11 Technical Papers Every Programmer Should Read

These papers provide a breadth of information that is generally useful and interesting from the perspective of a computer programmer.

Contents

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon’s platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon’s software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon’s platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon’s platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon’s platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

ACID Properties: ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9th percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will

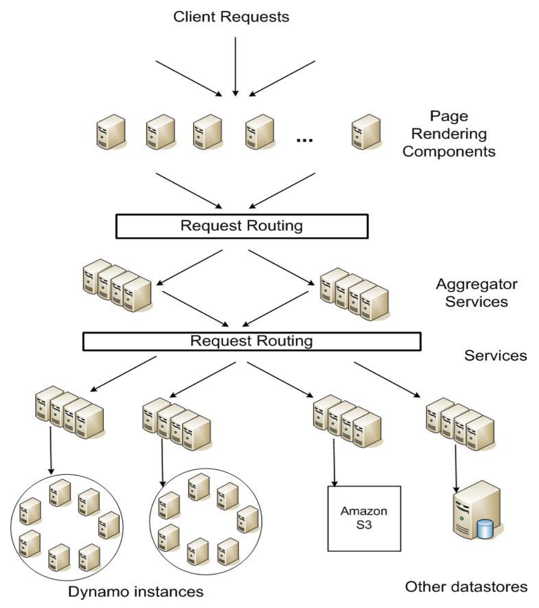


Figure 1: Service-oriented architecture of Amazon's platform

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9th percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9th percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node"*) at a time, with minimal impact on both operators of the system and the system itself.

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

3. RELATED WORK

3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

[1] http://freenetproject.org/, http://www.gnutella.org

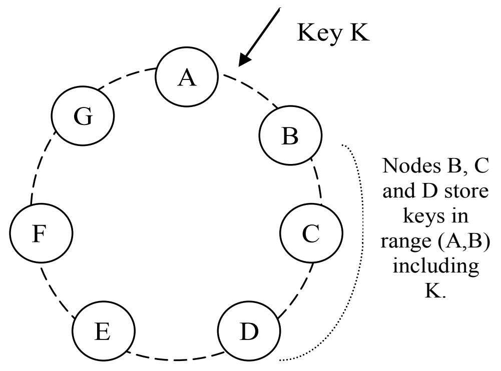


Figure 2: Partitioning and replication of keys in Dynamo ring.

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its N^{th} predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an *"Add to Cart"* operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if

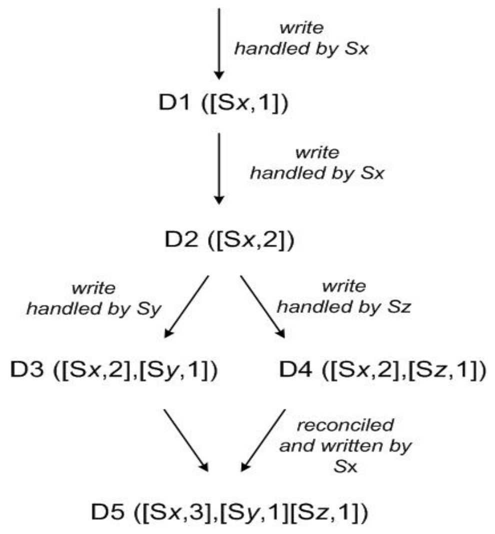


Figure 3: Version evolution of an object over time.

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that $R + W > N$ yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

4.8 Membership and Failure Detection

4.8.1 Ring Membership

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

4.8.2 External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

4.8.3 Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

[2] http://www.oracle.com/database/berkeley-db.html

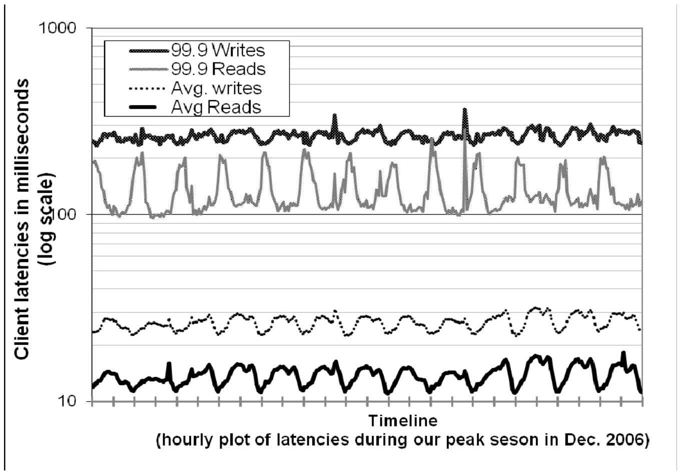


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

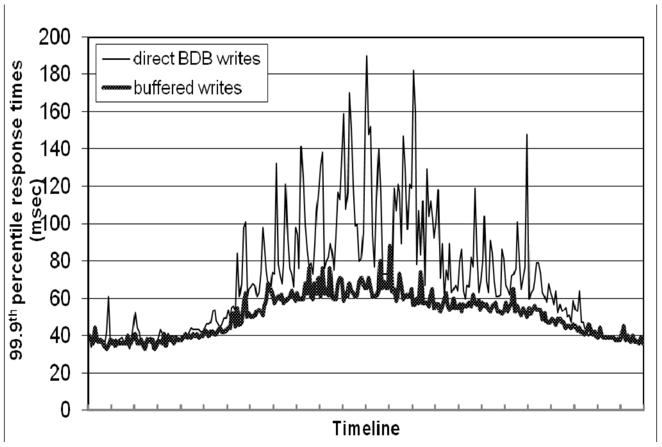


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

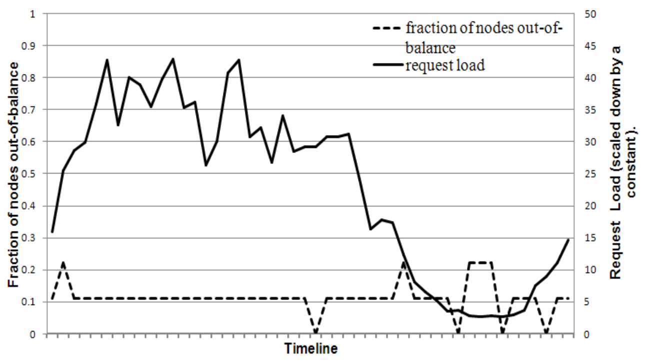


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9th or 99.99th percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9th percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9th percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9th percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9th percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8th

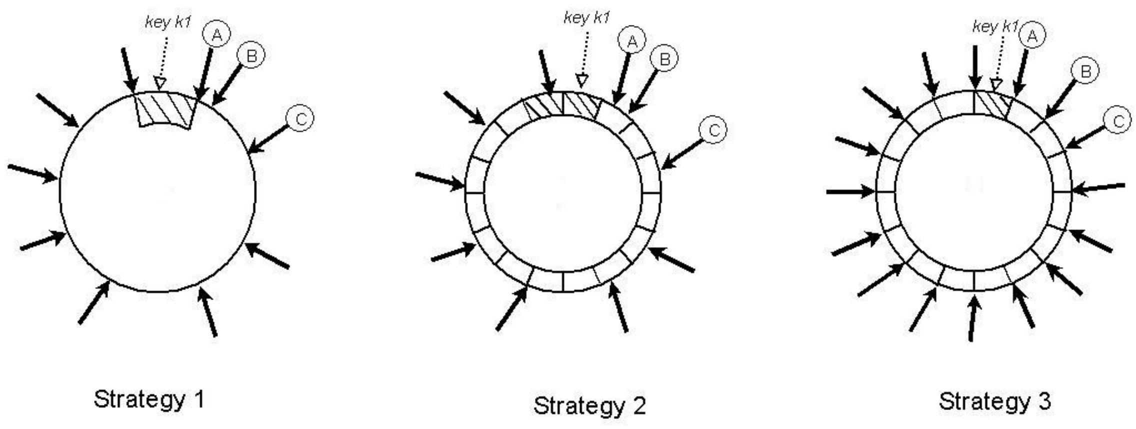


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

Strategy 1: T random tokens per node and partition by token value: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

Strategy 2: T random tokens per node and equal sized partitions: In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that Q >> N and Q >> S*T, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

Strategy 3: Q/S tokens per node, equal-sized partitions: Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to

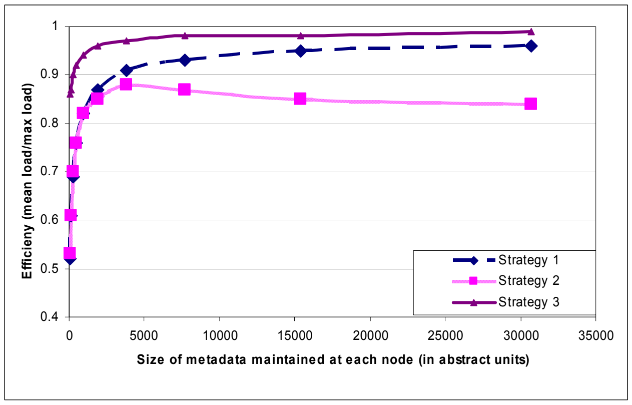


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9th percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9th percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9th percentile than the average.

6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database), shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99th percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., “Notes on Distributed Databases”, Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.

Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli
AT&T Bell Laboratories, Murray Hill, NJ 07974
(current address: DEC SRC, 130 Lytton Ave, Palo Alto CA 94301)

Peter Wegner
Dept. of Computer Science, Brown University
Providence, RI 02912

Abstract

Our objective is to understand the notion of *type* in programming languages, present a model of typed, polymorphic programming languages that reflects recent research in type theory, and examine the relevance of recent research to the design of practical programming languages.

Object-oriented languages provide both a framework and a motivation for exploring the interaction among the concepts of type, data abstraction, and polymorphism, since they extend the notion of type to data abstraction and since type inheritance is an important form of polymorphism. We develop a λ-calculus-based model for type systems that allows us to explore these interactions in a simple setting, unencumbered by complexities of production programming languages.

The evolution of languages from untyped universes to monomorphic and then polymorphic type systems is reviewed. Mechanisms for polymorphism such as overloading, coercion, subtyping, and parameterization are examined. A unifying framework for polymorphic type systems is developed in terms of the typed λ-calculus augmented to include binding of types by quantification as well as binding of values by abstraction.

The typed λ-calculus is augmented by universal quantification to model generic functions with type parameters, existential quantification and packaging (information hiding) to model abstract data types, and bounded quantification to model subtypes and type inheritance. In this way we obtain a simple and precise characterization of a powerful type system that includes abstract data types, parametric polymorphism, and multiple inheritance in a single consistent framework. The mechanisms for type checking for the augmented λ-calculus are discussed.

The augmented typed λ-calculus is used as a programming language for a variety of illustrative examples. We christen this language Fun because fun instead of λ is the functional abstraction keyword and because it is pleasant to deal with.

Fun is mathematically simple and can serve as a basis for the design and implementation of real programming languages with type facilities that are more powerful and expressive than those of existing programming languages. In particular, it provides a basis for the design of strongly typed object-oriented languages.

Contents

1. From Untyped to Typed Universes

1.1. Organizing Untyped Universes

Instead of asking the question *What is a type?* we ask why types are needed in programming languages. To answer this question we look at how types arise in several domains of computer science and mathematics. The road from untyped to typed universes has been followed many times in many different fields, and largely for the same reasons. Consider, for example, the following untyped universes:

(1) Bit strings in computer memory
(2) S-expressions in pure Lisp
(3) λ-expressions in the λ-calculus
(4) Sets in set theory

The most concrete of these is the universe of bit strings in computer memory. 'Untyped' actually means that there is only one type, and here the only type is the memory word, which is a bit string of fixed size. This universe is untyped because everything ultimately has to be represented as bit strings: characters, numbers, pointers, structured data, programs, etc. When looking at a piece of raw memory there is generally no way of telling what is being represented. The meaning of a piece of memory is critically determined by an external interpretation of its contents.

Lisp's S-expressions form another untyped universe, one which is usually built on top of the previous bit-string universe. Programs and data are not distinguished, and ultimately everything is an S-expression of some kind. Again, we have only one type (S-expressions), although this is somewhat more structured (atoms and cons-cells can be distinguished) and has better properties than bit strings.

In the λ-calculus, everything is (or is meant to represent) a function. Numbers, data structures and even bit strings can be represented by appropriate functions. Yet there is only one type: the type of functions from values to values, where all the values are themselves functions of the same type.

In set theory, everything is either an element or a set of elements and/or other sets. To understand how untyped this universe is, one must remember that most of mathematics, which is full of extremely rich and complex structures, is represented in set theory by sets whose structural complexity reflects the complexity of the structures being represented. For example, integers are generally represented by sets of sets of sets whose level of nesting represents the cardinality of the integer, while functions are represented by possibly infinite sets of ordered pairs with unique first components.

As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

In computer memory, we distinguish characters and operations, both represented as bit strings. In Lisp, some S-expressions are called lists while others form legal programs. In λ-calculus some functions are chosen to represent boolean values, others to represent integers. In set theory some sets are chosen to denote ordered pairs, and some sets of ordered pairs are then called functions.

Untyped universes of computational objects decompose naturally into subsets with uniform behavior. Sets of objects with uniform behavior may be named and are referred to as types. For example, all integers exhibit uniform behavior by having the same set of applicable operations. Functions from integers to integers behave uniformly in that they apply to objects of a given type and produce values of a given type.

After a valiant organization effort, then, we may start thinking of untyped universes as if they were typed. But this is just an illusion, because it is very easy to violate the type distinctions we have just created. In computer memory, what is the bit-wise boolean *or* of a character and a machine operation? In Lisp, what is the effect of treating an arbitrary S-expression as a program? In the λ-calculus, what is the effect of a conditional over a non-boolean value? In set theory, what is the set-union of the function successor and the function predecessor?

Such questions are the unfortunate consequence of organizing untyped universes without going all the way to typed systems; it is then meaningful to ask about the (arbitrary) representations of higher-level concepts and their interactions.

1.2. Static and Strong Typing

A type system has as its major purpose to avoid embarrassing questions about representations, and to forbid situations where these questions might come up. In mathematics as in programming, types impose constraints which help to enforce correctness. Some untyped universes, like naive set theory, were found to be logically inconsistent, and typed versions were proposed to eliminate inconsistencies. Typed versions of set theory, just like typed programming languages, impose constraints on object interaction which prevent objects (in this case sets) from inconsistent interaction with other objects.

A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects. In an untyped system untyped objects are *naked* in that the underlying representation is exposed for all to see. Violating the type system involves removing the protective set of clothing and operating directly on the naked representation.

Objects of a given type have a representation that respects the expected properties of the data type. The representation is chosen to make it easy to perform expected operations on data objects. For example, positional notation is favored for numbers because it allows arithmetic operations to be easily defined. But there are nevertheless many possible alternatives in choosing data representations. Breaking the type system allows a data representation to be manipulated in ways that were not intended, with potentially disastrous results. For example, use of an integer as a pointer can cause arbitrary modifications to programs and data.

To prevent type violations, we generally impose a static type structure on programs. Types are associated with constants, operators, variables, and function symbols. A *type inference* system can be used to infer the types of expressions when little or no type information is given explicitly. In languages like Pascal and Ada, the type of variables and function symbols is defined by redundant declarations and the compiler can check the consistency of definition and use. In languages like ML, explicit declarations are avoided wherever possible and the system may infer the type of expressions from local context, while still establishing consistent usage.

Programming languages in which the type of every expression can be determined by static program analysis are said to be *statically typed*. Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile time is sometimes too restrictive. It may be replaced by the weaker requirement that all expressions are guaranteed to be type-consistent although the type itself may be statically unknown; this can be generally done by introducing some run-time type checking. Languages in which all expressions are type-consistent are called strongly typed languages. If a language is strongly typed its compiler can guarantee that the programs it accepts will execute without type errors. In general, we should strive for strong typing, and adopt static typing whenever possible. Note that every statically typed language is strongly typed but the converse is not necessarily true.

Static typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent. It facilitates early detection of type errors and allows greater execution-time efficiency. It enforces a programming discipline on the programmer that makes programs more structured and easier to read. But static typing may also lead to a loss of flexibility and expressive power by prematurely constraining the behavior of objects to that associated with a particular type. Traditional statically typed systems exclude programming techniques which, although sound, are incompatible with early binding of program objects to a specific type. For example they exclude generic procedures, e.g. sorting, that capture the structure of an algorithm uniformly applicable to a range of types.

1.3. Kinds of Polymorphism

Conventional typed languages, such as Pascal, are based on the idea that functions and procedures, and hence their operands, have a unique type. Such languages are said to be *monomorphic*, in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with *polymorphic* languages in which some values and variables may have more than one type. Polymorphic functions are functions whose operands (actual parameters) can have more than one type. Polymorphic types are types whose operations are applicable to values of more than one type.

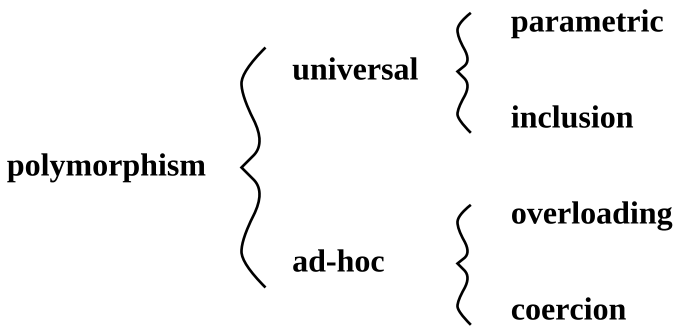


Figure 1: Varieties of polymorphism.

Strachey [Strachey 67] distinguished, informally, between two major kinds of polymorphism. *Parametric polymorphism* is obtained when a function works uniformly on a range of types: these types normally exhibit some common structure. *Ad-hoc polymorphism* is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.

Our classification of polymorphism in Figure 1 refines that of Strachey by introducing a new form of polymorphism called *inclusion polymorphism* to model subtypes and inheritance. Parametric and inclusion polymorphism are classified as the two major subcategories of *universal polymorphism*, which is contrasted with nonuniversal or ad-hoc polymorphism. Thus Figure 1 reflects Strachey's view of polymorphism but adds inclusion polymorphism to model object-oriented programming.

Parametric polymorphism is so called because the uniformity of type structure is normally achieved by type parameters, but uniformity can be achieved in different ways, and this more general concept is called *universal polymorphism*. Universally polymorphic functions will normally work on an infinite number of types (all the types having a given common structure), while an ad-hoc polymorphic function will only work on a finite set of different and potentially unrelated types. In the case of universal polymorphism, one can assert with confidence that some values (i.e., polymorphic functions) have many types, while in ad-hoc polymorphism this is more difficult to maintain, as one may take the position that an ad-hoc polymorphic function is really a small set of monomorphic functions. In terms of implementation, a universally polymorphic function will execute the *same* code for arguments of any admissible type, while an ad-hoc polymorphic function may execute *different* code for each type of argument.

There are two major kinds of universal polymorphism, i.e., two major ways in which a value can have many types. In *parametric polymorphism*, a polymorphic function has an implicit or explicit type parameter, which determines the type of the argument for each application of that function. In *inclusion polymorphism* an object can be viewed as belonging to many different classes which need not be disjoint, i.e. there may be inclusion of classes. These two views of universal polymorphism are not unrelated, but are sufficiently distinct in theory and in practice to deserve different names.

The functions that exhibit parametric polymorphism are also called *generic functions*. For example, the length function from lists of arbitrary type to integers is called a generic length function. A generic function is one which can work for arguments of many types, generally doing the same kind of work independently of the argument type. If we consider a generic function as a single value, it has many functional types and is therefore polymorphic. Ada generic functions are a special case of this concept of *generic*.

There are also two major kinds of ad-hoc polymorphism. In *overloading* the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. We may imagine that a preprocessing of the program will eliminate overloading by giving different names to the different functions; in this sense overloading is just a convenient syntactic abbreviation. A *coercion* is instead a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error. Coercions can be provided statically, by automatically inserting them between arguments and functions at compile time, or may have to be determined dynamically by run-time tests on the arguments.

The distinction between overloading and coercion blurs in several situations. This is particularly true when considering untyped languages and interpreted languages. But even in static, compiled languages there may be confusion between the two forms of ad-hoc polymorphism, as illustrated by the following example.

```
3     +   4
3.0   +   4
3     +   4.0
3.0   +   4.0
```

Here, the ad-hoc polymorphism of + can be explained in one of the following ways:

- The operator + has four overloaded meanings, one for each of the four combinations of argument types.
- The operator + has two overloaded meanings, corresponding to integer and real addition. When one of the argument is of type integer and the other is of type real, then the integer argument is coerced to the type real.
- The operator + is defined only for real addition, and integer arguments are always coerced to corresponding reals.

In this example, we may consider the same expression as exhibiting overloading or coercion, or both (and also changing meaning), depending on an implementation decision.

Our definition of polymorphism is applicable only to languages with a very clear notion of both type and value. In particular, there must be a clear distinction between the inherent type of an object and the apparent type of its syntactic representations in languages that permit overloading and coercion. These issues are further discussed below.

If we view a type as partially specifying the behavior, or intended usage, of associated values, then monomorphic type systems constrain objects to have just one behavior, while polymorphic type systems allow values to be associated with more than one behavior. Strictly monomorphic languages are too restrictive in their expressive power because they do not allow values, or even syntactic symbols that denote values, to exhibit different behavior in different contexts of use. Languages like Pascal and Ada have ways of relaxing strict monomorphism, but polymorphism is the exception rather than the rule and we can say that they are *mostly* monomorphic. Real and apparent exceptions to the monomorphic typing rule in conventional languages include:

(1) *Overloading*: integer constants may have both type integer and real.
Operators such as + are applicable to both integer and real arguments.

(2) *Coercion*: an integer value can be used where a real is expected, and vice versa.
(3) *Subtyping*: elements of a subrange type also belong to superrange types.
(4)*Value sharing*: nil in Pascal is a constant which is shared by all the pointer types.

These four examples, which may all be found in the same language, are instances of four radically different ways of extending a monomorphic type system. Let us see how they fit in the previous description of different kinds of polymorphism.

Overloading is a purely syntactic way of using the same name for different semantic objects; the compiler can resolve the ambiguity at compile time, and then proceed as usual.

Coercion allows the user to omit semantically necessary type conversions. The required type conversions must be determined by the system, inserted in the program, and used by the compiler to generate required type conversion code. Coercions are essentially a form of abbreviation which may reduce program size and improve program readability, but may also cause subtle and sometimes dangerous system errors. The need for run-time coercions is usually detected at compile time, but languages like (impure) Lisp have plenty of coercions that are only detected and performed at run time.

Subtyping is an instance of *inclusion polymorphism*. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures such as a type representing *Toyotas* which is a subtype of a more general type such as *Vehicles*. Every object of a subtype can be used in a supertype context in the sense that every Toyota is a vehicle and can be operated on by all operations that are applicable to vehicles.

Value sharing is a special case of *parametric polymorphism*. We could think of the symbol nil as being heavily overloaded, but this would be some strange kind of open-ended overloading, as nil is a valid element of an infinite collection of types which haven't even been declared yet. Moreover, all the uses of nil denote the same value, which is not the common case for overloading. We could also think that there is a different nil for every type, but all the nil's have the same representation and can be identified. The fact that an object having many types is uniformly represented for all types is characteristic of parametric polymorphism.

How do these relaxed forms of typing relate to polymorphism? As is implicit in the choice of names, universal polymorphism is considered *true* polymorphism, while ad-hoc polymorphism is some kind of *apparent* polymorphism whose polymorphic character disappears at close range. Overloading is not true polymorphism: instead of a value having many types, we allow a symbol to have many types, but the values denoted by that symbol have distinct and possibly incompatible types. Similarly, coercions do not achieve true polymorphism: an operator may appear to accept values of many types, but the values must be converted to some representation before the operator can use them; hence that operator really works on (has) only one type. Moreover, the output type is no longer dependent on the input type, as is the case in parametric polymorphism.

In contrast to overloading and coercion, subtyping is an example of true polymorphism: objects of a subtype can be uniformly manipulated as if belonging to their supertypes. In the implementation, the representations are chosen very carefully, so that no coercions are necessary when using an object of a subtype in place of an object of the supertype. In this sense the same object has many types (for example, in Simula a member of a subclass may be a longer memory segment than a member of its superclass, and its initial segment has the same structure as the member of the superclass). Similarly, operations are careful to interpret the representations uniformly so that they can work uniformly on elements of subtypes and supertypes.

Parametric polymorphism is the purest form of polymorphism: the same object or function can be used uniformly in different type contexts without changes, coercions or any kind of run-time tests or special encodings of representations. However, it should be noted that this uniformity of behavior requires that all data be represented, or somehow dealt with, uniformly (e.g., by pointers).

