

Introduction to ScalPL and L23

Revision 1.0/Prelim

Copyright © 2005 David DiNucci

All Rights Reserved

Abstract/Executive Summary

Current formal computer and modeling languages lack one or more important properties required for many complex planning and software engineering tasks, properties such as:

- The ability to describe a system as it continuously evolves through the development cycle, from high-level design to low-level design to executable code
- Object orientation—i.e. ability to build and use abstract data types w/inheritance, and express the relationships between those objects/classes
- The ability to express and exploit data parallelism, functional parallelism, and non-deterministic and partial orderings
- The ability to describe tasks such that they will be portable and efficient among different sequential, parallel and distributed platforms, including those with high communication latencies
- Sufficient formality to permit static analysis for various correctness properties (e.g. liveness, safety, lack of unwanted non-determinism)
- Debuggability (i.e. ability to reveal actual behavior in real-world situations)
- The ability to leverage existing tools, languages, and techniques where they apply
- The ability to mitigate hardware unpredictability by facilitating efficient recovery of lost state
- Allowance for production systems to grow and change to accommodate unforeseen needs

A good concurrent software engineering language would require all of these properties, and the lack of such languages is negatively impacting the economics of computing across a wide spectrum, including:

- the acceptance of multicore and multithreaded chips
- the practicality of increasing chip speed through asynchronous logic
- the viability of parallel/cluster and grid computing
- the exploitability of web services
- the manageability of outsourced software development

The Scalable Planning Language (ScalPL™) described here is a high-level visual language which can be used much like the popular Unified Modeling Language (UML) to develop, specify, and model plans and programs, but fills some needs above which UML doesn't. A ScalPL visual plan can evolve all the way from high level design to final executable code, in which case the design remains an integrated part of the code. A ScalPL plan/program is composed of modules with interfaces engineered to selectively and dynamically bind those modules tightly together into a monolithic program or to let them easily decompose into tasks/threads for environments where that is advantageous. When so decomposed, intermodule communication is efficient in all sorts of multi-threaded, multi-core, parallel, distributed, and grid environments, without rewriting the plan/program.

A high-level ScalPL plan, called a strategy, is created from three primary kinds of graphical constructs: Situations, represented by circles, where activities occur; Places, represented by rectangles, where passive content (data or information) resides while waiting to be acted upon; and Roles, represented by lines/arrows between circles and rectangles, specifying the roles which various places (and their contents) will play in various situations. This simple notation, reminiscent of Petri Nets, is expressive enough to diagram how computer modules, human organizations, or any number of other active and passive entities should (or do) interrelate and interact. The underlying communication model between activities (via places) does not favor any particular implementation, being efficient and portable between platforms based on traditional "whiteboard" memory, shared memory, and message-based communication.

Instead of specifying the precise order in which various activities *must* take place, a ScalPL planner uses colors to specify important local states scattered throughout the plan, and uses color matching to specify the combinations of states which must hold for each activity to *be allowed to* take place. The states (and therefore colors) transition as the result of each activity,

allowing overall action to be visualized as a set of color-based finite state machines. Unlike other methodologies, this approach makes both data flow and control flow statically apparent in a single diagram, and allows the plan to be statically analyzed for reachability, liveness, and safety properties in a piecewise, localized fashion. Instead of being totally sequenced/ordered, activities take place in partial orders, which can be visualized as directed acyclic graphs (DAGs).

The planner can specify implementations for activities within a strategy (high-level plan), each either in the form of another "sub" strategy, facilitating hierarchical decomposition of strategies to any depth, or in the form of a task (low-level ScalPL plan). Tasks can be specified using any language (e.g. computer language or specification language) chosen by the planner to fulfill project requirements. ScalPL does not facilitate nor impede further analysis inside of tasks, but is designed to allow tasks to efficiently access place contents to facilitate I/O, persistence, and communication with other tasks. A task-based activity can be technically regarded as either a special form of function or as an atomic transaction, and since all plans can ultimately be specified down to the task level, all ScalPL plans can be considered as collections of atomic transactions or as an unconventional composition of functions.

Using additional notations and extensions, ScalPL provides for strategies to also serve as atomic transactions (like tasks), and/or to serve as classes from which objects can be instantiated. (In the class/object interpretation, situations are object bindings, activities are object activations, roles are method interfaces, and places are messages and/or variables.) Other notations permit the specification of arrays of places and of delegation of roles, and other mechanisms allow the replication of activities and situations, the dynamic assignment of activities to situations, and the dynamic selection of array elements to play roles within specific situations. Together, these constructs and notations facilitate the construction of plans/programs with dynamic dependences, of classes which inherit interfaces and functionality (activities) from others, and of data parallel plans in which the amount of concurrency scales with the amount of data being acted upon.

To illustrate the power, flexibility, and expressiveness of these constructs, two examples are covered in detail: A parallel sort, closely related to quicksort, and a matrix multiplication. This document does not go into any depth regarding software engineering aspects or theory of the language, but does describe how to use a simple tool, called L23™ to enter and edit ScalPL plans.

Table of Contents

Abstract/Executive Summary.....	1
Table of Contents.....	3
Introduction.....	4
The Basics.....	7
Situations, Places, and Roles.....	7
Cause and Effect.....	8
Interacting with the Outside World.....	9
Summary.....	10
Refining Plans.....	12
Nesting: One Man’s Ceiling is Another Man’s Floor.....	12
Strategies: Branching Big Plans Into Smaller Plans.....	12
Tasks: The Leaves of the Activity Tree.....	13
Partial Bindings, Binding Constraints, and Overloading.....	14
Buffering and Shared Readers as Optimistic Execution.....	15
Plans as Place Contents.....	16
Controlling and Sensing Activation.....	17
Atomicity and Serializability.....	18
Standard tasks: Copy and Null.....	18
Object–Oriented Planning.....	20
Creating Objects with Sharable Persistent Places.....	20
Instantiation–time Activities, and Garbage Collection.....	21
Multiple Instantiation Levels.....	21
Delegation.....	24
Inheritance (Is–a) Relationship.....	25
Has–a Relationships (Aggregation).....	27
Uses–a Relationships (Passing Objects and Classes).....	27
Other Relationships.....	28
Arrays.....	29
Indexing, Subsetting, Translating.....	29
Type A Binding Modifiers.....	29
Type B Binding Modifiers.....	30
Rules in common for Type A and Type B Binding Modifiers.....	32
Delimited (Finite) Arrays.....	32
Replicating Situations for Data Parallelism.....	33
Examples.....	36
Matrix Multiply.....	36
fmatmult.....	37
fdotprod.....	38
ac_red.....	39
Matrix Multiply Variations.....	39
ac_red revisited.....	39
fmatmult revisited.....	41
Matrix Multiply Optimization.....	42
Quicksort.....	43
Specification.....	43
High–Level Implementation Description.....	44
Parallelism Analysis.....	44
Finer Points.....	45
Conclusion.....	47
Acknowledgements.....	47
Alphabetical Index.....	48
Software Cabling Translations.....	49

Introduction

Natural languages are very expressive, but their complexity, generality, and varied origins imbue them with ambiguity, informality, and context sensitivity which make them unsuitable for many tasks. Computer languages have addressed some of these issues by severely restricting their scope and using a basic textual English– or math–like syntax to provide the formality and precision required to express complex sets of specific unambiguous instructions, or algorithms, for traditional computers. In turn, computer architectures have evolved to efficiently execute algorithms expressed in such languages.

Over time, it has become clear that these languages leave significant gaps in utility and expressiveness. For example, they are not particularly good at expressing "the big picture" of how signals or information move or flow among several different entities or tasks, especially when the paths between entities are plentiful and irregular. These languages are also not very good at expressing complex cause–and–effect relationships. They are generally sequence based, making it difficult to express cases where many actions are to occur at once or cases where the order of some actions is unimportant. Traditionally, in the latter case, faulty logic has been used–i.e. "if the order is unimportant, we'll express one specific ordering and that will be as good as any other", but this forces the unimportant ordering to masquerade as an important ordering, making it difficult to recognize and reprioritize these cases at a later time. And because these languages were designed for describing all aspects of an action in an integrated way, the only way to omit detail, to be filled in later, is often to introduce ambiguity. These shortcomings severely limit the utility of such languages for systems design and modeling, and/or for describing concurrent behavior.

Some more recent or experimental languages and extensions have attempted to step in to address some of these issues. Visual diagramming languages used for software engineering methodologies have evolved into UML (Unified Modeling Language) which is popular for describing how entities (objects) within large systems relate to one another and how they interact. But UML is more of a collection of diagramming and modeling techniques than a single language, is not expressive enough to treat as a computer language, and is generally inadequate for describing portable concurrent systems. Various approaches have been used to augment traditional textual languages with inter–task communication mechanisms to enable the construction of concurrent systems, but making communication so low–level and integrating communication with low–level computational code has detrimental effects such as mixing temporary state with long–lasting state, tailoring the resulting programs to specific platforms, and hiding communication patterns by burying each transfer into not only one low–level process but dividing it into two (i.e. the originator and receiver). Distributed object languages and functional and dataflow languages have addressed some of the software engineering and portability issues, but often at the cost of existing programming languages and tools, not to mention inefficiencies related to extra copying by either the user or the language support system.

This lack of languages is now having significant economic impacts along the entire spectrum of computing, including:

- **Multicore:** While Moore's law continues to fuel the growth in the number of transistors that can be packed onto a computer chip, heat/power factors, atomic feature sizes, and speed–of–light limits have already put an end to the corresponding growth in performance, unless the programs for those chips are decomposed into concurrent tasks, or threads, each to use a different part (or core) of the chip. Thus, multi–core chips are now being built and marketed by virtually all major makers, but the programs to exploit them are slow to materialize due to the language issue. Game machines are among the first mainstream platforms to utilize this technology, in hopes that game vendors, already accustomed to exploiting arcane architectures for maximum performance, will be willing to lead the way in software.
- **Multithread:** Even a single processor core will often stall while waiting for data, e.g. from memory. To circumvent this, some processors have been built to bounce very quickly among several tasks (threads) to find one which is done waiting and ready to go at any one time. To keep such processors busy even when there's only one program to run, that program must be decomposed into several tasks.
- **Parallel and clusters:** When one computer alone is not capable of delivering the necessary speed for an important application, multiple computers can be bound together, either loosely into clusters (cable–connected cabinets) or more tightly into parallel processors (in single cabinets). Though these approaches have been used for decades, the

difficulty of splitting applications up to run effectively on these platforms, and of porting such applications to different platforms, have made it cost effective only for a handful of well-behaved or "cost is no object" applications.

- **Grids:** Growing capabilities in WANs and the internet have made it practical to consider even widely-distributed computers, even computers belonging to different organizations, as single entities to promote efficient use or sharing of resources and/or to harness their combined compute power in high-demand applications (so-called "on-demand computing"). In addition to the concurrency aspects, applications must be able to cope with high network latencies, sudden withdrawal and/or availability of resources, and the differences (heterogeneity) in the computers provided by the different sites. The power of these so-called grids is currently being un-harnessed due to the lack of software methods.
- **Web Services:** The internet has also introduced the concept of web services—essentially, web pages that are meant for other programs to use to retrieve and/or provide information, instead of for humans. The complexity of integrating and coordinating different web services from different sites, overlapping the delay/latencies of using several such, and of nesting services, have limited their exploitation.
- **Outsourced software:** Software development has gone global, and offshoring requires clear specification and testability of modules among disparate organizations, formalizing the wall between module interfaces and implementations, and the ability to remotely refine designs.
- **Asynchronous Logic:** Processors are currently designed to work off of a clock pulse which must propagate throughout the processor, and the speed of these processors is limited by the amount of time it takes for that propagation. By making the processor more event- and/or data-driven, different portions of the processor can potentially be made to operate at different speeds, obviating clock skew and speeding up the processors.

ScalPL™ (Scalable Planning Language) is a methodology for developing plans, for people, computers, and/or organizations. It is different from most other planning methodologies in that the planner isn't required to specify the order in which every subtask should be performed, only the orderings contributing directly to a correct outcome. Instead of ordering subtasks, the planner graphically shows the conditions under which they should be performed, and how each subtask can affect or be affected by others. This leaves significant flexibility, even in a concrete formal plan, for waiting until the last moment to decide the order in which some subtasks should be performed to maximize utility of available resources and to deal with unforeseen delays or slack. Also, tasks which do not need to be performed in any particular order can very naturally be performed in no order at all—i.e. all at the same time—when sufficient resources are available. When taken together with the very portable notion of communication used in ScalPL, these features are valuable for formulating very portable plans/programs to be performed by collections of computer processors, in so-called parallel, distributed, grid, multi-threaded, and multi-core environments. Since ScalPL carefully defines notions related to interfaces and hierarchy, it is also valuable in the ever-more common case where large and/or distributed projects are *planned* (not just performed) by distributed concurrent entities like disparate teams or members, including cases where the planning and performance are themselves concurrent with one another.

L23™ is a tool which helps to create, modify, and execute ScalPL plans. The planner does not strictly need L23 in order to work with ScalPL: Other graphical tools, or even pencil and paper, will suffice for many entry and editing needs. However, since L23 knows the precise rules of ScalPL outlined here, it can provide shortcuts and guidance not available in other tools, and since L23 will evolve over time to automatically analyze and convert the ScalPL program in useful ways, it will generally be a good choice for those who expect to do significant work with ScalPL.

The pages that follow will briefly describe ScalPL, and how to use L23 to enter and edit plans/programs. Although these two aspects are intermingled in one document, the L23 instructions are underscored, to be easily found by those seeking usage information or ignored by those who do not have immediate access to L23. Each important term in ScalPL will be presented in **boldface** in its first mention/definition and indexed at the end. ScalPL has descended from techniques with names like LGDF2, F-Nets, and Software Cabling, so for those readers familiar with some of the hardware-oriented terminology used in Software Cabling descriptions (e.g. sockets, memories, cables), a translation table is available at the end of this document. Aside from terminology and a few minor extensions, ScalPL is Software Cabling.

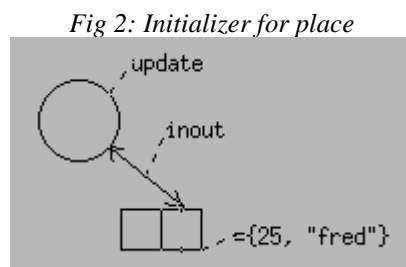
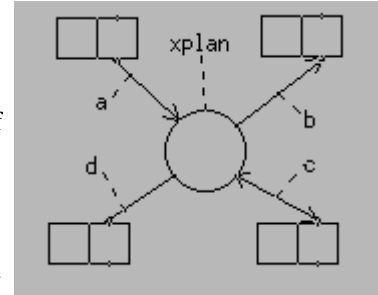
This document is not a formal definition of ScalPL, and in the interest of broadening the audience, examples and brevity and informal language are often preferred over concise precise language and complete specifications. This is also not a software engineering guide. It will not delve into aspects of debugging, analysis, and/or proofs, even though the methodologies described here have profound advantageous effects on these aspects. As a result, to evaluate this methodology for adoption, the reader would best be served by consulting other documentation as well as this to fully appreciate the interplay between ScalPL and software engineering. ScalPL is a work in progress. Feedback and suggestions will be carefully considered.

The Basics

Situations, Places, and Roles

The three basic graphical constructs of ScalPL are (1) **situations**, represented by circles, in which **activities** take place, (2) **places**, represented by (partitioned) rectangles, where passive **contents** (generally in the form of data or information) are held, and (3) **roles**, each represented by a line between a place and a situation illustrating how the contents of the place may affect or be affected by the activity in the situation. A role line R between place P and situation S will be described alternately as "place P plays role R in S" or "role R of S is bound to P". Situations are typically labeled for the activity which will take place there. Role lines are drawn with arrowheads to denote the kind of role the place will play for the associated activity, or put another way, the **permission** which the activity has to that place: An arrowhead on the situation end if the activity will/can look at (read) the

Fig 1: Activity (xplan) in situation, with places playing In (a), Out (b), Inout (c), and unaccessed (d) roles



contents of the place (called an **In** role), on the place end if the activity will/can provide new contents (called an **Out** role) but not read the old contents, and on both ends if the activity will/can modify the contents (which is observationally equivalent to first reading the contents and then replacing them), called an **Inout** role.

The contents of each place will usually be initialized to some sensible default value when the plan starts, but the planner can override this and initialize it to any value x by labeling the place (or following its label with) " $=x$ ". Instead of a value, a name in angle brackets (" $=\langle name \rangle$ ") initializes with a constant which has been assigned that name elsewhere. If a place is being initialized just so it can be read (but it is never modified), the place rectangle is not required: A role line (with no modify arrowhead) can be drawn directly from the textual initializer constant to the situation.

In L23, a situation or place is created by selecting the circle or rectangle from the palette and then clicking the mouse in the appropriate location on the canvas. With either the circle or rectangle selected in the palette, either can also be dragged around on the canvas with the mouse. To create a role in L23, select the role (straight horizontal line) from the palette, then move to the canvas, start the mouse within a situation, and drag the role out until it ends within a place—anywhere other than the leftmost region. When you release the mouse button, the role will automatically reposition to connect to the top or bottom of the rectangle, whichever is best. If you release the end of a role somewhere where it doesn't belong, it will just remain there until repositioned (as described next).

To reposition a role line once it has been created, select the finger from the palette, and select the end of the role (i.e. the one not impinging the situation) in the canvas and drag it to where it's needed. If you try to drag a role starting anywhere other than from the end, L23 will create a bend point in the role, and the finger will drag that instead. To remove such a bend, drag the bend with the finger until the role is straight and release. (If the bend appears as a large black dot, you may have a newer version of L23, in which case the bend is removed by dragging it to any extreme end of the role.) In addition to dragging bends and ends of roles, the finger can also be used to drag many other constructs, like situations and places.

To add arrowheads to a role, select the arrowhead from the palette and click the mouse in the palette on the role anywhere near the end where you want the arrowhead. If there is already an arrowhead on that end, it will be removed.

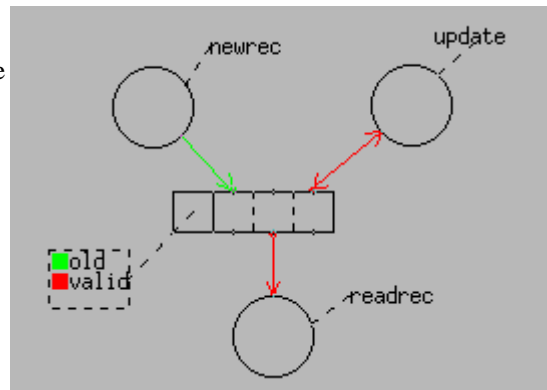
To add a label to a place, situation, or role, select the "L" in the palette, then place the mouse in (or on) the object being labeled and drag the label from it. A thin line will automatically connect the label to a point

on the object. The label text will always start as " nolabel", but this can be edited in the text window within the palette, either immediately after creating the label, or after reselecting a label with the "L" selected in the palette.

Cause and Effect

These basic circle, line, rectangle drawings are sufficient to show which activities (in the situations) will interact with which places, and the flow of information in each interaction, but it doesn't show the circumstances or constraints under which these interactions will take place. While some other methodologies would show this information in a separate diagram, ScalPL shows them in the same diagram using colors. That is, each place will not only hold different contents during its lifetime, as mentioned earlier, but will also transition between different **stages** during its lifetime, represented by different colors. Except for the convention that the initial stage is always green, the planner chooses the number of stages for each place and the colors and meanings of those stages for each place. Any time a place has finished playing a role in an activity, the activity will specify a new stage (color) for it, in a way described shortly, thereby determining which activity/situation bind to the place next. When using a tool like L23 to debug a running ScalPL program, the stage of each place at any given time will be reflected on the screen as the color of the rectangle.

