` to the machinery we have developed for elementwise operations. This operator is an extension of `grid-transpose`, and its use is restricted to the constant-factor remapping expressions in `map-grid`. Unlike the other remapping operators, `grid-stretch` does not return a new grid object, but a different view of its grid argument. The elements of the result are shared with those of the argument, and assigning into one of them affects both objects.

The definition of stretch is similar to that of transpose:

$$\text{stretch}(G, S)[p \otimes S] = G[p]. \tag{3.13}$$

The dimensionality of stretch(G, S), n_S, is the length of S, and it is independent of n, the dimensionality of G. The elements of $S = [s_0, \ldots, s_{n_S - 1}]$ are either integers between 0 and $n - 1$ included, or the *undefined index*, denoted by a diamond ($\diamond$). (In the implementation we use the value -1 to signify a diamond). The integer elements of S are all different, that is $i \neq j \Leftrightarrow (s_i \neq s_j \vee s_i \neq \diamond)$. The meaning of $\otimes$ changes slightly:

$$(x \otimes y)[i] = \begin{cases} x[y[i]] & \text{when } y[i] \neq \diamond \\ \diamond & \text{when } y[i] = \diamond \end{cases}$$

It is obvious then that transpose is a special case of stretch, with $n_S = n$ and no diamonds.

When a vector with diamonds is used as a grid index, as in 3.13, the diamond stands for "any integer coordinate." Equation 3.13 then stands for an infinite number of equations, each of which is obtained by replacing diamonds in $p \otimes S$ with integer numbers. By necessity then if any element of S is a diamond, stretch(G, S) has an infinite domain. Since the rest of the system does not deal with infinite domains, this is one reason why the use of `grid-stretch` is restricted.

To identify precisely the stretch factor S outside the context of an expression stretch(G, S), it is necessary to specify the input dimensionality n of G. We refine the definition of S to be the pair $([s_0, \ldots, s_{n_S-1}], n)$. For convenience we abbreviate such pair as $[s_0, \ldots, s_{n_S-1}]_n$.

Certain values of S are invertible. We say that S^{-1} is the inverse of S if for all p, $(p \otimes S) \otimes S^{-1} = p$. For instance, to verify that $[0, \diamond]_1^{-1} = [0]_2$, consider:

$$\begin{aligned} [p_0] \otimes [0, \diamond]_1 &= [p_0, \diamond] \\ [p_0, p_1] \otimes [0]_2 &= [p_0]. \end{aligned}$$

However, $[0]_2$ is not invertible, as it induces some loss of information. We require that the second argument of `grid-stretch` be invertible.

Because `grid-stretch` returns an assignable object that always shares its storage with its grid argument, it can be used in a left-hand-side position; for instance:

```
(map-grid (domain-of B)
          (slambda (x y) (setf x (+ x y)))
          :grids ((grid-stretch A [0 -1])
                  B))
```

The effect of this code is approximately the following:

for all p from domainOf(B)
$A[p \otimes [0, \diamond]_1^{-1}] = A[p \otimes [0, \diamond]_1^{-1}] + B[p]$

If we make the further assumption that operations for different indices p are serialized (in some order), then this code accumulates in A the elements of B along the 0-th dimension. A should have been initialized to the identity for the operation (0 in this case).

Depending on the shape of each computation section, it can be more convenient to vectorize along one of the reduction dimensions, or orthogonally to it. Thus two versions of the elementwise code are produced, and the most efficient one is chosen at run time on a section-by-section basis. Each version itself contains two sub-versions, one with masking for the bitmap/thick domain case, the other without masking. This quadruplication of code is not an issue, because in general the elementwise code is quite short.

A `grid-reduce` operator is also available in INFIDEL (it is implemented in terms of `map-grid` and `grid-stretch`). We include the functionality of `grid-stretch` in the interface because it exposes opportunities for optimization. Specifically, the reduction could be combined with other constant-factor remappings in the same map-grid operation, making it possible to obtain improved vector-register allocation. We do none of it, but someone might in the future.