The four ways of relaxing monomorphic typing discussed thus far become more powerful and interesting when we consider them in connection with operators, functions and procedures. Let us look at some additional examples. The symbol + could be overloaded to denote at the same time integer sum, real sum, and string concatenation. The use of the same symbol for these three operations reflects an approximate similarity of algebraic structure but violates the requirements of monomorphism. The ambiguity can usually be resolved by the type of the immediate operands of an overloaded operator, but this may not be enough. For example, if 2 is overloaded to denote integer 2 and real 2.0, then 2+2 is still ambiguous and is resolvable only in a larger context such as assignment to a typed variable. The set of possibilities can explode if we allow user-defined overloaded operators.

Algol 68 is well known for its baroque coercion scheme. The problems to be solved here are very similar to overloading, but in addition coercions have run-time effects. A two-dimensional array with only one row can be coerced to a vector, and a vector with only one component can be coerced to a scalar. The conditions for performing a coercion may have to be detected at run time, and may actually arise from programming errors, rather than planning. The Algol 68 experience suggests that coercions should generally be explicit, and this view has been taken by many later language designs.

Inclusion polymorphism can be found in many common languages, of which Simula 67 is the earliest example. Simula's *classes* are user-defined types organized in a simple inclusion (or inheritance) hierarchy where every class has a unique immediate superclass. Simula's objects and procedures are polymorphic because an object of a subclass can appear wherever an object of one of its superclasses is required. Smalltalk [Goldberg

83], although an untyped language, also popularized this view of polymorphism. More recently, Lisp Flavors [Weinreb 81] (untyped) have extended this style of polymorphism to multiple immediate superclasses, and Amber (typed) [Cardelli 85] further extends it to higher-order functions.

The paradigmatic language for parametric polymorphism is ML [Milner 84], which was entirely built around this style of typing. In ML, it is possible to write a polymorphic identity function which works for every type of argument, and a length function which maps a list of arbitrary element type into its integer length. It is also possible to write a generic sorting package that works on any type with an ordering relation. Other languages that used or helped develop these ideas include CLU [Liskov 81], Russell [Demers 79, Hook 84], Hope [Burstall 80], Ponder [Fairbairn 82] and Poly [Matthews 85].

Finally, we should mention generic procedures of the kind found in Ada, which are parametrized templates that must be instantiated with actual parameter values before they can be used. The polymorphism of Ada's generic procedures is similar to the parametric polymorphism of languages like ML, but is specialized to particular kinds of parameters. Parameters may be type parameters, procedure parameters, or value parameters. Generic procedures with type parameters are polymorphic in the sense that formal type parameters can take different actual types for different instantiations. However, generic type polymorphism in Ada is syntactic since generic instantiation is performed at compile time with actual type values that must be determinable (manifest) at compile time. The semantics of generic procedures is macro-expansion driven by the type of the arguments. Thus, generic procedures can be considered as abbreviations for sets of monomorphic procedures. With respect to polymorphism, they have the advantage that specialized optimal code can be generated for the different forms of inputs. On the other hand, in true polymorphic systems code is generated only once for every generic procedure.

1.3. The Evolution of Types In Programming Languages

In early programming languages, computation was identified with numerical computation and values could be viewed as having a single arithmetic type. However, as early as 1954, Fortran found it convenient to distinguish between integers and floating-point numbers, in part because differences in hardware representation made integer computation more economical and in part because the use of integers for iteration and array computation was logically different from the use of floating point numbers for numerical computation.

Fortran distinguished between integer and floating point variables by the first letter of their names. Algol 60 made this distinction explicit by introducing redundant identifier declarations for integer real and Boolean variables. Algol 60 was the first significant language to have an explicit notion of type and associated requirements for compile time type checking. Its block-structure requirements allowed not only the type but also the scope (visibility) of variables to be checked at compile time.

The Algol 60 notion of type was extended to richer classes of values in the 1960s. Of the numerous typed languages developed during this period, PL/I, Pascal, Algol 68, and Simula, are noteworthy for their contributions to the evolution of the concept of type.

PL/I attempts to combine the features of Fortran, Algol 60, Cobol, and Lisp. Its types include typed arrays, records, and pointers. But it has numerous type loopholes, such as not requiring the type of values pointed to by pointer variables to be specified, which weaken the effectiveness of compile-time type checking.

Pascal provides a cleaner extension of types to arrays records and pointers, as well as user-defined types. However, Pascal does not define type equivalence, so that the question of when two type expressions denote the same type is implementation-dependent. There are also problems with type granularity. For example, Pascal's notion of array type, which includes the array bounds as part of the type, is too restrictive in that procedures that operate uniformly on arrays of different dimensions cannot be defined. Pascal leaves loopholes in strong type specification by not requiring the full type of procedures passed as parameters to be specified, and by allowing the tag field of variant records to be independently manipulated. The ambiguities and insecurities of the Pascal type system are discussed in [Welsh 77].

Algol 68 has a more rigorous notion of type than Pascal, with a well-defined notion of type equivalence (structural equivalence). The notion of type (*mode* in Algol 68) is extended to include procedures as first-class values. Primitive modes include int, real, char, bool, string, bits, bytes, format, file, while mode constructors (type constructors) include array, struct, proc, union, and ref for respectively constructing array types, record types, procedure types, union (variant) types, and pointer types. Algol 68 has carefully defined rules for coercion, using dereferencing, deproceduring, widening, rowing, uniting, and voiding to transform values to the type required for further computation. Type checking in Algol 68 is decidable, but the type-checking algorithm is so complex that questions of type equivalence and coercion cannot always be checked by the user. This complexity was felt by some to be a flaw, resulting in a reaction against complex type systems. Thus, later languages, like Ada, had simpler notion of type equivalence with severely restricted coercion.

Simula is the first *object-oriented* language. Its notion of type includes classes whose instances may be assigned as values of class-valued variables and may persist between execution of the procedures they contain. Procedures and data declarations of a class constitute its interface and are accessible to users. Subclasses inherit declared entities in the interface of superclasses and may define additional operations and data that specialize the behavior of the subclass. Instances of a class are like data abstractions in having a declarative interface and a state that persists between invocation of operations, but lack the information-hiding mechanism of data abstractions. Subsequent object-oriented languages like Smalltalk and Loops combine the class concept derived from Simula with a stronger notion of information hiding.

Modula-2 [Wirth 83] is the first widespread language to use modularization as a major structuring principle (these ideas were first developed in Mesa). Typed interfaces specify the types and operations available in a module; types in an interface can be made *opaque* to achieve data abstraction. An interface can be specified separately from its implementation, thereby separating the specification and implementation tasks. Block-structured scoping, preserved within modules, is abandoned at a more global level in favor of flexible inter-module visibility rules achieved by import and export lists. Module interfaces are similar to class declarations (except for the above-mentioned scoping rules), but unlike class instances, module instances are not first-class values. A linking phase is necessary to interconnect module instances for execution; this phase is specified by the module interfaces but is external to the language.

ML has introduced the notion of parametric polymorphism in languages. ML types can contain type variables which are instantiated to different types in different contexts. Hence it is possible to partially specify type information and to write programs based on partially specified types which can be used on all the instances of those types. A way of partially specifying types is just to omit type declarations: the most general (less specific) types which fit a given situation are then automatically inferred.

The above historical framework provides a basis for a deeper discussion of the relations between types, data abstraction, and polymorphism in real programming languages. We consider the untyped data abstractions (packages) of Ada, indicate the impact on methodology of requiring data abstractions to have type and inheritance, discuss the interpretation of inheritance as subtype polymorphism, and examine the relation between the subtype polymorphism of Smalltalk and the parametric polymorphism of ML.

Ada has a rich variety of modules, including subprograms to support procedure-oriented programming, packages to support data abstractions, and tasks to support concurrent programming. But it has a relatively weak notion of type, excluding procedures and packages from the domain of typed objects, and including task types relatively late in the design process as an afterthought. Its choice of name equivalence as type equivalence is weaker than the notion of structural equivalence used in Algol 68. Its severe restriction against implicit coercion weakens its ability to provide polymorphic operations applicable to operands of many types.

Packages in Ada have an interface specification of named components that may be simple variables, procedures, exceptions, and even types. They may hide a local state either by a *private* data type or in the package body. Packages are like record instances in having a user interface of named components. Ada packages differ from records in that record components must be typed values while package components may be procedures, exceptions, types, and other named entities. Since packages are not themselves types they cannot be parameters, components of structures, or values of pointer variables [Wegner 83]. Packages in Ada are second-class objects while class instances in Simula or objects in object-oriented languages are first-class objects.

The differences in behavior between packages and records in Ada is avoided in object-oriented languages by extending the notion of type to procedures and data abstractions. In the context of this discussion it is useful to define object-oriented languages as extensions of procedure-oriented languages that support typed data abstractions with inheritance. Thus we say that a language is object-oriented iff it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state
- Objects have an associated object type
- Types may inherit attributes from supertypes

These requirements may be summarized as:

object-oriented = data abstractions + object types + type inheritance

The usefulness of this definition may be illustrated by considering the impact of each of these requirements on methodology. Data abstraction by itself provides a way of organizing data with associated operations that differs considerably from the traditional methodology of procedure- oriented programming. The realization of *data abstraction methodology* was one of the primary objectives of Ada, and this methodology is described at length in the Ada literature in publications such as [Booch 83]. However Ada satisfies only the first of our three requirements for object-oriented programming and it is interesting to examine the impact of object types and inheritance on data abstraction methodology [Hendler 86].

The requirement that all objects have a type allows objects to be first-class values so that they can be managed as data structures within the language as well as used for computation. The requirement of type inheritance allows relations among types to be specified. Inheritance may be viewed as a type composition mechanism which allows the properties of one or more types to be reused in the definition of a new type. The specification *B inherits A* may be viewed as an abbreviation mechanism which avoids redefining the attributes of type A in the definition of type B. However, inheritance is more than a shorthand, since it imposes structure among a collection of related types that can greatly reduce the conceptual complexity of a system specification. This is illustrated by the Smalltalk object hierarchy in [Goldberg 83].

The Smalltalk object hierarchy is a description of the Smalltalk programming environment in Smalltalk. It is conceptually similar to the Lisp *apply* function which describes the Lisp language interpreter in Lisp, but is a great deal more complex. It describes a collection of over 75 related system object types by an inheritance hierarchy. The object types include numerical, structured, input-output, concurrent, and display objects. The object hierarchy carefully factors out properties common to numeric objects into the supertype Number. It factors out properties common to different kinds of structured objects into the supertype Collection. It further factors out properties common to numbers, collections, and other kinds of objects into the supertype Object. In doing this the collection of over 75 object types that comprise the Smalltalk environment is described as a relatively simple structured hierarchy of object types. The shorthand provided by the object hierarchy in reusing superclasses whose attributes are shared by subclasses is clearly incidental to the conceptual parsimony achieved by imposing a coherent structure on the collection of object types.

The Smalltalk object hierarchy is also significant as an illustration of the power of polymorphism. We may characterize a polymorphic function as a function applicable to values of more than one type and *inclusion polymorphism* as a relation among types which allows operations to be applied to object of different types related by inclusion. Objects are seen as collections of such polymorphic operations (attributes). This view emphasizes the sharing of operations by operands of many types as a primary feature of polymorphism.

The Smalltalk object hierarchy realizes polymorphism in the above sense by factoring out attributes common to a collection of subtypes into a supertype. Attributes common to numerical types are factored out into the supertype Number. Attributes common to structured types are factored out into the supertype Collection. Attributes common to all types are factored out into the supertype Object. Thus polymorphism is intimately related to the notion of inheritance, and we can say that the expressive power of object-oriented type systems is due in large measure to the polymorphism they facilitate.

In order to complete our discussion of the evolution of types in programming languages we examine the type mechanisms of ML [Milner 84]. ML is an interactive functional programming language in which type specifications omitted by the user may be reintroduced by type inference.If the user enters "3+4" the system responds "7:int", computing the value of the expression and inferring that the operands and the value are of type int. If the user enters the function declaration "fun f x = x+1" the system responds "f:int→int", defining a function value for f and inferring that it is of type "int→int". ML supports type inference not only for traditional types but also for parametric (polymorphic) types, such as the length function for lists. If "fun rec length x = if x = nil then 0 else 1+length(tail(x));" is entered, ML will infer that "length" is a function from lists of arbitrary element type to integers (length: 'a list → int). If the user then enters "length[1;2;3]", applying length to a list of integers, the system infers that length is to be specialized to the type "int list → int" and then applies the specialized function to the list of integers.

When we say that a parametric function is applicable to lists of arbitrary type we really mean that it may be specialized by (implicitly or explicitly) providing a type parameter T, and that the specialized function may then be applied to the specialized operands. There is an important distinction between the parametric function length for lists of arbitrary type and the specialized function for lists of type int. Functions like length are applicable to lists of arbitrary type because they have a uniform parametric representation that allows them to be specialized by supplying a type parameter. This distinction between a parametric function and its specialized versions is blurred in languages like ML, because type parameters omitted by the user are automatically reintroduced by the type inference mechanism.

Supertypes in object-oriented languages may be viewed as parametric types whose parameter is omitted by the user. In order to understand the similarity between parametric types and supertypes it is useful to introduce a notation where supertype parameters must be explicitly supplied in specializing a supertype to a subtype. We shall see below that Fun has explicit type parameters for both parametric types and supertypes in order to provide a uniform model for both parametric and subtype polymorphism. This results in a uniform treatment of type inference when parameters are omitted in parametric types and supertypes.

1.5. Type Expression Sublanguages

As the set of types of a programming language becomes richer, and its set of definable types becomes infinite, it becomes useful to define the set of types by a type expression sublanguage. The set of type expressions of current strongly typed programming languages is generally a simple sublanguage of the complete language that is nevertheless not altogether trivial. Type expression sublanguages generally include basic types like integer and boolean and composite types like arrays, records, and procedures constructed from basic types.

```
Type ::= BasicType | ConstructedType
BasicType ::= Int | Bool | ...
ConstructedType ::= Array(Type) | Type → Type | ...
```

The type expression sublanguage should be sufficiently rich to support types for all values with which we wish to compute, but sufficiently tractable to permit decidable and efficient type checking. One of the purposes of this paper is to examine tradeoffs between richness and tractability for type expression sublanguages of strongly typed languages.

The type expression sublanguage can generally be specified by a context-free grammar. However, we are interested not only in the syntax of the type expression sublanguage but also in its semantics. That is, we are interested in what types denote and in relations among type expressions. The most basic relation among type expressions is type equivalence. However, we are also interested in similarity relations among types that are weaker than equivalence, such as inclusion which is related to subtypes. Similarity relations among type expressions that permit a type expression to denote more than one type, or to be compatible with many types, are referred to as polymorphism.

The usefulness of a type system lies not only in the set of types that can be represented but also in the kinds of relationships among types that can be expressed. The ability to express relations among types involves some ability to perform computations on types to determine whether they satisfy the desired relationship. Such computations could in principle be as powerful as computations performable on values. However, we are concerned only with simple, easily computable relationships that express uniform behavior shared by collections of types.

The reader interested in a discussion of type expression languages and type-checking algorithms for languages like Pascal and C is referred to chapter 6 of [Aho 85], which considers type checking for overloading, coercion, and parametric polymorphism. Fun adds abstract data types to the set of basic types and adds subtype and inheritance to the forms of polymorphism that are supported.

1.6. Preview of Fun

Fun is a λ-calculus-based language that enriches the first-order typed λ-calculus with second-order features designed to model polymorphism and object-oriented languages.

Section 2 reviews the untyped and typed λ-calculus and develops first-order features of the Fun type expression sublanguage. Fun has the basic types Bool, Int, Real, String and constructed types for record, variant, function, and recursive types. This set of *first-order types* is used as a base for introducing parametric types, abstract data types, and type inheritance by means of second-order language features in subsequent sections.

Section 3 briefly reviews theoretical models of types related to features of Fun, especially models which view types as sets of values. Viewing types as sets allows us to define parametric polymorphism in terms of set intersection of associated types and inheritance polymorphism in terms of subsets of associated types. Data abstraction may also be defined in terms of set operations (in this case unions) on associated types.

Sections 4, 5, and 6 respectively augment the first-order λ-calculus with universal quantification for realizing parameterized types, existential quantification for realizing data abstraction, and bounded quantification for realizing type inheritance. The syntactic extensions of the type expression sublanguage determined by these features may be summarized as follows:

```
Type ::= ... | QuantifiedType
QuantifiedType ::=
      ∀A. Type |                              Universal Quantification
      ∃A. Type |                              Existential Quantification
      ∀A⊆Type. Type |  ∃A⊆Type. Type          Bounded Quantification
```

Universal quantification enriches the first-order λ-calculus with parameterized types that may be specialized by substituting actual type parameters for universally quantified parameters. Universally quantified types are themselves first-class types and may be actual parameters in such a substitution.

Existential quantification enriches first-order features by allowing abstract data types with hidden representation. The interaction of universal and existential quantification is illustrated in section 5.3 for the case of stacks with a universally quantified element type and an existentially quantified hidden data representation.

Fun supports information hiding not only through existential quantification but also through its let construct, which facilitates hiding of local variables of a module body. Hiding by means of let is referred to as first-order hiding because it involves hiding of local identifiers and associated values, while hiding by means of existential quantifiers is referred to as second-order hiding because it involves hiding of type representations. The relation between these two forms of hiding is illustrated in section 5.2 by contrasting hiding in package bodies with hiding in private parts of Ada packages.

Bounded quantification enriches the first-order λ-calculus by providing explicit subtype parameters. Inheritance (i.e. subtypes and supertypes) is modeled by explicit parametric specialization of supertypes to the subtype for which the operations will actually be executed. In object-oriented languages every type is potentially a supertype for subsequently defined subtypes and should therefore be modelled by a bounded quantified type. Bounded quantification provides an explanatory mechanism for object-oriented polymorphism that is cumbersome to use explicitly but useful in illuminating the relation between parametric and inherited polymorphism.

Section 7 briefly reviews type checking and type inheritance for Fun. It is supplemented by an appendix listing type inference rules.

Section 8 provides a hierarchical classification of object-oriented type systems. Fun represents the topmost (most general) type system of this classification. The relation of Fun to less general systems associated with ML, Galileo, Amber, and other languages with interesting type systems is reviewed.

It is hoped that readers will have as much fun reading about Fun as the authors have had writing about it.

2. The λ-Calculus

2.1. The Untyped λ-Calculus

The evolution from untyped to typed universes may be illustrated by the λ-calculus, initially developed as an untyped notation to capture the essence of the functional application of operators to operands. Expressions in the λ-calculus have the following syntax (we use fun instead of the traditional λ to bring out the correspondence with programming language notations):

```
e ::= x            -- a variable is a λ-expression
e ::= fun(x)e      -- functional abstraction of e
e ::= e(e)         -- operator e applied to operand e
```

The identity function and successor function may be specified in the λ-calculus as follows (with some syntactic sugar explained later). We use the keyword value to introduce a new name bound to a value or a function:

```
value id = fun(x) x              -- identity function
value succ = fun(x) x+1          -- successor function (for integers)
```

The identity function may be applied to an arbitrary λ-expression and always yields the λ-expression itself. In order to define addition on integers in the pure λ-calculus we pick a representation for integers and define the addition operation so that its effect on λ-expressions representing the integers n and m is to produce the λ-expression that represents n + m. The successor function should be applied only to λ-expressions that represent integers and suggests a notion of typing. The infix notation x+1 is an abbreviation for the functional notation + (x) (1). The symbols 1 and + above should in turn be viewed as abbreviations for a pure λ-calculus expression for the number 1 and addition.

Correctness of integer addition requires no assumptions about what happens when the λ-expression representing addition is applied to λ-expressions that do not represent integers. However, if we want our notation to have good error-checking properties, it is desirable to define the effect of addition on arguments that are not integers as an error. This is accomplished in typed programming languages by type checking that eliminates, at compilet time, the possibility of operations on objects of an incorrect type.

Type checking in the λ-calculus, just as in conventional programming languages, has the effect that large classes of λ-expressions legal in the untyped λ-calculus become illegal. The class of illegally-typed expressions depends on the type system one adopts, and, although undesirable, it may even depend on a particular type-checking algorithm.

The idea of λ-expressions operating on functions to produce other functions can be illustrated by the function twice which has the following form:

```
value twice = fun(f) fun(y) f(f(y))      -- twice function
```

The application of twice to the successor function yields a λ-expression that computes the successor of the successor.

```
twice(succ)           ⇒    fun(y) succ(succ(y))
twice (fun(x)x+1)     ⇒    fun(y) (fun(x)x+1) ((fun(x)x+1) (y))
```

The above discussion illustrates how types arise when we specialize an untyped notation such as the λ-calculus to perform particular kinds of computation such as integer arithmetic. In the next section we introduce explicit types into the λ-calculus. The resulting notation is similar to functional notation in traditional typed programming languages.

2.2. The Typed λ-Calculus

The typed λ-calculus is like the λ-calculus except that every variable must be explicitly typed when introduced as a bound variable. Thus the successor function in the typed λ-calculus has the following form:

```
value succ = fun (x: Int) x+1
```

The function twice from integers to integers has a parameter f whose type is Int → Int (the type of functions from integers to integers) and may be written as follows:

```
value twice = fun(f: Int → Int) fun (y:Int) f(f(y))
```

This notation approximates that of functional specification in typed programming languages but omits specification of the result type. We may denote the result type with a returns keyword as follows:

```
value succ = fun(x: Int) (returns Int) x + 1
```

However, the type of the result can be determined from the form of the function body x + 1. We shall omit result type specifications in the interests of brevity. Type inference mechanisms that allow this information to be recovered during compilation are discussed in a later section.

Type declarations are introduced by the keyword type. Throughout this paper, type names begin with upper-case letters while value and function names begin with lower-case letters.

```
type IntPair = Int × Int
type IntFun = Int → Int
```

Type declarations introduce names (abbreviations) for type expressions; they do not *create* new types in any sense. This is sometimes expressed by saying that we used *structural equivalence* on types instead of *name equivalence*: two types are equivalent when they have the same structure, regardless of the names we use as abbreviations.

The fact that a value v has a type T is indicated by v:T.

```
(3,4): IntPair
succ: IntFun
```

We need not introduce variables by type declarations of the form var:T because the type of a variable may be determined from the form of the assigned value. For example the fact that intPair below has the type IntPair can be determined by the fact that (3,4) has type Int × Int, which has been declared equivalent to IntPair.

```
value intPair = (3,4)
```

However, if we want to indicate the type of a variable as part of its initialization we can do so by the notation value var:T = value.

```
value intPair: IntPair = (3,4)
value succ: Int → Int = fun(x: Int) x + 1
```

Local variables can be declared by the let-in construct, which introduces a new initialized variable (following let) in a local scope (an expression following in). The value of the construct is the value of that expression.

```
let a = 3 in a + 1              yields 4
```

If we want to specify types, we can also write:

```
let a : Int = 3 in a + 1
```

The let-in construct can be defined in terms of basic fun-expressions:

```
let a : T = M in N        ≡       (fun(a:T) N)(M)
```

2.3. Basic Types, Structured Types and Recursion

The typed λ-calculus is usually augmented with various kinds of basic and structured types. For basic types we shall use:

Unit	the trivial type, with only element ()

Bool	with an if-then-else operation
Int	with arithmetic and comparison operations
Real	with arithmetic and comparison operations
String	with string concatenation (infix) ^

Structured types can be built up from these basic types by means of type constructors. The type constructors in our language include function spaces ($\rightarrow$), Cartesian products ($\times$), record types (also called labeled Cartesian products) and variant types (also called labeled disjoint sums).

A pair is an element of a Cartesian product type, e.g.

```
value p = 3,true : Int × Bool
```

Operations on pairs are selectors for the first and second components:

```
fst(p)    yields  3
snd(p)    yields  true
```

A record is an unordered set of labeled values. Its type may be specified by indicating the type associated with each of its labels. A record type is denoted by a sequence of labeled types, separated by commas and enclosed in curly braces:

```
type ARecordType = {a: Int, b: Bool, c: String}
```

A record of this type may be created by initializing each of the record labels to a value of the required type. It is written as a sequence of labeled values separated by commas and enclosed in curly braces:

```
value r: ARecordType = {a = 3, b = true, c = "abcd"}
```

The labels must be unique within any given record or record type. The only operation on records is field selection, denoted by the usual *dot* notation:

```
r.b  yields  true
```

Since functions are first-class values, records may in general have function components.

```
type FunctionRecordType = {f1: Int → Int, f2: Real → Real}
value functionRecord = {f1 = succ, f2 = sin}
```

A record type can be defined in terms of existing record types by an operator & which concatenates two record types:

```
type NewFunctionRecordType = FunctionRecordType & {f3: Bool → Bool}
```

This is intended as an abbreviation, instead of writing the three fields f1, f2, and f3 explicitly. It is only valid when used on record types, and when no duplicated labels are involved.

A data structure can be made local and private to a collection of functions by let-in declarations. Records with function components are a particularly convenient way of achieving this; here is a private counter variable shared by an increment and a total function:

```
value counter =
    let count = ref(0)
    in   {increment = fun(n:Int) count := count + n,
           total = fun() count
         }

counter.increment(3)
counter.total()             yields 3
```

This example involves side-effects, as the main use of private variables is to update them privately. The primitive ref returns an updateable reference to an object, and assignments are restricted to work on such references. This is an common form of information hiding that allows updating of local state by using static scoping to restrict visibility.