Fig 3: Place with stage legend (stage names)



Each role is colored to denote the stage which a place must be in for the associated activity to access it. We will say that a role is **ready** (which is shorthand for "the place is ready to play that role") when the stage (color) of the place matches the stage (color) of the role. In L23, to color a role, select the color in the palette and then select the role to be given that color. In newer versions of L23, one doesn't color the role, but the so-called "solder dot" which connects the role to the place.

The different stages for a place represent different conditions under which different activities should be performed to that place, so the planner must ensure (1) that each role is colored to reflect the condition under which the associated activity should access the associated place, and (2) that the activity assigns the proper new color/stage to that place when the activity is done with the place (as described shortly). The leftmost zone in each place can be used for a **stage legend**, to assign a meaningful **stage name** (usually an adjective) to each color for that place.

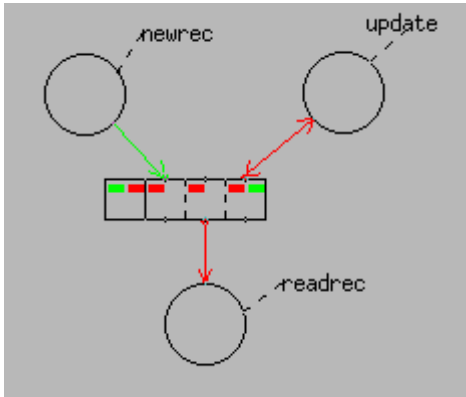
In L23, the stage legend is created by selecting the "dot" in the palette and clicking the mouse in an empty spot in the left zone of the place to create a dot, or legend entry. Clicking on an existing dot deletes it. A legend entry is colored using the same color palette as described above for coloring roles. If there's not enough room for all the dots you want to add, you can vertically stretch the place by selecting the "touch-up brush" in the palette (resembling a small artist's brush) and then using the mouse to drag the bottom of the place rectangle.

To see and modify the descriptions in the legend, one must first magnify the legend by selecting the magnifying glass (circle containing dots) in the palette and clicking on the legend. This creates a magnified legend box holding the same colored dots, slightly enlarged, each labeled with its description, all which start as " nolabel". This magnified legend can be dragged to wherever desired using the finger from the palette, and the descriptions can be modified by using the same label editing approach mentioned earlier (i.e. the "L" from the palette). The magnified descriptions can be re-hidden or recovered by clicking in the left place region again with the magnifying glass.

Given this much, one can tell from inspection not only the sorts of access that each activity will make to the contents of each place, but also the circumstances (stages) under which each activity can make those accesses. What is still not apparent in the diagram are the stages to which an activity might change each of its places when it is done with them. This is remedied by adding a **transition table** adjacent to each role-

place connection, within the place rectangle. In its simplest form, a transition table is just a set of one or more colored dots denoting the stages that might result after the place plays that role in the activity.

Fig 4: Transition dots (in transition tables)

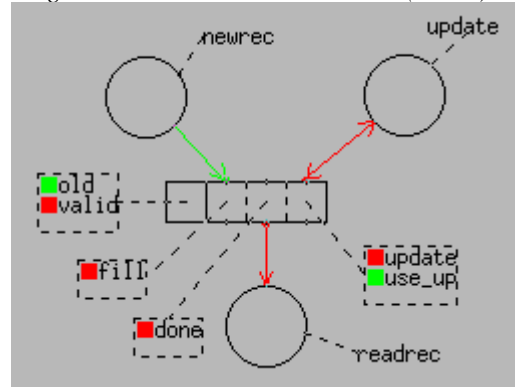


In L23, these dots are added just as to the stage table, with the "dot" from the palette, followed by coloring them with colors from the palette. By including transition tables, the ScalPL diagram shows both the stage which the places must have before being accessed by each activity (in the role color), and the stage that may result from that access (in the transition table colors), allowing cause-and-effect reasoning. When taken together with the arrowheads on the roles, a big picture is painted of the conditions and order in which different activities might occur and how contents can flow between them.

A transition table can specify more than just a set of stages that the activity might leave the place in. It can also specify

the conditions under which the place will be left in each of those stages. To include this information, the planner can choose a **transition name** (usually a verb) for each dot in the transition table, denoting that when that activity performs the transition of that name to that role, the stage of the place will change to the given color. As always, the meaning of the color is described in the stage legend for the place. In L23, the transition names in the transition table are edited just like the mnemonic names in the stage legend—i.e. using the magnifying glass and the label editor.

Fig 5: Roles with transition names (tables)



Read-only constants (where the place rectangle is omitted and the role is drawn directly to the textual constant) can only be used when the stage is unimportant. The role must always be (or is assumed to be) green, and a transition table containing only a green "done" transition is implied, so the role is effectively always ready.

In addition to being a mnemonic device, the transition names serve an important engineering role by allowing an activity in a situation to remain oblivious to the number of stages for each place, or their names or colors. The activity knows only that it is sometimes allowed to access the place (i.e. when the role is ready), and it knows that when it's done accessing a place, it can perform a transition by name. The transition table then allows the situation to convert that name into an appropriate stage/color for the place.

An activity can even remain oblivious to the names of the places it will access. That is, an activity doesn't really know about place names, it only knows about role names. Role lines should therefore be labeled according to the needs or expectations of the activity, and only when the role line is not labeled does the role name default to the name of the place to which it is bound.

In summary, an activity accesses place contents of places via roles, and when each access is done, the activity transitions the role. It is the activity's situation that determines (a) the place which is to play each role, denoted by where the role lines are connected, (b) the condition (stage) under which each place will be considered to be ready to play that role for the activity, denoted by the color of each role line, and (c) the new stage that will result from each transition to that role, denoted by the transition table on the role.

Interacting with the Outside World

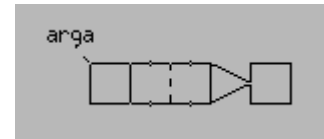
A plan generally operates within some larger context, which for now we'll call the "outside world". We call a place that can move between a plan and the outside world a **visiting place**, or just a **visitor** for short. A visitor is distinguished from other places by attaching a small box, called a **goodbye box**, to its right end, usually with a small triangle (i.e. with its base attached to the right end of the place rectangle and its

apex attached to the goodbye box). In L23, a place can be changed to a visitor and back by selecting the "touch-up brush" from the palette and then clicking on the right edge of the place rectangle.

There are three basic kinds of state that a visitor may be in:

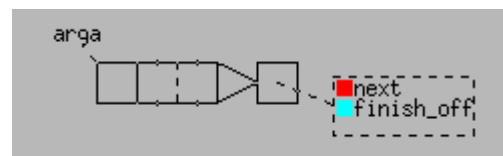
- (1) **Present:** Under the exclusive control of the plan, with no interference from the external world.
- (2) **Absent:** Under the exclusive control of the outside world, with no accessibility by the plan.
- (3) **Visible:** Up for grabs between the plan and the outside world.

Fig 6: Visiting place (i.e. visitor), with goodbye box



These are reflected in the stage of the visiting place. That is, if a visitor is green, it is visible. If the external world takes control, the visitor will become absent, and the stage will become undefined/colorless so that nothing in the plan can access the visitor, and if/when the outside world is done, the visitor will become visible (green) again. On the other hand, if an activity in the plan accesses a visible visitor (via a green role), then that visitor automatically becomes present within the plan, and remains present as it transitions among stages in the usual way. If/when the plan wishes to provide access to the outside world again, it cannot just change the visitor's stage back to green/visible, for reasons that will become clear soon, so green transition dots are not allowed on visitors. Instead, the plan explicitly hands control of the visitor over to the outside world through the use of one or more colored **goodbye dots** in the visitor's goodbye box. These dots are added and edited in L23 using the same approach as the stage legend or transition dots. That is, if the color of the visitor ever matches the color of one of these goodbye dots, the visitor becomes absent until/unless it becomes visible again—

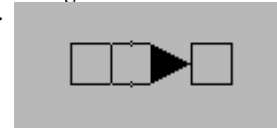
Fig 7: Goodbye notes



which may happen instantaneously if the outside world has no immediate use for the visitor. Each different goodbye dot color can be labeled with a different **goodbye note** (usually in the form of a verb) to inform the outside world what it may want to do with the place it is now getting, or why the activity is sending the visitor on its way. In L23, goodbye notes are added to the goodbye dots just as transition names and stage names are added to transition dots and stage dots. Note that since a visitor becomes absent as soon as its color matches a goodbye dot, it is nonsensical (and therefore illegal) to have a role on a visitor be the same color as a goodbye dot.

To just get some initial contents from the outside world into a plan, and/or to pass some final contents from a plan to the outside world, one can avoid a visitor's visible (contentious) state by flagging the visitor as an **initial visitor**, distinguished from other visitors by filling in the triangle adjoining the finishing box. In L23, this notation is added by clicking multiple times on the right end of the place with the touch-up brush selected. Initial visitors are different from normal visitors in that they are present (not just visible) even when they are green. In fact, they are guaranteed to be present (green) when the plan begins, very much like non-visiting places. This has several implications. First, green transition dots are allowed on initial visitors, as for non-visiting places. Second, the plan is guaranteed not to begin until all initial visitors are indeed available to visit the plan, as described shortly. However, like other visitors, the content of initial visitors may be set by the outside world, and they contain goodbye dots to allow control (and therefore contents) to be passed out to the world. Once a plan bids goodbye to an initial visitor, that visitor will never return—i.e. will never become present (or visible) again.

Fig 8: Initial visitor



Summary

So far, a plan consists of activities taking place within situations, and places which play roles within those activities. Places transition between different stages which define the situations in which they are prepared to play a role. When the place does play a role in an activity, that activity may read the place's contents, supply brand new contents, update its contents (externally indistinguishable from reading its contents and supplying new contents), or not even touch its contents, but when the activity is done with the place, it will specify a transition for it, in the form of a name, which in turn will be translated (by the transition table)

into a new stage for the place. Visiting places transfer contents to and/or from the outside world. The activity can invite them in (make them Present) whenever they are Visible, and when the activity is done, it can bid them goodbye by changing them to a color of one of their goodbye notes. Initial visitors don't need to be invited in—they are present from the moment the activity starts, but once they are bid goodbye, they are gone for good.

Refining Plans

Nesting: One Man's Ceiling is Another Man's Floor

As described earlier, all activities take place within situations. What may not be clear yet is that all activities start as plans. When a plan is assigned to a situation (e.g. by labeling the situation with the name of the plan, formerly described as the name of the activity), that plan may or may not **activate**—i.e. become an activity—or may activate multiple times, all depending upon the form of the plan and some conditions surrounding the situation. By default, activities are completely distinct from one another, even in cases where multiple activities are created from the same plan and activate within the same situation. This section will describe the circumstances under which a plan may activate in a situation to become an activity, and once it does, how it behaves within the situation.

There are two forms of plans. The first, called a **strategy**, is the sort of plan that has been described here so far, consisting of situations, places, and roles. Since each situation in such a plan may contain an activity which in turn is defined in terms of another plan, a hierarchy of plans/activities is created. This can help to break large complex plans down into smaller and simpler ones, but eventually, this hierarchy of plans must end. That's where the second form of plan, called a **task**, comes in. Tasks are plans which are simple enough that they are easily expressed using familiar languages, so there's no benefit to breaking them down into smaller plans. Tasks are described in the section following strategies.

Strategies: Branching Big Plans Into Smaller Plans

When one strategy, which we'll call the **child** strategy, activates within a situation in another strategy, which we'll call the **parent** strategy, the visitors in the child represent the roles of the same names on the situation in the parent. The contents of the visitors in the child correspond to the contents of the places in the parent playing those roles. A visitor in the child is visible (green) exactly when its corresponding role on the parent situation is ready. If (an activity in) the child accesses a visitor while it is green so that the visitor becomes present in the child, then the stage/color is removed from the place in the parent which is playing that role. When the child bids goodbye to the visitor, the goodbye note in the child becomes a transition in the parent, and the transition table corresponding to the role in the parent determines the new stage to assign to the place in the parent based on the goodbye note.

This explains why green transition dots are not allowed on visitors (except initial visitors). If a child was to be able to change the color of a visitor back to green, the place playing the role in the parent would logically change back to the "ready" stage (i.e. the same stage/color as the role), but if there was no transition dot in the parent representing that ready stage on the role in the parent, the parent might not even consider such a transition possible. By forcing the child to send a goodbye note when it is done, which is then interpreted as a transition in the parent, the parent remains in control of what to do, and has the option of using the transition table to change the place immediately back to the ready stage or not, as it needs.

For a strategy to fit into a given situation, the visitors in the strategy must conform, in various ways, to the roles of the same names on the situation. Primarily, each visitor is statically categorized (manually or automatically by tools) into one of four **usages** according to the sequences of roles it may play between the time it becomes visible (green) and the time it is bid goodbye (i.e. the color of a goodbye dot), as determined by the transition tables within the visitor, according to the following definitions:

- **Noaccess** usage means that the visitor does not play any In, Out, or Inout roles.
- **In** usage means that the visitor does not play an Out or Inout role, but may play an In role.
- **Out** usage means the visitor will play an Out role, and that will never follow an In or Inout role (but may precede it).
- **Inout** usage means everything else—i.e. that the visitor only sometimes play an Out role, or at least sometimes plays an Inout role or an In role followed by an Out role.

These usages for a plan's visitors define the arrowheads (permissions) which the roles of a situation must have for the plan to be used within the situation—i.e. for the visitors of the plan to conform to the roles of the situation:

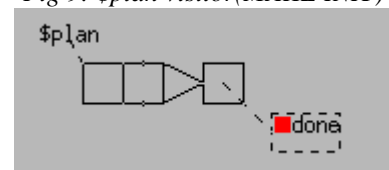
A visitor with usage...	...conforms to a role of same name if it has permissions ...
Inout	Inout
In	In or Inout
Out	Out or Inout
Noaccess	Anything but Out

This table is built to have the important property that a role with the usage on the left can pretend to have the usage (rather than permission) on the right, so a planner can predict how the activity within a situation will behave just by observing the role permissions and interpreting them as usages. In most cases, this means that the right column are the sets containing the left, but in the last row, a Noaccess visitor cannot pretend it is Out, because Out means that the role always puts out new content which is unrelated to the original content.

If a child strategy *does not* contain *any* initial visitors, it may activate at any time—e.g. as soon as its situation exists. If the strategy *does* contain *at least one* initial visitor, it cannot activate until the roles in the parent corresponding to those visitors are ready. If/when the strategy does activate, the initial visitors in the child will immediately become green (i.e. present, as described earlier), and in the parent, the stages will be immediately removed from the places playing those roles, until they are returned when the child strategy bids goodbye to those visitors.

If *all* of the visitors within an activity are initial, the activity is oblivious to anything new happening outside of itself. Even though such an oblivious activity may continue to affect its environment by bidding goodbye to visitors, whatever it will do has already been determined (as much as it can be) by the places it accessed before becoming oblivious (e.g. when it activated). Strategies with non-initial visitors can be made oblivious, too, by transforming the non-initial visitors into initial ones. This is accomplished by explicitly including a special initial visitor named **\$plan** in the strategy, with one goodbye note named "done". (It is not necessary to explicitly include a role for this visitor on situations holding this plan, for reasons that will become clear.) As soon as an activity bids goodbye to this \$plan visitor, the non-initial visitors undergo a transformation, turning them into initial visitors for the remainder of the activity. If a visitor is green (i.e. visible) when the transformation occurs, it becomes absent (stageless/colorless).

Fig 9: \$plan visitor(MAKE INIT)



There is no way for other activities (e.g. in the parent) to sense the order in which an oblivious activity bids goodbye to its visitors, so behaviorally, as far as they can tell, the oblivious activity may have bid them goodbye the instant that it became oblivious. (The proof is beyond the scope of this paper.) For this reason, an oblivious activity will be said to have already **finished**, even if it still has work to do and visitors to bid goodbye. Note that by this definition, a plan with only initial visitors will finish the instant that it activates, so we'll call an activity resulting from such a plan **atomic**. Since a finished activity is oblivious and the order of its remaining goodbyes cannot be sensed, it can be fully described by the contents and goodbye notes (if any) that it gives to each of its remaining visitors.

A finished activity loses the exclusive right to its situation, so it won't prevent another (identical or different) plan from activating within the same situation if it is ready according to its normal activation conditions. However, the finished activity may prevent those normal activation conditions from holding for the subsequent activity, such as when the finished activity has not yet bid goodbye to visitors which are needed by initial visitors in the new activity. When reasoning about this case, the planner only needs to understand that nothing in a new activity can affect anything in a finished (oblivious) activity. By default, the new activity gets a brand new set of places, so the finished one won't interfere with it, either.

Tasks: The Leaves of the Activity Tree

When the plan to go into a situation has simple, straightforward behavior, it may be more natural to express it in some conventional language that is more familiar, suitable, or convenient for the planner or

programmer, like a natural human language or computer language, than to break it down further into more situations and places and roles, as one does with strategies. Such a simple, straightforward plan is called a task. Activities resulting from a task act very much like those from a strategy with only initial visitors, so they are atomic, and they will typically activate over and over again, whenever their roles become ready. The phrase "simple, straightforward behavior" used above means that, to qualify as a task, the behavior of each resulting activity (as defined by the transitions—or goodbye notes—posted to all of its roles and the new contents written to the places playing Out or Inout roles) must be completely determined by (i.e. must be a function of) the content on the places playing In or Inout roles when it activates. Terms like "deterministic", "side-effect free", and "stateless" are sometimes used by programmers to describe this sort of behavior. It is pretty easy to ensure that traditional algorithms or computer programs (scripts, subroutines, methods, or functions) meet these constraints by using ScalPL constructs to communicate with the environment, and avoiding the use of persistent memorized data (e.g. constructs like static/common/saved variables in programming languages) or any I/O or inter-procedure communication that is not sanctioned by ScalPL. Though this may initially seem restrictive, places external to the task already provide the equivalent of I/O and persistence capabilities. Sequential algorithms and programs are deterministic by nature, so they naturally qualify.

Instead of using visitors to move information between the plan and the outside world like a strategy, a task uses a specially-identified set of variables called **arguments**, a construct familiar to computer programmers, and it is these which are associated with, and must conform to, roles on the enclosing situation. A task starts with a textual header that introduces these arguments and specifies the type, usage, dimensionality, and goodbye notes for each, in the form

```
usage argname ^ dimality ( gbnote ) type
```

where

- *usage* is the usage: in, out, inout, or noaccess
- *argname* is the name of the variable to be treated as an argument
- *dimality* is an integer representing the dimensionality of the argument/role, as described later under arrays. If 0 (as usual), the dimality and the preceding caret (^) can be omitted.
- *gbnote* is zero or more goodbye notes that can be issued for the argument, separated by spaces. If zero goodbye notes are specified, the single goodbye note "done" is assumed, and the enclosing parens can be omitted.
- *type* is the data type of the argument (content).