Chapter 4

INFIDEL Reference Manual

4.1 Support types

This section describes miscellaneous abstract types, constructs, and features.

4.1.1 Vectors and points

These are simple but useful extensions to create and manipulate arrays.

`cons-vector` *type* `&rest` *elements* [*Macro*]

`cons-vector` returns a pointer to a vector of type *type* initialized with *elements*. The vector is stack-allocated and no larger than needed to contain *elements*.

`cons-point` `&rest` *coords* [*Macro*]

`cons-point` returns a pointer to a vector of integers with the values given in *coords*. It expands into a `cons-vector` with type `int`. The L reader has been modified to translate a bracket-enclosed list into a `cons-point`. Example: `[1 2 3]` is read as `(cons-point 1 2 3)`.

4.1.2 Virtual vectors

A virtual vector is an L object identified by a symbol. This object represents a vector, but its elements are not stored in adjacent memory locations: they are implemented as separate variables, typically to be used as loop indices. Some L macros accept both vectors (arrays) and virtual vectors as their arguments: one of them is the macro `aref`.

`aref` *v i* [*Operator*]

`aref` is the generic indexing operator in L. It maps directly into the C indexing operator, producing *v*[*i*]. In addition, *v* can be a virtual vector. In this case, if *i* is not compile-time constant, an error occurs. Otherwise, `aref` expands into the *i*-th component of *v*.

4.2 Domains

4.2.1 Generic domain operators

`domain` *n* [*Type*]

Locations containing domain values have static type `(domain` n`)`, where n is the number of dimensions.

`domcons` *lower-bound upper-bound ndim* [*Operator*]

`domcons` returns a value of type `(domain` *ndim*`)` that represents the set of points in a rectangle of dimensions *ndim* whose bounds are specified in *lower-bound* and *upper-bound*. *ndim* is an integer variable, *lower-bound* and *upper-bound* are pointers to integers.

`domcons-t` *points np ndim* [*Operator*]

np and *ndim* are integers. *points* is an array of size *np* `*` *ndim* representing *np* points, each with *ndim* coordinates. The j-th coordinate of point i is stored in *points* `[`i `*` *ndim* `+` j`]`. `domcons-t` returns the domain of dimensions *ndim* representing such set of points.

`to-domain` *grid* [*Operator*]

grid is an integer grid, whose values should be 0 or 1. `to-domain` returns the domain $p : grid[p] = 1$.

`domain-union` *x y* [*Function*]
`domain-intersection` *x y* [*Function*]
`domain-difference` *x y* [*Function*]
`domain-accrete` *x* [*Function*]
`domain-boundary` *x* [*Function*]
`domain-shift` *x p* [*Function*]
`domain-contract` *x p* [*Function*]
`domain-project` *x p* [*Function*]
`domain-inject` *x p* [*Function*]

These are the standard operations on domains as defined in the FIDIL reference manual. *x* and *y* are domains. *p* is an integer array.

`domain-reduce` *x p* [*Function*]

This is the domain counterpart of the FIDIL reduce operator on grids. Given $x =$ domainOf(G), it returns domainOf(reduce(G, f, p, v_0)).

`domain-init` [*Function*]

`domain-init` is an initialization procedure that must be called before any operations on domains. Its return type is `void`.

`null-domain` *n* [*Operator*]

`null-domain` returns a descriptor for an empty domain in n dimensions.

`null-domain-p` *x* [*Operator*]

`null-domain-p` returns true if *x* is the null domain, false otherwise.

`setf-lowerbound` *l x* [*Function*]
`setf-upperbound` *u x* [*Function*]
`setf-bounds` *l u x* [*Function*]

`setf-lowerbound` sets the integer array *l* to contain the lower bound of domain *x*. *l* must point to an integer array with at least as many elements as the dimensionality of *x*. `setf-upperbound` sets *u* to the upper bound of *x*. `setf-bounds` returns in *l* and *u* both lower and upper bounds and may be more efficient than obtaining them separately. These functions do not return a value.

`point-in-domain` *p d* [*Function*]

`point-in-domain` is true if *p* is in *d*, false otherwise. *p* is an array of integers, *d* a domain.