A variant type is also formed from an unordered set of labeled types, which are now enclosed in brackets:

```
type AVariantType = [a: Int, b:Bool, c: String]
```

An element of this type can either be an integer labeled a, a boolean labeled b, or a string labeled c:

```
value v1 = [a = 3]
value v2 = [b = true]
value v3 = [c = "abcd"]
```

The only operation on variants is case selection. A case statement for a variant of type AVariantType, has the following form:

```
case *variant* of
  [a = *variable of type* Int] *action for case* a
  [b = *variable of type* Bool] *action for case* b
  [c = *variable of type* String] *action for case* c
```

where in each case a new variable is introduced and bound to the respective contents of the variant. That variable can then be used in the respective action.

Here is a function which, given an element of type AVariantType above, returns a string:

```
value f = fun (x: AVariantType)
      case x of
        [a = anInt] "it is an integer"
        [b = aBool] "it is a boolean"
        [c = aString] "it is the string: " ^ aString
      otherwise "error"
```

where the contents of the variant object x are bound to the identifiers anInt, aBool or aString depending on the case.

In the untyped λ-calculus it is possible to express recursion operators and to use them to define recursive functions. However, all computations expressible in the typed λ-calculus must terminate (roughly, the type of a function is always strictly more complex than the type of its result, hence after some number of applications of the function we obtain a basic type; moreover, we do not have non-terminating primitives). Hence, recursive definitions are introduced as a new primitive concept. The factorial function can be expressed as:

```
rec value fact =
        fun (n:Int) if n=0 then 1 else n * fact(n-1)
```

For simplicity we assume that the only values which can be recursively defined are functions.

Finally, we introduce recursive type definitions. This allows us, for example, to define the type of integer lists out of record and variant types:

```
rec type IntList =
        [nil:Unit,
         cons: {head: Int, tail: IntList}
        ]
```

A integer list is either nil (represented as [nil = ()]) or the cons of an integer and an integer list (represented as, e.g., [cons = {head = 3, tail = nil}]).

3. Types are Sets of Values

What is an adequate notion of *type* which can account for polymorphism, abstraction and parametrization? In the previous sections we have started to describe a particular type system by giving informal typing rules for the linguistic constructs we use. These rules are enough to characterize the type system at an intuitive level, and can be easily formalized as a type inference system. The rules are sound and can stand on their own, but have been discovered and justified by studying a particular semantics of types, developed in [Hindley 69] [Milner 78] [Damas 82] [MacQueen 84a] and [Mitchell 84].

Although we do not need to discuss that semantic theory of types in detail, it may be useful to explain the basic intuitions behind it. These intuitions can in turn be useful in understanding the typing rules, particularly regarding the concept of subtypes which will be introduced later.

There is a universe V of all values, containing simple values like integers, data structures like pairs, records and variants, and functions. This is a complete partial order, built using Scott's techniques [Scott 76], but in first approximation we can think of it as just a large set of all possible computable values.

A type is a set of elements of V. Not all subsets of V are legal types: they must obey some technical properties. The subsets of V obeying such properties are called *ideals*. All the types found in programming languages are ideals in this sense, so we don't have to worry too much about subsets of V which are not ideals.

Hence, a type is an ideal, which is a set of values. Moreover, the set of all types (ideals) over V, when ordered by set inclusion, forms a lattice. The top of this lattice is the type Top (the set of all values, i.e. V itself). The bottom of the lattice is, essentially, the empty set (actually, it is the singleton set containing the least element of V).

The phrase *having a type* is then interpreted as *membership* in the appropriate set. As ideals over V may overlap, a value can have many types.

The set of types of any given programming language is generally only a small subset of the set of all ideals over V. For example any subset of the integers determines an ideal (and hence a type), and so does the set of all pairs with first element equal to 3. This generality is welcome, because it allows one to accommodate many different type systems in the same framework. One has to decide exactly which ideals are to be considered *interesting* in the context of a particular language.

A particular type system is then a collection of ideals of V, which is usually identified by giving a language of type expressions and a mapping from type expressions to ideals. The ideals in this collection are elevated to the rank of *types* for a particular language. For example, we can choose the integers, integer pairs and integer-to-integer functions as our type system. Different languages will have different type systems, but all these type systems can be built on top of the domain V (provided that V is rich enough to start with), using the same techniques.

A monomorphic type system is one in which each value belongs to at most one type (except for the least element of V which, by definition of ideal, belongs to all types). As types are sets, a value may belong to many types. A polymorphic type system is one in which large and interesting collections of values belong to many types. There is also a grey area of *mostly* monomorphic and *almost* polymorphic systems, so the definitions are left imprecise, but the important point is that the basic model of ideals over V can explain all these degrees of polymorphism.

Since types are sets, subtypes simply correspond to subsets. Moreover, the semantic assertion *T1 is a subtype of T2* corresponds to the mathematical condition $T1 \subseteq T2$ in the type lattice. This gives a very simple interpretation for subrange types and inheritance, as we shall see in later sections.

Finally, if we take our type system as consisting of the single set V, we have a type-free system in which all values have the same type. Hence we can express typed and untyped languages in the same semantic domain, and compare them.

The type lattice contains many more points than can be named in any type language. In fact it includes an uncountable number of points, since it includes every subset of the integers. The objective of a language for talking about types is to allow the programmer to name those types that correspond to interesting kinds of behavior. In order to do this the language contains type constructors, including function type constructors (e.g., type T = T1 → T2) for constructing a function type T from domain and range types T1, T2. These constructors allow an unbound number of interesting types to be constructed from a finite set of primitive types. However, there may be useful types of the type lattice that cannot be denoted using these constructors.

In the remaining sections of this paper we introduce more powerful type constructors that allow us to talk about types corresponding to infinite unions and intersections in the type lattice. In particular, universal quantification will allow us to name types whose lattice points are infinite intersections of types, while existential quantification will allow us to name types corresponding to infinite unions. Our reason for introducing universal and existential quantification is the importance of the resulting types in increasing the expressive power of typed programming languages. It is fortunate that these concepts are also mathematically simple and that they correspond to well-known mathematical constructions.

The ideal model is not the only model of types which has been studied. With respect to other denotational models, however, it has the advantage of explaining simple and polymorphic types in an intuitive way, namely as sets of values, and of allowing a natural treatment of inheritance. Less satisfactory is its treatment of type parametrization, which is rather indirect since types cannot be values, and its treatment of type operators, which involves getting out of the model and considering functions over ideals. In view of this intuitive appeal, we have chosen the ideal model as our underlying view of types, but much of our discussion could be carried over, and sometimes even improved, if we chose to refer to other models.

The idea of types as parameters is fully developed in the second-order λ-calculus [Bruce 84]. The (only known) denotational models the second-order λ-calculus are *retract models* [Scott 76]. Here, types are not sets of objects but special functions (called retracts); these can be interpreted as identifying sets of objects, but are objects themselves. Because of the property that types are objects, retract models can more naturally explain explicit type parameters, while ideal models can more naturally explain implicit type parameters.

4. Universal Quantification

4.1. Universal Quantification and Generic Functions

The typed λ-calculus is sufficient to express monomorphic functions. However it cannot adequately model polymorphic functions. For example, it requires the previously defined function twice to be unnecessarily restricted to functions from integers to integers when we would have liked to define it polymorphically for functions a → a from an arbitrary type a to itself. The identity function can similarly be defined only for specific types such as integers: fun(x:Int)x. We cannot capture the fact that its form does not depend on any specific type. We cannot express the idea of a functional form that is the same for a variety of types, and we must explicitly bind variables and values to a specific type at a time when such binding may be premature.

The fact that a given functional form is the same for all types may be expressed by universal quantification. In particular, the identity function may be expressed as follows:

```
value id = all[a] fun(x:a) x
```

In this definition of id, a is a type variable and all[a] provides type abstraction for a so that id is the identity for all types. In order to apply this identity function to an argument of a specific type we must first supply the type as a parameter and then the argument of the given type:

```
id [Int] (3)
```

(We use the convention that type parameters are enclosed in square brackets while typed arguments are enclosed in parentheses.)

We refer to functions like id which require a type parameter before they can be applied to functions of a specific type as *generic functions*. id is the generic identity function.

Note that all is a binding operator just like fun and requires a matching actual parameter to be supplied during function application. However, all[a] serves to bind a type while fun(x:a) serves to bind a variable of a given (possibly generic) type.

Although types are applied, there is no implication that types can be manipulated as values: types and values are still distinct and type abstractions and application serve type-checking purposes only, with no run-time implications. In fact we may decide to omit the type information in square brackets:

```
value id = fun(x:a) x          where a is now a free type variable
id(3)
```

Here the type-checking algorithm has the task of recognizing that a is a free type variable and reintroducing the original all[a] and [Int] information. This is part of what a polymorphic type-checker can do, like the one used in the ML language. In fact ML goes further and allows the programmer to omit even the remaining type information:

```
value id = fun(x) x
id(3)
```

ML has a type inference mechanism that allows the system to infer the types of both monomorphic and polymorphic expressions, so that type specifications omitted by the programmer can be reintroduced by the system. This has the advantage that the programmer can use the shorthand of the untyped λ-calculus while the system can translate the untyped input into fully typed expressions. However, there are no known fully automatic type inference algorithms for the powerful type systems we are going to consider. In order for us to clarify what is happening, and not to depend on the current state of type-checking technology, we shall always write down enough type information to make the type checking task trivial.

Going back to the fully explicit language, let's extend our notation so that the type of a polymorphic function can be explicitly talked about. We denote the type of a generic function from an arbitrary type to itself by ∀a. a → a:

```
type GenericId = ∀a. a → a
id: GenericId
```

Here is an example of a function taking a parameter of a universally quantified type. The function inst takes a function of the above type and returns two instances of it, specialized for integers and booleans:

```
value inst = fun(f: ∀a. a → a) (f[Int],f[Bool])

value intid = fst(inst(id))          : Int → Int
value boolid = snd(inst(id))                                : Bool → Bool
```

In general, function parameters of universally quantified types are most useful when they have to be used on different types in the body of a single function, e.g., a list length function passed as a parameter and used on lists of different types.

In order to show some of the freedom we have in defining polymorphic functions, we now write two versions of twice which differ in the way type parameters are passed. The first version, twice1, has a function parameter f which is of a universal type. The specification

```
fun(f: ∀a. a → a) body-of-function
```

specifies the type of function parameter f to be generic and to admit functions from any given type into the same type. Applied instances of f in the body of twice1 must have a formal type parameter f[t] and require an actual type to be supplied when applying twice1. The full specification of twice1 requires binding of the type parameter t as a universally quantified type and binding of x to t.

```
value twice1 = all[t] fun(f: ∀a. a → a) fun(x: t) f[t](f[t](x))
```

Thus twice1 has three bound variables for which actual parameters must be supplied during function application.

```
all[t]               -- requires an actual parameter which is a type
fun(f: ∀a. a → a)    -- requires a function of the type ∀a. a → a
fun(x: t)            -- requires an argument of the type substituted for t
```

An application of twice1 to the type Int, the function id, and the argument 3 is specified as follows:

```
twice1[Int](id)(3)
```

Note that the third argument 3 has the type Int of the first argument and that the second argument id is of a universally quantified type. Note also that twice1[Int](succ) would not be legal because succ does not have the type ∀a. a → a.

The function twice2 below differs from twice1 in the type of the argument f, which is not universally quantified. Now we do not need to apply f[t] in the body of twice:

```
value twice2 = all[t] fun(f: t → t) fun(x: t) f(f(x))
twice2[Int]                  yields  fun(f: Int → Int) fun(x: Int) f(f(x))
```

It is now possible to compute twice of succ:

```
twice2[Int](succ)            yields  fun(x: Int) succ(succ(x))
twice2[Int](succ)(3)         yields  5
```

Thus twice2 first receives the type parameter Int which serves to specialize the function f to be Int → Int, then receives the function succ of this type, and then receives a specific element of the type Int to which the function succ is applied twice.

An extra type application is required for twice2 of id, which has to be first specialized to Int:

```
twice2[Int](id[Int])(3)
```

Note that both λ-abstraction (function abstraction) and universal quantification (generic type abstraction) are binding operators that require formal parameters to be replaced by actual parameters. Separation between types and values is achieved by having different binding operations for types and values and different parenthesis syntax when actual parameters are supplied.

The extension of the λ-calculus to support two different kinds of binding mechanism, one for types and one for variables, is both practically useful in modeling parametric polymorphism and mathematically interesting in generalizing the λ-calculus to model two qualitatively different kinds of abstraction in the same mathematical model. In the next few sections we introduce still a third kind of abstraction and associated binding mechanism, but first we have to introduce the notion of parametric types.

In Fun, types and values are rigorously distinguished (values are objects and types are sets); hence we need two distinct binding mechanisms: fun and all. These two kinds of bindings can be unified in some type models where types are values, achieving some economy of concepts, but this unification does not fit our underlying semantics. In such models it is also possible to unify the parametric type-binding mechanism described in the next section with fun and all.

4.2. Parametric Types

If we have two type definitions with similar structure, for example:

```
type BoolPair = Bool × Bool
type IntPair = Int × Int
```

we may want to factor the common structure in a single *parametric* definition and use the parametric type in defining other types:

```
type Pair[T] = T × T
type PairOfBool = Pair[Bool]
type PairOfInt = Pair[Int]
```

A type definition simply introduces a new name for a type expression and it is equivalent to that type expression in any context. A type definition does not introduce a *new* type. Hence 3,4 is an IntPair because it has type Int × Int, which is the definition of IntPair.

A parametric type definition introduces a new *type operator.* Pair above is a type operator mapping any type T to a type T × T. Hence Pair[Int] is the type Int × Int, and it follows that 3,4 has type Pair[Int].

Type operators are not types: they operate on types. In particular, one should not confuse the following notations:

```
type A[T] = T → T
type B = ∀T. T → T
```

where A is a type operator which, when applied to a type T, gives the type of functions from T to T, and B is the type of the identity function and is never applied to types.

Type operators can be used in recursive definitions, as in the following definition of generic lists. Note that we cannot think of List[Item] below as an abbreviation which has to be *macro-expanded* to obtain the real definition (this would cause an infinite expansion). Rather, we should think of List as a new type operator which is recursively defined and maps any type to lists of that type:

```
rec type List[Item] =
     [nil: Unit,
      cons: {head: Item, tail: List[Item]}
     ]
```

A generic empty list can be defined, and then specialized, as:

```
value nil = all Item. [nil = ()]
value intNil = nil[Int]
value boolNil = nil[Bool]
```

Now, [nil = ()] has type List[Item], for any Item (as it matches the definition of List[Item]). Hence the types of the generic nil and its specializations are:

```
nil : ∀Item. List[Item]
intNil : List[Int]
boolNil : List[Bool]
```

Similarly, we can define a generic cons function, and other list operations:

```
value cons : ∀Item. (Item × List[Item]) → List[Item] =
     all Item.
         fun (h: Item, t: List[Item])
              [cons = {head = h, tail = t}]
```

Note that cons can only build homogeneous lists, because of the way its arguments and result are related by the same Item type.

We should mention that there are problems in deciding, in general, when two parametric recursive type definitions represent the same type. [Solomon 78] describes the problem and a reasonable solution which involves restricting the form of parametric type definitions.

5. Existential Quantification

Type specifications for variables of a universally quantified type have the following form, for any type expression t(a):

p: ∀a. t(a) (e.g. id: ∀a. a → a)

By analogy with universal quantification, we can try to give meaning to existentially quantified types. In general, for any type expression t(a),

p : ∃a. t(a)

has the property

For some type a, p has the type t(a)

For example:

(3,4): ∃a. a × a
(3,4): ∃a. a

where a = Int in the first case, and a = Int × Int in the second.

Thus we see that a given constant such as (3,4) can satisfy many different existential types. (Warning: for didactic purposes we assign here existential types to ordinary values, like (3,4). Although this is conceptually correct, in later sections it will be disallowed for type-checking purposes, and we shall require using particular constructs to obtain objects of existential type).

Every value has type ∃a. a because for every value there exists a type such that that value has that type. Thus the type ∃a. a denotes the set of all values, which we shall sometime call Top (the biggest type):

```
type Top = ∃a. a          -- the type of any value whatsoever
```

The set of all ordered pairs may be denoted by the following existential type.

```
∃a. ∃b. a × b             -- the type of any pair whatsoever
```

This is the type of any pair p,q because, for some type a (take a type of p) and some type b (take a type of q), p,q has type a × b.

The type of any object together with an integer-valued operation that can be applied to it may be denoted by the following existential type.

∃a. a × (a → Int)

The pair (3,succ) has this type, if we take a = Int. Similarly the pair ([1;2;3],length) has this type, if we take a = List[Int].

Because the set of types includes not only simple types but also universal types and the type Top, existentially quantified types have some properties that may at first appear counterintuitive. The type ∃a. a × a is not simply the type of pairs of equal type (e.g. 3,4), as one might expect. In fact even 3,true has this type. We know that both 3 and true have type Top; hence there is a type a = Top such that 3,true : a × a. Therefore, ∃a. a × a is the type of all pairs whatsoever, and is the same as ∃a. ∃b. a × b. Similarly, any function whatsoever has type ∃a. a → a, if we take a = Top.

However, ∃a. a × (a → Int) forces a relation between the type of an object and the type of an associated integer-valued function. For example, (3,length) does not have this type (if we consider 3 as having type Top, then we would have to show that length has type Top → Int, but we only know that length: ∀a. List[a] → a maps integer lists to integers, and we cannot assume that any arbitrary object of type Top will be mapped to integer).

Not all existential types turn out to be useful. For example, if we have an (unknown) object of type ∃a. a, we have absolutely no way of manipulating it (except passing it around) because we have no information about it. If we have an (unknown) object of type ∃a. a × a, we can assume that it is a pair and apply fst and snd to it, but then we are stuck because we have no information about a.

Existentially typed objects can be useful, however, if they are sufficiently structured. For example, x : ∃a. a × (a → Int) provides sufficient structure to allow us to compute with it. We can execute:

```
(snd(x)) (fst(x))
```

and obtain an integer.

Hence, there are *useful* existential types which hide some of the structure of the objects they represent but show enough structure to allow manipulations of the objects through operations the objects themselves provide.

These existential types can be used, for example, in forming apparently heterogeneous lists:

```
[(3,succ); ([1;2;3],length)]    : List[∃a. a × (a → Int)]
```

We can later extract an element of this list and manipulate it, although we may not know which particular element we are using and what its exact type is. Of course, we can also form totally heterogeneous lists of type List[∃a.a], but these are quite unusable.

5.1. Existential Quantification and Information Hiding

The real usefulness of existential types becomes apparent only when we realize that ∃a. a × (a → Int) is a simple example of an *abstract type* packaged with its set of operations. The variable a is the abstract type itself, which hides a representation. The representation was Int and List[Int] in the previous examples. Then a × (a → Int) is the set of operators on that abstract type: a constant of type a and an operator of type a → Int. These operators are unnamed, but we can have a named version by using record types instead of Cartesian products:

```
x: ∃a. {const: a, op: a → Int}
x.op(x.const)
```

As we do not know what the representation a really is (we only know that there is one), we cannot make assumptions about it, and users of x will be unable to take advantage of any particular implementation of a.

As we announced earlier, we have been a bit liberal in applying various operators directly to objects of existential types (like x.op above). This will be disallowed from now on, for the only purpose of making our formalism easier to type-check. Instead, we shall have explicit language constructs for creating and manipulating objects of existential types, just as we had type abstractions all[t] and type applications exp[t] for creating and using objects of universal types.

An ordinary object (3,succ) may be converted to an abstract object having type ∃a. a × (a→Int) by *packaging* it so that some of its structure is hidden. The operation pack below encapsulates the object (3,succ) so that the user knows only that an object of the type a × (a→Int) exists without knowing the actual object. It is natural to think of the resulting object as having the existential type ∃a. a × (a→Int).

```
value p = pack [a=Int in a × (a→Int)] (3,succ)    : ∃a. a × (a→Int)
```

Packaged objects such as p are called *packages*. The value (3,succ) is referred to as the *content* of the package. The type a × (a→Int) is the *interface*: it determines the structure specification of the contents and corresponds to the specification part of a data abstraction. The binding a=Int is the type *representation*: it binds the abstract data type to a particular representation Int, and corresponds to the hidden data type associated with a data abstraction.

The general form of the operation pack is as follows:

```
pack [a = typerep in interface] (contents)
```

The operation pack is the only mechanism for creating objects of an existential type. Thus, if a variable of an existential type has been declared by a declaration such as:

```
p : ∃a. a × (a → Int)
```

then p can take only values created by a pack operation.

A package must be opened before it can be used:

```
open p as x in (snd(x))(fst(x))
```

Opening a package introduces a name x for the contents of the package which can be used in the scope following in. When the structure of x is specified by labeled components, components of the opened package may be referred to by name:

```
value p = pack [a = Int in {arg:a, op:a→Int}] (3, succ)
open p as x in x.op(x.arg)
```

We may also need to refer to the (unknown) type hidden by the package. For example, suppose we wanted to apply the second component of p to a value of the abstract type supplied as an external argument. In this case the unknown type b must be explicitly referred to and the following form can be used:

```
open p as x [b] in ... fun(y:b) (snd(x))(y) ...
```

Here the type name b is associated with the hidden representation type in the scope following in. The type of the expression following in must not contain b, to prevent b from escaping its scope.

The function of open is mostly to bind names for representation types and to help the type checker in verifying type constraints. In many situations we may want to abbreviate open p as x in x.a to p.a. We are going to avoid such abbreviations to prevent confusion, but they are perfectly admissible.

Both pack and open have no run-time effect on data. Given a smart enough type checker, one could omit these constructs and revert to the notation used in the previous section.

5.2. Packages and Abstract Data Types

In order to illustrate the applicability of our notation to *real* programming languages, we indicate how records with function components may be used to model Ada packages and how existential quantification may be used to model data abstraction in Ada [D.O.D. 83]. Consider the type Point1 for creating geometric points of a globally defined type Point from pairs of real numbers and for selecting x and y coordinates of points.

```
type Point = Real × Real
type Point1  =
        {makepoint: (Real × Real) → Point,
          x_coord: Point → Real,
          y_coord: Point → Real
        }
```

Values of the type Point1 can be created by initializing each of the function names of the type Point1 to functions of the required type.

```
value point1 : Point1 =
        {makepoint = fun(x:Real,y:Real) (x,y),
         x_coord = fun(p:Point) fst(p),
         y_coord = fun(p:Point) snd(p)
        }
```

In Ada, a package point1 with makepoint, x_coord, and y_coord functions may be specified as follows:

```
package point1 is
        function makepoint (x:Real, y:Real) return Point;
        function x_coord (P:Point) return Real;
        function y_coord (P:Point) return Real;
end point1;
```

This package specification is not a type specification but part of a value specification. In order to complete the value specification in Ada, we must supply a package body of the following form:

```
package body point1 is
        function makepoint (x:Real, y:Real) return Point;
              -- implementation of makepoint
        function x_coord (P:Point) return Real;
              -- implementation of x_coord
        function y_coord (P:Point) return Real;
              -- implementation of y_coord
end point1;
```

The package body supplies function bodies for function types of the package specification. In contrast to our notation, which allows different function bodies to be associated with different values of the type, Ada does not allow packages to have types, and directly defines the function body for each function type in the package body.

Packages allow the definition of groups of related functions that share a local hidden data structure. For example a package localpoint with a local data structure point has the following form:

```
package body localpoint is
     point: Point;   -- shared global variable of makepoint, x_coord, y_coord
     procedure makepoint(x,y: Real); ...
     function x_coord return Real; ...
     function y_coord return Real; ...
end localpoint;
```

Hidden local variables can be realized in our notation by the let construct:

```
value localpoint =
      let p: Point = ref((0,0))
      in   {makepoint = fun(x: Real, y: Real) p := (x, y),
              x_coord = fun() fst(p),
              y_coord = fun() snd(p)
            }
```

Although Ada does not have the concept of a package type it does have the notion of a package template, which has some, but not all, the properties of a type. Package templates are introduced by the keyword generic.

```
generic
    package Point1 is
         function makepoint (x:Real, y:Real) return Point;
         function x_coord (P:Point) return Real;
         function y_coord (P:Point) return Real;
    end Point1;
```

Values point1 and point2 of the generic package template Point1 can be introduced as follows:

```
package point1 is new Point1;
package point2 is new Point1;
```

All package values associated with a given generic package template have the same package body. The specification of an Ada package is statically associated with its body prior to execution, while the typed values of record types are dynamically associated with function bodies when the value-creation command is executed.

Components of package values created from a generic package can be accessed using the record notation.

```
type p is Point;
p = point1.makepoint(3,4);
```

Thus packages are like record values in allowing their components to be accessed by the same notation as is used for selection of record components. But packages are not first-class values in Ada. They cannot be passed as parameters of procedures, cannot be components of arrays or record data structures, and cannot be assigned as values of package variables. Moreover, generic package templates are not types, although they are like types in allowing instances to be created. In effect, Ada has two similar but subtly different language mechanisms for handling record-like structures, one for handling data records with associated record types, and one for handling packages with associated generic templates. By contrasting the two mechanisms of Ada for record types and generic packages with the single mechanism of our notation we gain appreciation of and insight into the advantages of uniformly extending types to records with function components.