In L23, a file *task.c* containing task *task* expressed in the C language will start with the header

```
#include task.h
/*ScalPL chip { arglist } */
```

where *arglist* is one or more argument declarations as described above. L23 will read the header and create the contents of the *task.h* file to make the *task.c* file compile correctly with a standard C compiler with no modification.

Elsewhere in the body of the task, these arguments can be used as standard program variables with a few restrictions, but specifics will change from language to language. Generally, arguments with *noaccess* usage cannot be used as variables, those with *in* usage cannot be modified, and those with *out* usage may be automatically initialized to some constant (e.g. zeroes) at the beginning of each activity to prevent the old values from being used. To bid goodbye to an argument, a special construct is used which specifies the argument name, a set of indices (or index ranges) if the dimensionality is non-zero, and the goodbye note.

In L23, this is a function or macro call of the form *ScalPL signal(argname gbnote, indices)* where *argname* is the name of the argument, *gbnote* is the goodbye note, and *indices* is zero or more indices depending on the dimensionality of the argument, as described later under arrays. Variants exist to specify regions. Once an argument is bid goodbye, it generally can no longer be accessed, though extensions may be available both to continue accessing the values in such arguments locally.

Partial Bindings, Binding Constraints, and Overloading

By default, for a plan to activate within a situation, the names of all of the visitors in the plan must correspond exactly to, and conform to, the roles on the situation (excluding anonymous roles, described later). However, the planner can tailor this, requiring that only a subset of visitors in a plan must be bound by roles on its parent situation, by specifying a **binding constraint expression** (or **BCE**) for the plan. This

BCE consists of a list of visitor names connected by "&" (meaning logical "and", or conjunction) and "|" (meaning logical inclusive "or", or disjunction). Normally, "&" will take precedence over "|", but parentheses can be used to override this. A visitor name in a BCE is equivalent to an assertion that a role of the same name exists (and is bound) on the plan's situation in the parent. A plan can activate within a situation only when the plan's BCE evaluates as true with respect to the role bindings on that situation. Any visitor which is not bound will not be usable at all within the plan—i.e. any situations bound to that visitor will be effectively non-existent, or GC'd. A BCE consisting of a single "&" is a shortcut meaning that all of the visitors must be bound (which is the default), and one consisting of a single "|" is a shortcut meaning that any (combination) of the visitors may be bound, as long as one of them is. In L23, the BCE is just filled into the BCE field for the plan as text.

BCEs make it possible to treat a single plan as a collection of different (possibly inter-related) plans, or viewed another way, to allow different parts of a larger plan to be utilized in different situations. A plan which can be bound in different ways in different situations can be considered as a library, or as an object having different access methods. (The latter will be discussed in much greater detail in a subsequent section.)

Putting multiple plans into a single place or constant allows for **overloading**—i.e. the ability to use different plans with the same name and same visitors, with the decision of which to use based both on (1) which BCE permits the binding on the situation, and (2) which plan has visitors of the appropriate data types. If more than one plan on the place or constant fits both the bindings and the data types on the situation, the binding is considered illegal. Multiple plans may be depicted graphically as a single unit by sectioning a single diagram with dashed lines or some other clear notation. Union constructors can also be used to combine separate plan constants into new constants.

Buffering and Shared Readers as Optimistic Execution

Consider a role which is not Out or Inout (i.e. has no arrow on the place end of its line) and for which all of entries of the transition table are the same stage/color. Now assume that the activity in the associated situation is actively using that role—i.e. that the visitor or argument corresponding to the role is present in the activity. If one is asked what the new stage and contents of the place will be after it plays that role, even without knowing anything about the plan/activity in the situation, a likely guess will be that the content will not change (since the role is not writable) and the resulting stage will be the only one in the transition table. This is almost right, but it ignores the case where the activity never bids goodbye to the visitor, so no transition at all occurs to the role and the place stays stageless/colorless.

A role like that described (having In or Nodata permissions and all transitions having the same color) will be called a **predictable role**, and ScalPL has a special rule that, for such roles, it's OK to accept the "almost right" answer as the right answer—i.e. to assume that a transition will eventually occur for any "active" (i.e. present in the activity) predictable role. In fact, such a transition can be assumed to occur even in cases where there is clearly nothing in the activity that will ever perform that transition. This is effectively the same as saying that even if the activity never explicitly performs a transformation (bids goodbye) to a predictable role, somehow the transition will magically occur anyway. If such an assumption is not desired (which should be very rare if ever), a role can be forced to be unpredictable by prefixing the role name with an exclamation point aka "bang" (!).

This is all one really needs to know to correctly use predictable roles within plans, but further insight into the motivations and applications of this rule can help to use it more effectively to optimize performance. For example, consider the following case where a single place plays a readable role in two different situations:

Fig 10: Two predictable, multiple readers

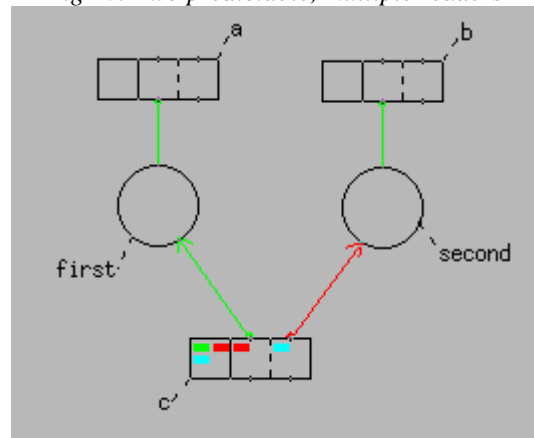
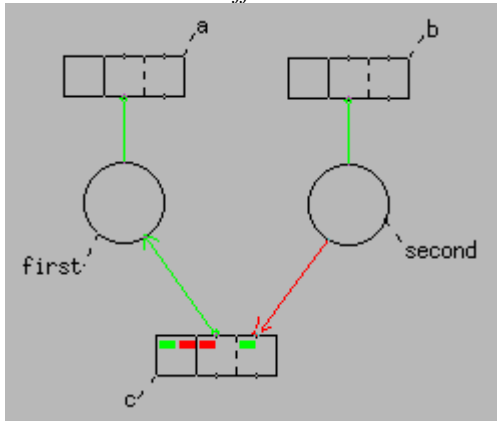


Fig 11: Predictable reader, potentially buffered



As soon as the place begins to play a role in the first situation, it can be assumed that the role for the second situation will eventually become ready. If all other necessary roles for the second situation are ready and there are sufficient resources, there's no reason that the place can't begin to play the role in the second situation even while it is still playing the role in the first, as long as any transition (goodbye note) that does eventually explicitly come from the first activity for the role is suppressed (since its effect has already been felt). This technique of having multiple activities reading from the same place at the same time is sometimes called **multiple readers**. Note that this decision of whether to go ahead with the second activity is not made by the planner, but by the entity carrying out the plan.

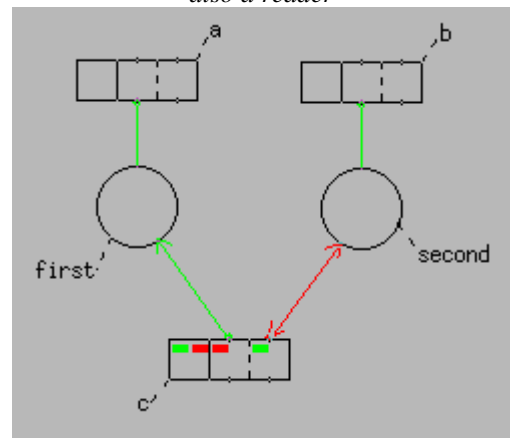
Now consider a similar case where the second activity has an Out role:

If the same trick was played to allow the place to play the role in the second activity even while still playing the role in the first, the second could change the contents of the place and those altered contents could be seen by the first. Since that would be a different outcome than the un-optimized case, it would be an incorrect optimization. However, since the second doesn't read the old contents anyway, the entity carrying out the plan can give the second a brand new place as if it was the original one, and discard the place from the first activity as soon as it is done with it. This technique of creating new versions while retaining old versions for reading is sometimes called **multi-buffering**.

Finally, consider the same case, but this time where the second activity's role has both read and write access to the place. It can work just the same, as long as the entity carrying out the plan copies the contents of the old place into the new place. This technique of allowing a writer to use a new copy of something being read is sometimes called **copy-on-write**.

These examples are meant to demonstrate that when there are available resources, predictable roles can help the entity carrying out the plan to optimize performance. Even though the planner doesn't need to be involved in carrying out these optimizations, it is often to the overall benefit of the planner to be aware of these opportunities.

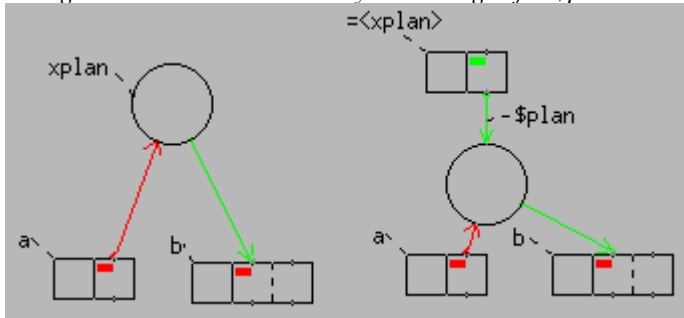
Fig 12: Buffering still possible when writer is also a reader



Plans as Place Contents

For a plan to activate, it must be assigned to a situation, but until it is, it can be considered as just a piece of data, and therefore can be held as contents on a place. To direct a situation to find the plan it is to activate on a place, leave the label off of the situation, and instead use a role named "\$plan" from the place holding the plan to the situation. Note that this role is used by the situation itself, and may or may not be explicitly represented as a visitor in the plan in that situation (described later). The \$plan role must be ready for the situation to determine the plan thereon to in turn determine whether the bound roles truly correspond to visitors, and which of those (corresponding to initial visitors) must be ready for activation to occur.

Fig 13: Labeled situation is syntactic sugar for \$plan role



As data, a plan is treated just as any other named constant, in this case named for the plan, so the place can be initialized with the "`=<planname>`" construct. In fact, a labeled situation can just be considered as a shorthand ("syntactic sugar") for having a separate place holding the plan and bound to the situation with a green \$plan role. Constructs described later will take more advantage of regarding plans as contents, but even those described so far can be used

for passing plans into strategies. That is, by binding a situation in a strategy to a visitor with a \$plan role, the strategy's parent can supply different plans to be plugged into that situation. Of course, the bindings for the situation must be compatible with the plan supplied.

Controlling and Sensing Activation

To control when a plan can activate, or to sense when a plan does activate, even if the plan has no initial visitors, one uses extra roles for the situation that the activity doesn't even know about, called **anonymous roles**. Anonymous roles always have one transition named "done", never have an arrow on their place end, and are denoted as being anonymous by either having a slash drawn through their line or by having a label starting with (or entirely consisting of) an underscore. Each anonymous role acts as though the plan assigned to the situation has an initial visitor for it, so the plan won't activate until the role is ready. Since anonymous roles always have one transition and are In or Nodata, they are always predictable, so a transition will eventually be applied to them after activation even though the activity in the situation isn't even aware of their presence.

Taking anonymous roles into account, for a plan to activate within a situation, first all anonymous roles must be ready, then the \$plan role must be ready, at which time the plan on the \$plan role can be consulted to determine which other roles correspond to initial visitors and therefore must also be ready. When all of these conditions are true, the plan may activate. In fact, if these **activation conditions** are continuously (rather than intermittently) true, then the plan *will* activate in a finite time. As described, when the plan activates, the color is stolen from all of the initial roles, and at any time after that, transitions may be optimistically performed to any predictable roles.

Instead of depending on a particular place being in a particular stage before a role becomes ready, a **fork** can be used to make a role become ready based on any one of a number of stages, perhaps on multiple places. As its name implies, the role line forks (i.e. splits) and the different branches may be given different colors, different transition tables, and may be bound to different places, but they will all share the same name and have the same data type. A forked role will become ready when any of the branches becomes ready, and any activity created as a result of that role being ready will behave as though the role is bound through just one of the ready branches.

One common special case of the fork is when multiple branches go to the same place and have the same transition table, but different colors. Such a role is different from a non-fork role only in that it will become ready when the place reaches any one of the stages specified by the fork colors, instead of just one. For example, consider a place which holds a data structure like a stack or queue which may be empty (green stage), full (red stage), or neither (blue stage). A situation to add a new element should not be able to activate if it is full (only if empty or neither), and one to remove an element should not be able to activate if it is empty (only if full or neither). If role colors are signified by solder dot colors rather than line colors, this case is often shown by just adding multiple solder dots of different colors to the single role line and transition table. In L23, to add extra solder dots to a role, select the dot from the palette and then click on the role line. To delete solder dots, click on the solder dot to delete while the dot is selected in the palette.

Atomicity and Serializability

As described, all plans can be ultimately refined into tasks, and the activities resulting from tasks are atomic (which in this case means they finish as soon as they activate). Activities resulting from strategies will also be atomic if all of their visitors initial. Atomic activities have important properties that make plans which use them easier to construct and understand.

For example, to understand how an atomic activity can affect other activities within a situation (in its parent), there is no need to take into account how long it will take to finish, or the order in which it bids its visitors goodbye, though it may be important to know which visitors it will never bid goodbye, if any. From the perspective of the situation holding the activity, the only important behavioral qualities of an atomic activity are the new contents it puts into its Out and Inout visitors before bidding them goodbye, and the goodbye notes (if any) that it gives to each of its visitors. When bound by the rules of ScalPL, the only influence which the world outside of the atomic activity can have on these things is the contents of the In and Inout visitors when the activity begins. (The stage of the visitors is not an influence, because as far as the activity is concerned, they are always green at activation, and if they are not green, it does not activate.)

These properties make it easier to develop a plan, especially when different parts are to be developed by different entities. Since atomic activities are epitomized by the mapping (i.e. input–output correspondence) between the data on their In and Inout visitors when they begin and the data on their Out and Inout visitors and the goodbye notes on all of their visitors when they're finished, a specification of that mapping alone constitutes a complete specification of the task. The planner can provide such a specification to a subplanner, who/which can implement that specification in any way, and as long as the resulting implementation meets that specification, it will function as expected in the context of the larger plan. While that may seem like common sense, it is not so common when activities are not atomic, in which case the various orders in which events occur (such as the order in which visitors are invited into the activity and/or bid goodbye) can be as important in a specification as the actual outputs and transitions provided by the activity. If the atomic activity is created from a task, the mapping will not only be atomic, but will be deterministic/functional, with only one possible result for every combination of inputs.

These properties also have important implications for understanding plans. Since activities outside of an atomic activity cannot have any influence on the activity after it has started (and therefore finished), a planner never needs to consider what is going on in more than one atomic activity at a time to understand the logic of the plan. And since all plans can ultimately be refined into only tasks, and tasks create only atomic activities, entire concurrent systems can be understood one task at a time, which is good because our brain usually works best when it can concentrate on one thing at a time. Regardless of what might really happen when a plan executes, the atomic activities can always be treated as though they occur one at a time, a property called **serializability**. This is related to the fact that each atomic activity can be treated as though it takes no time at all to execute.

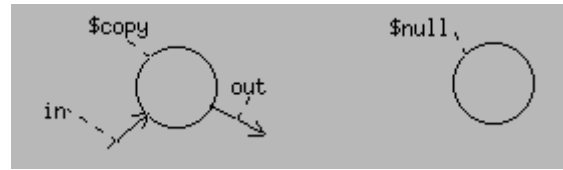
Even an atomic task might never complete (i.e. might never bid goodbye to one or more of its visitors). This can be because of some sort of permanent interruption (a "fatal error" in computer lingo), or because it just gets stuck running in circles (an "infinite loop" or "divergent evaluation" in computer lingo). It may not even be possible to determine whether or not a task will ever complete ("halting problem" in computer lingo). To prevent these special cases from interfering with the simple task specifications, we simply imagine that a task always bids goodbye to all of its visitors, but some of them (i.e. the ones that it really doesn't get a special goodbye note, called **bottom** (for historical reasons). Bottom is special because it doesn't get converted into a new stage for the corresponding place in the parent by the corresponding transition table, so the place remains stageless/colorless. From the perspective of the situation in the parent, the effect is the same regardless of whether visitors were never bid goodbye or whether those same visitors were given goodbye notes of bottom—i.e. the corresponding places remain colorless.

Standard tasks: Copy and Null

Most tasks are created by the planner or supplied by other planners. Two tasks, called **\$copy** and **\$null**, are actually part of ScalPL, and guaranteed to be available to the planner.

\$copy has two roles, called in and out, of any type (even plans) as long as the types must match. (ScalPL tools will often coerce them to be the same type if one type is unspecified.) Each has one transition, named done. \$Copy behaves simply as though it exactly copies the contents from its in role to its out role whenever it activates. When a computer carries out a plan containing \$copy tasks, it can often do so without actually copying anything, by simply remembering what is supposed to be copied to where, using techniques like renaming and copy-on-write.

Fig 14: \$copy and \$null tasks



\$null has no roles. It is the only task allowed to have no roles. It does nothing but activate. By adding anonymous roles as necessary, it is useful for changing the stage of places when the conditions are right and no other work needs to be done.

The use of the \$null task is sometimes made more obvious by using a dashed situation circle. The use of \$copy can be abbreviated by simply drawing one role-like arrow from the side of one place to the side of another. Put another way, in the simple case where a situation contains a \$copy task with no anonymous roles or binding modifiers, the situation circle and labeling of the roles can be omitted. The two ends (or their solder dots) may still need to be colored differently to represent the stage that each place must be in for the \$copy to occur.

Object-Oriented Planning

Creating Objects with Sharable Persistent Places

It is possible to make plans which share some or all of their places (and their stages and contents) among some or all of their activities, whether those activities overlap in time or don't. Though plans so far have been considered from a standpoint of how their activities can achieve a goal on their own, this sort of plan can also be seen as a collection of shared places and a controlled set of mechanisms for other entities to access and/or alter members if this collection. In computer lingo, these matters are often called objects, or instances of abstract data types (ADTs) or classes. Since much has been written about productive and constructive ways to utilize ADTs or objects to manage complexity, this section will relate some of the terminology used in the literature to ScalPL plans.

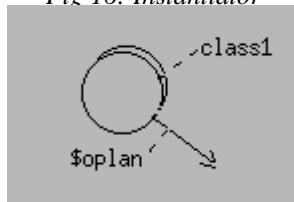
A place which can be shared by some or all of the activities created from a strategy is called **instantiated**. A place is instantiated (i.e. turned into an instantiated place) by a process called **instantiation**, described shortly. A place in a plan which can be instantiated is called **instantiatable**, and is denoted by adding a shadow behind the place rectangle—i.e. an extra parallel line above and to the right of it. In L23, this shadow is added by using the touch-up brush and touching the mouse to the top of the place rectangle. Up to four shadows can be added, for reasons described later, after which they recycle to zero. For now, we consider that instantiation of a strategy instantiates all of the instantiatable places within the strategy all at once.