`domain-size` *d* [*Function*]

`domain-size` returns the number of points in its domain argument.

4.2.2 Low-level domain operators

This section describes a lower-level part of the domain interface. Mostly, these operators expose choices of representation for domain values. We do not recommend using these operators in portable programs. We include their description for two reasons. First, we do not have sufficient programming experience with INFIDEL to guarantee that the high-level, generic domain interface is always adequate for producing efficient code. The low-level interface gives opportunities for experimentation. Second, these operators can be useful building blocks for a multiprocessor port of INFIDEL.

A domain value is represented by an instance of one of several structure types. Currently these types are defined: *manhattan*, *thin*, *thick*, *bitmap*, *collage*. The domain library is meant to be extensible, and new domain types should be added as needed.

`manhattan` [*Structure*]
`thind` [*Structure*]
`thickd` [*Structure*]
`bitmapd` [*Structure*]
`collaged` [*Structure*]

These structures implement the domain types. They have associated predicates (`manhattan-p`, etc.) and symbolic names to be used in the `typecase` and `type-p` macros. For better isolation, the names of their fields are not part of the interface. Their constructor functions (`make-manhattan`, etc.), should not be used directly.

`manhattan-component` *domain* [*Operator*]

`manhattan-component` returns the manhattan component of *domain*. Argument and return type are both `t`. It is an error if *domain* does not have a manhattan component. Currently, only thin domains do not have a manhattan component. Manhattan domains are their own manhattan component. For thick and bitmap domains, the manhattan component is some superset of points with a manhattan structure, which depends on how they have been created. For collage domains, the manhattan component is the union of the bases of the tiled components.

`to-manhattan` *x* [*Function*]
`to-bitmap` *x* [*Function*]
`to-thin` *x* [*Function*]

These functions convert their domain argument respectively to manhattan, bitmap, or thin form.

4.3 Grids

Grids represent mappings between n-dimensional integer tuples and values of some type. Grids are optimized for implementing "large" scalar maps.

`grid` *n eltype* [*Type*]

This is the type of an *n*-dimensional grid value, with elements of type *eltype*.

`grid-alloc` *domain eltype* [*Operator*]

`grid-alloc` returns a grid with index set *domain* and element type *eltype*. The grid's data is not initialized. The effect of reading a grid's element before it has been written is undefined.[1]

[1]It could return a random value or cause a run-time error, depending on whether the chunk for that element has yet been allocated or not.

It is foreseen that in the near future this operator will take another optional argument, containing hints on the allocation and partitioning of the grid.

`grid-free` *grid* [*Operator*]

`grid-free` releases all memory associated with *grid*. Under certain circumstances, a grid that is no longer in use is freed automatically by the garbage collector. The use of `grid-free` is recommended for large grids. Garbage collection should be adequate for small grids.

`domain-of` *grid* [*Operator*]

`domain-of` returns the domain of *grid*.

`grid-index` *grid point elsize* [*Operator*]
`grid-index*` *grid point elsize* [*Operator*]

`grid-index` returns a pointer to the element of *grid* indexed by *point*. *elsize* is the size of an element of the grid, in words. The type returned by `grid-index` is `(pointer void)`. The caller should cast the return value into the desired pointer type. This pointer can then be used for reading or setting the element's value.

`grid-index*` is the same as `grid-index`, but it should be used when the pointer is used only for reading the element.

If the element is undefined, the returned pointer value is undefined, and the effect of reading or writing through this pointer is also undefined.

If *point* is outside the grid's domain, an error is signalled.

4.3.1 Remapping operations

`grid-copy` *grid* [*Function*]

Return a copy of *grid*.

`grid-shift` *grid factor* [*Function*]
`grid-project` *grid factor* [*Function*]
`grid-inject` *grid factor* [*Function*]

`grid-transpose` *grid factor* [*Function*]
`grid-restrict` *grid domain* [*Function*]
`grid-merge` *gx gy* [*Function*]

The FIDIL map operators shift, project, inject, transpose, on, and disjoint union. *factor* is an array of integers.