Ada packages which simply encapsulate a set of operations on a publicly defined datatype, do not need fancy type operators. They can be modelled in our notation by the simple typed λ-calculus without existential quantification. It is only when we hide the type representation using private data types that existential quantification is needed.

The let construct was used in the previous example to realize information hiding. We call this *first order* information hiding because it is achieved by restricting scoping at the value level. This is contrasted to *second order* information hiding that is realized by existential quantifiers, which restrict scoping at the type level.

An Ada point package point2 with a private type Point may be defined as follows:

```
package point2
     type Point is private;
     function makepoint (x:Real, y:Real) return Point;
     function x_coord (P:Point) return Real;
```

```
            function y_coord (P:Point) return Real;
            private
                -- hidden local definition of the type Point
        end point2;
```

The private type Point may be modelled by existential quantification:

```
        type Point2 =
            ∃Point.
                {makepoint: (Real × Real) → Point,
                  x_coord: Point → Real,
                  y_coord: Point → Real
                }
```

It is sometimes convenient to view the type specifications of an existentially quantified type as a parametric function of the hidden type parameter. In the present example we may define Point2WRT[Point] as follows:

```
        type Point2WRT[Point] =
            {makepoint: (Real × Real) → Point,
              x_coord: Point → Real,
              y_coord: Point → Real
            }
```

The notation WRT in Point2WRT[Point], to be read as *with respect to*, underlines the fact that this type specification is relative to a type parameter.

A value point2 of the existential type Point2 may be created by the pack operation.

```
        value point2 : Point2 = pack [Point = (Real × Real) in Point2WRT[Point]]
            point1
```

The pack operation hides the representation Real × Real of Point, has the existentially parametrized type Point2WRT[Point] as its specification part, and provides as its hidden body the previously defined value point1 that implements operations for the given data representation.

Note that Point2WRT[Point] represents a parameterized type expression which, when supplied with an actual type parameter such as Real, determines a type (in this case a record type with three components). The relation between this kind of parameterization and the other kinds of parameterization introduced so far is illustrated by the following table:

1. Function abstraction: fun(x: type) value-expr(x). The parameter x is a value and the result of substituting an actual parameter for the formal parameter determines a value.

2. Quantification: all(a) value-expr(a). The parameter a is a type and the result of substituting an actual type for the formal parameter determines a value.

3. Type Abstraction: TypeWRT[T]   = type-expr(T). The parameter T is a type and the result of substituting an actual type for the formal parameter is also a type.

Actual type parameters are restricted to be types, while actual value parameters may be arbitrarily complex values. However, when the class of namable types is enriched to include universally and existentially quantified types, this also enriches the arguments that may be substituted for formal type parameters.

Existential quantification can be used to model the private types of Ada. However, it is much more general than the data abstraction facility of Ada, as shown in the following examples.

5.3. Combining Universal and Existential Quantification

In this section we give an example that demonstrates the interaction between universal and existential quantification. Universal quantification yields generic types while existential quantification yields abstract data types. When these two notions are combined we obtain parametric data abstractions.

Stacks are an ideal example to illustrate the interaction between generic types and data abstraction. The simplest form of a stack has both a specific element type such as *integer* and a specific data structure implementation such as a *list* or an *array*. Generic stacks parameterize the element type, while abstraction from the data representation may be accomplished by creating a package that has an existential data type. A stack with parameterized element type and a hidden data representation is realized by combining universal quantification to realize the parameterization with existential quantification to realize the data abstraction.

The following operations on lists and arrays will be used:

```
nil:      ∀a. List[a]
cons:     ∀a. (a × List[a]) → List[a]
hd:       ∀a. List[a] → a
tl:       ∀a. List[a] → List[a]
null:     ∀a. List[a] → Bool

array:    ∀a. Int → Array[a]
index:    ∀a. (Array[a] × Int) → a
update:   ∀a. (Array[a] × Int × a) → Unit
```

We start with a concrete type IntListStack with integer elements and a list data representation. This concrete type can be implemented as a tuple of operations with no quantification.

```
type IntListStack =
      {emptyStack: List[Int],
       push: (Int × List[Int]) → List[Int],
       pop: List[Int] → List[Int],
       top: List[Int] → Int
      }
```

An instance of this stack type with components initialized to specific function values may be defined as follows:

```
value intListStack : IntListStack =
      {emptyStack = nil[Int],
       push = fun(a:Int,s:List[Int]) cons[Int](a,s),
       pop = fun(s:List[Int]) tl[Int](s),
       top = fun(s:List[Int]) hd[Int](s)
      }
```

We could also have a stack of integers implemented via pairs consisting of an array and a top-of-stack index into the array; this concrete stack may again be implemented as a tuple without any quantification.

```
type IntArrayStack =
      {emptyStack: (Array[Int] × Int),
       push: (Int × (Array[Int] × Int)) → (Array[Int] × Int),
       pop: (Array[Int] × Int) → (Array[Int] × Int),
       top: (Array[Int] × Int) → Int
      }
```

An instance of IntArrayStack is an instance of the above tuple type with operation fields initialized to operations on the array stack representation.

```
value intArrayStack : IntArrayStack =
      {emptyStack = (Array[Int](100),-1),
       push = fun(a:Int,s:(Array[Int] × Int))
                  update[Int](fst(s),snd(s)+1,a); (fst(s),snd(s)+1),
       pop = fun(s:(Array[Int] × Int)) (fst(s),snd(s)-1),
       top = fun(s:(Array[Int] × Int)) index[Int](fst(s),snd(s))
      }
```

The concrete stacks above may be generalized both by making the element type generic and by hiding the stack data representation. The next example illustrates how a generic element type may be realized by universal quantification. We first define the type GenericListStack as a universally quantified type:

```
type GenericListStack =
      ∀Item.
          {emptyStack: List[Item],
 push: (Item × List[Item]) → List[Item],
 pop: List[Item] → List[Item],
 top: List[Item] → Item
}
```

An instance of this universal type may be created by universal quantification of a record instance whose fields are initialized to operations parameterized by the generic universally quantified parameter.

```
value genericListStack : GenericListStack =
    all[Item]
        {emptyStack = nil[Item],
         push = fun(a:Item,s:List[Item]) cons[Item](a,s),
         pop = fun(s:List[Item]) tl[Item](s),
         top = fun(s:List[Item]) hd[Item](s)
        }
```

The genericListStack has, as its name implies, a concrete list implementation of the stack data structure. An alternative type GenericArrayStack with a concrete array implementation of the stack data structure may be similarly defined:

```
type GenericArrayStack =  ...

value genericArrayStack : GenericArrayStack = ...
```

Since the data representation of stacks is irrelevant to the user, we would like to hide it so that the stack interface is independent of the hidden stack data representation. We would like to have a single type GenericStack which can be implemented as a generic list stack or a generic array stack. Users of GenericStack should not have to know which implementation of GenericStack they are using.

This is where we need existential types. For any item type there should exist an implementation of stack which provides us with stack operations. This results in a type GenericStack defined in terms of a universally quantified parameter Item and an existentially quantified parameter Stack as follows:

```
type GenericStack =
    ∀Item. ∃Stack. GenericStackWRT[Item][Stack]
```

The two-parameter type GenericStackWRT[Item][Stack] may in turn be defined as a tuple of doubly parameterized operations:

```
type GenericStackWRT[Item][Stack] =
    {emptystack: Stack,
     push: (Item,Stack) → Stack,
     pop: Stack → Stack,
     top: Stack → Item
    }
```

Note that there is nothing in this definition to distinguish the rôle of the two parameters Item and Stack. However, in the definition of GenericStack the parameter Item is universally quantified, indicating that it represents a generic type, while the parameter Stack is existentially quantified, indicating that it represents a hidden abstract data type.

We can now abstract our genericListStack and genericArrayStack packages into packages of type GenericStack:

```
value listStackPackage : GenericStack =
    all[Item]
        pack[Stack = List[Item] in GenericStackWRT[Item][Stack]]
            genericListStack[Item]

value arrayStackPackage : GenericStack =
    all[Item]
        pack[Stack = (Array[Item] × Item) in GenericStackWRT[Item][Stack]]
            genericArrayStack[Item]
```

Both listStackPackage and arrayStackPackage have the same type and differ merely in the form of the hidden data representation.

Moreover, functions like the following useStack can work without any knowledge of the implementation:

```
value useStack =
        fun(stackPackage: GenericStack)
              open stackPackage[Int] as p [stackRep]
              in p.top(p.push(3,p.emptystack));
```

and can be given any implementation of GenericStack as a parameter:

```
useStack(listStackPackage)
useStack(arrayStackPackage)
```

In the definition of GenericStack, the type Stack is largely unrelated to Item, while it is our intention that, whatever the implementation of Stack, stacks should be collections of items (actually, there is a weak dependency of Stack upon Item given by the order of the quantifiers). Because of this, it is possible to build objects of type GenericStack where stacks have nothing to do with items, and do not obey properties like pop(push(a,s)) = a. This limitation is corrected in more powerful type systems like [MacQueen 86] and [Burstall 84], where it is possible to abstract on type operators (e.g. List) instead of just types (e.g. List[Int]), and one can directly express that representations of Stack must be based on Item (but even in those more expressive type systems it is possible to *fake* stack packages which do not obey stack properties).

5.4. Quantification and Modules

We are now ready for a major example: geometric points. We introduce an abstract type with operations mkpoint (make a new point from two real numbers), x-coord and y-coord (extract the x and y coordinates of a point):

```
type Point =
        ∃PointRep.
              {mkpoint: (Real × Real) → PointRep,
               x-coord: PointRep → Real,
               y-coord: PointRep → Real
              }
```

Our purpose is to define values of this type that hide both the representation PointRep and the implementation of the operations mkpoint, x-coord, and y-coord with respect to this representation. In order to accomplish this we define the type of these operations as a parametric type with the point representation PointRep as a parameter. The type name PointWRT emphasizes that the operations are defined with respect to a particular representation and that in contrast the abstract datatype Point is representation-independent.

```
type PointWRT[PointRep] =
        {mkpoint: (Real × Real) → PointRep,
         x-coord: PointRep → Real,
         y-coord: PointRep → Real
        }
```

The existential type Point may be defined in terms of PointWRT by existential abstraction with respect to PointRep:

```
type Point = ∃PointRep. PointWRT[PointRep]
```

The relationship between representation-dependent point operations and the associated abstract datatype becomes even clearer when we illustrate the abstraction process for some specific point representations. Let's define a Cartesian point package whose point representation is by pairs of reals and whose operations mkpoint, x-coord, y-coord are as follows:

```
value cartesianPointOps =
        {mkpoint = fun (x:Real, y:Real) (x,y),
         x-coord = fun (p: Real × Real) fst(p),
         y-coord = fun (p: Real × Real) snd(p)
        }
```

A package with point representation Real × Real and with the above implementations of point operations as its content can be specified as follows:

```
value cartesianPointPackage =
        pack [PointRep = Real × Real in PointWRT[PointRep]]
              cartesianPointOps
```

Similarly we can make a polar point package whose point representation Real × Real is the same as that for the Cartesian point package but whose content is a different (polar-coordinate) implementation of the operations:

```
value polarPointPackage =
        pack[PointRep = Real × Real  in  PointWRT[PointRep]]
              {mkpoint = fun (x:Real, y:Real) ... ,
               x-coord = fun (p: Real × Real) ... ,
               y-coord = fun (p: Real × Real) ...
              }
```

These examples illustrate how a package realizes data abstraction by hiding both the data representation and the implementation of its operations. The Cartesian and polar packages have the same existential type Point, use the same parametric type PointWRT[PointRep] to specify the structure of point operations, and have the same type Real × Real for data representation. They differ only in the content of the package that determines the function implementations. In general, a given existential type forces all packages of that type to have the same structure for operations. But both the type of the internal data representation and the value (implementation) of the operations may differ for different realizations of an abstract data type.

An abstract data type packaged with its operators, like Point, is also a simple example of a *module*. In general modules can import other (known) modules, or can be parameterized with respect to other (as yet unknown) modules.

Parametric modules can be treated as functions over existential types. Here is a way of extending the Point package with another operation (add). Instead of doing this extension for a particular Point package, we write a procedure to do the extension for any Point package over an unknown representation of point. Recall that & is the record type concatenation operator:

```
type ExtendedPointWRT[PointRep] =
        PointWRT[PointRep] & {add: (PointRep × PointRep) → PointRep}

type ExtendedPoint = ∃PointRep. ExtendedPointWRT[PointRep]

value extendPointPackage =
        fun (pointPackage: Point)
              open pointPackage as p [PointRep] in
                    pack[PointRep' = PointRep in ExtendedPointWRT[PointRep']]
                          p &
                                {add = fun (a:PointRep, b:PointRep)
                                 p.mkpoint(p.x-coord(a)+p.x-coord(b),
                                 p.y-coord(a)+p.x-coord(b))
                                }

value extendedCartesianPointPackage =
        extendPointPackage(cartesianPointPackage)

value extendedPolarPointPackage =
        extendPointPackage(polarPointPackage)
```

We now go back to the Point module and show how other modules can be built on top of it. In particular, we build modules Circle and Rectangle on top of Point and then define a module Picture which uses both Circle and Rectangle. As different instances of Point may be based on different data representations, we have to make sure that circles and rectangles are based on the same representation of Point, if we want to make them interact.

A circle package provides operations to create a circle out of a point (the center) and a real (the radius), and operations to extract the center and the radius of a circle. An operation diff of circle difference (distance between the centers of two circles) is also defined. The two parameters to diff are circles based on the same implementation of Point. A circle package also provides a point package, to allow one to access point operations working on the same representation of point used in the circle package.

```
type CircleWRT2[CircleRep,PointRep] =
        {pointPackage: PointWRT[PointRep],
         mkcircle: (PointRep × Real) → CircleRep,
         center: CircleRep → PointRep,
         radius: CircleRep → Real,
         diff: (CircleRep × CircleRep) → Real
        }

type CircleWRT1[PointRep] =
        ∃CircleRep. CircleWRT2[CircleRep,PointRep]

type Circle = ∃PointRep. CircleWRT1[PointRep]

type CircleModule =
        ∀PointRep. PointWRT[PointRep] → CircleWRT1[PointRep]

value circleModule : CircleModule =
        all[PointRep]
              fun (p: PointWRT[PointRep])
                      pack[CircleRep = PointRep × Real in CircleWRT2[CircleRep,PointRep]]
                            {pointPackage = p,
                             mkcircle = fun (m:PointRep,r:Real) (m,r),
                             center = fun (c: PointRep × Real) fst(c),
                             radius = fun (c: PointRep × Real) snd(c),
                             diff = fun (c1: PointRep × Real, c2: PointRep × Real)
                                          let p1 = fst(c1)
                                          and p2 = fst(c2)
                                          in sqrt((p.x-coord(p1) - p.x-coord(p2))**2+
                                                (p.y-coord(p1) - p.y-coord(p2))**2)
                            }
```

We can now build some particular circle packages by applying circleModule to various point packages. We could also define different versions of circleModule based on different representations of circle, and all of those could be applied to all the different point packages to obtain circle packages. Here we apply circleModule to cartesianPointPackage and to polarPointPackage to obtain cartesian and polar circle packages.

```
value cartesianCirclePackage =
        open cartesianPointPackage as p [Rep] in
              pack[PointRep = Rep in CircleWRT1[PointRep]]
                      circleModule[Rep](p)

value polarCirclePackage =
        open polarPointPackage as p [Rep] in
              pack[PointRep = Rep in CircleWRT1[PointRep]]
                      circleModule[Rep](p)
```

To use a circle package we have to open it. We actually have to open it twice (note that the type Circle has a double existential quantification) to bind PointRep and CircleRep to the point and circle representations used in that package. Here we use an abbreviated form of open which is equivalent to two consecutive opens:

```
open cartesianCirclePackage as c [PointRep] [CircleRep]
in ... c.mkcircle(c.pointPackage.mkpoint(3,4),5) ...
```

A rectangle is determined by two points: the upper left and the bottom right corner. The definition of rectangle module is very similar to that of the circle module. In addition, we have to make sure that the two points determining a rectangle are based on the same representation of Point.

```
type RectWRT2[RectRep,PointRep] =
        {pointPackage: PointWRT[PointRep],
         mkrect: (PointRep × PointRep) → RectRep,
         toplft: RectRep → PointRep,
         botrht: RectRep → PointRep
        }
```

```
type RectWRT1[PointRep] =
        ∃RectRep. RectWRT2[RectRep,PointRep]

type Rect = ∃PointRep. RectWRT1[PointRep]

type RectModule =
        ∀PointRep. PointWRT[PointRep] → RectWRT1[PointRep]

value rectModule =
        all[PointRep]
              fun (p: PointWRT[PointRep])
                     pack[PointRep = PointRep in RectWRT1[PointRep]]
                           {pointPackage = p,
                            mkrect = fun (tl: PointRep, br: PointRep) (tl,br),
                            toplft = fun (r: PointRep × PointRep) fst(r),
                            botrht = fun (r: PointRep × PointRep) snd(r)
                           }
```

We now put it all together in a module of figures, which uses circles and rectangles (based on the same implementation of point) and defines an operation boundingRect which returns the smallest rectangle containing a given circle.

```
type FiguresWRT3[RectRep,CircleRep,PointRep] =
        {circlePackage: CircleWRT[CircleRep,PointRep]
         rectPackage: RectWRT[RectRep,PointRep]
         boundingRect: CircleRep → RectRep
        }

type FiguresWRT1[PointRep] =
        ∃RectRep. ∃CircleRep. FigureWRT3[RectRep,CircleRep,PointRep]

type Figures = ∃PointRep. FigureWRT1[PointRep]

type Figures =
        ∀PointRep. PointWRT[PointRep] → FiguresWRT1[PointRep]

value figuresModule =
        all[PointRep]
              fun (p: PointWRT[PointRep])
                     pack[PointRep = PointRep in FiguresWRT1[PointRep]]
                           open circleModule[PointRep](p) as c [CircleRep]
                           in    open rectModule[PointRep](p) as r [RectRep]
                                 in     {circlePackage = c,
                                         rectPackage = r,
                                         boundingRect =
                                              fun(c: CircleRep) ..r.mkrect(..c.center(c)..)..
                                        }
```

5.5. Modules are First-Class Values

In the previous section we have shown that packages and modules are first-class citizens: they are legal values which can be passed and returned from functions and stored in data structures. For example, it is possible to write programs which, depending on conditions, produce one or another package of the same existential type implementing an interface, and return it to be used in the construction of larger packages.

The process of linking modules can also be expressed: we have done this in the previous example, e.g., when we produced cartesianCirclePackage by linking cartesianPointPackage and circleModule. Hence, the process of building systems out of modules can be expressed in the same language used to program modules, and the full power of the language can be applied during the linking phase.

Although we have shown that we can express parametric modules and linking mechanisms, we do not claim that this is the most convenient notation to work with. Our purpose is to show that all these concepts can be captured in a relatively simple framework. There is more to be done, however, to prevent the notation from getting out of hand. The major problem here is that one must be aware of the dependency graph of modules when creating new module instances, and the linking must be done *by hand* for every new instance. These problems are particularly addressed in the Standard ML module mechanism [MacQueen 84b].

6. Bounded Quantification

6.1. Type Inclusion, Subranges, and Inheritance

We say that a type A *is included in*, or *is a subtype of* another type B when all the values of type A are also values of type B, i.e. exactly when A, considered as a set of values, is a subset of B. This general notion of inclusion specializes to different inclusion rules for different type constructors. In this section we discuss inclusions of subranges, records, variants and function types. Inclusions of universally and existentially quantified types are discussed in later sections.

As an introduction to inclusions on record types, we first present a simple theory of inclusions on integer subrange types. Let n..m denote the subtype of the type Int associated with the subrange n to m, extremes included, where n and m are known integers. The following type inclusion relations hold for integer subrange types:

```
n..m ≤ n'..m'    iff    n' ≤ n   and   m ≤ m'
```
where the ≤ on the left is type inclusion and those on the right are *less or equal to*.

Subrange types may occur as type specifications in λ-expressions:

```
value f = fun (x: 2..5) x + 1
f : 2..5 → 3..6
f(3)
```

The constant 3 has the type 3..3 and also has the type of any supertype, including the type 2..5 of x above. It is therefore a legal argument of f. Similarly the following should be legal:

```
value g = fun (y: 3..4) f(y)
```

as the type of y is a subtype of the domain of f. An actual parameter of an application can have any subtype of the corresponding formal parameter.

Consider a function of type 3..7 → 7..9. This can also be considered a function of type 4..6 → 6..10, as it maps integers between 3 and 7 (and hence between 4 and 6) to integers between 7 and 9 (and hence between 6 and 10). Note that the domain shrinks while the codomain expands. In general we can formulate the inclusion rules for functions as follows:

```
s → t   ≤   s' → t'    iff    s' ≤ s   and   t ≤ t'
```

note the (rather accidental) similarity of this rule and the rule for subranges, and how the inclusion on the domain is swapped.

The interesting point of these inclusion rules is that they also work for higher functional types. For example:

```
value h = fun (f: 3..4 → 2..7) f(3)
```

can be applied to f above:

```
h(f)
```

because of the inclusion rules for subranges, arrows and application.

The same line of reasoning applies to record types. Suppose we have types:

```
type Car = {age:Int, speed:Int, fuel:String}
type Vehicle = {age:Int, speed:Int}
```

We would like to claim that all cars are vehicles, i.e. that Car is a subtype of Vehicle. To achieve this we need the following inclusion rule for record types:

$$\{a_1{:}t_1, \ldots, a_n{:}t_n, \ldots, a_m{:}t_m\} \le \{a_1{:}u_1, \ldots, a_n{:}u_n\}$$
$$\text{iff} \quad t_i \le u_i \quad \text{for} \quad i \in 1..n.$$

i.e., a record type A is a subtype of another record type B if A has all the attributes (fields) of B, and possibly more, and the types of the common attributes are respectively in the subtype relation.

The meaning of the type Vehicle is the set of all records that have at least an integer field age and an integer field speed, and possibly more. Hence any car is in this set, and the set of all cars is a subset of the set of all vehicles. Again, subtypes are subsets.

Subtyping on record types corresponds to the concept of inheritance (subclasses) in languages, especially if records are allowed to have functional components. A class instance is a record with functions and local variables, and a subclass instance is a record with at least those functions and variables, and possibly more.

In fact we can also express multiple inheritance. If we add the type definitions:

```
type Object = {age:Int}
type Machine = {age:Int, fuel:String}
```

then we have that car is a subtype (inherits properties from) both vehicle and machine, and those are both subtypes of object. Inheritance on records also extends to higher functional types, as in the case of subranges, and the inclusion rule for function spaces is also maintained.

In the case of variant types, we have the following inclusion rule:

$$[a_1{:}t_1, \ldots, a_n{:}t_n] \le [a_1{:}u_1, \ldots, a_n{:}u_n, \ldots, a_m{:}u_m]$$
$$\text{iff} \quad t_i \le u_i \quad \text{for} \quad i \in 1..n.$$

For example, every bright color is a color:

```
type brightColor = [red:Unit, green:Unit, blue:Unit]
type color = [red:Unit, green:Unit, blue:Unit, gray:Unit, brown:Unit]
```

and any function working on colors will be able to accept a bright color.

More detailed examples of this kind of inheritance can be found in the first half of [Cardelli 84b].

6.2. Bounded Universal Quantification and Subtyping

We now come to the problem of how to mix subtyping and parametric polymorphism. We have seen the usefulness of those two concepts in separate applications; and we shall now show that it is useful, and sometimes necessary, to merge them.

Let us take a simple function on records of one component:

```
value f_0 = fun(x: {one: Int}) x.one
```

which can be applied to records like {one = 3, two = true}. This can be made polymorphic by:

```
value f = all[a] fun(x: {one: a}) x.one;
```

We can use f[t] on records of the form {one = y} for any y of type t, and on records like {one = y, two = true}.

The notation all[a]e allows us to express the notion that a type variable ranges over all types but does not allow us to designate type variables that range over a subset of the set of types. A general facility for specifying variables that range over arbitrary subsets of types could be realized by quantification over type sets defined by specified predicates. However we do not need this generality and can be satisfied with specifying just a particular class of subsets – namely the set of all subtypes of a given type. This may be accomplished by bounded quantification.

A type variable ranging over the set of all subtypes of a type T may be specified by bounded quantification as follows:

```
all[a ≤ T] e                -- a ranges over all subtypes of T in the scope e
```

Here is a function which accepts any record having integer component one and extracts its contents:

```
value g₀ = all[a ≤ {one: Int}] fun(x: a) x.one
g₀ [{one: Int, two: Bool}]({one=3, two=true})
```

Note that there is little difference between g_0 and f_0; all we have done is to move the constraint that the argument must be a subtype of {one :Int} from the fun parameter to the all parameter. We now have two ways of expressing inclusion constraints: implicitly by function parameters and explicitly by bounded quantifiers. Now that we have bounded quantifiers we could remove the other mechanism, requiring exact matching of types on parameter passing, but we shall leave it for convenience.