Fig 15: Instantiatable place



One set of instantiated places doesn't need to be shared with all of the strategy's activities. That is, the instantiatable places for a particular strategy might be instantiated once to be used in some activities, but they could be instantiated again to yield a different set of places for a different set of activities, even while the first set of places still exists. Following object-oriented (OO) lingo, a strategy with instantiatable places will often be called a **class** (or abstract data type, ADT), each instance (i.e. set of instantiated places) created by instantiating the class will then be called an **object**, and each situation that an object can be put into will be called a **binding**. The planner's goal is to design each class so that its instances (objects) are cohesive entities that interact with their environment (via bindings) in an intuitive way such as mimicking real-life objects, making it easy for others to remember and understand their behavior without necessarily considering their precise internal implementation. It is the generalization of how all operations, on a computer or in life, make sense on some types of objects, but may have no meaning or different meaning on others. (You can tune a piano, but you can't tune a fish.) Object-oriented programming is based on the ability to define the hidden internal characteristics (instantiatable places) that define a type (strategy) and the operations (roles) that others may apply to instances of that type and how those are implemented internally to use or change these hidden characteristics.

Fig 16: Instantiator



To instantiate (the instantiatable places in) a class to create (the instantiated places in) an object, ScalPL uses a construct called an **instantiator**, shown like a situation except with a shadow and with an extra initial role called **\$oplan**. An instantiator reads the strategy named in its label (or if none, from its \$oplan role), creates/instantiates the instantiatable places, and then writes a new plan to its \$oplan role which is identical to the input plan except that in place of the instantiatable places, there is an annotation which describes where to find the corresponding instantiated places. When a subsequent situation

activates such a strategy, the situation will not bother building instantiated places itself as it does with the other non-instantiated places, but will instead just find the ones already instantiated. This means that the new plan which is written to the \$oplan role is effectively an object, because when it is activated repeatedly, each activation will share the same instantiated places. The plan which the instantiator reads (from its label or \$oplan role) is effectively a class, because it serves as the model for a new object each time it is instantiated.

Instantiation–time Activities, and Garbage Collection

The instantiator is actually a special case of a situation. Any situation can create places—standard situations just usually create the non–instantiatable places, while an instantiator instead creates just the instantiatable places. Instantiatable places may seem to have longer lifetimes than uninstantiatable ones, but in fact, both kinds can be considered to hang around forever, at least logically, as long as care is taken to make (and use) new instantiations whenever the plan demands it. Practically speaking, however, it makes sense for the host to dispense with a place (whether instantiated or not) as soon as it can be determined that it will no longer contribute to the outcome of the plan. This act of having the host automatically dispense with useless constructs is usually called **garbage collection** (or **GC**). In some languages, GC can result in hidden and/or uncontrollable actions to find and compact garbage, but those can be mostly avoided here. The most obvious time to garbage collect uninstantiated places is when their encompassing activity finishes. An instantiated place typically remains accessible (by design) through the new plan (object) written to the instantiator’s \$oplan role, so the most obvious time for their garbage collection is when all of these plans/objects are gone—i.e. when the contents of all of the places holding this new plan/object are replaced, or when those places holding the object are themselves GC’d.

GC rules for all places may easily be made far more aggressive, such as collecting places which have entered a stage where there are no situations that can access them. Then, if a place is collected for such a reason, any situations which require that place in order to create an activity can themselves be collected, which may in turn lead to other places being collected from lack of situations. In any case, GC is the job of the host entity executing the plan (the runtime system), but the planner may be able to simplify the host’s job by understanding that job.

In addition to creating places, an instantiator can also bind visitors to roles, just like a situation. Unsurprisingly, the only visitors an instantiator will bind are those which have shadows. Such visitors will be referred to as **instantiatable visitors**, even though that term is somewhat misleading, since visitors aren’t really instantiated, just bound at the time of instantiation.

Instantiatable visitors can be used to initialize instantiatable places within the strategy during instantiation. This is possible because the instantiator, like other situations, creates an activity. Since uninstantiatable places won’t have been created yet, only situations (if any) that are bound only to instantiatable places will play a part in an instantiation activity. For example, if a plan contains a situation holding a \$copy task which copies contents from an instantiatable visitor (playing \$copy’s *in* role) to an instantiatable place (playing \$copy’s *out* role), then this copy would be performed in the instantiation activity. For cases like this example where the action is to be performed once at instantiation time, it is common to make the instantiatable visitors initial, so that the instantiation activity (other than the instantiated places) can be GC’d after the initialization takes place.

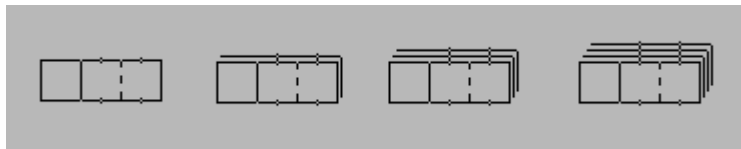
Instantiation activities can also be long lasting, such as to repeatedly move contents between instantiatable visitors and instantiated places, and/or to wait for specific events (e.g. specific stages in specific instantiated places) to take actions like copying contents from some instantiated places to the outside world. In this sense, an instantiator can serve as a back door to funnel information between the innards of an object and the outside world using methods only available to the plan originally creating (instantiating) the object. If an instantiation activity has any non–initial visitors (for purposes such as these) but needs to eventually become oblivious, the \$plan place used must also be instantiated.

Multiple Instantiation Levels

In the above, we considered that an instantiator instantiated all of the instantiatable places in a plan all at once. In fact, it is possible to instantiate just some of them at a time by giving the places different

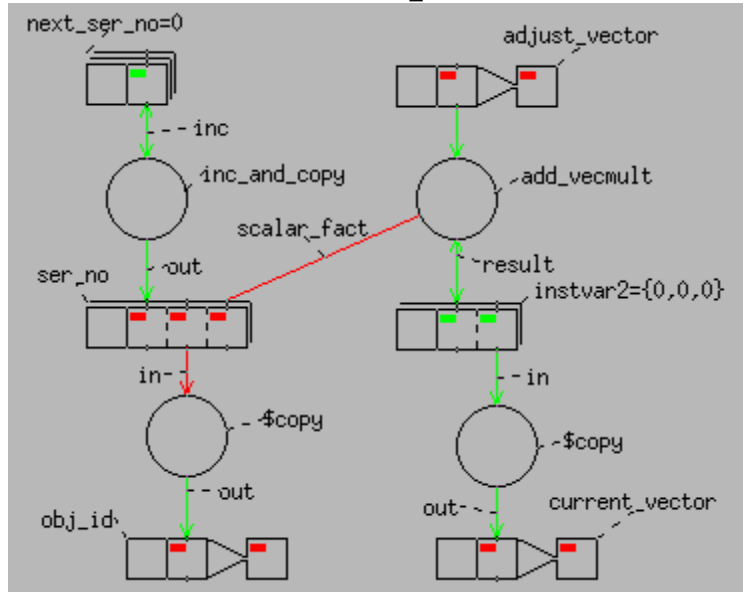
instantiation levels (ILs)—i.e. number of shadows behind the place. So far, we have been using the term "instantiatable places" to describe places with instantiation level 1 (one shadow), but any number of shadows can be added. In L23, shadows are added by selecting the touch–up brush from the palette and

Fig 17: Places with Instantiation Level (IL) 0, 1, 2, and 3



clicking near the top of the place.
 An instantiator also has an instantiation level, marked by the number of shadows it has, so those described so far have also had instantiation level 1. In I23, shadows are added to situations/instantiators by selecting the touch-up brush from the palette and clicking near the center of the circle. An instantiator will instantiate only the instantiatable places having the same instantiation level as the instantiator. Places and situations with no shadows are sometimes referred to as having IL 0, since they behave perfectly analogously to the other ILs, except that a situation at IL 0 has no \$oplan role to "save" the places which it instantiates.

Fig 18: Class with class variable next_ser_no, instance variables ser_no and instvar2



Places with higher instantiation levels are generally shared more widely and instantiated less frequently than those with lower levels. This guideline is enforced by instantiators, since an instantiator will fail if the plan it receives has uninstantiated

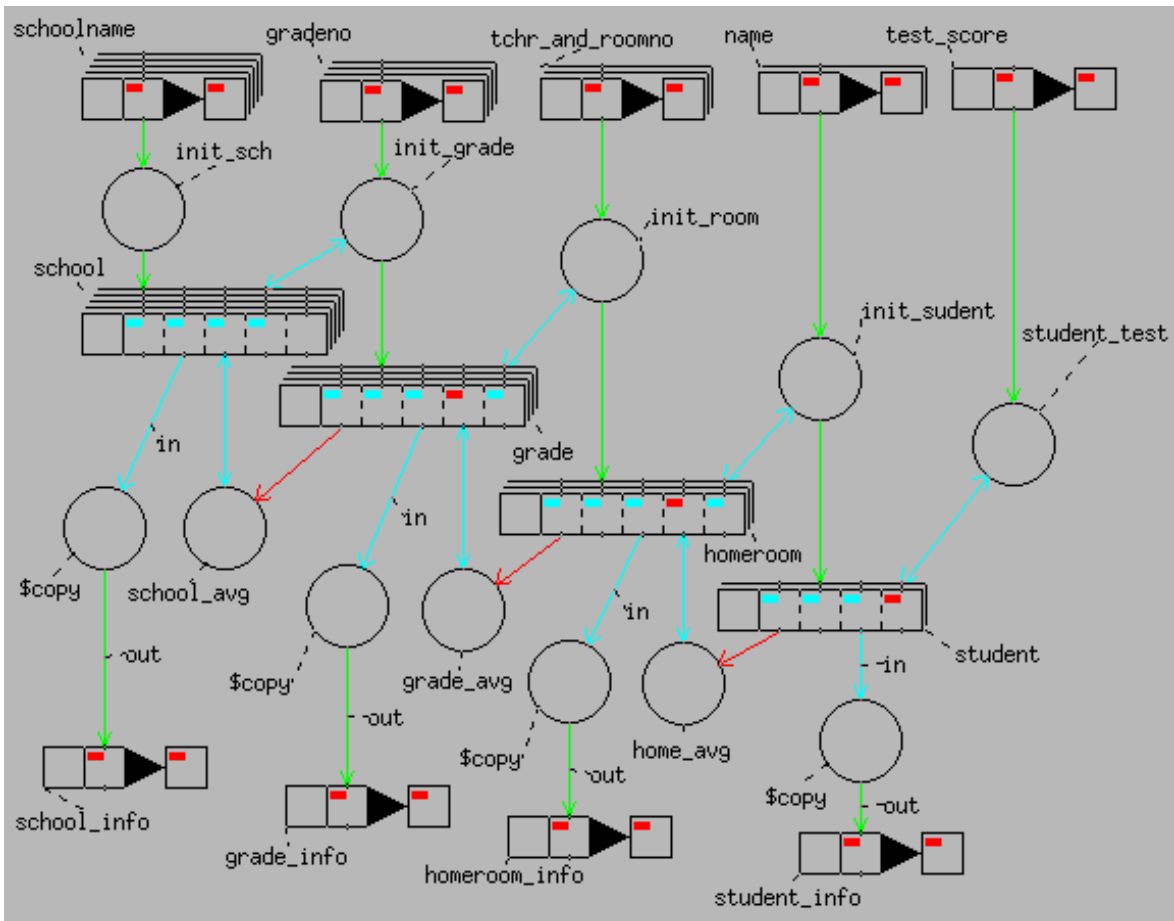


Fig 19: sgrs_factory for creating school-grade-room-student hierarchy

places with IL higher than itself. For example, if a plan has some places with IL 1 and others with IL 2, then using an instantiator with IL 2 first will create a new class with the IL 2 places instantiated. Each time that resulting class is instantiated with an IL 1 instantiator, it will create a new object which has new IL 1 places but shares the IL 2 places with the other objects in the class. In OOP vernacular, the IL 2 places are similar to **class variables**. In OOP languages in general, though, it is neither common to have the equivalent of ILs higher than 2, nor to even create different instances of class variables at IL 2 (as would happen, for example, if the original plan in this example was subjected a second time to an instantiator with IL 2). In fact, there is no common OOP term for the original plan in this example (e.g. it is not what is ordinarily termed a "superclass", which will be described in the next section).

Simply put, ILs are useful for creating a hierarchy (tree) of activities where all activities below level *n* in that hierarchy will share information with activities at level *n* or greater. In many cases, the instantiation activities higher in the hierarchy have little use except to serve as ancestors to show how the lowest level activities are related. Instantiation levels may work well in situations where different kinds of experiences

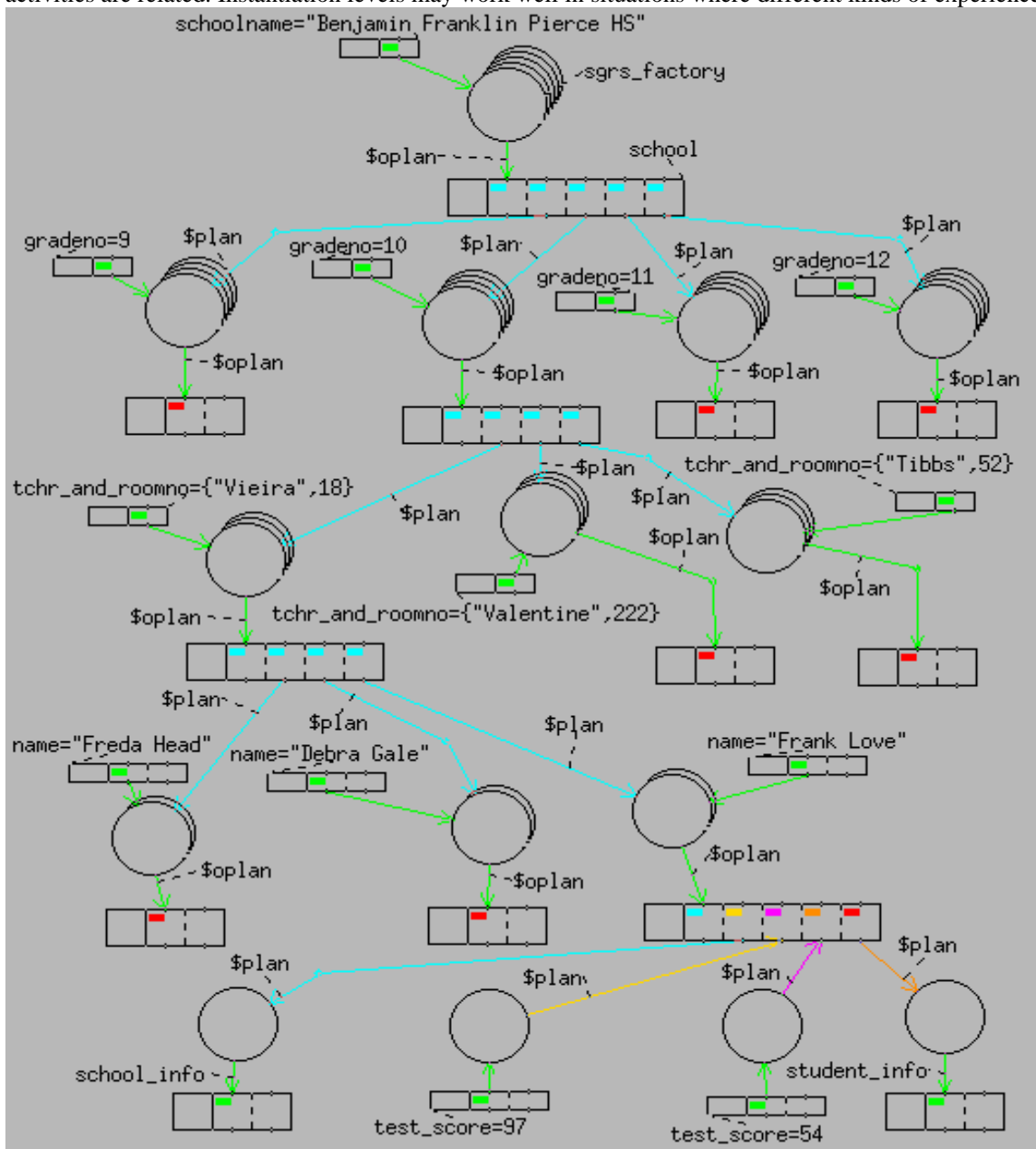


Illustration 20: Using the sgrs_factory

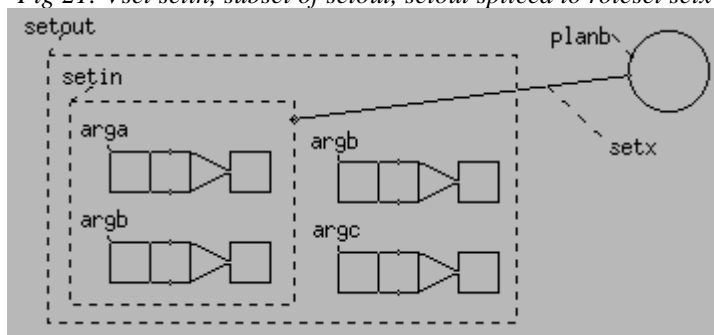
are or information shared among hierarchical groups—e.g. students may have experiences and information of their own (held in places with IL 1), others shared with students in the same classroom (IL 2), others with students in the same grade (IL 3), others with students in the same school (IL 4), etc. See the next diagram for an implementation of "sgrs_factory", a strategy implementing this school–grade–homeroom–student hierarchy, and the following diagram for a rather contrived example of its use to create a school with several rooms, grades, and students.

Delegation

Object-oriented design implies more than the ability to instantiate classes into objects. It also implies the ability for one class to inherit or extend functionality from another. That is, one class can act as a representative, or **subclass**, of another so-called **superclass**, perhaps with the subclass offering some additional and/or better defined functionality than the superclass. Such a relationship is often called an "is-a" relationship, as in, the subclass "is a" representative of the superclass. To support inheritance and other techniques, ScalPL provides support for **delegation**—i.e. allowing an object to collectively pass some of its **interface** (i.e. visitors) off to another object to handle while intercepting or adding other parts of the interface on its own.

As a step toward simplifying delegation, any number of visitors in a strategy can be grouped together into a set, called a **visitor set** or **vset**, shown as a labeled dashed box around those visitors. Vsets can be nested arbitrarily to create subsets, but cannot intersect. (The placement of vset boxes has no special meaning with regard to other constructs such as non-visitor places or situations). A situation in the parent must be aware of the vsets within the child plan, and of how visitors and other vsets are nested within those vsets, to bind those vsets or visitors. A role line coming from the situation may represent a vset just as easily as representing a single visitor, but a role line representing a vset (sometimes called a **roleset**) can be differentiated from one representing a visitor by where it is bound. That is, a roleset cannot be bound to a place, but can be bound to a vset in the parent, called a **splice binding**, in which case the vset in the child

Fig 21: Vset setin, subset of setout, setout spliced to roleset setx

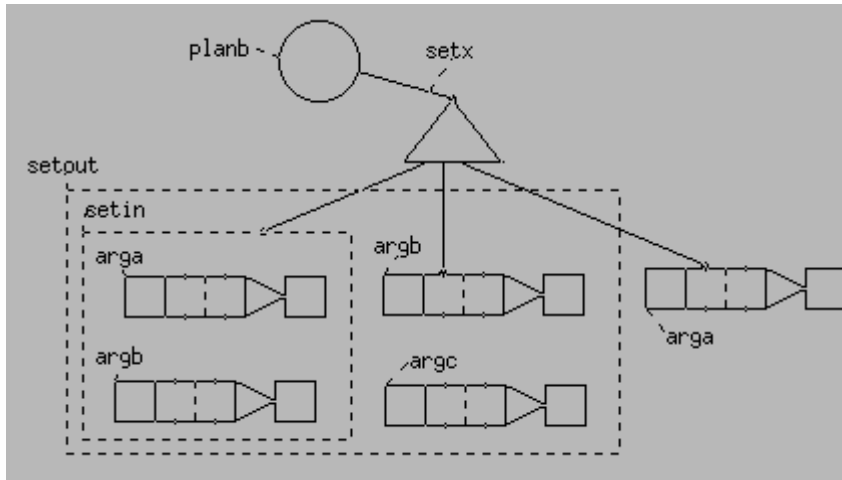


and the vset in the parent must match, based on names. For splice bindings, visitors in the child vset must match to visitors in the parent vset with identical goodbye notes (and types, when used), and vsets in the child must match to vsets in the parent, for which the splice binding occurs recursively.