4.3.2 Elementwise operations

`grid-reduce` *g op ival* [*Operator*]

`grid-reduce` is the equivalent of the FIDIL reduce operator when the type of the result is a grid element. *g* is the grid to reduce, *op* the binary operator used in the reduction, and *ival* an initial value for the result: typically the null value for the operator.

`set-grid-reduce` *result g op ival n factor* [*Operator*]

`set-grid-reduce` computes in *result* the reduction of grid *g* by the binary operator *op*, with initial value *ival*, and along the *n* dimensions specified by *factor*. *result* must be a previously allocated grid with the correct domain (obtainable by `domain-reduce`).

`map-grid` *domain op* `&key` *grids others* [*Operator*]
`map-grid*` *domain index op* `&key` *grids others* [*Operator*]

`map-grid` and `map-grid*` execute an operation in a data-parallel fashion over the domain D equal to the intersection of *domain* and the domains of the grid arguments. *op* specifies the operation to be applied to the grids' elements at each point. *index* is a virtual vector, available within *op*, that at execution time is bound to each point of D. *grids* is a list of *grid specifiers*, and *others* a list of *non-local variable specifiers*.

The *grids* argument

A grid specifier has the form `(`*grid-expression* `&key` *free partition*`)`. *Grid-expression* is either a variable, or an expression containing only constant-factor remappings: that is, any combination of `grid-shift`, `grid-inject`, `grid-project`, `grid-transpose`, and `grid-stretch` with a compile-time known factor (the second argument).

The keyword arguments are called *grid directives*. They provide information that may help optimize the operation.

The *free* directive is used to indicate that the data of the grid is dead after the operation, and the memory allocator can reclaim the data area. This is done transparently. After the completion of `map-grid`, a grid with the value `t` for its free directive behaves just as if it had just been allocated.

The *partition* directive provides a way to specify the partition of a grid, when the partition is known at compile time (an alternative way is through the use of a partial value: see the implementation note below). If the partitions of all grids are provided, `map-grid` can perform several optimizations.

The *others* argument

The *others* argument is a list of variables used by the operator *op*, other than the variables in its lambda list. The necessity for *other* is due to a deficiency of L: the lack of closures. During expansion of `map-grid`, the code generated by applying *op* to the grid elements is placed in a separate L function called a *looper*. Variables in *op* that are lexically visible where the `map-grid` statement appears, may no longer be so after the code is moved to the looper. Such variables must be included in the *other* list or `map-grid` will generate incorrect code.

Examples

This section presents examples of the use of `map-grid`.

```
(map-grid (domain-of a)
          (slambda (x) (setf x 0))
          :grids ((a)))
```

The above code sets to 0 all elements of the integer grid `a`.

```
(map-grid (domain-intersection (domain-of b)
                               (domain-of c))
          (slambda (x y z) (setf x (+ y z)))
          :grids ((a)
                  (b)
                  (c)))
```

The above code adds the elements of b and c on the intersection of their domains, and stores the result in a.

```
(map-grid d
          incf
          :grids (((grid-stretch a [-1 0 1]))
                  (b)))
```

The above code is usable only internally for execution on a uniprocessor. a is a 2-dimensional grid, and b a 3-dimensional one. "Rows" of b on d along the 1st dimension (index 0) are added into a. More specifically, if i, j, k are the coordinates of the point P as it spans d, the operation performed is: a[j, k] += b[i, j, k].

Implementation note

map-grid makes use of the compile-time capabilities of L. In the general case, map-grid expands into code that calls library functions, and those call back the looper function. Under certain conditions, map-grid expands into a simplified inlined code. The conditions are:

1. *domain* is a compile-time constant;
2. all grids expand into partial values with a known domain;
3. the computation partition has a small number of sections.

This is likely to be useful for operations on small or medium-sized grids with a simple domain, when the domain is known at compile time.