To express the type of g_0, we need to introduce bounded quantification in type expressions:

```
g₀ : ∀a ≤ {one: Int}. a → Int
```

Now we have a way of expressing both inheritance and parametric polymorphism. Here is a new version of g_0 in which we abstract Int to any type:

```
value g = all[b] all[a ≤ {one: b}] fun(x: a) x.one
g[Int][{one: Int, two: Bool}]({one=3, two=true})
```

where all[b] e is now an abbreviation for all[b ≤ Top] e. The new function g could not be expressed by parametric polymorphism or by inheritance separately. Only their combination, achieved by bounded quantifiers, allows us to write it.

So far, bounded quantifiers have not shown any extra power, because we can rephrase g_0 as f_0 and g as f, given that we allow type inclusion on parameter passing. But bounded quantifiers are indeed more expressive, as is shown in the next example.

The need for bounded quantification arises very frequently in object-oriented programming. Suppose we have the following types and functions:

```
type Point = {x: Int, y: Int}

value moveX₀ = fun(p:Point, dx:Int) p.x := p.x + dx; p
value moveX = all[P ≤ Point] fun(p:P, dx:Int) p.x := p.x + dx; p
```

It is typical in (type-free) object-oriented programming to reuse functions like moveX on objects whose type was not known when moveX was defined. If we now define:

```
type Tile = {x: Int, y:Int, hor: Int, ver:Int}
```

we may want to use moveX to move tiles, not just points. However, if we use the simpler $moveX_0$ function, it is only sound to assume that the result will be a point, even if the parameter was a tile and we allow inclusion on function arguments. Hence, we lose type information by passing a tile through the $moveX_0$ function and, for example, we cannot further extract the hor component from the result.

Bounded quantification allows us to better express input/output dependencies: the result type of moveX will be the same as its argument type, whatever subtype of Point that happens to be. Hence we can apply moveX to a tile and get a tile back without losing type information:

```
moveX[Tile]({x=0,y=0,hor=1,ver=1},1).hor
```

This shows that bounded quantification is useful even in the absence of proper parametric polymorphism to express adequately subtyping relations.

Earlier we saw that parametric polymorphism can be either explicit (by using ∀ quantifiers) or implicit (by having free type variables, implicitly quantified). We have a similar situation here, where inheritance can be either explicit, by using bounded quantifiers, or left implicit in the inclusion rules for parameter passing. In object-oriented languages, subtype parameters are generally implicit. We may consider such languages as abbreviated version of languages using bounded quantification. Thus, bounded quantification is useful not only to increase expressive power, but also to make explicit the parameter mechanisms through which inheritance is achieved.

6.3. Comparison with Other Subtyping Mechanisms

How does the above inheritance mechanisms compare with those in Simula, Smalltalk and Lisp Flavors? For many uses of inheritance the correspondence is not exact, although it can be obtained by paraphrases. Other uses of inheritance cannot be simulated, especially those which make essential use of dynamic typing. On the other hand, there are some things that can be done with bounded quantification, which are impossible in some object-oriented languages.

Record types are used to model classes and subclasses. Record types are matched by structure, and the (multiple) inheritance relations are implicit in the names and types of record components. In Simula and Smalltalk classes are matched by name, and the inheritance relations are explicit; only single inheritance is allowed. Lisp Flavors allow a form of multiple inheritance. Smalltalk's *metaclasses* cannot be emulated in the present framework.

Records are used to model class instances. Records have to be constructed explicitly (there is no *create new instance of class X* primitive), by specifying at construction time the values of the components. Hence, different records of the same record type can have different components; this gives a degree of flexibility which is not shared by Simula and Smalltalk. Simula distinguishes between functional components (operations), which must be shared by all the instances of a class, and non-functional components (variables) which belong to instances. Simula's *virtual* procedures are a way of introducing functional components that may change in different instances of a class, but must still be uniform within subclasses of that class. Smalltalk also distinguishes between *methods*, shared by all instances of a class, and *instance variables*, local to instances. Unlike Simula's variables declared in classes, Smalltalk instance variables are *private* and cannot be directly accessed. This behavior can be easily obtained in our framework by limiting visibility of local variables via static scoping techniques.

Functional record components are used to model *methods*. As remarked in the previous paragraph, record components are conceptually bound to individual records, not to record types (although implementations can optimize this). In Simula and Smalltalk it is possible for a subclass automatically to inherit the methods of its superclass, or to redefine them. When considering multiple inheritance, this automatic way of inheriting methods creates problems in case more than one superclass defines the same method: which one should be inherited? We avoid this problem by having to create records explicitly. At record creation time one must choose explicitly which field values a particular record should have; whether it should *inherit* them by using some predefined function (or value) used in the allocation of other records, or *redefine* them, by using a new function (or value). Everything is allowed as long as the type constraints are respected.

Record field selection is used to model *message passing*. A message sent to an object with some parameters translates to the selection of a functional component of a record and its application to the parameters. This is very similar to what Simula does, while Smalltalk goes through a complex name-binding procedure to associate message names with actual methods. Simula can compute statically the precise location of a variable or operation in an instance. Smalltalk has to do a dynamic search, which can be optimized by caching recently used methods. Our field selections have intermediate complexity: because of multiple inheritance it is not possible to determine statically the precise location of a field in a record, but caching can achieve an almost constant-time access to fields, on the average, and achieves exactly constant time in programs which only use single inheritance.

Smalltalk's concept of *self*, corresponding to Simula's *this* (a class instance referring to its own methods and variables), can also be simulated without introducing any special construct. This can be done by defining a record recursively, so that an ordinary variable called *self* (though we could use a different name) refers to the record itself, recursively. Smalltalk's concept of *super* (a class instance referring to the methods of its immediate superclass) and similar constructs in Simula (*qua*) and Flavors cannot be simulated because they imply an explicit class hierarchy.

Simula has a special construct, called *inspect*, which is essentially a case statement on the class of an object. We have no way of emulating this directly; it turns out, however, that *inspect* is often used because Simula does not have variant types. Variant types in Simula are obtained by declaring all the variant cases as subclasses of an (often) dummy class and then doing an *inspect* on objects of that class. As we have variants, we just have to rephrase the relevant Simula classes and subclasses as variants and then use an ordinary *case* for discrimination.

Smalltalk and Lisp Flavors have some idioms which cannot be reproduced because they are essentially impossible to type-check statically. For example, in Flavors one can ask whether an object supports a message (although it may be possible to paraphrase some of these situations by variant types). Generally, the freedom of type-free languages is hard to match, but we have shown in previous sections that polymorphism can go a long way in achieving flexibility, and bounded quantification can extend that flexibility to inheritance situations.

6.4. Bounded Existential Quantification and Partial Abstraction

As we have done for universal quantifiers, we can modify our existential type quantifiers, restricting an existential variable to be a subtype of some type:

```
∃a ≤ t. t'
```

We retain the notation ∃a. t as an abbreviation for ∃a ≤ Top. t.

Bounded existentials allow us to express *partially abstract* types: although a is abstract, we know it is a subtype of t, so it is no more abstract than t is. If t is itself an abstract type, we know that those two abstract types are in a subtype relation.

We can see this in the following example, in which we use a version of the pack construct modified for bounded existentials:

```
pack [a ≤ t = t' in t''] e
```

Suppose we have two abstract types, Point and Tile, and we want to use them and make them interact with each other. Suppose also that we want Tile to be a subtype of Point, but we do not want to know *why* the inclusion holds, because we want to use them abstractly. We can satisfy these requirements by the following definition:

```
type Tile = ∃P. ∃T ≤ P. TileWRT2[P,T]
```

Hence, there is a type P (point) such that there is a type T (tile) subtype of P which supports tile operations. More precisely:

```
type TileWRT2[P,T] =
      {mktile: (Int × Int × Int × Int) → T,
       origin: T → P,
       hor: T → Int,
       ver: T → Int
      }

type TileWRT[P] = ∃T ≤ P. TileWRT2[P,T]
type Tile = ∃P. TileWRT[P]
```

A tile package can be created as follows, where the concrete representations of points and tiles are as in the previous sections:

```
type PointRep = {x:Int,y:Int}
type TileRep = {x:Int,y:Int,hor:Int,ver:Int}

pack [P = PointRep in TileWRT[P]]
pack [T ≤ PointRep = TileRep in TileWRT2[P,T]]
    {mktile = fun(x:Int,y:Int,hor:Int,ver:Int) <x=x,y=y,hor=hor,ver=ver>,
     origin = fun(t:TileRep) t,
     hor = fun(t:TileRep) t.hor,
     ver = fun(t:TileRep) t.ver
    }
```

Note that origin returns a TileRep (a PointRep is expected), but tiles can be considered as points.

A function using abstract tiles can treat them as points, although how tiles and points are represented, and why tiles are subtypes of points, are unknown:

```
fun(tilePack:Tile)
    open tilePack as t [pointRep] [tileRep]
        let f = fun(p:pointRep) ...
        in f(t.tile(0,0,1,1))
```

In languages with both type inheritance and abstract types, it is natural to be able to extend inheritance to abstract types without having to reveal the representation of types. As we have just seen, bounded existential quantifiers can explain these situations and achieve a full integration of inheritance and abstraction.

7. Type Checking and Type Inference

In conventional typed languages, the compiler assigns a type to every expression and subexpression. However, the programmer does not have to specify the type of every subexpression of every expression: type information need only be placed at critical points in a program, and the rest is deduced from the context. This deduction process is called *type inference*. Typically, type information is given for local variables and for function arguments and results. The type of expressions and statements can then be inferred, given that the type of variables and basic constants is known.

Type inference is usually done bottom-up on expression trees. Given the type of the leaves (variables and constants) and type rules for the ways of combining expressions into bigger expressions, it is possible to deduce the type of whole expressions. For this to work it is sufficient to declare the type of newly introduced variables. Note that it may not be necessary to declare the return type of a function or the type of initialized variables.

```
fun (x:Int) x+1
let x = 0 in x+1
```

The ML language introduced a more sophisticated way of doing type inference. In ML it is not even necessary to specify the type of newly introduced variables, so that one can simply write:

```
fun (x) x+1
```

The type inference algorithm still works bottom-up. The type of a variable is initially taken to be unknown. In x+1 above, x would initially have type a, where a is a new *type variable* (a new type variable is introduced for every program variable). Then the Int operator would retroactively force a to be equivalent to Int. This instantiation of type variables is done by Robinson's unification algorithm [Robinson 65], which also takes care of propagating information across all the instances of the same variable, so that incompatible uses of the same variable are detected. An introductory exposition of polymorphic type inference can be found in [Cardelli 84a].

This inference algorithm is not limited to polymorphic languages. It could be added to any monomorphic typed language, with the restriction that at the end of type checking all the type variables should disappear. Expressions like `fun (x) x` would be ambiguous, and one would have to write `fun (x:Int) x`, for example, to disambiguate them.

The best type inference algorithm known is the one used in ML and similar languages. This amounts to saying that the best we know how to do is type inference for type systems with little existential quantification, no subtyping, and with a limited (but quite powerful) form of universal quantification. Moreover, in many extensions of the ML type system the type-checking problem has been shown to be undecidable.

Type inference reduces to type checking when there is so much type information in a program that the type inference task becomes trivial. More precisely we can talk of type checking when all the type expressions involved in checking a program are already explicitly contained in the program text, i.e., when there is no need to generate new type expressions during compilation and all one has to do is match existing type expressions.

We probably cannot hope to find fully automatic type-inference algorithms for the type system we have presented in this paper. However, the type-checking problem for this system turns out to be quite easy, given the amount of type information which has to be supplied with every program. This is probably the single most important property of this type system: it is very expressive without posing any major type-checking problem.

There is actually one problem, which is however shared by all polymorphic languages, and this has to do with type checking side-effects. Some restrictions have to be imposed to prevent violating the type system by storing and fetching polymorphic objects in memory locations. Examples can be found in [Gordon 79] and [Albano 83]. There are several known practical solutions to this problem [Damas 84] [Milner 84] which trade off flexibility with complexity of the type checker.

8. Hierarchical Classification of Type Systems

Type systems can be classified in terms of the type operators they admit. Figure 2 is a (partial) diagram of type systems ordered by *generality*. Each box in the diagram denotes a particularly clearcut type system; other type systems may fall in between. At the bottom of each box, we enumerate the type operators present in the type system (going from the bottom up, we only show the *new* operators). At the top of each box is a name for that type system and in the middle is the set of features it can model (again, going from the bottom up, we only list the *new* features). The diagram could be made more symmetrical, but it would then reflect the structure of existing classes of languages less precisely.

This is a classification of type systems, not of languages. A particular language may not fall on any particular point of this diagram, as it can have features which position it, to different degrees, at different points of the diagram. Also, existing language type systems will seldom fall exactly on one of the points we have highlighted; more often they will have a combination of features which positions them somewhere between two or more highlighted points.

At the bottom we have simple first-order type systems, with Cartesian products, disjoint sums and first-order function spaces, which can be used to model records, variants and first-order procedures, respectively. A ° sign indicates an incomplete use of a more general type operator.

First-order type systems have evolved into higher-order type systems (on the left) and inheritance-based type systems (on the right). On the left side we could find Algol 68, a higher-order monomorphic language. On the right side we could find Simula 67, a single-inheritance language, and multiple-inheritance languages higher up (again, these allocations are not so clear-cut). These two classes of type systems are dominated by higher-order inheritance systems, as in Amber [Cardelli 85].

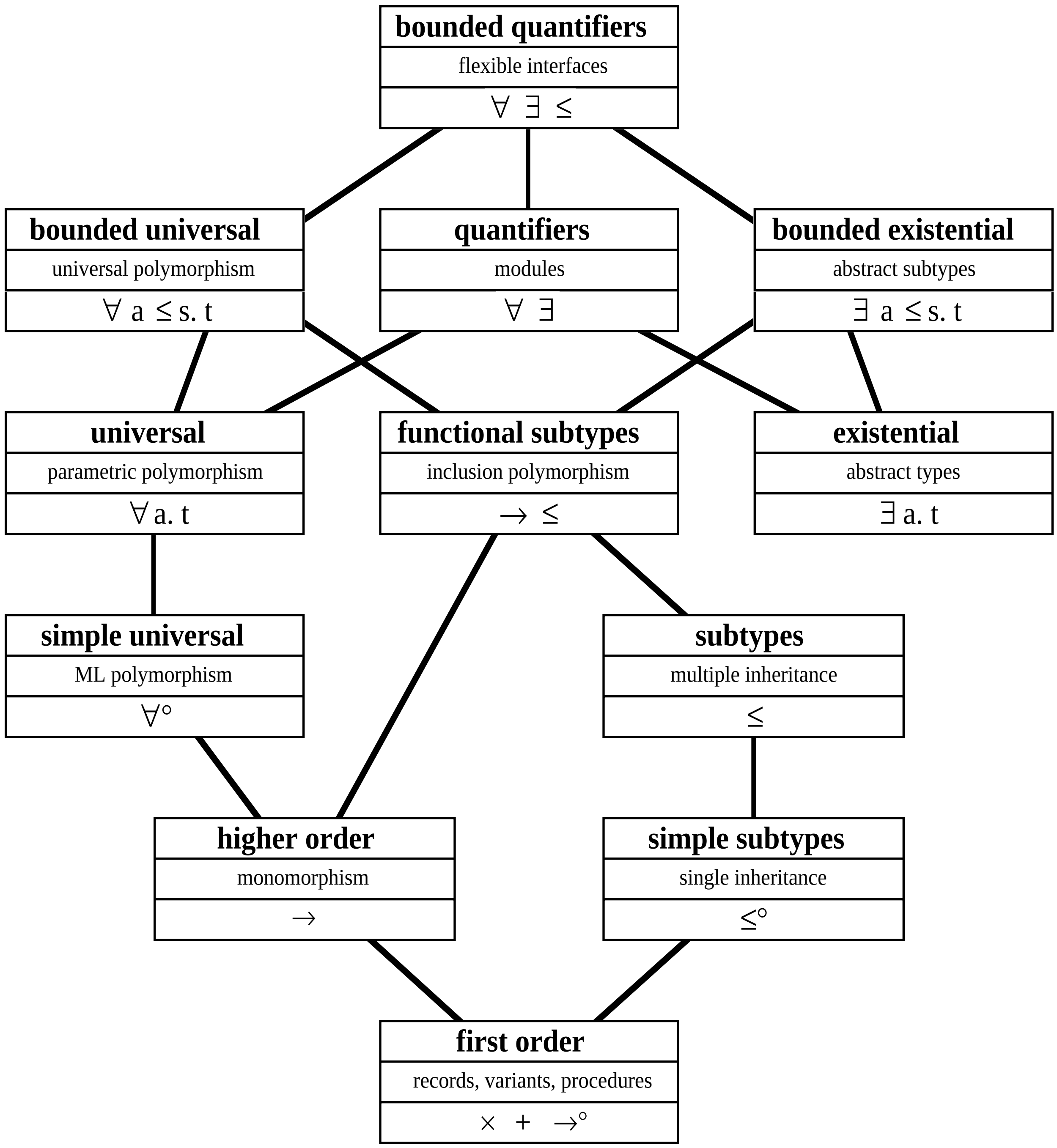


Figure 2: Classification of typ systems.

Higher-order languages have developed into parametric polymorphic languages. These can have restricted top-level universal quantification (this is Milner's type system [Milner 78], with roots in Curry [Curry 58] and Hindley [Hindley 69]) or general universal quantification (this is the Girard-Reynolds type system [Girard 71] [Reynolds 74]).

Up on the right we have type systems with type abstraction, characterized by existential quantification. Joining universal and existential quantifiers we obtain SOL's [Mitchell 85] type system, which can be used to explain basic module features.

The remaining points at the top have to do with inclusion. We have shown that the bounded universal quantifiers are needed to model object-oriented programming, and bounded existential quantifiers are needed to mix inheritance with data abstraction.

Three powerful concepts (inclusion, universal and existential quantification) are sufficient to explain most programming features. When used in full generality, they go much further than most existing languages. We have been careful to maintain the ability to type-check these features easily. However, this is not the whole picture. Many interesting type systems lie well above our diagram [Reynolds 85]. These include the Coquand and Huet theory of constructions [Coquand 85], Martin-Löf's dependent types [Martin-Löf 80], Burstall and Lampson's language Pebble [Burstall 84], and MacQueen's language DL [MacQueen 84b].

There are benefits in going even higher up: Pebble and DL have a more general treatment of parametric modules; dependent types have an almost unlimited expressive power. But there are also extra complications, which unfortunately reflect pragmatically on the complexity of type checking. The topmost point of our diagram is thus a reasonable place to stop for a while, to gain some experience, and to consider whether we are willing to accept extra complications in order to achieve extra power.

9. Conclusions

The augmented λ-calculus supports type systems with a very rich type structure in a functional framework. It can model type systems as rich, or richer, than those in real programming languages. It is sufficiently expressive to model the data abstractions of Ada and the classes of object-oriented languages. Its ability to express computations on types is strictly weaker than its ability to express computations on values.

By modeling type and module structures of real programming languages in the augmented λ-calculus, we gain an understanding of their abstract properties independent of the idiosyncrasies of programming languages in which they may be embedded. Conversely, we may view type and module structures of real programming languages as syntactically sugared versions of our augmented λ-calculus.

We started from the typed λ-calculus and augmented it with primitive types such as Int, Bool, and String and with type constructors for pairs, records, and variants.

Universal quantification was introduced to model parametric polymorphism, and existential quantification was introduced to model data abstraction. The practical application of existential quantification was demonstrated by modeling Ada packages with private data types. The usefulness of combining universal with existential abstraction was demonstrated by a generic stack example, using universal quantification to model the generic type parameter and existential quantification to model the hidden data structure.

Both universal and existential quantification become more interesting when we can restrict the domain of variation of the quantified variable. Bounded universal quantification allows more sensitive parameterization by restricting parameters to the set of all subtypes of a type. Bounded existential quantification allows more sensitive data abstraction by allowing the specification of subtyping relations between abstract types.

The insight that both subrange types of integers and subtypes defined by type inheritance are type inclusion polymorphisms extends the applicability of bounded quantification to both these cases. The case of record subtypes such as {a:T1} ≤ {a:T1, b:T2} is particularly interesting in this connection. It allows us to assert that a record type obtained by adding fields to a given record type is a subtype of that record type.

Types such as *Cars* may be modeled by record types whose fields are the set of data attributes applicable to cars. Subtypes such as *Toyotas* may be modeled by record types that include all fields of car records plus additional fields for operations applicable only to *Toyotas*. Multiple inheritance may generally be modeled by record subtypes.

Records with functional components are a very powerful mechanism for module definitions, especially when combined with mechanismd for information hiding, which are here realized by existential types. Type inclusion of records provides a paradigm for type inheritance and may be used as a basis for the design of strongly typed object-oriented languages with multiple inheritance. Although we have used a unified language (Fun) throughout the paper, we have not presented a language design for a practical programming language. In language design there are many important issues to be solved concerning readability, easy of use, etc. which we have not directly attacked.

Fun provides a framework to classify and compare existing languages and to design new languages. We do not propose it as a programming language, as it may be clumsy in many areas, but it could be the basis of one.

Appendix: Type Inference Rules

The type system discussed in this paper can be formalized as a set of type inference rules which prescribes how to establish the type of an expression from the type of its subexpressions. These rules can be intended as the specification of a typechecking algorithm. An acceptable algorithm is one which partially agrees with these rules, in the sense that if it computes a type, that type must be derivable from the rules.

The inference rules are given in two groups: the first group is for deducing that two types are in the inclusion relation, and the second group is for deducing that an expression has a type (maybe using the first group in the process).

Type expressions are denoted by s, t and u, type variables by a and b, type constants (e.g. int) by k, expressions by e and f, variables by x, and labels by l. We identify all the type expressions which differ only because of the names of bound type variables.

Here are the rules of type inclusion $t \leq s$. C is a set of *inclusion constraints* for type variables. $C.\ a \leq t$ is the set C extended with the constraint that the type variable a is a subtype of the type t.

$C \vdash t \leq s$ is an *assertion* meaning that from C we can infer $t \leq s$. A horizontal bar is a *logic implication*: if we can infer what is above it, then we can infer what is below it.

Note: this set of rules is not complete with respect to some semantic models; some valid rules have been omitted to make typechecking easier.

{TOP} $\quad C \vdash t \leq Top$

{VAR} $\quad C.\ a \leq t \vdash a \leq t$

{BAS1} $\quad C \vdash a \leq a$

{BAS2} $\quad C \vdash k \leq k$

{ARROW}
$$\frac{C \vdash s' \leq s \quad C \vdash t \leq t'}{C \vdash s \to t \leq s' \to t'}$$

{RECD}
$$\frac{C \vdash s_1 \leq t_1\ ..\ C \vdash s_n \leq t_n}{C \vdash \{l_1{:}s_1, .. , l_n{:}s_n, .. , l_m{:}s_m\} \leq \{l_1{:}t_1, .. , l_n{:}t_n\}}$$

{VART}
$$\frac{C \vdash s_1 \leq t_1\ ..\ C \vdash s_n \leq t_n}{C \vdash [l_1{:}s_1, .. , l_n{:}s_n] \leq [l_1{:}t_1, .. , l_n{:}t_n, .. , l_m{:}t_m]}$$

{FORALL}
$$\frac{C.\ a \leq s \vdash t \leq t'}{C \vdash (\forall a \leq s.t) \leq (\forall a \leq s.t')} \quad a \text{ not free in } C$$

{EXISTS}
$$\frac{C.\ a \leq s \vdash t \leq t'}{C \vdash (\exists a \leq s.t) \leq (\exists a \leq s.t')} \quad a \text{ not free in } C$$

{TRANS}
$$\frac{C \vdash s \leq t \qquad C \vdash t \leq u}{C \vdash s \leq u}$$

Here are the typing rules for expressions e. A is a set of type assumptions for free program variables. A. x:t is the set A extended with the assumption that variable x has type t. C,A |— e: t means that from the set of constraints C and the set of assumptions A we can infer that e has type t .