A roleset can also be bound to a **bundling**, which explicitly breaks out each component (role or

roleset) of the roleset so that each can be bound independently. A bundling is represented as an isosceles triangle, with the apex (point) attached to the roleset, and labeled roles for each of the components of that vset emanating from the base. In L23, a bundling is created using by selecting the bundling triangle from the palette and then clicking at the desired spot in the canvas. The incoming role is attached by dragging the end anywhere into the triangle. Outgoing roles are created by selecting the role in the palette, and then in the canvas pulling the roles out of the base, just as one does with a situation circle. The bundling is originally created to only support a limited number of roles on the base, but this can be increased or decreased by selecting the touch-up brush from the menu and then stretching and/or shrinking the base by dragging the end with the mouse.

Fig 22: Bundling roleset setx from setin, arga, and argb



A role/roleset from a situation or bundling can be labeled with a **path**, rather than a single name. A path is a sequence of names separated by forward slashes (/), where the first name represents a roleset in the situation or bundling as usual, and each subsequent name represents an element nested immediately within the roleset named prior. The last name is the role or roleset actually being bound. If a splice binding implies a binding for some particular role within a

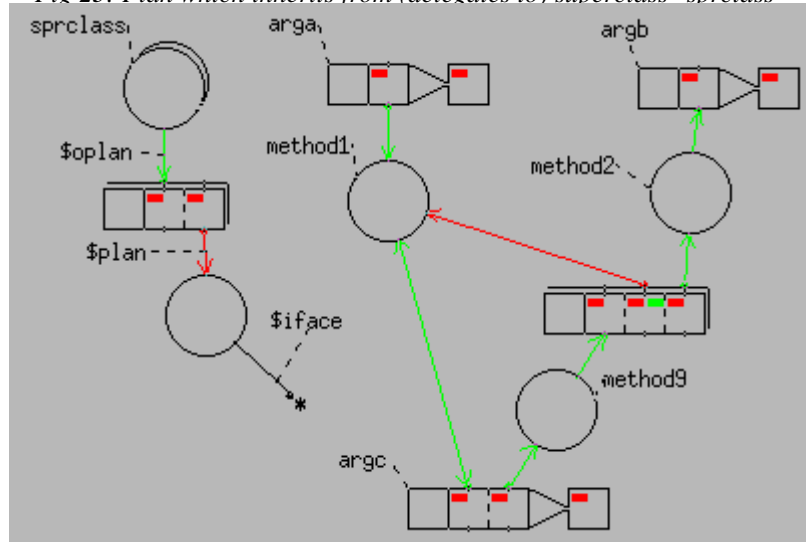
roleset, and an explicit binding is also specified for that role (e.g. using a path), then the explicit binding takes precedence for that element over the splice rules.

Any vset can contain a **wildcard**, represented as an asterisk, which will represent an arbitrary (possibly empty) collection of visitors and vsets. When the plan holding the wildcard is activated as a child in a situation, the precise collection represented by any particular wildcard is **resolved** (defined) to be equal to the visitors and vsets which are necessary to make the child conform to any roles or rolesets on the situation in the parent. A wildcard is not a vset *per se*, but if a named roleset is bound to a wildcard, the wildcard is interpreted as though it is a vset containing those elements, and the roleset is bound to them with a splice binding.

The top level variables, vsets, and wildcard (if any) in a plan are considered as being elements of a vset called **\$iface**, whether or not that vset is explicitly shown as a dashed box. This implies that all of the roles or rolesets from a situation should technically have a path starting with "\$iface/", but this prefix is implied if not specified.

Vsets and wildcards can also be instantiatable—i.e. to be bound at instantiation time by an instantiator. Since it can be clumsy or inconvenient to put a shadow on a vset box or wildcard asterisk, a vset box without a shadow will still be considered to have IL n if all of the visitors and/or vsets within it have IL n, and a wildcard without a shadow will still be considered to have IL n if it is the only wildcard in an a vset having IL n.

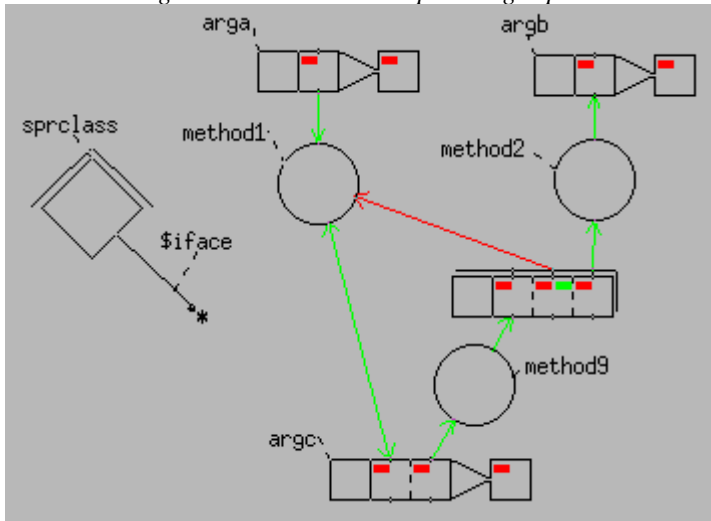
Fig 23: Plan which inherits from (delegates to) superclass "sprclass"



Inheritance (Is-a) Relationship

If a subclass is to add more methods to the superclass without augmenting or replacing/overriding any already there, it can just instantiate the superclass into an instance variable of the subclass and delegate (with a splice binding) all of the visitors in the child's wildcard to it, as shown in the left-most construction in figure 23.

Fig 24: Inheritance example using super



That construction is used frequently enough that ScalPL has another construct which abbreviates and generalizes it, called a **super**, shown as an augmented diamond shape. Figure 24 shows the equivalent of figure 23 after replacing the left construct with a super. In L23, a super is created by selecting the super from the palette and clicking on the canvas where the construct is desired. Lines are pulled from a super much as they are pulled from a situation, and shadows can be added to each side separately by selecting the touch-up brush from the palette and clicking near that side.

the maximum and minimum number of shadows (extra parallel lines) shown on any of its four sides. The super as a whole represents instantiators with this high IL and with this low IL as well as instantiators with all ILs in between, and places with ILs from high to low+1, such that the place with IL n plays the \$oplan role for the instantiator with IL n and the \$plan role for the instantiator with IL $n-1$. See diagram. In addition, any side with n shadows represents the instantiator with IL n , so that any lines from that side represent roles from that instantiator. If the super is labeled, it corresponds to a label on the instantiator with the highest IL. For example, a super showing IL3, IL 2, and IL 0 as in Fig 25 would expand to a plan like that in Fig 26.

More generally, a super has a high and low instantiation level, represented by

Fig 25: Example of super with ILs 3, 2, and 0

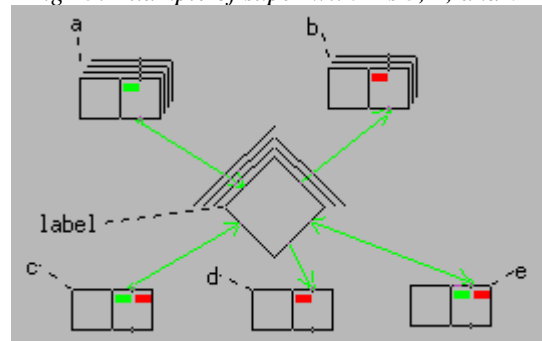
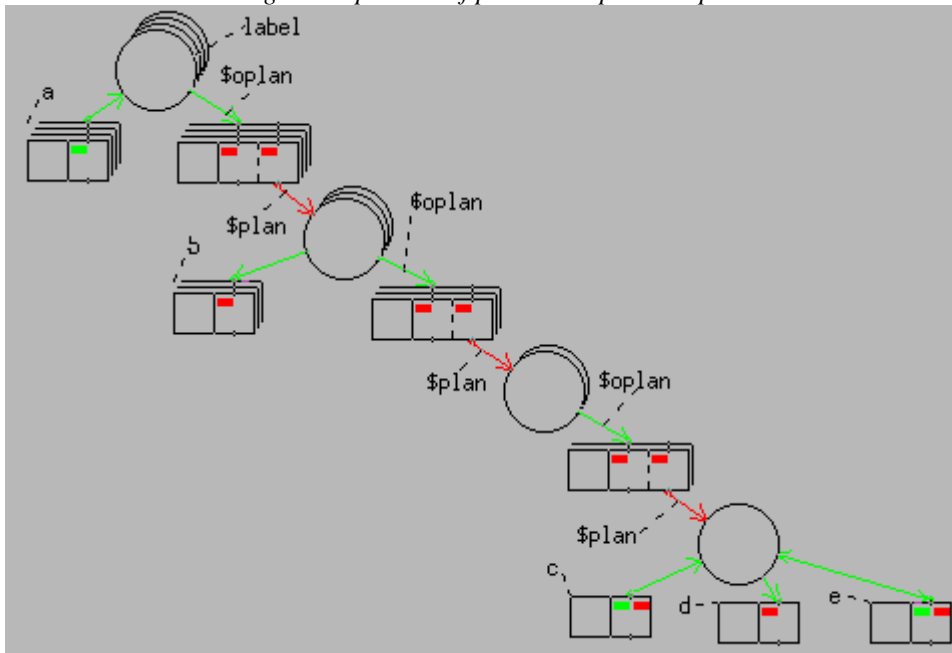


Fig 26: Expansion of previous super example



For the more common inheritance case, where a subclass augments or replaces existing methods in the superclass, additional path bindings are used from the IL 0 side of the super to places within the subclass to override one or more of the \$iface bindings to the wildcard, while placing new visitors with the same names into the subclass interface. If the

superclass roles are being completely replaced by the subclass, the roles from the super may never even become ready. If the superclass roles are just being augmented by the subclass, content may be moved between the place playing the superclass role and the new visitor in the subclass while processing with intermediate situations. In any case, if the ability to use the subclass anywhere that its superclass could be used is to be preserved, a property called contravariance should be maintained—i.e.

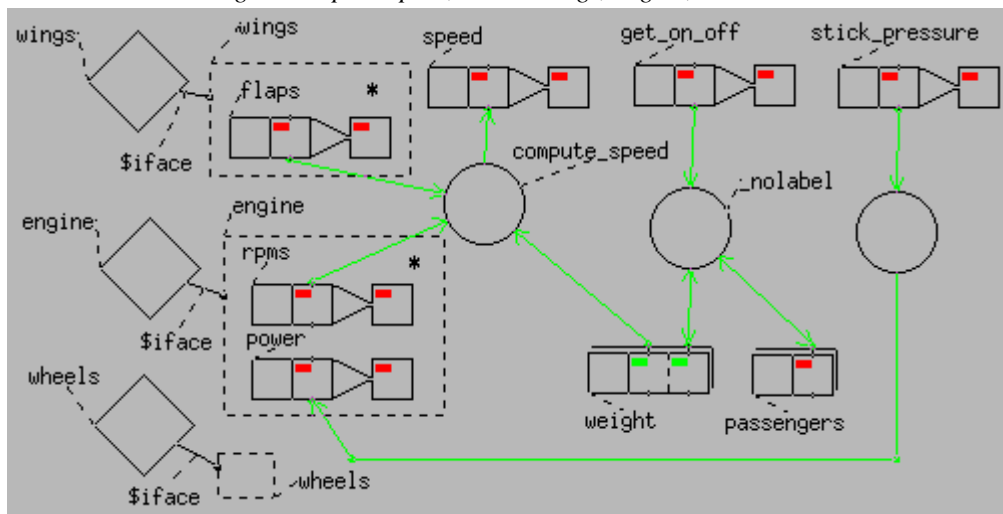
1. a new visitor should be provided in the subclass for every intercepted role from the super;
2. if the replacement visitors are In or Inout, they should adequately deal with at least the same content (i.e. as wide of a domain) as the ones they replace; and
3. if they are Out or Inout, content should not be written that could never have been written to the ones that they replace (i.e. no larger domain).

This is, in fact, another reason to use the super construct rather than its equivalent lower-level constructs: ScalPL tools may be able to recognize that an inheritance relation is being implemented, and thereby to give appropriate warnings if proper covariance is not obeyed, in some cases.

Has-a Relationships (Aggregation)

Vsets are also useful when defining a new class constructed from a number of other classes, each of which are to remain visible to the new class's instances and subclasses, forming so called "has a" relationships (class A "has a" B and C and D) as opposed to inheritance's "is a" relationship (Class E "is a" G). In some interpretations, "has a" and "is a" are very close—i.e. the term "class A has a B and C and D" can almost be interpreted as "class A is a (B and a C and a D and possibly other stuff)". The primary difference is that, when a class is built from "has a" relationships, inheritors from the new class not only inherit functionality from all of its "has a" classes (rather than just one in "is a"), but those inheritors get to choose which of those "has a" classes they are accessing at any one time, even when multiple of them have identical interface names, etc.

Fig 27: Airplane plan, has-a wings, engine, and wheels



Because of their similarities, a "has a" relationship looks much like an "is a" relationship in ScalPL, but instead of having only one situation being bound to the one wildcard in the \$iface vset of the new class (subclass), each "has a" relationship is represented by a situation bound to a different interface element (usually vset) in the \$iface vset of the new class, usually named for the class.

Uses-a Relationships (Passing Objects and Classes)

Vsets also play a valuable role when passing objects and classes as arguments to other objects and classes, in so-called "uses a" relationships, by allowing the new parent to manipulate parts of the argument's (new child's) interface, even without necessarily understanding the structure of other parts of that interface. For example, when object A passes object B (having some interface I) to object C, C can use B and bind all or part of B's interface element I to a known vset (e.g. J) in its C's own interface, thereby allowing A to

communicate directly with B through J in the interface of C. In the extreme, C could even inherit from B in an "is a" relationship by binding all or part of B's interface to the wildcard in the top level of C.

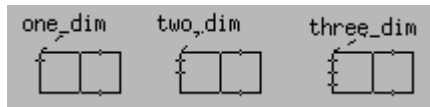
Other Relationships

The relationships above ("is a", "has a", and "uses a") have been singled out for discussion primarily because they are commonly addressed in classical object-oriented design, and so may serve as familiar paradigms to the reader, but ScalPL in no way restricts programs to these constructs.

Arrays

A place can hold contents of any size, including a collection of items such as a file or array or database, but controlling access to the entire collection through a place's single stage can be a bottleneck, thwarting the ability to divide work to be done on different portions of the contents among different parts of the plan. Splitting the contents among a fixed number of places can help some, but this approach can be untenable when the number of places becomes large, and it is clumsy to handle changes in the size of the collection

Fig 28: One-, two-, and three-dimensional arrays



and/or changing relationships between its elements. To circumvent these bottlenecks, ScalPL supports homogeneous collections of places called **arrays**, each array represented as a place (or visitor) rectangle with one or more hash marks on the left edge. An array denotes an infinite set of same-type places, called elements of the array, each with its own control and data state (value and color), and each with a unique address consisting of one or more integers. The number of hash marks on the place rectangle, called the dimensionality of the array, indicates the number of integers in the address for each element. The word **scalar** will be used to mean having zero dimensionality (e.g. no need to supply an index, such as a place with no hash marks). In L23, hash marks are added to a place by selecting the touch-up brush from the palette and clicking on the left end of the place rectangle until the desired number of hash marks appears.

Indexing, Subsetting, Translating

As with scalar places, an activity accesses an array place via a role from the activity's situation. By default, the role has the same dimensionality as the place to which it is bound, and the indices of the role elements correspond to those of the array place one-to-one. If the activity corresponds to a strategy, the dimensionality of the role is reflected in the dimensionality (i.e. number of hash marks) of the corresponding visitor within the strategy. If the activity corresponds to a task, the dimensionality of the role is declared as part of the task header.

There are many cases when a one-to-one infinite mapping between role elements and place elements is not needed or desired, and in some cases, having an infinite number of role elements does not even make sense, such as when the child is a task, or the role corresponds to an initial visitor in a child strategy. (In these cases, all infinity elements of the parent would need to have reached the specified stage in order for the child to activate which would, in general, take an infinite amount of time, as would the child bidding goodbye to all of the infinity elements.) Even in cases where having an infinite number of role elements is desirable, a one-to-one correspondence may not be—e.g. the child may want to access only certain infinite rows or columns, or to transpose or offset the infinite array in certain dimensions. To make a non-one-to-one correspondence between the elements of an array and the elements of the role it plays, the role and/or situation is annotated with a **binding modifier**. There are two forms of binding modifiers, called type A and type B, which are meant to have identical expressive power, but which may differ in how intuitive they may be in different cases or to different planners.

Type A Binding Modifiers

A type A binding modifier consists of a comma-separated list of expressions after the role name (i.e. role line label), in brackets, the number of expressions matching the dimensionality of the role array. Each expression represents one array index, so a type A binding modifier resembles a computed subscript. Each expression can be:

1. an integer constant—e.g. 10—or
2. a placeholder for an index to be supplied by the activity when accessing the role, denoted by the index ordinal prefixed with an at sign (@)—e.g. @2 would represent the second index supplied by the activity—or
3. an integer field from another role on the same situation, denoted by that role name optionally followed by a field specifier path using a dot delimiter—e.g. dimension.height would represent the integer stored within the height field of the dimension role—or
4. two or more expressions connected with "+" or "-" symbols, denoting their sum or difference—or

- an expression followed by an **"in" clause**—i.e. the word "in" followed by the beginning and end of the range, each being either a constant or a fixed value read from another role (i.e. one not dependent on an index from this role). Attempted accesses outside of the range will simply act as references to elements which never become ready (even if they are ready).

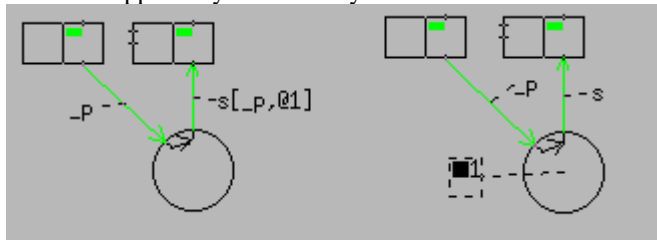
The other role mentioned in rule 3 is referred to as a **primary role**, in which case the role being bound is called a **secondary role**. It is customary to draw an arrow within the situation from the primary role to the secondary role. In L23, this is done by selecting the touch-up brush from the palette, clicking the mouse on the primary role and then on the secondary role. If the primary role is an array (which is not necessarily the same thing as saying that it is bound to an array), its name in the secondary's binding modifier must itself be followed by parentheses holding an appropriate number of indices, and those will be processed through the primary's binding modifier, if any, before fetching the integer index or indices from the place to which the primary is bound. Of course, if the primary role's binding modifier also uses rule 3, then the primary may in turn be a secondary to another primary.

If an activity-supplied index @n is used from rule 2, then all other activity-supplied indices from @1 to @(n-1) must also be used. The dimensionality of the primary role is defined as the number of activity-supplied indices used.