4.4 Caveats

Not all the described types and operators are available in the current system, mostly because we don't have yet any applications that use them. At the time this report is written, the following applies:

- all types and operations related to thin and thick domains are not implemented; only manhattan, collage, and bitmap domains are available;
- the grid allocator does not accept hints and always partitions the domain automatically;

- vector operations are not optimized dynamically; however, the layout for a grid section is longest-major, that is, the stride along the longest dimension is 1. This is a good choice in most cases.
- the `map-grid` operation does not fold into inline code when all descriptors have known values. It used to do it in an older version. I believe there is no major obstacle to fixing it back.

Bibliography

[Sem93] Luigi Semenzato. The High-Level Intermediate Language L. Technical Report UCB/CSD 93-760, Computer Science Division (EECS), Univ. California, Berkeley, July 1993.

[Sku90] Cory F. Skutt. Alternative evaluation of array expressions. In Kluwer Academic Publishers, editor, *Proceedings of the First International Workshop on Arrays, Functional Programming, and Parallel Systems*, pages 269–294, 1990.

Appendix A

Facts of modular arithmetics

A diophantine equation of the form

$$(ax + b) \bmod c = 0 \tag{A.1}$$

where a, b, and c are integer, and we are interested in integer solutions, is always solvable if a and c are *mutually prime*, that is $\gcd(a, c) = 1$. The solutions can be obtained by computing[1] the *modular inverse* of a with respect to c, which we denote $(a)^{-1}_{(\bmod\ c)}$, defined by:

$$\left[(a)^{-1}_{(\bmod\ c)}\, c\right] \bmod c = 1.$$

The solutions have the form

$$x = -b\,(a)^{-1}_{(\bmod\ c)} + kc \quad \text{for all integer } k.$$

If a and c are not mutually prime, let $d = \gcd(a, c) > 1$. If $b \bmod d = 0$, we can rewrite equation A.1 as:

$$\left(\frac{a}{d}x + \frac{b}{d}\right) \bmod \frac{c}{d} = 0.$$

and solve it as described. If $b \bmod d \neq 0$, there are no solutions.

A system of equations of the form:

$$\begin{cases} (x + b) \bmod c = 0 \\ (x + b') \bmod c' = 0 \end{cases} \tag{A.2}$$

[1]In our application, a brute-force search of all integers between 1 and $a \bmod c$ is adequate.

is solved by finding the intersection of the set of solutions of each equation. We know that

$$\begin{cases} x = -b + kc \\ x = -b' + k'c' \end{cases} \tag{A.3}$$

We derive

$$k' = \frac{kc - b + b'}{c'}$$

and, because k' must be integer,

$$(kc - b + b') \bmod c' = 0$$

which is just equation A.1. Given $d = \gcd(c, c')$, if $(b - b') \bmod d \neq 0$, then the system has no solutions; otherwise,

$$k = \frac{b - b'}{d} \left(\frac{c}{d}\right)^{-1}_{(\text{mod } c)} + \lambda \frac{c'}{d} \quad \text{for all integer } \lambda$$

and substituting k in the first equation of A.3 we obtain

$$x = -b + c\frac{b - b'}{d} \left(\frac{c}{d}\right)^{-1}_{(\text{mod } c)} + \lambda \frac{cc'}{d}$$

which is the same as saying

$$(x + \alpha) \bmod \beta = 0$$

with

$$\begin{aligned} \alpha &= b - c\frac{b - b'}{d} \left(\frac{c}{d}\right)^{-1}_{(\text{mod } c)} \\ \beta &= \frac{cc'}{d}. \end{aligned}$$

Appendix B

FIDIL Domain and map operators

Tables B.1 and B.2 give the standard operators and functions on domains. Tables B.3–B.6 give the standard operators and functions on maps.