[TOP] $$C,A \vdash e: Top$$

[VAR] $$C, A. x{:}t \vdash x: t$$

[ABS] $$\frac{C,A.x{:}s \vdash e: t}{C,A \vdash (fun(x{:}s)e): s \to t}$$

[APPL] $$\frac{C,A \vdash e: s \to t \quad C,A \vdash e': s}{C,A \vdash (e\ e'): t}$$

[RECD] $$\frac{C,A \vdash e_1{:}t_1 \ \ ..\ \ C,A \vdash e_n{:}t_n}{C,A \vdash \{l_1{=}e_1, .. ,l_n{=}e_n\} : \{l_1{:}t_1, .. ,l_n{:}t_n\}}$$

[SEL] $$\frac{C,A \vdash e: \{l_1{:}t_1, .. ,l_n{:}t_n\}}{C,A \vdash e.l_i: t_i} \qquad i \in 1..n$$

[VART] $$\frac{C,A \vdash e{:}t_i}{C,A \vdash [l_i{=}e]: [l_1{:}t_1, .. ,l_n{:}t_n]} \qquad i \in 1..n$$

[CASE] $$\frac{C,A \vdash e: [l_1{:}t_1, .. ,l_n{:}t_n] \quad C,A \vdash f_1: t_1 \to t \ \ ..\ \ C,A \vdash f_n: t_n \to t}{C,A \vdash (case\ e\ of\ l_1 \Rightarrow f_1, .. , l_n \Rightarrow f_n): t}$$

[GEN] $$\frac{C.a \le s,A \vdash e: t}{C,A \vdash all[a \le s]e : \forall a \le s.t} \quad a \text{ not free in } C,A$$

[SPEC] $$\frac{C,A \vdash e: \forall a \le s.t \quad C \vdash s' \le s}{C,A \vdash e[s']: t\{s'/a\}}$$

[PACK] $$\frac{C,A \vdash e: s\{t/a\} \quad C \vdash t \le u}{C,A \vdash pack\ [a \le u = t\ in\ s]\ e : \exists a \le u.s}$$

[OPEN] $$\frac{C,A \vdash e: \exists b \le u.s \quad C.a \le u, A.x{:}s\{a/b\} \vdash e': t}{C,A \vdash open\ e\ as\ x\ [a]\ in\ e' : t} \quad a \text{ not free in } t,C,A$$

[DEFN] $$\frac{C,A \vdash e: t\{s/b\}}{C,A \vdash e: a[s]} \quad \text{if } a[b] = t \text{ is a type definition}$$

$$[\text{TRANS}] \qquad \frac{C,A \vdash e: t \qquad C \vdash t \le u}{C,A \vdash e: u}$$

Acknowledgements

Research by Peter Wegner was supported in part by IBM, Yorktown Heights, N.Y., and in part by the Office of Naval Research under Contract No. N00014-74-C0592.

Thanks are due to Dave MacQueen for countless discussions, and to John Mitchell for insights. Also to Doug McIlroy and Malcolm Atkinson for a very thorough reading of the manuscript, and to Rishiyur Nikhil and Albert Meyer for many useful comments.

References

[Aho 85] A.V.Aho, R.Sethi, J.D.Ullman: **Compilers. Principles, techniques and tools**, Addison-Wesley 1985.

[Albano 85] A.Albano, L.Cardelli, R.Orsini: **Galileo: a strongly typed, interactive conceptual language**, *Transactions on Database Systems*, June 1985, 10(2), pp. 230-260.

[Booch 83] G. Booch, **Software Engineering with Ada**, Benjamin Cummings 1983.

[Bruce 84] K.B.Bruce, R.Meyer: **The semantics of second order polymorphic lambda calculus**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: **Hope: an experimental applicative language**, *Conference Record of the 1980 LISP Conference*, Stanford, August 1980, pp. 136-143.

[Burstall 84] R.Burstall, B.Lampson, **A kernel language for abstract data types and modules**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 84a] L.Cardelli: **Basic polymorphic typechecking**, AT&T Bell Laboratories Computing Science Technical Report 119, 1984. Also in "Polymorphism newsletters", II.1, Jan 1984.

[Cardelli 84] L.Cardelli: **A semantics of multiple inheritance**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Cardelli 85] L.Cardelli: **Amber**, *Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP*, Le Val D'Ajol, Vosges (France), May 1985.

[Coquand 85] T.Coquand, G.Huet: **Constructions: a higher order proof system for mechanizing mathematics**, Technical report 401, INRIA, May 1985.

[Curry 58] H.B.Curry, R.Feys: **Combinatory logic**, North-Holland, Amsterdam, 1958.

[Damas 82] L.Damas, R.Milner: **Principal type-schemes for functional programs**, *Proc. POPL 1982*, pp.207-212.

[Damas 84] PhD Thesis, Dept of Computer Science, University of Edinburgh, Scotland.

[Demers 79] A.Demers, J.Donahue: **Revised Report on Russell**, TR79-389, Computer Science Department, Cornell University, 1979.

[D.O.D. 83] US Department of Defense: **Ada reference manual**, ANSI/MIS-STD 1815, Jan 1983.

[Fairbairn 82] J.Fairbairn: **Ponder and its type system**, Technical report No 31, Nov 82, University of Cambridge, Computer Laboratory.

[Girard 71] J-Y.Girard: **Une extension de l'interprétation de Gödel a l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types**, *Proceedings of the second Scandinavian logic symposium*, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Goldberg 83] A.Goldberg, D.Robson: **Smalltalk-80. The language and its implementation**, Addison-Wesley, 1983.

[Gordon 79] M.Gordon, R.Milner, C.Wadsworth. **Edinburgh LCF**, *Lecture Notes in Computer Science, n.78*, Springer-Verlag, 1979.

[Hendler 86] J.Hendler, P.Wegner: **Viewing object-oriented programming as an enhancement of data abstraction methodology**, *Hawaii Conference on System Sciences*, January 1986.

[Hook 84] J.G.Hook: **Understanding Russell, a first attempt**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, pp. 51-67, Springer-Verlag, 1984.

[Hindley 69] R.Hindley: **The principal type scheme of an object in combinatory logic**, *Transactions of the American Mathematical Society*, Vol. 146, Dec 1969, pp. 29-60.

[Liskov 81] B.H.Liskov: **CLU Reference Manual**, *Lecture Notes in Computer Science*, 114, Springer-Verlag, 1981.

[McCracken 84] N.McCracken: **The typechecking of programs with implicit type structure**, in *Semantics of Data Types*, Lecture Notes in Computer Science n.173, pp. 301-316, Springer-Verlag 1984.

[MacQueen 84a] D.B.MacQueen, G.D.Plotkin, R.Sethi: **An ideal model for recursive polymorphic types**, *Proc. Popl 1984*. Also to appear in *Information and Control*.

[MacQueen 84b] D.B.MacQueen: **Modules for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp 198-207. ACM, New York.

[MacQueen 86] D.B.MacQueen: **Using dependent types to express modular structure**, *Proc. POPL 1986.*

[Martin-Löf 80] P.Martin-Löf, **Intuitionistic type theory**, Notes by Giovanni Sambin of a series of lectures given at the University of Padova, Italy, June 1980.

[Matthews 85] D.C.J.Matthews, **Poly manual**, Technical Report No. 63, University of Cambridge, Computer Laboratory, 1985.

[Milner 78] R.Milner: **A theory of type polymorphism in programming**, *Journal of Computer and System Science 17*, pp. 348-375, 1978.

[Milner 84] R.Milner: **A proposal for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp. 184-197. ACM, New York.

[Mitchell 79] J.G.Mitchell, W.Maybury, R.Sweet: **Mesa language manual**, Xerox PARC CSL-79-3, April 1979.

[Mitchell 84] J.C.Mitchell: **Type Inference and Type Containment**, in *Semantics of Data Types, Lecture Notes in Computer Science 173*, 51-67, Springer-Verlag, 1984.

[Mitchell 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, *Proc. Popl 1985*.

[Reynolds 74] J.C.Reynolds: **Towards a theory of type structure**, in *Colloquium sur la programmation*, pp. 408-423, Springer-Verlag *Lecture Notes in Computer Science*, n.19, 1974.

[Reynolds 85] J.C.Reynolds: **Three approaches to type structure**, *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science 185, Springer-Verlag, Berlin 1985, pp. 97-138.

[Robinson 65] J.A.Robinson: **A machine-oriented logic based on the resolution principle**, *Journal of the ACM*, vol. 12, no. 1, Jan. 1956, pp. 23-49.

[Schmidt 82] E.E.Schmidt: **Controlling large software development in a distributed environment**, Xerox PARC, CSL-82-7, Dec. 1982.

[Scott 76] D.Scott: **Data types as lattices**, *SIAM Journal of Computing*, Vol 5, No 3, September 1976, pp. 523-587.

[Solomon 78] M.Solomon: **Type Definitions with Parameters**, *Proc. POPL 1978*, Tucson, Arizona.

[Strachey 67] C.Strachey: **Fundamental concepts in programming languages**, lecture notes for the *International Summer School in Computer Programming*, Copenhagen, August 1967.

[Welsh 77] J.Welsh, W.J.Sneeringer, C.A.R.Hoare: **Ambiguities and insecurities in Pascal**, *Software Practice and Experience*, November 1977.

[Wegner 83] P.Wegner: **On the unification of data and program abstraction in Ada**, *Proc POPL 1983*.

[Weinreb 81] D.Weinreb, D.Moon: **Lisp machine manual, Fourth Edition, Chapter 20: Objects, message passing, and flavors**, Symbolics Inc., 1981.

[Wirth 83] N.Wirth: **Programming in Modula-2**, Springer-Verlag 1983.

Higher-Order and Symbolic Computation, 13, 11–49, 2000

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

There are numerous references throughout the course to CPL [1–3]. This is a programming language which has been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for research into both programming languages and the design of compilers. Partial implementations exist at Cambridge and London. The language is still evolving so that there is no definitive manual available yet. We hope to reach another resting point in its evolution quite soon and to produce a compiler and reference manuals for this version. The compiler will probably be written in such a way that it is relatively easy to transfer it to another machine, and in the first instance we hope to establish it on three or four machines more or less at the same time.

The lack of a precise formulation for CPL should not cause much difficulty in this course, as we are primarily concerned with the ideas and concepts involved rather than with their precise representation in a programming language.

Keywords: programming languages, semantics, foundations of computing, CPL, L-values, R-values, parameter passing, variable binding, functions as data, parametric polymorphism, ad hoc polymorphism, binding mechanisms, type completeness

1. Preliminaries

1.1. Introduction

Any discussion on the foundations of computing runs into severe problems right at the start. The difficulty is that although we all use words such as ‘name’, ‘value’, ‘program’, ‘expression’ or ‘command’ which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious. These misunderstandings arise in at least two ways. The first is straightforwardly incorrect or muddled thinking. An investigation of the meanings of these basic terms is undoubtedly an exercise in mathematical logic and neither to the taste nor within the field of competence of many people who work on programming languages. As a result the practice and development of programming languages has outrun our ability to fit them into a secure mathematical framework so that they have to be described in ad hoc ways. Because these start from various points they often use conflicting and sometimes also inconsistent interpretations of the same basic terms.

A second and more subtle reason for misunderstandings is the existence of profound differences in philosophical outlook between mathematicians. This is not the place to discuss this issue at length, nor am I the right person to do it. I have found, however, that these differences affect both the motivation and the methodology of any investigation like this to such an extent as to make it virtually incomprehensible without some preliminary warning. In the rest of the section, therefore, I shall try to outline my position and describe the way in which I think the mathematical problems of programming languages should be tackled. Readers who are not interested can safely skip to Section 2.

1.2. Philosophical considerations

The important philosophical difference is between those mathematicians who will not allow the existence of an object until they have a construction rule for it, and those who admit the existence of a wider range of objects including some for which there are no construction rules. (The precise definition of these terms is of no importance here as the difference is really one of psychological approach and survives any minor tinkering.) This may not seem to be a very large difference, but it does lead to a completely different outlook and approach to the methods of attacking the problems of programming languages.

The advantages of rigour lie, not surprisingly, almost wholly with those who require construction rules. Owing to the care they take not to introduce undefined terms, the better examples of the work of this school are models of exact mathematical reasoning. Unfortunately, but also not surprisingly, their emphasis on construction rules leads them to an intense concern for the way in which things are written—i.e., for their representation, generally as strings of symbols on paper—and this in turn seems to lead to a preoccupation with the problems of syntax. By now the connection with programming languages as we know them has become tenuous, and it generally becomes more so as they get deeper into syntactical questions. Faced with the situation as it exists today, where there is a generally known method of describing a certain class of grammars (known as BNF or context-free), the first instinct of these mathematicians seems to be to investigate the limits of BNF—what can you express in BNF even at the cost of very cumbersome and artificial constructions? This may be a question of some mathematical interest (whatever that means), but it has very little relevance to programming languages where it is more important to discover better methods of describing the syntax than BNF (which is already both inconvenient and inadequate for ALGOL) than it is to examine the possible limits of what we already know to be an unsatisfactory technique.

This is probably an unfair criticism, for, as will become clear later, I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development. In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.

But first we must try to get a better understanding of the processes of computing and their description in programming languages. In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention?) of calculus. We are still intellectually at the stage that calculus was at when it was called the 'Method of Fluxions' and everyone was arguing about how big a differential was. We need to develop our insight into computing processes and to recognise and isolate the central concepts—things analogous to the concepts of continuity and convergence in analysis. To do this we must become familiar with them and give them names even before we are really satisfied that we have described them precisely. If we attempt to formalise our ideas before we have really sorted out the important concepts the result, though possibly rigorous, is of very little value—indeed it may well do more harm than good by making it harder to discover the really important concepts. Our motto should be 'No axiomatisation without insight'.

However, it is equally important to avoid the opposite of perpetual vagueness. My own view is that the best way to do this in a rapidly developing field such as computing, is to be extremely careful in our choice of terms for new concepts. If we use words such as 'name', 'address', 'value' or 'set' which already have meanings with complicated associations and overtones either in ordinary usage or in mathematics, we run into the danger that these associations or overtones may influence us unconsciously to misuse our new terms—either in context or meaning. For this reason I think we should try to give a new concept a neutral name at any rate to start with. The number of new concepts required may ultimately be quite large, but most of these will be constructs which can be defined with considerable precision in terms of a much smaller number of more basic ones. This intermediate form of definition should always be made as precise as possible although the rigorous description of the basic concepts in terms of more elementary ideas may not yet be available. Who when defining the eigenvalues of a matrix is concerned with tracing the definition back to Peano's axioms?

Not very much of this will show up in the rest of this course. The reason for this is partly that it is easier, with the aid of hindsight, to preach than to practice what you preach. In part, however, the reason is that my aim is not to give an historical account of how we reached the present position but to try to convey what the position is. For this reason I have often preferred a somewhat informal approach even when mere formality would in fact have been easy.

2. Basic concepts

2.1. Assignment commands

One of the characteristic features of computers is that they have a store into which it is possible to put information and from which it can subsequently be recovered. Furthermore the act of inserting an item into the store erases whatever was in that particular area of the store before—in other words the process is one of overwriting. This leads to the assignment command which is a prominent feature of most programming languages.

The simplest forms of assignments such as

```
x := 3
x := y + 1
x := x + 1
```

lend themselves to very simple explications. ‘Set `x` equal to `3`’, ‘Set `x` to be the value of `y` plus `1`’ or ‘Add one to `x`’. But this simplicity is deceptive; the examples are themselves special cases of a more general form and the first explications which come to mind will not generalise satisfactorily. This situation crops up over and over again in the exploration of a new field; it is important to resist the temptation to start with a confusingly simple example.

The following assignment commands show this danger.

```
                  i := a > b → j,k                    (See note 1)
               A[i] := A[a > b → j,k]
A[a > b → j, k] := A[i]
   a > b → j, k := i                                  (See note 2)
```

All these commands are legal in CPL (and all but the last, apart from minor syntactic alterations, in ALGOL also). They show an increasing complexity of the expressions written on the left of the assignment. We are tempted to write them all in the general form

ε_1 := ε_2

where ε_1 and ε_2 stand for expressions, and to try as an explication something like ‘evaluate the two expressions and then do the assignment’. But this clearly will not do, as the meaning of an expression (and a name or identifier is only a simple case of an expression) on the left of an assignment is clearly different from its meaning on the right. Roughly speaking an expression on the left stands for an ‘address’ and one on the right for a ‘value’ which will be stored there. We shall therefore accept this view and say that there are two values associated with an expression or identifier. In order to avoid the overtones which go with the word ‘address’ we shall give these two values the neutral names: L-value for the address-like object appropriate on the left of an assignment, and R-value for the contents-like object appropriate for the right.

2.2. L-values and R-values

An L-value represents an area of the store of the computer. We call this a *location* rather than an address in order to avoid confusion with the normal store-addressing mechanism of the computer. There is no reason why a location should be exactly one machine-word in size—the objects discussed in programming languages may be, like complex or multiple precision numbers, more than one word long, or, like characters, less. Some locations are addressable (in which case their numerical machine address may be a good representation) but some are not. Before we can decide what sort of representation a general, non-addressable location should have, we should consider what properties we require of it.

The two essential features of a location are that it has a content—i.e. an associated R-value—and that it is in general possible to change this content by a suitable updating operation. These two operations are sufficient to characterise a general location which are consequently sometimes known as 'Load-Update Pairs' or LUPs. They will be discussed again in Section 4.1.

2.3. Definitions

In CPL a programmer can introduce a new quantity and give it a value by an initialised definition such as

```
let  p = 3.5
```

(In ALGOL this would be done by **real** `p; p := 3.5;`). This introduces a new use of the name `p` (ALGOL uses the term 'identifier' instead of name), and the best way of looking at this is that the activation of the definition causes a new location not previously used to be set up as the L-value of `p` and that the R-value 3.5 is then assigned to this location.

The relationship between a name and its L-value cannot be altered by assignment, and it is this fact which makes the L-value important. However in both ALGOL and CPL one name can have several different L-values in different parts of the program. It is the concept of scope (sometimes called lexicographical scope) which is controlled by the block structure which allows us to determine at any point which L-value is relevant.

In CPL, but not in ALGOL, it is also possible to have several names with the same L-value. This is done by using a special form of definition:

```
let  q ≃ p
```

which has the effect of giving the name of the same `L`-value as `p` (which must already exist). This feature is generally used when the right side of the definition is a more complicated expression than a simple name. Thus if `M` is a matrix, the definition

```
let  x ≃ M[2,2]
```

gives `x` the same L-value as one of the elements of the matrix. It is then said to be sharing with `M[2,2]`, and an assignment to `x` will have the same effect as one to `M[2,2]`.

It is worth noting that the expression on the right of this form of definition is evaluated in the L-mode to get an L-value at the time the definition is obeyed. It is this L-value which is associated with `x`. Thus if we have

```
let i = 2
let x ≃ M[i,i]
    i := 3
```

the L-value of `x` will remain that of `M[2,2]`.

`M[i,i]` is an example of an anonymous quantity i.e., an expression rather than a simple name—which has both an L-value and an R-value. There are other expressions, such as `a+b`, which only have R-values. In both cases the expression has no name as such although it does have either one value or two.

2.4. Names

It is important to be clear about this as a good deal of confusion can be caused by differing uses of the terms. ALGOL 60 uses ‘identifier’ where we have used ‘name’, and reserves the word ‘name’ for a wholly different use concerned with the mode of calling parameters for a procedure. (See Section 3.4.3.) ALGOL X, on the other hand, appears likely to use the word ‘name’ to mean approximately what we should call an L-value, (and hence something which is a location or generalised address). The term *reference* is also used by several languages to mean (again approximately) an L-value.

It seems to me wiser not to make a distinction between the meaning of ‘name’ and that of ‘identifier’ and I shall use them interchangeably. The important feature of a name is that it has no internal structure at any rate in the context in which we are using it as a name. Names are thus atomic objects and the only thing we know about them is that given two names it is always possible to determine whether they are equal (i.e., the same name) or not.

2.5. Numerals

We use the word ‘number’ for the abstract object and ‘numeral’ for its written representation. Thus 24 and XXIV are two different numerals representing the same number. There is often some confusion about the status of numerals in programming languages. One view commonly expressed is that numerals are the ‘names of numbers’ which presumably means that every distinguishable numeral has an appropriate R-value associated with it. This seems to me an artificial point of view and one which falls foul of Occam’s razor by unnecessarily multiplying the number of entities (in this case names). This is because it overlooks the important fact that numerals in general do have an internal structure and are therefore not atomic in the sense that we said names were in the last section.

An interpretation more in keeping with our general approach is to regard numerals as R-value expressions written according to special rules. Thus for example the numeral 253 is a syntactic variant for the expression

$$2 \times 10^2 + 5 \times 10 + 3$$

while the CPL constant **8** 253 is a variant of

$$2 \times 8^2 + 5 \times 8 + 3$$

Local rules for special forms of expression can be regarded as a sort of ‘micro-syntax’ and form an important feature of programming languages. The micro-syntax is frequently used in a preliminary ‘pre-processing’ or ‘lexical’ pass of compilers to deal with the recognition of names, numerals, strings, basic symbols (e.g. boldface words in ALGOL) and similar objects which are represented in the input stream by strings of symbols in spite of being atomic inside the language.

With this interpretation the only numerals which are also names are the single digits and these are, of course, constants with the appropriate R-value.

2.6. Conceptual model

It is sometimes helpful to have a picture showing the relationships between the various objects in the programming language, their representations in the store of a computer and the abstract objects to which they correspond. Figure 1 is an attempt to portray the conceptual model which is being used in this course.

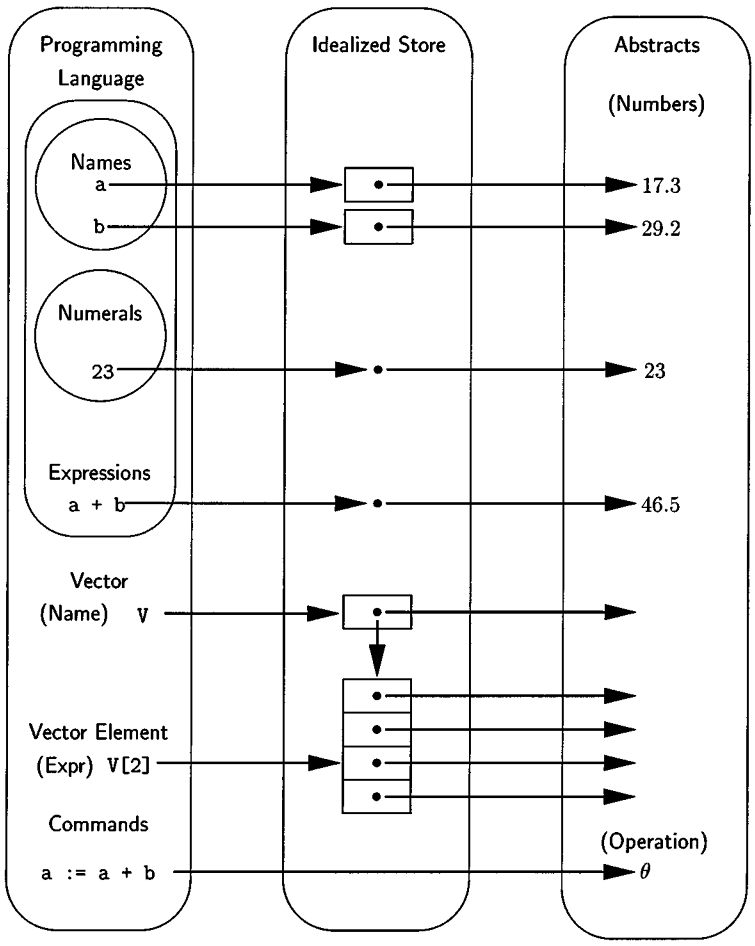


Figure 1. The conceptual model.

On the left are some of the components of the programming language. Many of these correspond to either an L-value or an R-value and the correspondence is indicated by an arrow terminating on the value concerned. Both L-values and R-values are in the idealised store, a location being represented by a box and its contents by a dot inside it. R-values without corresponding L-values are represented by dots without boxes, and R-values which are themselves locations (as, for example, that of a vector) are given arrows which terminate on another box in the idealised store.

R-values which correspond to numbers are given arrows which terminate in the right hand part of the diagram which represents the abstract objects with which the program deals.

The bottom section of the diagram, which is concerned with vectors and vector elements will be more easily understood after reading the section on compound data structures. (Section 3.7.)

3. Conceptual constructs

3.1. Expressions and commands

All the first and simplest programming language—by which I mean machine codes and assembly languages—consist of strings of commands. When obeyed, each of these causes the computer to perform some elementary operation such as subtraction, and the more elaborate results are obtained by using long sequences of commands.

In the rest of mathematics, however, there are generally no commands as such. Expressions using brackets, either written or implied, are used to build up complicated results. When talking about these expressions we use descriptive phrases such as ‘the sum of x and y’ or possibly ‘the result of adding x to y’ but never the imperative ‘add x to y’.

As programming languages developed and became more powerful they came under pressure to allow ordinary mathematical expressions as well as the elementary commands. It is, after all, much more convenient to write as in CPL, `x := a(b+c)+d` than the more elementary

```
CLA b
ADD c
MPY a
ADD d
STO x
```

and also, almost equally important, much easier to follow.

To a large extent it is true that the increase in power of programming languages has corresponded to the increase in the size and complexity of the right hand sides of their assignment commands for this is the situation in which expressions are most valuable. In almost all programming languages, however, commands are still used and it is their inclusion which makes these languages quite different from the rest of mathematics.

There is a danger of confusion between the properties of expressions, not all of which are familiar, and the additional features introduced by commands, and in particular those introduced by the assignment command. In order to avoid this as far as possible, the next section will be concerned with the properties of expressions in the absence of commands.