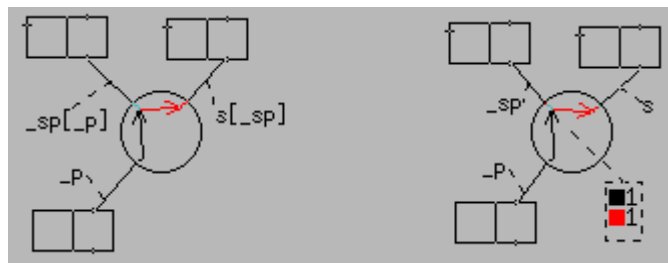
Type B Binding Modifiers

Type B binding modifiers mostly consist of labels on the primary-to-secondary arrows within the situation circle, instead of adding notations to role labels. In L23, primary-to-secondary arrows are automatically color-coded. Selecting the magnifying glass from the palette and clicking on the situation circle shows the label associated with each color, in the same format as magnified transition dots, and the arrow labels are edited just as the labels on magnified dots. In general, type B binding modifiers are much more abbreviated than type A modifiers, which were invented primarily for those who may find type A modifiers too cryptic. Each type B example presented here will be preceded by the corresponding type A expression.

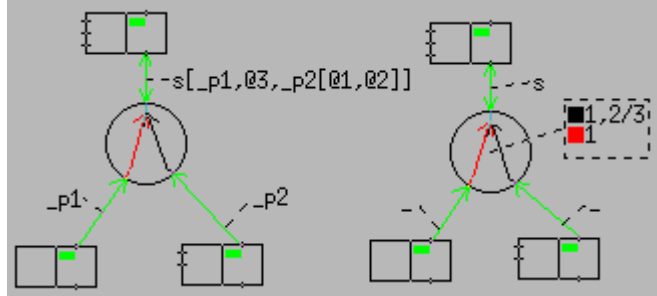
Selection: Labeling an arrow with a sequence of integers specifies that the corresponding number of integers (equal to the length of the sequence) will be read from the primary role and used for the specified indices in the secondary role, providing some functionality from rule 3 from type A. The example here shows both type A and type B modifiers, using an integer read from an anonymous role (_p) as the first index for another two-dimensional array. As a result, the activity in the situation will see s as a one-dimensional role, and the index supplied by that activity will be used as the second index into the array.



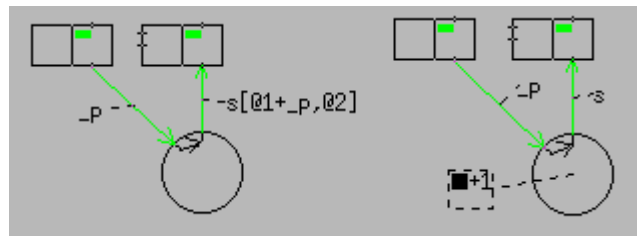
It is quite common to use selections transitively—e.g. where role _p indexes role _sp which then indexes role s:



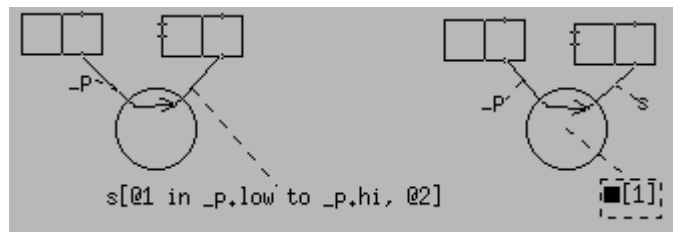
Indirect selection: Prefixing any selection sequence with another sequence and a slash makes the selection indirect, such that when indices are directed toward the secondary role, the indices named in the first sequence are extracted and instead used to denote an element on the primary, and that element is then used to supply indices for the secondary role as in a normal selection. This provides some functionality from rule 2 of type A. The example here shows that the first index for the top array place is obtained from the `_p1` role (as a simple selection), and the third index is obtained from an element of `_p2` denoted by taking the first and second indices originally directed toward `s` and using them to index `_p2` instead.



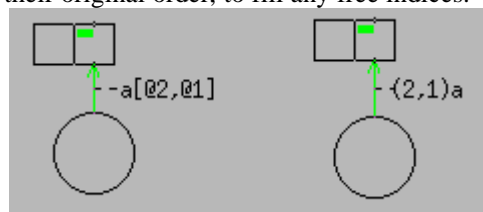
Translation: Prefixing an integer sequence with a + or - specifies that the corresponding number of integers (equal to the length of the sequence) will be read from the primary role and added or subtracted, one-for-one, to the specified indices in the secondary role, providing some functionality from rule 4 from type A.



Range selection: Enclosing an integer sequence in square brackets delimits the indices in the specified dimensions to ranges read from the primary role, providing functionality from rule 5 of type A (i.e. an "in" clause).



Permutation: A sequence of integers in parentheses appearing before a role name will be interpreted as specifying which activity-supplied indices will be used to fill any free indices for the role, and in what order those activity-supplied indices will be used. A "free index", in this context, is one that has not been supplied by the role as the secondary of a (direct or indirect) selection. If a permutation sequence is not specified, all of the activity-supplied indices that have not been used for indirect selections (i.e. in their first sequence) will be used, in their original order, to fill any free indices.



Type B binding modifiers do not specifically support constants, as in rule 1 of type A, but they are already supported via anonymous primary roles bound to constant places. When using type B modifiers, the dimensionality of a role is defined as the number of unique integers (indices) named in the first sequences of indirect selections for which that role is a secondary, plus either (a) the length of the permutation sequence, if one is specified, or (b) the number of free indices if a permutation is not specified.

In general, type A binding modifiers should be considered as syntactic sugar for type B modifiers. Sets of type A modifiers that do not correspond exactly to some set of type B binding modifiers will generally not be considered legal.

Rules in common for Type A and Type B Binding Modifiers

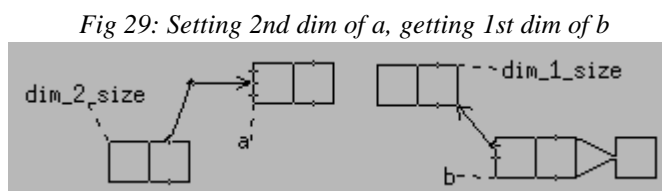
Using these constructs, an array as viewed through the role may appear to be translated up or down in one or more dimensions (by adding or subtracting constants from those indices in the modifier), to be transposed in one or more dimensions (by permuting the order that the role indices are used to index the array), to have had one or more dimensions removed (by completely specifying the array index in those dimensions without use of any role indices) or added (by using multiple role indices to resolve fewer array indices), to have been restricted in one or more dimensions (by using range selections or "in" clauses), to have had some template or sparse items reorganized into a linear pattern (by using role indices to index into a secondary role to extract only certain indices from the array), or several other combinations and variations of these.

With the advent of binding modifiers, the earlier-mentioned activation conditions must be extended. A plan will not be activated in a situation until all anonymous roles and all primary roles (which may or may not be anonymous) are ready. If some of those primaries (say A) are also secondaries, then of course *their* primaries (say B) will need to be ready and consulted before it is understood which elements of A must be ready. Only when these are all ready will the \$plan role be consulted (when it is ready) to determine which other roles must be bound and which subset of those must be ready, for activation to occur. Even though this creates logical dependence relationships between the readiness of primary and anonymous roles, the \$plan role, and any other initial roles in the plan being activated, this does not affect the atomicity of an activity, since consulting a primary or \$plan has no outward effect. Only when a plan finally activates within the situation are the primaries considered to be accessed, atomically, and a snapshot of the values in those primary elements will effectively be taken and referenced to resolve any further references to the secondaries, even if the new activity or another one modifies the contents of the primary elements during the plan's execution.

Delimited (Finite) Arrays

If every access to an array is to have one or more of its indices delimited to a certain range, the array itself can be delimited by drawing a **delimiting arrow** (appearing like a role) directly from the place containing the range to the hash mark denoting the index to be delimited. This is exactly equivalent to putting an "in" clause or range selection denoting that place on every role accessing the array. In L23, a delimiting arrow is drawn by selecting the role line from the palette and pulling the line out of the desired hash mark of the array and connecting it to the place holding the range in the same way as if the role came from a situation circle. The arrowhead will appear automatically.

If a place is delimited in certain dimensions for a situation, using an "in" clause, range selection, or the delimiting arrow just described, then that restriction will logically apply to the associated visitor in the child plan within that situation. This child plan can determine the range that any index of a visitor has been delimited to by drawing a **delimited arrow** from the hash mark denoting that index to the corner of another (non-interface) scalar place. The result is that the contents of the scalar place



will be initialized with the range to which the visitor was delimited at the time that the visitor is bound. In L23, a delimited arrow is drawn by selecting the role line from the palette and pulling the line out of the

place to get the range, connecting the end to the desired array hash mark. The arrowhead will appear automatically.

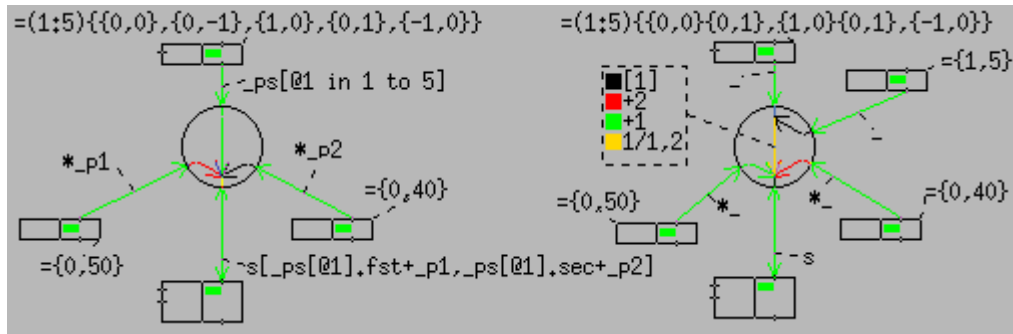
Replicating Situations for Data Parallelism

To replicate a situation and thereby the opportunities for plans to activate within it, use a **dup** construct. Dup is especially useful for creating one situation for every element of an array that needs to be processed. Not only can this be a more natural way of expressing a plan than artificially ordering the individual array elements to feed them sequentially through a single situation, it can also often make it possible to process many of the elements at a time if there are resources available to do so.

A dup is represented by prefixing an asterisk (*) or plus sign (+) to the name of an In scalar role which has one transition (named done). That role may contain a pair of integers denoting the low and high end of a range, or a single integer denoting the high end of a range, in which case the low end is assumed to be 1. If the high end of the range is less than the low end, a null range is assumed. If the role contains other data, the names of the one or two integer field(s) denoting the range must be specified in parentheses after the asterisk or plus. When a dup role becomes ready, it can effectively replicate the associated situation once for every integer in the range, creating so-called **replicant** situations. For each replicant, the dup role being acted upon is replaced with a green role of the same name (minus the * or + prefix), having a single red "done" transition, bound to a new (green) temporary place containing a unique one of the integers from the range. This role is often used as a primary role to perform indexing or translation to bind each replicant to different array elements on a secondary role, thereby binding each replicant to one or more different array elements.

At most one plan can activate within each replicant. The difference between an asterisk (*) and a plus sign (+) is that an asterisk (called a **dupall**) denotes that one plan will be allowed to activate in each replicant, while the plus (called a **dupany**) denotes that a plan will be allowed to activate in only one of the replicants, during the dup, which finishes when all of the activated plans finish. In practice, a dupall is often used as the primary for an array when all of the elements indexed by the range must be processed at the same general rate (in phases), and a dupany is used when the elements indexed by the range may be processed at different rates, perhaps because some dynamically-determined elements within the range are to be omitted from processing altogether.

Below is an example of how dupalls can be used in conjunction with binding modifiers to create an array of activities, each accessing different, perhaps overlapping, parts of an array place. Here, the goal is to apply a calculation (the activity) to each element of a rectangular array, where the calculation involves that element plus the four elements immediately adjacent to that element in a "+"-shaped stencil. Dupalls on the lower left and right roles replicate the center situation once for each column and row of the array, in this case $51 \times 41 = 255$ times, such that each replicant sees an integer between 0 and 50 inclusive denoting the column on its left role (`_p1` in type A) and 0 and 40 inclusive denoting the row on its right role (`_p2` in type A), each replicant with a different combination. All of the replicants see the same 5 pairs of numbers on their top role (`_ps`), indexed from 1 to 5, representing x and y coordinates of 5 points: The center (0,0), top (0,-1), right (1,0), bottom (0,1), and left (-1,0) of a "+"-shaped stencil centered at (0,0). Even though the array place at the bottom is 2-dimensional, each replicant sees it only as the role it plays, `s`, which in this case is effectively a 1-dimensional array of 5 elements, where the `n`th element contains the corresponding element of the "+"-stencil from bottom the array place. This is because the binding modifiers say to take any `n` used to access `s`, and to instead redirect it to denote one of the coordinate pairs from the top role (`_ps`), then to offset those x and y coordinates with the integers obtained on the left and right roles (`_p1` and `_p2`), respectively, to find the element of the bottom array to act upon. The binding modifiers, as well as all of the (anonymous) roles other than `s`, are invisible to activities running within the replicants, so they have no perception of accessing anything other than a 1-dimensional 5-element array as `s`.



When at least one of the roles for a situation is a dup role, there are several options for how each other roles on the situation should behave, such as whether they act on the dup itself or the roles also replicate to act on each of the resulting replicants. In simple cases, such as when there is only one dup role, the answers can be inferred through common sense or specified with prefixes on the role names. For example, a role will be replicated for each replicant if it is not anonymous (i.e. it has a name not beginning with an underscore), since having a name suggests that the role has meaning to the activity in the replicants. A role will also be replicated if it is secondary to a primary dup role, since that suggests that the role will be bound to a different place (array element) for each replicant. A role will also be replicated if the role name (starting with an underscore) is prefixed with an at sign (@). In all other cases, such as with plain anonymous roles, the role will belong to the dup itself, and will thus keep the dup from replicating the situation until the role is ready. Usually, such a role will issue a transition as soon as the dup replicates, but if the role is prefixed with an ampersand (&), the transition will be delayed until any or all of the replicants have activated, for dupany and dupall respectively. This is also true for the dup role itself: It will usually issue a transition as soon as the dup replicates, but adding an ampersand after the + or * will delay the transition until any/all have individually performed the transition to the role.

In more complex cases, such as when there are multiple dups and different roles must act on different ones, or when the order of evaluation of the dups is important because they have differing types (e.g. a dupall and a dupand, or a dupand with an ampersand and one without), the planner can be more explicit by adding concentric circles to the outside of the situation, up to one for each dup. The circles are evaluated from outer to inner. Each role line is bound to the circle representing the dupall or dupany to which it applies, or to the inner situation circle if it is to be replicated to apply to each replicant. This obviates the need for the at sign prefix, but the ampersand prefix is still used to specify when a transition is to be delayed until all replicants activate.

When a dupall or dupany is specified for more than one role on a situation, they effectively multiply. As long as all are of the same kind (dupall or dupany), their rules of behavior are simple and unambiguous, regardless of the order in which they are interpreted—i.e. any of any is still any, all of all is still all. Dupands are not commutative, so the outcome depends on the order in which they are applied. If dupalls and/or dupanys are combined on a single situation with at most one dupand, then the dupand replicates first, followed by the dupalls replicating those replicants, followed by the dupanys replicating those. This precedence can be effectively overridden (i.e. so that the situation will finish when all of the dupall replicants finish for one element in the dupany range) by using the nested circle notation in the previous paragraph, or by further nesting of strategies and applying one dup at each level of nesting.

It is common for a dup to be used on an anonymous role containing the entire index range of an array, and for that role to be used as a primary role to bind each replicant to a unique array element on a secondary role, so a shorthand ("syntactic sugar") exists for this case. Specifically, if a role is bound to an array that has its dimensions defined by delimiting arrows and the name of the role is prefixed by an at sign (@), then that is equivalent to removing the at sign and treating the role as a scalar secondary role, with the primary being a new anonymous primary dupall role from the place holding the range (i.e. bound to the delimiting arrow). As a result, the situation is replicated once for every element in the array with the role bound to one element for each replicant. Prefixing a role with a question mark (?) has the same effect except that a dupany is implied, instead of a dupall. In fact, the question mark can be used even when the array is not delimited, in which case situations are effectively bound to all infinity elements of the array, though only

one will activate before the dupany completes (as usual). To allow similar functionality with an explicit dupany, a place initialized to the special constant **\$infrange** can be used for the primary.

Examples

Matrix Multiply

To illustrate some features of ScalPL, three related plans (strategies) are shown:

- `fmatmult`, which multiplies two two-dimensional arrays and produces a two-dimensional array as a result, such that element i,j in the result is defined to be the dot product of the i th row of the first with the j th column of the second (under the constraint that the number of columns in the first must equal the number of rows in the second).
- `fdotprod`, which performs a dot product. A dot product is defined on two equal-length vectors as the sum of the products of corresponding elements. For example, if one vector is (1, 6, 2, 4) and the other is (3, 5, 8, 7), the dot product would be $1*3+6*5+2*8+4*7=3+30+16+28=77$.
- `ac_red`, which performs a generic scalar reduction of a vector using any associative and commutative scalar operation. Examples of associative and commutative scalar operations are addition, multiplication, `min`, `max`, `and`, and `or`. A scalar reduction of the vector (3,6,9,2) with the multiplication operator would be $3 * 6 * 9 * 2 = 324$.

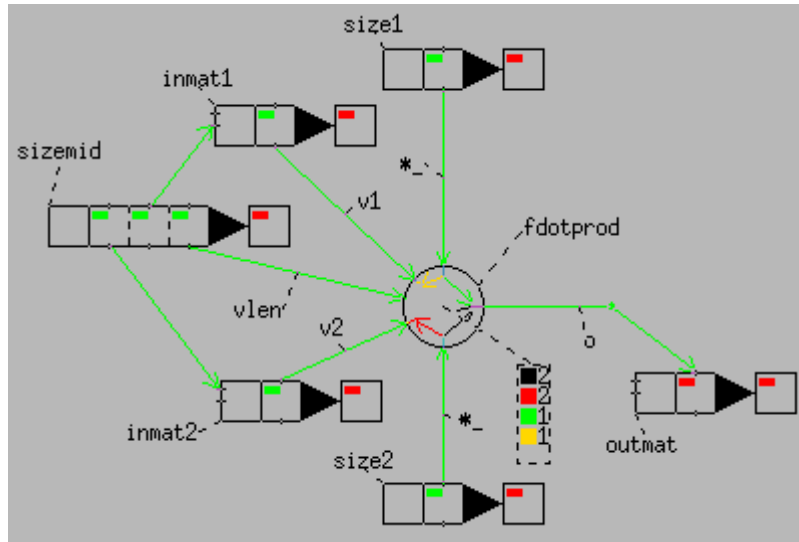
While most ScalPL plans do not refine plans all the way down to single mathematical operations (like scalar add and multiply) as these do, and one would expect to find plans like matrix multiply in a standard plan library, they have been chosen here because they are familiar and illustrate a number of issues. For example, a general problem with expressing parallel algorithms (such as these) is that the optimal way to execute them depends highly on the target architecture and/or on the size and shape of the inputs. In the context of the particular operations shown here, if just one very long dot product is being performed, then one might concentrate on parallelizing the sum reduction of all those products—e.g. by breaking the long vectors down into shorter ones and having each processor work on a shorter one, then adding them all together—but if many different dot products are being performed (as in a large matrix multiplication), then the individual products can be distributed among different processors. If special hardware is available for vector pipelined computations, then it can be used to perform the multiplications first, with the additions performed as a separate operation.