Expression	Meaning
nullDomain(n)	The empty domain of type **domain**$[n]$. The quantity n must be a compile-time integer constant.
$D_1 + D_2$	Union of D_1 and D_2.
$D_1 * D_2$	Intersection of D_1 and D_2.
$D_1 - D_2$	Set difference of D_1 and D_2.
p **in** D	where D is a domain of arity n and p is an array of type **valtype**(D): a logical expression that is **true** iff p is a member of D.
lwb(D) upb(D)	For a domain of arity n: An **integer** map with domain $[1..n]$ (for $n = 1$, an **integer**) whose k^{th} component is the minimum (lwb) or maximum (upb) value of of the k^{th} component of the elements of D.
arity(D)	yields n for a domain of arity n.
sizeOf(D)	The cardinality of D.
shift(D, S), D << S	Where S is of type **valtype**(D) and n is the arity of D: The domain $\{d + S \mid d \textbf{ in } D\}$.
shift(D)	Same as shift(D, -lwb(D)).
inject(D, S)	The domain $\{d{*}S \mid d \textbf{ in } D\}$.
project(D, S)	The domain $\{d \oslash S \mid d \textbf{ in } D\}$, where '$\oslash$' denotes elementwise integer division, rounding toward $-\infty$.
expand(D, S)	The domain $\{e \mid e \oslash S \textbf{ in } D\}$.
contract(E, S)	The domain D such that $E =$ expand(D, S), if it exists.

Table B.1: Operators and functions on domains, part 1.

Expression	**Meaning**
accrete(D)	The set of points that are within a distance 1 in all coordinates from some point of D.
boundary(D)	accrete$(D) - D$.
reduce(D, S)	the domain R such that if $D =$ domainOf(G) for some map G, then $R =$ domainOf(reduce(G, f, S, v_0)) for any f, v_0.

Table B.2: Operators and functions on domains, part 2.

Expression	Meaning
domainOf(X)	The domain of map X. This may also appear in a left-hand side context if X is a partial map variable. The result of an assignment to the domain of X is a map whose initial image consists of undefined values.
toDomain(X)	where X is a **logical** map: $\{p \in \text{domainOf}(X) \| X[p]\}$.
image(X)	where X is a map whose codomain is an integer map of arity n: the domain of dimension n whose elements are all elements in the image of X—that is, the set $\{d \| X[p] = d, \text{ for some } p\}$.
upb(X) lwb(X) arity(X)	upb(domainOf(X)) lwb(domainOf(X)) arity(domainOf(X))
X # Y	The composition of X and Y. X and Y are maps; Y's codomain must be **valtype**(`domainOf(`X`)`); and `image(`Y`)` must be a subset of `domainOf(`X`)`. X # Y is a map object (which is assignable if X is assignable) such that $(X\#Y)[p] \equiv X[Y[p]]$. Hence, its domain is `domainOf(`Y`)`.
shift(X, S), X << S shift(X)	where S is a $[1..n]$ **integer** (an integer for $n = 1$), with default value -lwb(X), and n is the arity of X: the map `X # [p` **from** `domainOf(`X`): p-`S`].`
inject(X, S)	`X # [p` **from** `inject(domainOf(`X`), `S`): p/`S`].`
project(X, S)	`X # [p` **from** `project(domainOf(`X`), `S`): `S`*p].`
contract(X, S)	`[p` **in** `expand([0..0, ..., 0..0], `S`) :` `[project(`X` << -p, `S`)].`
expand(X, S)	Produces a map defined by the relation `expand(contract(`X, S`),`S`) = `X.

Table B.3: Operators and functions on maps, part 1.

Expression	Meaning
X **on** D	The map X restricted to domain D.
X (+) Y	where domainOf(X) $\cap$ domainOf(Y) $= \{\}$: the union of the graphs of X and Y, whose codomains must be identical and whose domains must be of identical arity.
concat(E_1,...,E_n)	Concatenation of $E_1, \ldots, E_2$. The E_i must be 1-dimensional maps with contiguous domains and some (common) codomain T, or values of type T, which are treated as one-element maps with lower bound 0. At least one of the E_i must be a map on T. The result has the same lower bound as E_1 and an upper bound equal to the sum of the lengths of the E_i.
F`@`	Assuming that F takes arguments of type T_i and returns a result of type T, F`@` is a function extending F to arguments of type $[D_i]\ T_i$, where the D_i are domains of the same arity, and returns a result of type $[D]\ T$, where D is the intersection of the D_i. The result of applying this function is the result of applying F pointwise to the elements corresponding to the intersection of the argument domains.
F `<@>`	For F as above returning type T_1: The extension of F to arguments of types $[D_i]T$ as above, returning a value of type $[D_1]T_1$ defined by $F{<}\ @\ {>}(x_1, \ldots, x_n) = F@(x_1, \ldots, x_n)\ (+)\ (x_1\ \mathbf{on}\ (D_1 - D))$.