3.2. Expressions and evaluation

3.2.1. Values. The characteristic feature of an expression is that it has a *value*. We have seen that in general in a programming language, an expression may have two values—an L-value and an R-value. In this section, however, we are considering expressions in the absence of assignments and in these circumstances L-values are not required. Like the rest of mathematics, we shall be concerned only with R-values.

One of the most useful properties of expressions is that called by Quine [4] *referential transparency*. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are irrelevant to the value of the main expression.

We are quite familiar with this property of expressions in ordinary mathematics and often make use of it unconsciously. Thus we expect the expressions

$$sin(6) \quad sin(1+5) \quad sin(30/5)$$

to have the same value. Note, however, that we cannot replace the symbol string $1+5$ by the symbol 6 in all circumstances as, for example $21+52$ is not equal to 262. The equivalence only applies to complete expressions or sub-expressions and assumes that these have been identified by a suitable syntactic analysis.

3.2.2. Environments. In order to find the value of an expression it is necessary to know the value of its components. Thus to find the value of $a+5+b/a$ we need to know the values of a and b. Thus we speak of evaluating an expression in an environment (or sometimes relative to an environment) which provides the values of components.

One way in which such an environment can be provided is by a *where-clause*.

Thus

$$\begin{aligned} a + 3/a &\text{ where } a = 2 + 3/7 \\ a + b - 3/a &\text{ where } a = b + 2/b \end{aligned}$$

have a self evident meaning. An alternative syntactic form which has the same effect is the initialised definition:

$$\text{let } a = 2 + 3/7 \ \ldots\ a + 3/a$$
$$\text{let } a = b + 2/b \ \ldots\ a + b - 3/a$$

Another way of writing these is to use λ-expressions:

$$(\lambda a.\, a + 3/a)(2 + 3/7)$$
$$(\lambda a.\, a + b - 3/a)(b + 2/b)$$

All three methods are exactly equivalent and are, in fact, merely syntactic variants whose choice is a matter of taste. In each the letter a is singled out and given a value and is known as the *bound variable*. The letter b in the second expression is not bound and its value still has to be found from the environment in which the expression is to be evaluated. Variables of this sort are known as *free variables.*

3.2.3. Applicative structure. Another important feature of expressions is that it is possible to write them in such a way as to demonstrate an *applicative structure*—i.e., as an operator applied to one or more operands. One way to do this is to write the operator in front of its operand or list of operands enclosed in parentheses. Thus

$$\begin{array}{ll} a+b & \text{corresponds to } +(a,b) \\ a+3/a & \text{corresponds to } +(a,/(3,a)) \end{array}$$

In this scheme a λ-expression can occur as an operator provided it is enclosed in parentheses. Thus the expression

$$a+a/3 \text{ where } a=2+3/7$$

can be written to show its full applicative structure as

$$\{\lambda a.+(a,/(3,a))\}(+(2,/(3,7))).$$

Expressions written in this way with deeply nesting brackets are very difficult to read. Their importance lies only in emphasising the uniformity of applicative structure from which they are built up. In normal use the more conventional syntactic forms which are familiar and easier to read are much to be preferred—providing that we keep the underlying applicative structure at the back of our minds.

In the examples so far given all the operators have been either a λ-expression or a single symbol, while the operands have been either single symbols or sub-expressions. There is, in fact, no reason why the operator should not also be an expression. Thus for example if we use D for the differentiating operator, $D(sin)=cos$ so that $\{D(sin)\}(\times(3,a))$ is an expression with a compound operator whose value would be $cos(3a)$. Note that this is not the same as the expression $\frac{d}{dx}sin(3x)$ for $x=a$ which would be written $(D(\lambda x.sin(x(3,x))))(a)$.

3.2.4. Evaluation. We thus have a distinction between *evaluating* an operator and *applying* it to its operands. Evaluating the compound operator $D(sin)$ produces the result (or value) *cos* and can be performed quite independently of the process of applying this to the operands. Furthermore it is evident that we need to evaluate both the operator and the operands before we can apply the first to the second. This leads to the general rule for evaluating compound expressions in the operator-operand form viz:

1. Evaluate the operator and the operand(s) in any order.
2. After this has been done, apply the operator to the operand(s).

The interesting thing about this rule is that it specifies a partial ordering of the operations needed to evaluate an expression. Thus for example when evaluating

$$(a + b)(c + d/e)$$

both the additions must be performed before the multiplication, and the division before the second addition but the sequence of the first addition and the division is not specified. This partial ordering is a characteristic of algorithms which is not yet adequately reflected in most programming languages. In ALGOL, for example, not only is the sequence of commands fully specified, but the left to right rule specifies precisely the order of the operations. Although this has the advantage of precision in that the effect of any program is exactly defined, it makes it impossible for the programmer to specify indifference about sequencing or to indicate a partial ordering. The result is that he has to make a large number of logically unnecessary decisions, some of which may have unpredictable effects on the efficiency of his program (though not on its outcome).

There is a device originated by Schönfinkel [5], for reducing operators with several operands to the successive application of single operand operators. Thus, for example, instead of $+(2, p)$ where the operator $+$ takes two arguments we introduce another adding operator say $+'$ which takes a single argument such that $+'(2)$ is itself a function which adds 2 to its argument. Thus $(+'(2))(p) = +(2, p) = 2 + p$. In order to avoid a large number of brackets we make a further rule of association to the left and write $+'\ 2p$ in place of $((+'\ 2)p)$ or $(+'\ (2))(p)$. This convention is used from time to time in the rest of this paper. Initially, it may cause some difficulty as the concept of functions which produce functions as results is a somewhat unfamiliar one and the strict rule of association to the left difficult to get used to. But the effort is well worth while in terms of the simpler and more transparent formulae which result.

It might be thought that the remarks about partial ordering would no longer apply to monadic operators, but in fact this makes no difference. There is still the choice of evaluating the operator or the operand first and this allows all the freedom which was possible with several operands. Thus, for example, if p and q are sub-expressions, the evaluation of $p + q$ (or $+(p, q)$) implies nothing about the sequence of evaluation of p and q although both must be evaluated before the operator $+$ can be applied. In Schönfinkel's form this is $(+'\ p)q$ and we have the choice of evaluating $(+'\ p)$ and q in any sequence. The evaluation of $+'\ p$ involves the evaluation of $+'$ and p in either order so that once more there is no restriction on the order of evaluation of the components of the original expression.

3.2.5. Conditional expressions. There is one important form of expression which appears to break the applicative expression evaluation rule. A conditional expression such as

```
(x = 0) → 0,1/x
```

(in ALGOL this would be written **if** `x = 0` **then** `0` **else** `1/x`) cannot be treated as an ordinary function of three arguments. The difficulty is that it may not be possible to evaluate both arms of the condition—in this case when `x = 0` the second arm becomes undefined.

Various devices can be used to convert this to a true applicative form, and in essence all have the effect of delaying the evaluation of the arms until after the condition has been decided. Thus suppose that *If* is a function of a Boolean argument whose result is the selector *First* or *Second* so that $If(True) = First$ and $If(False) = Second$, the naive interpretation of the conditional expression given above as

$$\{If(x = 0)\}(0, 1/x)$$

is wrong because it implies the evaluation of both members of the list $(0, 1/x)$ before applying the operator $\{If(x = 0)\}$. However the expression

$$[\{If(x = 0)\}(\{\lambda a.\ 0\}, \{\lambda a.\ 1/x\})]a$$

will have the desired effect as the selector function $If\ (x = 0)$ is now applied to the list $(\{\lambda a.\ 0\}, \{\lambda a.\ 1/x\})$ whose members are λ-expressions and these can be evaluated (but not applied) without danger. After the selection has been made the result is applied to a and provided a has been chosen not to conflict with other identifiers in the expression, this produces the required effect.

Recursive (self referential) functions do not require commands or loops for their definition, although to be effective they do need conditional expressions. For various reasons, of which the principal one is lack of time, they will not be discussed in this course.

3.3. Commands and sequencing

3.3.1. Variables. One important characteristic of mathematics is our habit of using names for things. Curiously enough mathematicians tend to call these things ‘variables’ although their most important property is precisely that they do not vary. We tend to assume automatically that the symbol x in an expression such as $3x^2 + 2x + 17$ stands for the same thing (or has the same value) on each occasion it occurs. This is the most important consequence of referential transparency and it is only in virtue of this property that we can use the where-clauses or λ-expressions described in the last section.

The introduction of the assignment command alters all this, and if we confine ourselves to the R-values of conventional mathematics we are faced with the problem of variables which actually vary, so that their value may not be the same on two occasions and we can no longer even be sure that the Boolean expression $x = x$ has the value *True*. Referential transparency has been destroyed, and without it we have lost most of our familiar mathematical tools—for how much of mathematics can survive the loss of identity?

If we consider L-values as well as R-values, however, we can preserve referential transparency as far as L-values are concerned. This is because L-values, being generalised addresses, are not altered by assignment commands. Thus the command `x := x+1` leaves the address of the cell representing `x` (L-value of `x`) unchanged although it does alter the contents of this cell (R-value of `x`). So if we agree that the values concerned are all L-values, we can continue to use where-clauses and λ-expressions for describing parts of a program which include assignments.

The cost of doing this is considerable. We are obliged to consider carefully the relationship between L and R-values and to revise all our operations which previously took R-value operands so that they take L-values. I think these problems are inevitable and although much of the work remains to be done, I feel hopeful that when completed it will not seem so formidable as it does at present, and that it will bring clarification to many areas of programming language study which are very obscure today. In particular the problems of side effects will, I hope, become more amenable.

In the rest of this section I shall outline informally a way in which this problem can be attacked. It amounts to a proposal for a method in which to formalise the semantics of a programming language. The relation of this proposal to others with the same aim will be discussed later. (Section 4.3.)

3.3.2. The abstract store. Our conceptual model of the computing process includes an abstract store which contains both L-values and R-values. The important feature of this abstract store is that at any moment it specifies the relationship between L-values and the corresponding R-values. We shall always use the symbol σ to stand for this mapping from L-values onto R-values. Thus if α is an L-value and β the corresponding R-value we shall write (remembering the conventions discussed in the last section)

$$\beta = \sigma\alpha.$$

The effect of an assignment command is to change the contents of the store of the machine. Thus it alters the relationship between L-values and R-values and so changes σ. We can therefore regard assignment as an operator on σ which produces a fresh σ. If we update the L-value α (whose original R-value in σ was β) by a fresh R-value β' to produce a new store σ', we want the R-value of α in σ' to be β', while the R-value of all other L-values remain unaltered. This can be expressed by the equation

$$(U(\alpha, \beta'))\sigma = \sigma' \text{ where } \sigma' x = (x = \alpha) \longrightarrow \beta', \sigma x.$$

Thus U is a function which takes two arguments (an L-value and an R-value) and produces as a result an operator which transforms σ into σ' as defined.

The arguments of U are L-values and R-values and we need some way of getting these from the expressions written in the program. Both the L-value and the R-value of an expression such as `V[i+3]` depend on the R-value of `i` and hence on the store. Thus both must involve σ and if ε stands for a written expression in the programming language we shall write $\mathcal{L}\ \varepsilon\ \sigma$ and $\mathcal{R}\ \varepsilon\ \sigma$ for its L-value and R-value respectively.

Both $\mathcal{L}$ and $\mathcal{R}$ are to be regarded as functions which operate on segments of text of the programming language. The question of how those segments are isolated can be regarded as a matter of syntactic analysis and forms no part of our present discussion.

These functions show an application to Schönfinkel's device which is of more than merely notational convenience. The function $\mathcal{R}$, for example, shows that its result depends on both ε and σ, so it might be thought natural to write it as $\mathcal{R}(\varepsilon, \sigma)$. However by writing $\mathcal{R}\ \varepsilon\ \sigma$ and remembering that by our convention of association to the left this means $(\mathcal{R}\ \varepsilon)\sigma$ it becomes natural to consider the application of $\mathcal{R}$ to ε separately and before the application of $\mathcal{R}\ \varepsilon$ to σ. These two phases correspond in a very convenient way to the processes of compilation, which involves manipulation of the text of the program, and execution which involves using the store of the computer. Thus the notation allows us to distinguish clearly between compile-time and execution-time processes. This isolation of the effect of σ is a characteristic of the method of semantic description described here.

It is sometimes convenient to use the contents function C defined by $C\ \alpha\ \sigma\ = \sigma\ \alpha$. Then if

$$\alpha = \mathcal{L}\ \varepsilon\ \sigma$$
$$\beta = \mathcal{R}\ \varepsilon\ \sigma$$

we have $\beta = C\ \alpha\ \sigma\ = \sigma\ \alpha$. After updating α by β', we have

$$\sigma' = U(\alpha, \beta')\sigma$$

and

$$C\ \alpha\ \sigma' = \beta'.$$

3.3.3. Commands. Commands can be considered as functions which transform σ. Thus the assignment

$$\varepsilon_1\ \texttt{:=}\ \varepsilon_2$$

has the effect of producing a store

$$\sigma' = U(\alpha_1, \beta_2)\sigma$$

where

$$\alpha_1 = \mathcal{L}\ \varepsilon_1\ \sigma$$

and

$$\beta_2 = \mathcal{R}\ \varepsilon_2\ \sigma$$

so that

$$\sigma' = U(\mathcal{L}\ \varepsilon_1\ \sigma, \mathcal{R}\ \varepsilon_2\ \sigma)\sigma$$

and if θ is the function on σ which is equivalent to the original command we have

$$\sigma' = \theta\sigma$$

where

$$\theta = \lambda\ \sigma.\ U(\mathcal{L}\ \varepsilon_1\ \sigma, \mathcal{R}\ \varepsilon_2\ \sigma)\sigma$$

Sequences of commands imply the successive application of sequences of θ's. Thus, for example, if γ_1, γ_2, γ_3 are commands and θ_1, θ_2, θ_3 the equivalent functions on σ, the command sequence (or compound command)

$$\gamma_1\,;\gamma_2\,;\gamma_3\,;$$

applied to a store σ will produce a store

$$\begin{aligned}\sigma' &= \theta_3(\theta_2(\theta_1\ \sigma))\\ &= (\theta_3 \cdot \theta_2 \cdot \theta_1)\sigma\end{aligned}$$

where $f \cdot g$ is the function product of f and g.

Conditional commands now take a form similar to that of conditional expressions. Thus the command

```
Test ε₁ If so do   γ₁
        If not do  γ₂
```

corresponds to the operator

$$\lambda\sigma.\ \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\theta_1, \theta_2)\sigma$$

where θ_1 and θ_2 correspond to γ_1 and γ_2.

Conditional expressions can also be treated more naturally. The dummy argument introduced in the last section to delay evaluation can be taken to be σ with considerable advantages in transparency. Thus

$$\mathcal{R}(\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3)\sigma = \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\mathcal{R}\ \varepsilon_2, \mathcal{R}\ \varepsilon_3)\sigma$$

and

$$\mathcal{L}(\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3)\sigma = \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\mathcal{L}\ \varepsilon_2, \mathcal{L}\ \varepsilon_3)\sigma$$

Informally $\mathcal{R}\,\varepsilon_2$ and $\mathcal{L}\,\varepsilon_2$ correspond to the compiled program for evaluating ε_2 in the R-mode or L-mode respectively. The selector $\mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)$ chooses between these at execution time on the basis of the R-value of ε_1 while the final application to σ corresponds to running the chosen piece of program.

If we consider commands as being functions operating on σ, loops and cycles are merely recursive functions also operating on σ. There is, however, no time to go further into these in this course.

An interesting feature of this approach to the semantics of programming languages is that all concept of sequencing appears to have vanished. It is, in fact, replaced by the partially ordered sequence of functional applications which is specified by λ-expressions.

In the remaining sections we shall revert to a slightly less formal approach, and try to isolate some important ‘high level’ concepts in programming languages.

3.4. Definition of functions and routines

3.4.1. Functional abstractions. In order to combine programs hierarchically we need the process of functional abstraction. That is to say that we need to be able to form functions from expressions such as

```
let  f[x] = 5x²+ 3x + 2/x³
```

This could be thought of as defining `f` to be a function and giving it an initial value. Thus the form of definition given above is merely a syntactic variant of the standard form of definition (which has the quantity defined alone on the left side)

```
let  f = λx. 5x²+ 3x + 2/x³
```

This form makes it clear that it is `f` which is being defined and that `x` is a bound or dummy variable and could be replaced by any other non-clashing name without altering the value given to `f`.

3.4.2. Parameter calling modes. When the function is used (or called or applied) we write `f[`ε`]` where ε can be an expression. If we are using a referentially transparent language all we require to know about the expression ε in order to evaluate `f[`ε`]` is its value. There are, however, two sorts of value, so we have to decide whether to supply the R-value or the L-value of ε to the function `f`. Either is possible, so that it becomes a part of the definition of the function to specify for each of its bound variables (also called its formal parameters) whether it requires an R-value or an L-value. These alternatives will also be known as calling a parameter by *value* (R-value) or *reference* (L-value).

Existing programming languages show a curious diversity in their modes of calling parameters. FORTRAN calls all its parameters by reference and has a special rule for providing R-value expressions such as $a+b$ with a temporary L-value. ALGOL 60, on the other hand, has two modes of calling parameters (specified by the programmer): *value* and *name*. The ALGOL call by *value* corresponds to call by R-value as above; the call by name,[3] however, is quite different (and more complex). Only if the actual parameter (i.e., the expression ε above) is a simple variable is the effect the same as a call by reference. This incompatibility in their methods of calling parameters makes it difficult to combine the two languages in a single program.

3.4.3. Modes of free variables. The obscurity which surrounds the modes of calling the bound variables becomes much worse when we come to consider the free variables of a function. Let us consider for a moment the very simple function

```
f[x] = x + a
```

where `a` is a free variable which is defined in the surrounding program. When `f` is defined we want in some way to incorporate `a` into its definition, and the question is do we use its

R-value or its L-value? The difference is illustrated in the following pair of CPL programs. (In CPL a function definition using = takes its free variables by R-value and one using ≡ takes them by L-value.)

Free variable by R-value

```
let  a = 3
let  f[x] = x + a
   ... (f[5] = 8)...
   a := 10
   ... (f[5] = 8)...
```

Free variable by L-value

```
let  a = 3
let  f[x] ≡ x + a
   ...(f[5] = 8)...
   a := 10
   ...(f[5] = 15)...
```

The expressions in parentheses are all Booleans with the value **true**.

Thus the first example freezes the current R-value of `a` into the definition of `f` so that it is unaffected by any future alterations (by assignment) to `a`, while the second does not. It is important to realize, however, that even the second example freezes something (i.e., the L-value of `a`) into the definition of `f`. Consider the example

```
let  a = 3
let  f[x] ≡ x + a
     ...(f[5] = 8),(a = 3) ...
     § let a = 100
      ...(f[5] = 8),(a = 100) ...
          a := 10
      ...(f[5] = 8),(a = 10) ...
      ...........§
     ...(f[5] = 8),(a = 3) ...
```

Here there is an inner block enclosed in the statement brackets § .... § (which corresponds to **begin** and **end** in ALGOL), and inside this an entirely fresh `a` has been defined. This forms a hole in the scope of the original `a` in which it continues to exist but becomes inaccessible to the programmer. However as its L-value was incorporated in the definition of `f`, it is the original `a` which is used to find `f[5]`. Note that assignments to `a` in the inner block affect only the second `a` and so do not alter `f`.

It is possible to imagine a third method of treating free variables (though there is nothing corresponding for bound variables) in which the locally current meaning of the variables is used, so that in the example above the second and third occurrences of `f[5]` would have the values `105` and `15` respectively. I believe that things very close to this exist in LISP2 and are known as fluid variables. The objection to this scheme is that it appears to destroy referential transparency irrevocably without any apparent compensating advantages.

In CPL the facilities for specifying the mode of the free variables are considerably coarser than the corresponding facilities for bound variables. In the case of bound variables the mode has to be specified explicitly or by default for each variable separately. For the free variables, however, it is only possible to make a single specification which covers all the free variables, so that they must all be treated alike. The first method is more flexible and provides greater power for the programmer, but is also more onerous (although good default conventions can help to reduce the burden); the second is much simpler to use but sometimes does not allow a fine enough control. Decisions between methods of this sort are bound to be compromises reflecting the individual taste of the language designer and are always open to objection on grounds of convenience. It is no part of a discussion on the fundamental concepts of programming languages to make this sort of choice—it should rest content with pointing out the possibilities.

A crude but convenient method of specification, such as CPL uses for the mode of the free variables of a function, becomes more acceptable if there exists an alternative method by which the finer distinctions can be made, although at the cost of syntactic inelegance. Such a method exists in CPL and involves using an analogue to the *own* variables in ALGOL 60 proposed by Landin [6].

3.4.4. Own variables. The idea behind *own* variables is to allow some private or secret information which is in some way protected from outside interference. The details were never very clearly expressed in ALGOL and at least two rival interpretations sprang up, neither being particularly satisfactory. The reason for this was that *owns* were associated with blocks whereas, as Landin pointed out, the natural association is with a procedure body. (In this case of functions this corresponds to the expression on the right side of the function definition.)

The purpose is to allow a variable to preserve its value from one application of a function to the next—say to produce a pseudo-random number or to count the number of times the function is applied. This is not possible with ordinary local variables defined within the body of the function as all locals are redefined afresh on each application of the function. It would be possible to preserve information in a non-local variable—i.e., one whose scope included both the function definition and all its applications, but it would not then be protected and would be accessible from the whole of this part of the program. What we need is a way of limiting the scope of a variable to be the definition only. In CPL we indicate this by using the word in to connect the definition of the own variable (which is usually an initialised one) with the function definitions it qualifies.

In order to clarify this point programs using each of the three possible scopes (non-local, own and local) are written below in three ways viz. Normal CPL, CPL mixed with λ-expressions to make the function definition in its standard form, and finally in pure λ-expressions. The differences in the scope rules become of importance only when there is a clash of names, so in each of these examples one or both of the names `a` and `x` are used twice. In order to make it easy to determine which is which, a prime has been added to one of them. However, the scope rules imply that if all the primes were omitted the program would be unaltered.

1. Non-local variable

```
CPL          let a' = 6
             let x' = 10
             let a  = 3/x'
             let f[x] ≡ x + a
                 ....  f[a] ....
```

Mixed

```
let  a' = 6
let  x' = 10
let  a  = 3/x'
let  f ≡λx. x + a
     ....  f[a] ....
```

Pure λ

```
{λa'.{λx'. {λa. {λf. f  a}[λx. x + a]}[3/x']}10}6
```

2. Own variable

CPL

```
let  a' = 6
let  x' = 10
let  a  = 3/x'
in  f[x] ≡ x + a
 ....  f[a'] ....
```

Mixed

```
let  a' = 6
let  x' = 10
let  f  ≡{λa. λx. x + a}[3/x']
 ....  f[a'] ....
```

Pure λ

```
{λa'. {λx'. {λf. f  a'}[{λa. λx. x + a}[3/x']]}10}6
```

3. Local variable

CPL

```
let  a' = 6
let  x' = 10
let  f[x] ≡(x + a where a = 3/x)
    ....  f[a'] ....
```

Mixed

```
let  a' = 6
let  x' = 10
let  f  ≡ λx.{λa.x + a}[3/x]
    ....  f[a'] ....
```

Pure λ

```
{λa'. {λx'. {λf. f  a' }[λx. {λa. x + a}[3/x]]}10}6
```

We can now return to the question of controlling the mode of calling the free variables of a function. Suppose we want to define `f[x]` to be `ax + b + c` and use the *R*-value of `a` and `b` but the *L*-value of `c`. A CPL program which achieves this effect is

```
let a' = a and b' = b
in f[x] ≡ a'x + b' + c
  ....
```

(Again the primes may be omitted without altering the effect.)

The form of definition causes the *L*-values of `a'`, `b'` and `c` to be used, while the definition of the variables `a'` and `b'` ensures that these are given fresh *L*-values which are initialised to the *R*-values of `a` and `b`. As they are own variables, they are protected from any subsequent assignments to `a` and `b`.

3.4.5. Functions and routines. We have so far discussed the process of functional abstraction as applied to expressions. The result is called a *function* and when applied to suitable arguments it produces a value. Thus a function can be regarded as a complicated sort of expression. The same process of abstraction can be applied to a command (or sequence of commands), and the result is know in CPL as a *routine*. The application of a routine to a suitable set of arguments is a complicated command, so that although it affects the store of the computer, it produces no value as a result.

Functions and routines are as different in their nature as expressions and commands. It is unfortunate, therefore, that most programming languages manage to confuse them very successfully. The trouble comes from the fact that it is possible to write a function which also alters the store, so that it has the effect of a function and a routine. Such functions are sometimes said to have side effects and their uncontrolled use can lead to great obscurity in the program. There is no generally agreed way of controlling or avoiding the side effects of functions, and most programming languages make no attempt to deal with the problem at all—indeed their confusion between routines and functions adds to the difficulties.