Rather than asking the programmer to recode the program for every target architecture or data characteristic, some languages offer higher level data structures with operators that do not micromanage the lower-level operations, and may provide a mechanism for the user to hint or suggest how to optimize them. For example, Fortran90 provides array operations, so the above programs could be expressed directly in those languages, and HPF additionally provides "structured comments" that allow the programmer to say how the rows and columns of those arrays should be distributed among processors, which in turn implies which processor will be used to perform operations. Even so, for any particular language, that approach only solves the issue for those particular abstractions which the language designers have chosen to include as primitives, and even then, only in those cases and for those platforms which the compiler (and runtime) implementers have spent the time and effort to optimize. Hinting and mapping mechanisms, like those used in HPF, are also restricted in their expressiveness (i.e. using the "owner computes" rule).

ScalPL provides a much more general approach to solving such problems which does not depend on language designers and implementers to be steps in front of any particular programmer's wishes. The general approach is for the programmer to first state the algorithm in the most general and least restrictive terms possible, by specifying the operations that must be performed to each input to produce outputs while excluding mention of the order in which operations must execute relative to one another except for cases where one operation directly depends upon the other. The result of this step is an executable algorithm, or program, which can run on a number of platforms, but it may not be obvious to a runtime system how to execute it optimally on some or all. After the initial programming task, ScalPL then allows the programmer to offer much more help in suggesting how it might be executed on a particular platform using mapping and specialization.

fmatmult

One simple implementation of fmatmult (the matrix multiplication plan) in terms of fdotprod (the dot product plan) is shown here. Simply put, this plan declares that each element i, j of the output matrix is defined as the dot product of row i of the first input matrix and column j of the second input matrix.



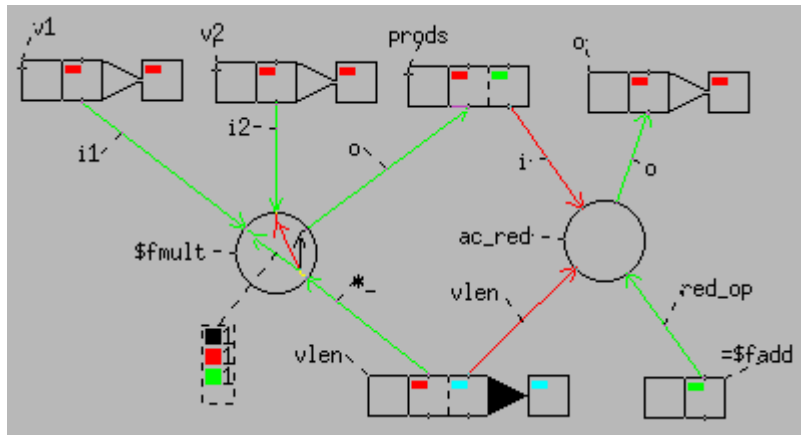
This plan is a strategy as opposed to a task, as indicated by its graphical schematic-like representation. It has 6 visitors, analogous to arguments: `inmat1` and `inmat2` are the input matrices, `outmat` is the result matrix, `size1` is the number of rows in `inmat1` and `outmat`, `size2` is the number of columns in `inmat2` and `outmat`, and `sizemid` is the other dimension of both input arrays. (ScalPL arrays don't really have rows and columns: The terms are used here only by convention to mean the first and second dimensions of the arrays, respectively.) `inmat1`, `inmat2`, and `outmat` are shown to be two-dimensional arrays by the two hash marks along their left edges. The size of `inmat1` in its second dimension, and of `inmat2` in its first dimension, is given by the range in `sizemid`, as indicated by the delimiting arrow from `sizemid` to the corresponding dimension hash marks on the others. The plan works by replicating the middle situation once for every element in the output array—i.e. once for every combination of values from the range on the `size1` place and the range on the `size2` place. This "dupall" replication is represented by the asterisk prefixing the label on the roles (lines) leading from those places to the situation. Each resultant situation replicant sees a unique combination of numbers on those roles from within those two ranges, but because those roles are anonymous (labels starting with underscore, after the asterisk), the plan within the replicants won't see them at all—they are used here only as primaries for the binding modifiers, shown by the arrows within the situation circle. The situation's old-style binding modifiers (shown by the the legend to its upper right) show that the value from the `size1` range for each replicant will be used to index the first dimension of both the `inmat1` array and the `outmat` array, while the value from the `size2` range for each replicant will be used to index the second dimension of both the `inmat2` array and the `outmat` array.

The result, then, is that each replicant will contain the `fdotprod` plan (as per the label), and for each, a single row of `inmat1` will play its `v1` role, a single column of `inmat2` will play its `v2` role, and the one element of `outmat` in the same row and column will play its `o` role. The `sizemid` place will play the `vlen` role for each replicant. The `fdotprod` plan, described below, will have no awareness of rows and columns: It knows only that it is getting two input vectors on its `v1` and `v2` roles and a length on its `vlen` role, and that it is producing a scalar on its `o` role.

As per the role colors, each situation replicant containing `fdotprod` can only access the places when they are in their initial green stage. The transition tables adjoining where the roles connect to the place rectangles show that the `inmat1` and `inmat2` elements will remain green, as will `vlen`, but the `outmat` elements will turn red as each is produced. `size1` and `size2` will turn red after plans have been activated in all of the situation replicants.

Every visitor here is marked as initial, meaning that the `fmatmult` strategy won't activate until all of the roles on its parent situation are ready, and they will not be under contention with the parent when they are green. Although the stage of the `inmat1`, `inmat2`, and `sizeid` visitors never change to the red stage of their goodbye dots, the parent strategy can assume that they will be bid goodbye nonetheless because they are predictable (i.e. not written to, and with only one posting dot).

fdotprod



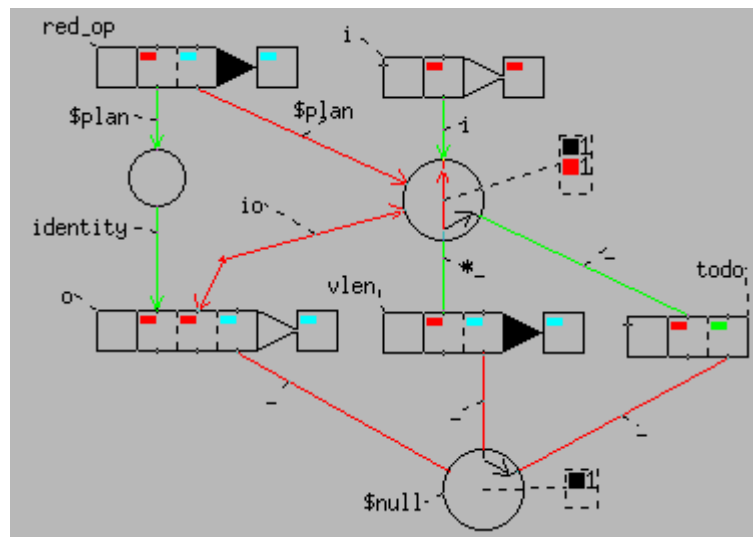
This next diagram represents the strategy named `fdotprod`, which is used above to perform a floating point dot product. The strategy has four visitors: `v1` and `v2` are the input vectors, `vlen` is the range of indices to be processed on those vectors, and `o` is the resulting scalar. `v1`, `v2`, and `prods`, are all one-dimensional arrays (vectors), as denoted by the single hashmark on their left side.

The situation (circle) on the left is replicated once for every number in the range found on `vlen`, as represented by the asterisk (dupall) prefix on the corresponding role's label. Each replicant performs the pairwise floating point multiplication (using the predefined constant plan `$fmult`) of corresponding elements from `v1` and `v2` into the corresponding element of `prods`, as shown by the three binding modifier arrows within the situation using the single integer from the dupall as an index into all three, but since it is again anonymous (starting with an underscore), it will not be seen by the `$fmult` plan in the replicated situation. As a result, each `$fmult` activity operates in a very simple situation where it gets one floating point number on its `i1` receptacle, another on its `i2` receptacle, and produces a result on its `o` receptacle. The `$fmult` activities are oblivious to the fact that array bindings are used.

The situation on the right adds the resulting floating point products from `prods` together, in no particular order, producing the output place, `o`. The plan in that situation, `ac_red`, is defined below, and can perform any associative and commutative scalar reduction which it finds on its `red_op` receptacle. In this case that operation is a floating point add (`$fadd`). `ac_red` finds the input elements for the reduction in the array playing its `i` role (in this case the `prods` vector), in the range found in the place playing its `vlen` role (in this case the `vlen` place), producing a result on the place playing its `o` role (here the `o` place).

ac_red

Here's one possible implementation of the `ac_red` (associative commutative reduction) plan:



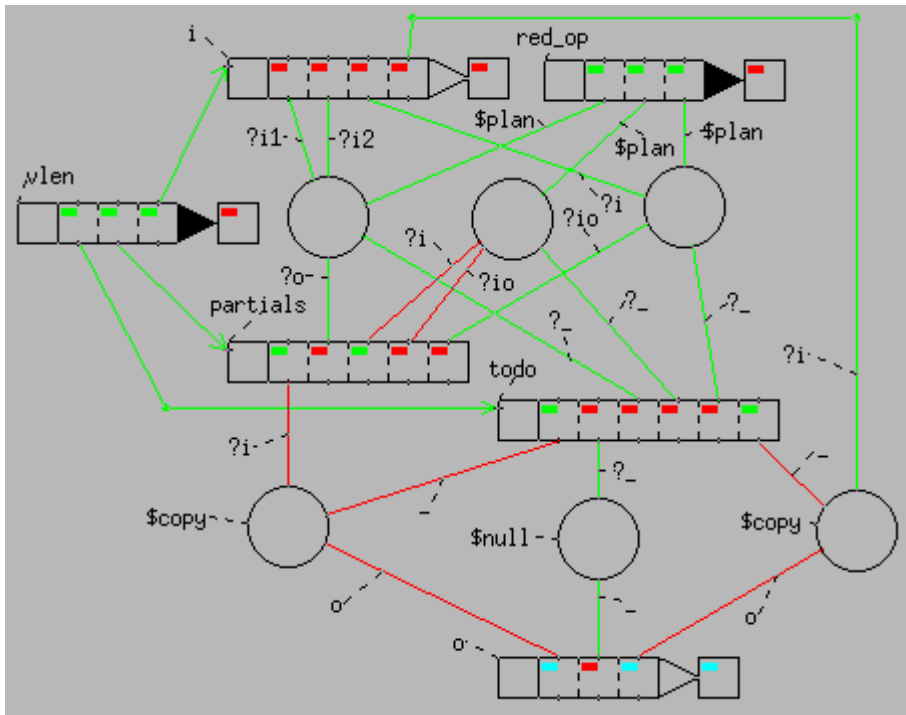
The situation on the left will necessarily be the first one to activate. Instead of being labeled with a plan name, it uses the `$plan` role played by the `red_op` visitor to find the plan to activate in the situation (in this case, the reduction operator provided by the parent), after which that place will transition to the red stage as per the associated transition dot. The reduction operator is assumed here to produce its identity (i.e. the value which can be applied to any other number by the operator to produce the number itself) to the place (o) playing its `identity` role. When the stage of `red_op` and o transition to red, the situation on the top right can replicate (as per the dupall role) and the replicants can execute, in no particular order, performing the reduction operator for every input element (on `i`) to o, while also transitioning the stage of the corresponding elements of `todo` to red. Note that the situation on the right binds different roles of the operation than the left situation did. When all of the elements of `todo` (within the range on `vlen`) have turned red, the situation at the bottom, holding the predefined do-nothing plan `$null`, can execute, having the primary function of bidding goodbye to the o place, thus returning the answer to the parent strategy and signifying completion. Note that the legend for the bottom situation shows that the traditional binding modifier is a dimension number ("1") in braces ("[.]"), meaning that the range on place `vlen` is used to restrict the elements of `todo` to which the situation is bound to those in the first dimension within that range. An "in" clause could have instead been used on the role bound to `todo`.

Matrix Multiply Variations

The three examples given work well in general, and are easy to understand, but there are some potential tradeoffs. This section will explore alternate implementations for `ac_red` and `fmatmult` which have some different properties.

ac_red revisited

The implementation for `ac_red` given above is unsatisfying in that it contains no parallelism! That is, even though the replicated situations containing the reduction operator can execute in any order, only one can execute at one time, due to the fact that they all write to the same place, o. The implementation also relies on an identity initialization and an initial reduction with that identity that are not strictly necessary. At the expense of some additional complexity, the following implementation addresses both of these issues:



Like the previous implementation, the top part of the strategy takes care of the reduction, the `todo` place keeps track of when the reductions are done, and the bottom part handles the finishing up, but now, instead of reducing all of the input into a single scalar place (`o`), a vector of partial results is kept (`partials`). The additional situations provide as few restrictions for this as possible. The question mark (?) prefix is used on many of the roles here, as shorthand for the dupany replication, to mean that that role can be bound to any of the elements of the associated place, so it can only be used for arrays with a delimiting arrow. If there are two roles from the same situation to the same place, they will not both be bound to the same element.

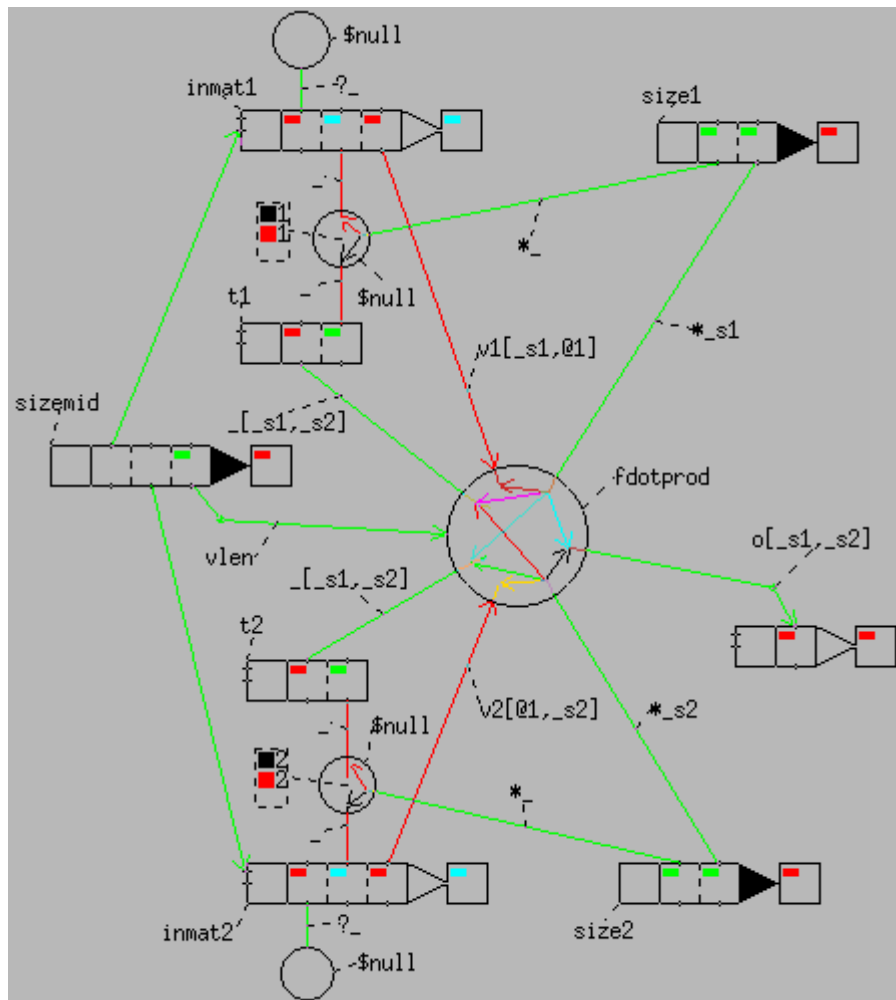
The three situations along the top each perform a single scalar reduction, but each in different ways. The first reduces any two elements of `i` (the input), playing its `i1` and `i2` roles, to produce a partial result into any element of the `partials` vector, playing its `o` role. The second reduces any one partial result playing its `i` role into any other playing its `io` role. The third reduces any one input element playing its `i` role into any partial result playing its `io` role. Stages of the `partials` elements are green when unused (or emptied), and red when they hold a valid partial value. These three situations can be performed in any order or in parallel. (In fact, assuming that the reduction `op` is atomic, which it will usually be, any one of these may execute concurrently with another instance of itself.) The strategy's `red_op` visitor plays the `$plan` role for each of these situations, and each binds only the roles it needs.

This implementation performs one fewer reduction operation than the original strategy, so one element of the `todo` vector must be transitioned to the red stage without a reduction. This is performed by the middle situation on the bottom holding the `$null` plan. The other two situations at the bottom (both containing a `$copy` task) can only activate when `todo` is all red, and one or the other is used to copy the answer to the output place, `o`. There are two of them because the answer could be in two different places: It will usually be in `partials`, as the only red element there, but in the special case where there is only one input element to the strategy, no reductions need to occur, so the answer will still be in the only (still green) element in `i`. Usually, the `$copy` task can play tricks to avoid really copy anything: For example, in this case, it could probably just internally rename the place holding the result to "o".

fmatmult revisited

The original matrix multiply strategy was completely atomic—i.e. all of its visitors were initial. This has both potential advantages and disadvantages. For advantages, all of the input roles (`size1`, `size2`, `inmat1`, and `inmat2`) are predictable, so the parent can immediately and accurately know the outcome for these roles the moment the strategy activates, and can predictively perform other actions based on that knowledge. A disadvantage (shared by solutions in many other parallel programming paradigms) is that the plan will not activate until all of the input elements are ready and all of the output elements are ready to accept the results. In some cases, this could result in the plan sitting idle waiting for one or two input or output elements to become ready even when there is plenty of work to do and plenty of compute resources available to support it.

For applications where those sorts of cases are likely, we can consider an alternate matrix multiply in a more streaming form, which is capable of performing any dot product as soon as its own unique input and output elements are ready, instead of waiting for all of them first. The dot product plan already works in a streaming fashion, allowing any single multiplication or addition to be performed as soon as possible, so that property can now extend to the entire matrix multiply:



The only differences between this and the original implementation is that the input and output matrices are no longer initial, and two place arrays (`t1` and `t2`) and four situations containing `$null` tasks have been added. These changes are necessary to record when the strategy is finished with each input element, and to bid goodbye to each at that time. Specifically, a row of `inmat1` can be bid goodbye when the associated

row of o has been computed, and a column of $inmat2$ can be bid goodbye when the associated column of o has been computed. However, since the elements of o are bid goodbye just as soon as they are calculated, and since some of those may even become available for the next iteration/matrix before this iteration/matrix has finished, the stages of o cannot be used to keep track of what has been done in this iteration. To make up for that, the stages of elements in local arrays $t1$ and $t2$ are used to record the completion of each element of o , and a $\$null$ task between $t1$ and $inmat1$, and between $t2$ and $inmat2$, activates when an entire output row or column have been calculated, bidding goodbye to the appropriate input row or column and resetting the stage of that row or column in $t1$ or $t2$ for the next iteration/matrix.