Table B.4: Operators and functions on maps, part 2.

Expression	Meaning
compress(X)	where X is a map on a domain of arity 1: The one-dimensional map, X' with a contiguous domain having a lower bound of 1 such that $X'[i]$ is the value of $X[p_i]$, for p_i the i^{th} smallest element in the domain of X.
compress(X, W)	where W is a one-dimensional map whose codomain is **logical**: compress(X **on** toDomain(W)).
decompress(X, W)	The map X' such that compress(X', W) = compress(X).
reduce(X, f, S, v_0)	where X is a rectangular map of arity n and codomain C; $S = [i_1, \ldots, i_r], 1 \leq i_1 < \ldots < i_r \leq n$; and f is a function taking two arguments, one of some type R, the second of type C, yielding a result of type R. The result, B, is of type $T = [*(n-r)]R$, or $T = R$ if $n = r$, and is defined as follows. $B[j_1, \ldots, j_{i_1-1}, j_{i_1+1}, \ldots]$ = $f(f(\cdots f(v_0, v_1), \cdots), v_m)$. where the v_i are the elements $X[j_1, \ldots, j_{i_1-1}, k, j_{i_1+1}, \ldots]$ for all k for which the expression is defined, taken in some undefined order.
reduce(X, f, v_0)	where X is any map with codomain C; v_0 is of some type R; and f is as above. The result is of type R and has the value v_0 if the domain of X is empty, and otherwise $f(f(\cdots f(v_0, v_1), \cdots), v_m)$ where the $v_i, i > 0$ are the elements of X is some undefined order.

Table B.5: Operators and functions on maps, part 3.

Expression	Meaning
sort(X, P)	where X is a contiguous, one-dimensional map with codomain T and P is logical-valued binary function with arguments of type T: the map X' with the same domain as X that results from permuting the image of X so that $i < j$ implies $P(X'[i], X'[j])$. The permutation is strict: the order of image elements x and y such that $P(x, y)$ and $P(y, x)$ is unchanged by the sort.
trace(A, S)	reduce(A,**proc +**, S, 0)
outerproduct(A, B)	where A and B are maps with rectangular domains of dimensions n_a and n_b and the same codomains: The map C defined as follows. $C[i_1, \ldots, i_{n_a}, j_1, \ldots, j_{n_b}] = A[i_1, \ldots, i_{n_a}]*B[j_1, \ldots, j_{n_b}]$
transpose(X[, π])	where $\pi = [\pi_1, \ldots, \pi_n]$ is a permutation of the integers between 1 and n, and n is the arity of the map X: The object, X', resulting from transposing the indices of X according to π. Specifically, $X'[i_{\pi_1}, \ldots, i_{\pi_n}] = X[i_1, \ldots, i_n]$. The default for π is [2,1].
flip(X, π)	where X is of type $[D_1] \ldots [D_n]\,T$: The map, X' defined by the following. $X'[p_{\pi_1}] \ldots [p_{\pi_n}] = X[p_1] \ldots [p_n]$. The default for π is [2,1].
flip(X)	where X is a record of maps with identical domains: produces the map taking p in the common domain to the record with field values $F_i[p]$, where the F_i are the fields of X. X can also be a map of records, in which case flip performs the inverse operation.
remap(X)	The object resulting from "reassociating" the indices of X, which must be of type `[*`m`][*`n`]`T to form an isomorphic object, Y of type `[*`$m+n$`]`T. If p is a valid index of X and q is a valid index of X`[`p`]`, then Y`[concat(`p, q`)] = `X`[`p`][`q`]`.
remap(Y, m)	If X, Y, and m are as above, then `remap(`Y, m`)=`X.

Table B.6: Operators and functions on maps, part 4.

Index