The other two $\$null$ situations (at the top and bottom) are solely to transition the stage of the input elements from green (Available) to red (Present): The input visitors are no longer initial so they are no longer automatically Present, and $v1$ and $v2$ can't be used to effect this transition because each will need to access each element multiple times and this transition can only be performed once for each element.

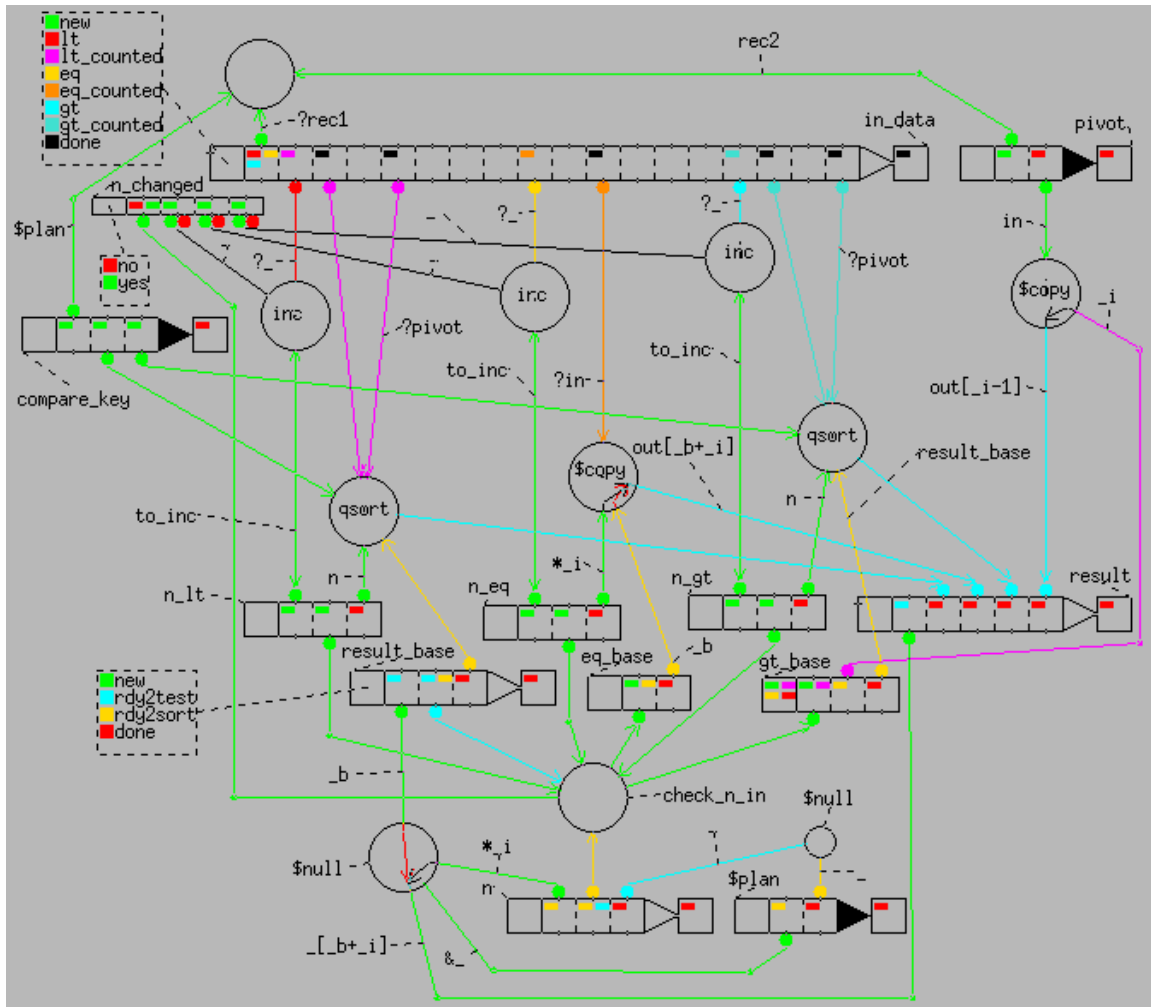
Note that the three size visitors in this plan are still initial, so they will be supplied just once and used throughout the lifetime of the plan, regardless of how many input matrices are supplied and output matrices are produced.

Matrix Multiply Optimization

The program, as specified, will run correctly on many different platforms, and there is a tremendous amount of parallelism to be exploited. For example, when the **fmult** fragment is executed, all of the **size1 * size2** dot products can execute in parallel, and for each of those, all of the multiplies (within **fdotprod**) can execute in parallel, and many of the additions (within the revised **ac_red**) can also occur in parallel. The problem is that the parallelism can only be exploited when potentially concurrent operations are on different processors at the same time, and it may not be obvious to a runtime system how to make that happen, especially since transferring data to a processor (so that an operation can execute) takes time in itself. The overhead of these operations can often be reduced by pre-scheduling operations on the processors (i.e. to determine which processors which will be free to perform each operation in advance, rather than sensing and deciding it at runtime), and also by transferring several data items at one time when the source and destination are the same for all. In some cases, the amount of data to be transferred (and therefore overhead) can be reduced by good scheduling: e.g. in the program under consideration, if each processor adds up all of the local products into a partial sum before summing those partial sums across processors into an overall sum, then at most only one item per processor needs to be transferred to accomplish the overall sum. The availability of special hardware can also affect scheduling decisions. For example, pipelined vector processors may reduce the time necessary for a single processor to perform many different scalar multiplications, thereby reducing the importance of subdividing a vector across processors to achieve parallelism, and in turn reducing the number of communications required to achieve the sum reduction. On the other hand, combining networks may be able to reduce the cost of additions between processors.

The general question is: How does one take these sorts of factors into account while still keeping the program portable—in fact, without changing the program at all, in some sense? ScalPL uses two approaches: Mapping and specialization. Mapping means providing instructions to the scheduler about how the operations should be assigned to processors. Specialization means providing instructions about which operations should not execute: it is used to help a scheduler make decisions when it has many different choices (such as in the top half of the revised **ac_red** strategy). Mapping cannot change the possible behavior of the program, only the performance. For specialization, care must be taken to ensure it does not cause some portion of the computation to stop before the original program says it is done (thereby violating either the liveness semantics of the model, or the semantics of the program). More specifics on mapping and specialization are beyond the scope of this document.

Quicksort



Specification

The quicksort plan (to be named "qsort") has 6 roles, described here textually:

in in_data ^ 1 (done): R	Input records (of type R)
in n (done): int	# of input records to process (including pivot)
in! pivot (done): R	Input record to treat as initial pivot (may be a member of in_data)
in! compare_key (done): plan {in rec1 (lt, eq, gt): R; in rec2 (done): R}	Input plan for comparing keys and deciding if the one in rec1 is less than (lt), equal to (eq), or greater than (gt) that in rec2
out result ^ 1 (done): R	Ordered output records (of type R)
in result_base (done): int	Index into result preceding first output record

The in_data and result roles are one-dimensional arrays, as shown by the "[^] 1" qualifiers. None of the input data to this plan, including the input records in in_data, is modified by this plan, as shown by the **in** usage qualifiers. Instead, the sorted list is written to elements result_base + 1 through result_base + in_count of result. As indicated by the "initial" qualifiers indicated by bang (!), the plan itself will not activate until pivot and compare_key are ready. All of the roles have just one

transition, "done", shown in parens, though the plan that plays the `compare_key` role has three transitions (`lt`, `eq`, and `gt`) on one of its roles (`rec1`). Although `in_data`, `pivot`, `result`, and the roles of `compare_key` all have contents of some type described here as `R`, there is no need to actually specify `R`.

The input records need not be within any particular index range within `in_data`, and need not be contiguous. Processing of input records may begin before `n` is ready, and regardless, it is important to not allow more than `n` elements of `in_data` to become ready, such as for subsequent sort iterations, until the current plan finishes, as indicated by a `done` transition to its `n` role.

High-Level Implementation Description

The activity at the top compares `pivot` against each element of `in_data` and transitions that element's stage based on the outcome: to `lt` (red) for less than the pivot; to `eq` (yellow) for equal to the pivot; or to `gt` (blue) for greater than the pivot. The three `inc` activities shown below `in_data` tally the number of elements with each of those stages into places `n_lt`, `n_eq`, and `n_gt`, respectively, while again transitioning the input element stages, to `lt_counted`, `eq_counted`, and `gt_counted`, respectively. Those with stage `lt_counted` or `gt_counted` are sorted again (recursively) by the `qsort` activities shown at the center left and right, while those with stage `eq_counted` will be moved directly to the output by the center `$copy` activity. Each time an `inc` activates, it changes the stage of `n_changed` in the upper left to `yes` (green) if it is not already. This makes it so `check_n_in` (at the bottom center) will not activate unless a tally has changed since it last activated, and unless its `n` and `result_base` roles are ready.

Each time `check_n_in` activates, which may be after each `inc` activity or only after several, it checks to see if all input has been tallied by ensuring that the total of the three tallies (`n_lt`, `n_eq`, and `n_gt`) is at least `n-1`. If not, the activity just resets the stage of `n_changed` back to `no` (red) to suppress re-activation at least until one of the three tallies changes again. If the sum *is* at least `n-1`, `check_n_in` computes where the elements equal to the pivot, and those greater than the pivot, are to end up in `result`, as `eq_base = result_base + n_lt`, and `gt_base = result_base + n_lt + n_eq + 1`, respectively. It then transitions the stage of the three bases to yellow or magenta, thereby allowing the previously-mentioned recursive `qsort` and `$copy` activities to complete, as well as the `$copy` on the right which moves the `pivot` to the `result`.

The only purpose of the `$null` activity at the bottom left is to ensure that the elements of `result` that will hold the output will be present even after the plan as a whole finishes. To do this, the activity replicates just as soon as both `n` and `result_base` are ready, and each replicant eventually changes one element in the range to be present (in this case, blue) as it becomes visible (green). The `$null` activity at the bottom right can finally finish the plan when these have all activated (as indicated by the fact that the stage of `$plan` has been changed to yellow by the `&` role on the left `$null` activity), as long as `n` elements of `in_data` are also present (as indicated by the fact that the stage of `n` has been changed to blue by `check_n_in`). There are simpler ways to pull in the elements of `result` before finishing than to use two `$null` activities as described here, e.g. by simply delaying finishing until all results have been produced, or not finishing at all, but these other approaches will generally not allow as much concurrency, either within the plan or between the plan and the parent which feeds it input and consumes its output.

There is nothing to prevent `checktotal` from running far less frequently than the `inc` plans, since the `inc` plans can activate whether or not `modflag` has the `recheck` stage.

Parallelism Analysis

Quicksort is usually considered to possess only moderate exploitable parallelism. This implementation has extreme parallelism. This is achieved, in part, by a specification which does not physically permute the input records, thereby allowing all data to be easily shared by parallel activities throughout the execution. It

is also enabled, in part, by the non-deterministic choice of pivot elements. Though this would be considered suboptimal for traditional quicksort, here it also allows a new pivot element to be chosen for a set of data (i.e. the set less than or greater than the previous pivot element) even if the set has not been fully defined yet.

The result is that all of the comparisons of all of the input records against the pivot element can be performed in parallel, the choosing of the two new pivot elements can occur in parallel, the indices of the input elements equal to the current pivot can be copied to the output array in parallel (once the input records have been tallied), and elements not equal to the current pivot can be passed immediately to the next recursive level of qsort to be processed even further in parallel (even before the current level has tallied all of the inputs, or before it even knows how many there should be). Also, up to three tally operations at each level can be performed in parallel (one lt, one eq, and one gt the pivot), and future optimizations could even increase that parallelism (by recognizing the permutativity of the tally operation).

Consider how each input element progresses through the different comparisons at the different recursive iterations to produce the output. Each iteration first selects one incoming element as a pivot, and then compares each other incoming element against its pivot, gathering those equal to it, or immediately sending those less/greater than it to the next iteration which either selects it as a pivot or compares it to its pivot which is respectively less/greater than the previous level's pivot. As many recursive iterations become active all at once, and pivots are chosen for each, the recursion tree exactly becomes a binary comparison tree, with each incoming element either sticking at the node if it compares equal to that pivot, or going down the "left" or "right" branch if it is less or greater than the pivot. When considered in this time/space form, this is perhaps as generic as a sort can get, simply comparing each input element with as many (and only as many) previous elements as necessary to determine where it belongs relative to those elements. Sequential sorts are required to temporally fold this tree up in different ways.

Finer Points

The top three situations read their plans from places (`in_extkey` and `in_compkey`). In the latter case, the plan descriptor with the corresponding name is found in the program (i.e. on the program data base) and inserted into the situation. Examples of these are `qsrt`, `inc`, and `checktotal`. The `qsrt` plan descriptor is this strategy itself, and possible implementations of `inc` and `checktotal` are shown below. The other two plans, `$null` and `$copy`, are part of a standard library supplied by ScalPL: `$null` does nothing (often used to just change the color of a place), and `$copy` copies whatever is on its `in` pin to its `out` pin.

A usable implementation of `inc` consists of a simple interface declaration plus one line of C:

```
{inout! count (done): int}
{(*count)++;}
```

That is, it has one pin, called `count`, of type integer, and it increments the integer present there whenever it fires. The `done` signal is posted by default when it finishes.

An implementation of `checktotal`, in C, is:

```
{ in! in_count (done): int;
  in! n_lt_pvt (sum_eq, sum_neq): int;
  in! n_eq_pvt (sum_eq, sum_neq): int;
  in! n_gt_pvt (sum_eq, sum_neq): int;
}
{int total = *n_lt_pvt + *n_eq_pvt + *n_gt_pvt;
  if (total == *in_count) {
    SCpost($n_lt_pvt, sum_eq)
    SCpost($n_eq_pvt, sum_eq)
    SCpost($n_gt_pvt, sum_eq)
  } else {
```

```

        SCpost($n_lt_pvt, sum_neq)
        SCpost($n_eq_pvt, sum_neq)
        SCpost($n_gt_pvt, sum_neq)
    }
}

```

That is, if the total of the three pivots equals `in_count`, the `sum_eq` signal is posted to each, else `sum_neq` is posted to them.

Roles (lines) which are not labeled adopt the label of the place to which they are bound. Roles with a label beginning with underscore (`_`), optionally preceded with a modifier ("`+`", "`*`", or "`@`"), are "anonymous". That is, they are invisible to the plan within the situation, and serve only to keep that plan from firing (if the color doesn't match the place to which it is bound) and/or to record the firing of that plan (by posting a single signal to the place to which it is bound). The "`+`" and "`*`" modifiers will be explained below, but for now, the "`@`" modifier is "dup each", and can be used on any role bound to an array to mean that the situation is replicated and each one is bound to just one element of that array. In the `qsort` program, for example,

A role can fork, with each branch bound to different places, or to the same place with different role or signal colors. An example in this program is where the three `inc` situations are bound to the `pvt_count_age` place. This is technically just a shorthand for replicating those situations that many times (i.e. 2 each), identical in their bindings except that each replicated situation adopts the role binding from just one of the forks. The overall effect is for such forked roles to represent a logical "or". For example, the `inc` plans in the `qsort` program can fire if the `pvt_count_age` place is either green or red, and in either case, they change the color (through the anonymous receptacle) to red.

Conclusion

This document has covered the constructs of ScalPL. The motivation behind the precise design of these constructs has, for the most part, been omitted, in part because much of it is best described in terms of portability, concurrency theory, or other factors that are unnecessary for the casual user of the methodology to understand. In fact, one could say that ScalPL was designed precisely so that people with little understanding of these underpinnings could nonetheless benefit from the understanding of experts in this field.

Also not covered here is a very large amount of work that has been performed by Elepar on designing and implementing tools to make ScalPL practical. This includes methods of implementing runtime systems on a variety of architectures, investigating user interface issues for building, debugging, and maintaining ScalPL plans, developing extensions and annotations for use in optimizing ScalPL plans and applying them to real-time environments, and establishing the theoretical underpinnings required to formally define behavior, to statically analyze plans for certain characteristics, and to optimally trace execution and recover from failures. Information on this work may be obtained through elepar.com or the author.

Acknowledgements

ScalPL has been developed over many years, primarily by the author, but the contributions of many others has helped to keep this work going. Work by Robert Babb II on Large Grain Data Flow (LGDF) in the 1980s and early 1990s served as an early seed for this work, and many others at Oregon Graduate Institute provided helpful guidance during those early years, most especially the committee reviewing the author's 1991 dissertation describing some of the theoretical underpinnings. More recently, the board of advisors for Elepar.com, consisting primarily of Robert (Buzz) Hill, Bob Dietrich, Pat Cauduro, Polly Jenkins, and Miki Tokola, has offered significant technical aid, feedback, and advice. However, errors or omissions in this specific document are the sole responsibility of the author, and should not reflect poorly on these other advisors, especially since this early draft is being released prior to incorporating much of their most recent feedback in the interest of timely dissemination of this technology. The author also wishes to thank those who have sustained him with food, shelter, entertainment, kindness, and patience while preparing this document.

Alphabetical Index

\$copy.....	18	instantiation.....	20
\$iface.....	26	instantiation levels.....	21
\$infrange.....	36	instantiator.....	20
\$null.....	18	interface.....	25
\$oplan.....	20	multi-buffering.....	16
\$plan.....	13, 16	multiple readers.....	16
Absent.....	10	Noaccess.....	12
activate.....	12	object.....	20
activation conditions.....	17	Out.....	7, 12
activities.....	7	overloading.....	15
anonymous roles.....	17	parent.....	12
arguments.....	14	path.....	25
arrays.....	30	permission.....	7
BCE.....	14	places.....	7
binding.....	20	predictable role.....	15
binding constraint expression.....	14	Present.....	10
binding modifier.....	30	primary role.....	31
bottom.....	18	ready.....	8
bundling.....	25	replicant.....	34
child.....	12	resolved.....	26
class.....	20	roles.....	7
class variables.....	22	roleset.....	25
contents.....	7	scalar.....	30
copy-on-write.....	16	secondary role.....	31
delegation.....	24	serializability.....	18
delimited arrow.....	33	situations.....	7
delimiting arrow.....	33	splice.....	25
dup.....	34	stage legend.....	8
dupall.....	34	stage name.....	8
dupany.....	34	stages.....	8
fork.....	17	strategy.....	12
garbage collection.....	21	subclass.....	24
GC.....	21	super.....	26
goodbye box.....	9	superclass.....	24
goodbye dots.....	10	task.....	12
goodbye note.....	10	transition name.....	9
ILs.....	21	transition table.....	8
In 7, 12		usages.....	12
in clause.....	31	Visible.....	10
initial visitor.....	10	visiting place.....	9
Inout.....	7, 12	visitor.....	9
instantiatable.....	20	visitor set.....	25
instantiatable visitors.....	21	vset.....	25
instantiated.....	20	wildcard.....	25

Software Cabling Translations

Software Cabling term	ScalPL term
Module	Plan
Chip	Task
Board	Strategy
Receptacle	Role
Pin	Role
Wire (cable)	Role
(Complex) cable	Roleset
Receptacle set	Roleset
Socket	Situation
Memory	Place
Control State	Stage
Data State	Contents
I-memory	Visitor
Atomic i-memory	Initial Visitor
Signal	Transition
I-set	Vset
Signal Table	Transition table
Posting box	goodbye box
posting dot	goodbye dot
signal name	goodbye note