The problem arises because we naturally expect referential transparency of R-values in expressions, particularly those on the right of assignment commands. This is, I think, a very reasonable expectation as without this property, the value of the expression is much harder to determine, so that the whole program is much more obscure. The formal conditions on expressions which have to be satisfied in order to produce this R-value referential transparency still need to be investigated. However in special cases the question is usually easy to decide and I suggest that as a matter of good programming practice it should always be done. Any departure of R-value referential transparency in a R-value context should either be eliminated by decomposing the expression into several commands and simpler expressions, or, if this turns out to be difficult, the subject of a comment.

3.4.6. Constants and variables. There is another approach to the problem of side effects which is somewhat simpler to apply, though it does not get round all the difficulties. This is, in effect, to turn the problem inside out and instead of trying to specify functions and expressions which have no side effect to specify objects which are immune from any possible side effect of others. There are two chief forms which this protection can take which can roughly be described as hiding and freezing. Their inaccessibility (by reason of the scope rules) makes them safe from alteration except from inside the body of the function or routine they qualify. We shall be concerned in this section and the next with different forms of protection by freezing.

The characteristic thing about variables is that their R-values can be altered by an assignment command. If we are looking for an object which is frozen, or invariant, an obvious possibility is to forbid assignments to it. This makes it what in CPL we call a *constant*. It has an L-value and R-value in the ordinary way, but applying the update function to it either has no effect or produces an error message. Constancy is thus an attribute of an L-value, and is, moreover, an invariant attribute. Thus when we create a new L-value, and in particular when we define a new quantity, we must decide whether it is a constant or a variable.

As with many other attributes, it is convenient in a practical programming language to have a default convention—if the attribute is not given explicitly some conventional value is assumed. The choice of these default conventions is largely a matter of taste and judgement, but it is an important one as they can affect profoundly both the convenience of the language and the number of slips made by programmers. In the case of constancy, it is reasonable that the ordinary quantities, such as numbers and strings, should be variable. It is only rather rarely that we want to protect a numerical constant such as `Pi` from interference. Functions and routines, on the other hand, are generally considered to be constants. We tend to give them familiar or mnemonic names such a `CubeRt` or `LCM` and we would rightly feel confused by an assignment such as `CubeRt := SqRt`. Routines and functions are therefore given the default attribute of being a constant.

3.4.7. Fixed and free. The constancy or otherwise of a function has no connection with the mode in which it uses its free variables. If we write a definition in its standard form such as

```
let f ≡ λx. x + a
```

we see that this has the effect of initialising `f` with a λ-expression. The constancy of `f` merely means that we are not allowed to assign to it. The mode of its free variables (indicated by ≡) is a property of the λ-expression.

Functions which call their free variables by reference (L-value) are liable to alteration by assignments to their free variables. This can occur either inside or outside the function body, and indeed, even if the function itself is a constant. Furthermore they cease to have a meaning if they are removed from an environment in which their free variables exist. (In ALGOL this would be outside the block in which their free variables were declared.) Such functions are called *free* functions.

The converse of a free function is a *fixed* function. This is defined as a function which either has no free variables, or if it has, whose free variables are all both constant and fixed. The crucial feature of a fixed function is that it is independent of its environment and is always the same function. It can therefore be taken out of the computer (e.g., by being compiled separately) and reinserted again without altering its effect.

Note that fixity is a property of the λ-expression—i.e., a property of the R-value, while constancy is a property of the L-value. Numbers, for example, are always fixed as are all ‘atomic’ R-values (i.e., ones which cannot be decomposed into smaller parts). It is only in composite objects that the distinction between fixed and free has any meaning. If such an object is fixed, it remains possible to get at its component parts, but not to alter them. Thus, for example, a fixed vector is a look-up table whose entries will not (cannot) be altered, while a free vector is the ordinary sort of vector in which any element may be changed if necessary.

3.4.8. Segmentation. A fixed routine or function is precisely the sort of object which can be compiled separately. We can make use of this to allow the segmentation of programs and their subsequent assembly even when they do communicate with each other through free variables. The method is logically rather similar to the FORTRAN Common variables.

Suppose `R[x]` is a routine which uses `a`, `b`, and `c` by reference as free variables. We can define a function `R'[a,b,c]` which has as formal parameters all the free variables of `R` and whose result is the routine `R[x]`. Then `R'` will have no free variables and will thus be a fixed function which can be compiled separately.

The following CPL program shows how this can be done:

```
§ let  R'[ref a,b,c] = value of
       § let  R[x] be
             § ... a,b,c ...
                 (body of R) §
          result is  R §

WriteFixedFunction [R']

finish  §
```

The command `WriteFixedFunction [R']` is assumed to output its argument in some form of relocatable binary or otherwise so that it can be read in later by the function `ReadFixedFunction`.

If we now wish to use `R` in an environment where its free variables are to be `p`, `q` and `r` and its name is to be `S` we can write

```
§ let  p,q,r = ... (Setting up the environment)
  let  S' = ReadFixedFunction
  let  S  = S'[p,q,r]

     .... S[u] ....     §
```

In this way `S'` becomes the same function as `R'` and the call `S'[p,q,r]`, which use the L-values of `p`, `q` and `r`, produces `S` which is the original routine `R` but with `p`, `q` and `r` as its free variables instead of `a`, `b` and `c`.

One advantage of this way of looking at segmentation is that it becomes a part of the ordinary programming language instead of a special ad hoc device. An unfamiliar feature will be its use of a function `R'` which has as its result another function or routine. This is discussed in more detail in the next section.

3.5. Functions and routines as data items.

3.5.1. First and second class objects. In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter), while we can write (in ALGOL still)

```
(if  x > 1 then a else b) + 6
```

when a and b are reals, we cannot correctly write

```
(if  x > 1 then sin else cos)(x)
```

nor can we write a type procedure (ALGOL's nearest approach to a function) with a result which is itself a procedure.

Historically this second class status of procedures in ALGOL is probably a consequence of the view of functions taken by many mathematicians: that they are constants whose name one can always recognise. This second class view of functions is demonstrated by the remarkable fact that ordinary mathematics lacks a systematic notation for functions. The following example is given by Curry [7, p. 81].

Suppose P is an operator (called by some a 'functional') which operates on functions. The result of applying P to a function $f(x)$ is often written $P[f(x)]$. What then does $P[f(x+1)]$ mean? There are two possible meanings (a) we form $g(x) = f(x+1)$ and the result is $P[g(x)]$ or (b) we form $h(x) = P[f(x)]$ and the result is $h(x+1)$. In many cases these are the same but not always. Let

$$P[f(x)] = \begin{cases} \dfrac{f(x) - f(0)}{x} & \text{for } x \neq 0 \\ f'(x) & \text{for } x = 0 \end{cases}$$

Then if $f(x) = x^2$

$$P[g(x)] = P[x^2 + 2x + 1] = x + 2$$

while

$$h(x) = P[f(x)] = x$$

so that $h(x+1) = x + 1$.

This sort of confusion is, of course, avoided by using λ-expressions or by treating functions as first class objects. Thus, for example, we should prefer to write $(P[f])[x]$ in place of $P[f(x)]$ above (or, using the association rule $P[f][x]$ or even $P\ f\ x$). The two alternatives which were confused would then become

$$P\ g\ x \quad \text{where } g\ x = f(x+1)$$

and $P\ f\ (x+1)$.

The first of these could also be written $P(\lambda x.\ f(x+1))x$.

I have spent some time on this discussion in spite of its apparently trivial nature, because I found, both from personal experience and from talking to others, that it is remarkably difficult to stop looking on functions as second class objects. This is particularly unfortunate as many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as a result. As usual with new or unfamiliar ways of looking at things, it is harder for the teachers to change their habits of thought than it is for their pupils to follow them. The difficulty is considerably greater in the case of practical programmers for whom an abstract concept such as a function has little reality until they can clothe it with a representation and so understand what it is that they are dealing with.

3.5.2. Representation of functions. If we want to make it possible to assign functions we must be clear what are their L-values and R-values. The L-value is simple—it is the location where the R-value is stored—but the R-value is a little more complicated. When a function is applied it is the R-value which is used, so that at least sufficient information must be included in the R-value of a function to allow it to be applied. The application of a function to its arguments involves the evaluation of its defining expression after supplying the values of its bound variables from the argument list. To do this it is necessary to provide an environment which supplies the values of the free variables of the function.

Thus the R-value of a function contains two parts—a rule for evaluating the expression, and an environment which supplies its free variables. An R-value of this sort will be called a *closure*. There is no problem in representing the rule in a closure, as the address of a piece of program (i.e., a subroutine entry point) is sufficient. The most straightforward way of representing the environment part is by a pointer to a *Free Variable List* (FVL) which has an entry for each free variable of the function. This list is formed when the function is initially defined (more precisely when the λ-expression which is the function is evaluated, usually during a function definition) and at this time either the R-value or the L-value of each of the free variables is copied into the FVL. The choice of R- or L-value is determined by the mode in which the function uses its free variables. Thus in CPL functions defined by $=$ have R-values in their FVL while functions defined by $\equiv$ have L-values. Own variables of the kind discussed in the previous section can also be conveniently accommodated in the FVL.

The concept of a closure as the R-value of a function makes it easier to understand operations such as passing a function as a parameter, assigning to a variable of type function, or producing a function as the value of an expression or result of another function application. In each case the value concerned, which is passed on or assigned, is a closure consisting of a pair of addresses.

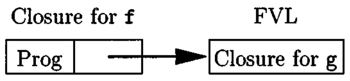
It is important to note that a function closure does not *contain* all the information associated with the function, it merely *gives access* (or points) to it, and that as the R-value of a function is a closure, the same applies to it. This is in sharp distinction to the case of data items such as *reals* or *integers* whose R-value is in some sense atomic or indivisible and contains all the information about the data items.

This situation, where some of the information is in the FVL rather than the R-value, is quite common and occurs not only with functions and routines, but also with labels, arrays and all forms of compound data structure. In these cases it is meaningful to ask if the information which is in the FVL or accessible through it is alterable or whether it cannot be changed at all, and this property provides the distinction between free and fixed objects.

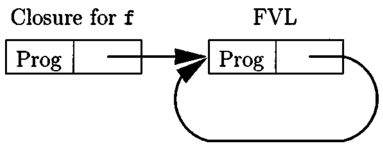
A function which has been defined recursively so that the expression representing it includes at least one mention of its own name, can also be represented rather simply by making use of closures. Suppose, for example, we take the non-recursive function

```
let  f[x]= (x=0) → 1,x*g[x-1]
```

This has a single free variable, the function g which is taken by R-value. Thus the closure for `f` would take the form



If we now identify g with `f`, so that the function becomes the recursively defined factorial, all we need to do is to ensure that the FVL contains the closure for `f`, Thus it will take the form



so that the FVL, which now contains a copy of the closure for `f`, in fact points to itself. It is a characteristic feature of recursively defined functions of all sorts that they have some sort of a closed loop in their representation.

3.6. Types and polymorphism

3.6.1. Types. Most programming languages deal with more than one sort of object—for example with integers and floating point numbers and labels and procedures. We shall call each of these a different *type* and spend a little time examining the concept of type and trying to clarify it.

A possible starting point is the remark in the CPL Working Papers [3] that “The Type of an object determines its representation and constrains the range of abstract object it may be used to represent. Both the representation and the range may be implementation dependent”. This is true, but not particularly helpful. In fact the two factors mentioned—representation and range—have very different effects. The most important feature of a representation is the space it occupies and it is perfectly possible to ignore types completely as far as representation and storage is concerned if all types occupy the same size of storage. This is in fact the position of most assembly languages and machine code—the only differences of type encountered are those of storage size.

In more sophisticated programming languages, however, we use the type to tell us what sort of object we are dealing with (i.e., to restrict its range to one sort of object). We also expect the compiling system to check that we have not made silly mistakes (such as multiplying two labels) and to interpret correctly ambiguous symbols (such as +) which mean different things according to the types of their operands. We call ambiguous operators of this sort *polymorphic* as they have several forms depending on their arguments.

The problem of dealing with polymorphic operators is complicated by the fact that the range of types sometimes overlap. Thus for example `3` may be an *integer* or a *real* and it may be necessary to change it from one type to the other. The functions which perform this operation are known as *transfer functions* and may either be used explicitly by the programmer, or, in some systems, inserted automatically by the compiling system.

3.6.2. Manifest and latent. It is natural to ask whether type is an attribute of an L-value or of an R-value—of a location or of its content. The answer to this question turns out to be a matter of language design, and the choice affects the amount of work, which can be done when a program is compiled as opposed to that which must be postponed until it is run.

In CPL the type is a property of an expression and hence an attribute of both its L-value and its R-value. Moreover L-values are invariant under assignment and this invariance includes their type. This means that the type of any particular written expression is determined solely by its position in the program. This in turn determines from their scopes which definitions govern the variables of the expression, and hence give their types. An additional rule states that the type of the result of a polymorphic operator must be determinable from a knowledge of the types of its operands without knowing their values. Thus we must be able to find the type of `a + b` without knowing the value of either `a` or `b` provided only that we know both their types.[4]

The result of these rules is that the type of every expression can be determined at compile time so that the appropriate code can be produced both for performing the operations and for storing the results.

We call attributes which can be determined at compile time in this way *manifest*; attributes that can only be determined by running the program are known as *latent*. The distinction between manifest and latent properties is not very clear cut and depends to a certain extent on questions of taste. Do we, for example, take the value of `2 + 3` to be manifest or latent? There may well be a useful and precise definition—on the other hand there may not. In either case at present we are less interested in the demarkation problem than in properties which are clearly on one side or other of the boundary.

3.6.3. Dynamic type determination. The decision in CPL to make types a manifest property of expressions was a deliberate one of language design. The opposite extreme is also worth examining. We now decide that types are to be attributes of R-values only and that any type of R-value may be assigned to any L-value. We can settle difficulties about storage by requiring that all types occupy the same storage space, but how do we ensure that the correct operations are performed for polymorphic operators? Assembly languages and other ‘simple’ languages merely forbid polymorphism. An alternative, which has interesting features, is to carry around with each R-value an indication of its type. Polymorphic operators will then be able to test this dynamically (either by hardware or program) and choose the appropriate version.

This scheme of dynamic type determination may seem to involve a great deal of extra work at run time, and it is true that in most existing computers it would slow down programs considerably. However the design of central processing units is not immutable and logical hardware of the sort required to do a limited form of type determination is relatively cheap. We should not reject a system which is logically satisfactory merely because today’s computers are unsuitable for it. If we can prove a sufficient advantage for it machines with the necessary hardware will ultimately appear even if this is rather complicated; the introduction of floating-point arithmetic units is one case when this has already happened.

3.6.4. Polymorphism. The difficulties of dealing with polymorphic operators are not removed by treating types dynamically (i.e., making them latent). The problems of choosing the correct version of the operator and inserting transfer functions if required remain more or less the same. The chief difference in treating types as manifest is that this information has to be made available to the compiler. The desire to do this leads to an examination of the various forms of polymorphism. There seem to be two main classes, which can be called ad hoc polymorphism and parametric polymorphism.

In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves ad hoc both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose results is of β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α **`list`**). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β **`list`**. We would like `Map` to work on all types of list provided f was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written

$$(\alpha \Rightarrow \beta, \alpha\ \mathbf{\texttt{list}}) \Rightarrow \beta\ \mathbf{\texttt{list}}$$

where α and β stand for any types.

Polymorphism of both classes presents a considerable challenge to the language designer, but it is not one which we shall take up here.

3.6.5. Types of functions. The type of a function includes both the types and modes of calling of its parameters and the types of its results. That is to say, in more mathematical terminology, that it includes the domain and the range of the function. Although this seems a reasonable and logical requirement, it makes it necessary to introduce the parametric polymorphism discussed above as without it functions such as `Map` have to be redefined almost every time they are used.

Some programming languages allow functions with a variable number of arguments; those are particularly popular for input and output. They will be known as *variadic* functions, and can be regarded as an extreme form of polymorphic function.[5]

A question of greater interest is whether a polymorphic function is a first class object in the sense of Section 3.5.1. If it is, we need to know what type it is. This must clearly include in some way the types of all its possible versions. Thus the type of a polymorphic function includes or specifies in some way the nature of its polymorphism. If, as in CPL, the types are manifest, all this information must be available to the compiler. Although this is not impossible, it causes a considerable increase in the complexity of the compiler and exerts a strong pressure either to forbid programmers to define new polymorphic functions or even to reduce all polymorphic functions to second class status. A decision on these points has not yet been taken for CPL.

3.7. Compound data structures

3.7.1. List processing. While programming was confined to problems of numerical analysis the need for general forms of data structure was so small that it was often ignored. For this reason ALGOL, which is primarily a language for numerical problems, contains no structure other than arrays. COBOL, being concerned with commercial data processing, was inevitably concerned with larger and more complicated structures. Unfortunately, however, the combined effect of the business man's fear of mathematics and the mathematician's contempt for business ensured that this fact had no influence on the development of general programming languages.

It was not until mathematicians began using computers for non-numerical purposes—initially in problems connected with artificial intelligence—that any general forms of compound data structure for programming languages began to be discussed. Both IPL V and LISP used data structures built up from lists and soon a number of other 'List Processing' languages were devised.

The characteristic feature of all these languages is that they are designed to manipulate more or less elaborate structures, which are built up from large numbers of components drawn from a very limited number of types. In LISP, for instance, there are only two sorts of object, an *atom* and a *cons-word* which is a doublet. The crucial feature is that each member of a doublet can itself be either an atom or another cons-word. Structures are built up by joining together a number of cons-words and atoms.

This scheme of building up complex structures from numbers of similar and much simpler elements has a great deal to recommend it. In some sense, moreover, the doublet of LISP is the simplest possible component from which to construct a structure and it is certainly possible to represent any other structure in terms of doublets. However from the practical point of view, not only for economy of implementation but also for convenience in use, the logically simplest representation is not always the best.

The later list processing languages attempted to remedy this by proposing other forms of basic building block with more useful properties, while still, of course, retaining the main plan of using many relatively simple components to form a complex structure. The resulting languages were generally very much more convenient for some classes of problems (particularly those they had been designed for) and much less suitable (possibly on grounds of efficiency) for others. They all, however, had an ad hoc look about them and arguments about their relative merits seemed somewhat unreal.

In about 1965 or 1966 interest began to turn to more general schemes for compound data structures which allowed the programmer to specify his own building blocks in some very general manner rather than having to make do with those provided by the language designer. Several such schemes are now around and in spite of being to a large extent developed independently they have a great deal in common—at least as far as the structures described in the next section as nodes are concerned. In order to illustrate these ideas, I shall outline the scheme which will probably be incorporated in CPL.

3.7.2. Nodes and elements. The building blocks from which structures are formed are known as *nodes*. Nodes may be of many types and the definition of a new node is in fact the definition of a new programmer-defined type in the sense of section 3.6. A node may be defined to consist of one or more components; both the number and the type of each component is fixed by the definition of the node. A component may be of any basic or programmer-defined type (such as a node), or may be an *element*. This represents a data object of one of a limited number of types; the actual type of object being represented is determined dynamically. An element definition also forms a new programmer-defined type in the sense of Section 3.6 and it also specifies which particular data types it may represent.

Both node and element definitions are definitions of new types, but at the same time they are used to form certain basic functions which can be used to operate on and construct individual objects of these types. Compound data structures may be built up from individuals of these types by using these functions.

The following example shows the node and element definitions which allow the lists of LISP to beformed.

```
node  Cons  is LispList : Car
       with   Cons : Cdr
element  LispList   is  Atom
                   or  Cons
node  Atom  is  string  PrintName
         with  Cons : PropertyList
```

These definitions introduce three new types: `Cons` and `Atom`, which are nodes, and `LispList` which is an element. They also define the basic selector and constructor functions which operate on them. These functions have the following effect.

If `x` is an object of type `Cons`, it has two components associated with it; the first, which is of manifest type `LispList` is obtained by applying the appropriate *selector function* `Car` to `x`, thus `Car[x]` is the first component of `x` and is of type LispList. The second component of `x` is `Cdr[x]` and is an object of type `Cons`.

If `p` is an object of type `LispList` and `q` is an object of type `Cons`, we can form a fresh node of type `Cons` whose first component is `p` and whose second component is `q` by using the *constructor function* `Cons[p,q]` which always has the same name as the node type. Thus we have the basic identities

```
Car[Cons[p,q]]= p
Cdr[Cons[p,q]]= q
```

In an exactly similar way the definition of the node `Atom` will also define the two selector functions `PrintName` and `PropertyList` and the constructor function `Atom`.

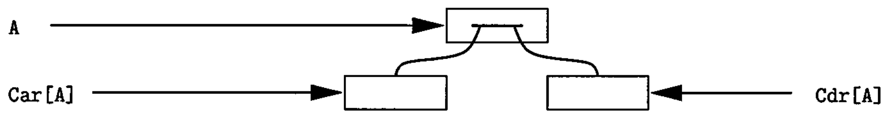
The number of components of a node is not limited to two—any non-zero number is allowed. There is also the possibility that any component may be the special object `NIL`.This can be tested for by the system predicate `Null`. Thus, for example, if end of a list is indicated by a `NIL` second component, we can test for this by the predicate `Null[Cdr[x]]`.

There is also a constructor function associated with an element type. Thus, for example if `n` is an atom, `LispList[n]` is an object of type `LispList` dynamically marked as being an atom and being in fact the atom `n`. There are two general system functions which apply to elements, both are concerned with finding their dynamically current type. The function `Type[p]` where `p` is a `LispList` will have the result either `Atom` or `Cons` according to the current type of `p`. In a similar way the system predicate `Is[Atom,p]` will have the value **`true`** if `p` is dynamically of type `Atom`.

These definitions give the basic building block of LISP using the same names with the exception of `Atom`. In Lisp `Atom[p]` is the predicate which would be written here as `Is[Atom,p]`. We use the function `Atom` to construct a new atom from a `PrintName` and a `PropertyList`.

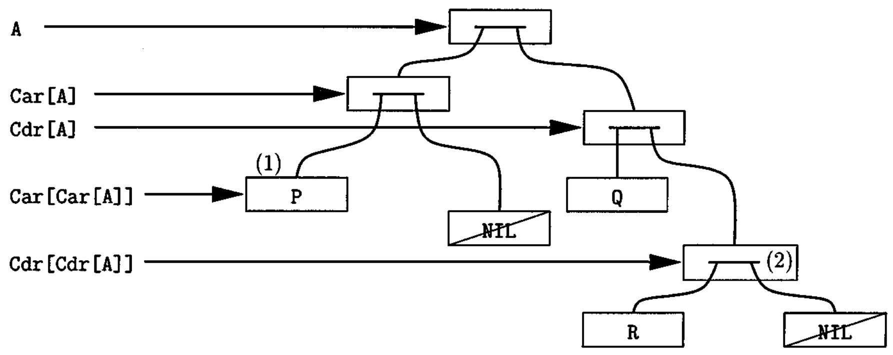
3.7.3. Assignments. In order to understand assignments in compound data structure we need to know what are the L- and R-values of nodes and their components.

Let us suppose that `A` is a *variable* of type `Cons`—i.e., that `A` is a named quantity to which we propose to assign objects of type `Cons`. The L-value of `A` presents no problems; like any other L-value it is the location where the R-value of `A` is stored. The R-value of `A` must give access to the two components of `A` (`Car[A]` and `Cdr[A]`)—i.e., it must give their L-values or locations. Thus, we have the diagram:



The L-values or locations are represented by boxes. The R-value of `A` is represented by the 'puppet strings' which lead from the inside of the L-value of `A` to the L-values of its components. One can think of the 'shape' of the box as representing its type and hence specifying the kind of occupant which may be put there.

Using this sort of diagram, it is now simple to determine the effect of an assignment. Consider the structure



The effect of obeying the assignment command

```
Car[Car[A]] := Cdr[Cdr[A]]
```

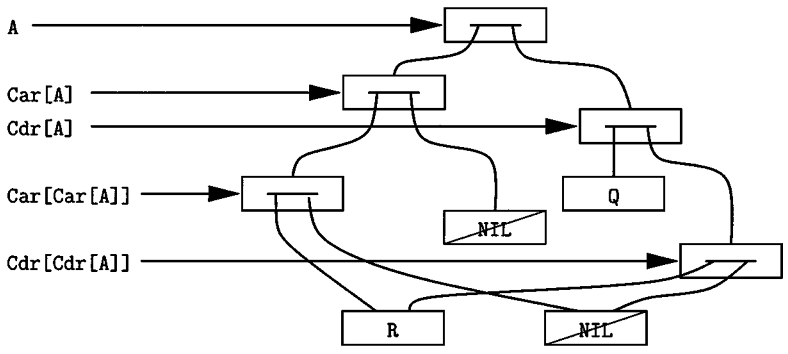
can be determined by the following steps

(1) Find the L-value of `[Car[Car[A]]`.
This is the box marked (1).
(2) Find the R-value of `[Cdr[Cdr[A]]`.
This is the puppet string marked (2).

(1) and (2) may be carried out in either order as neither actually alters the structure.

(3) Replace the contents of (1) by a copy of (2).

The resulting structure is as follows



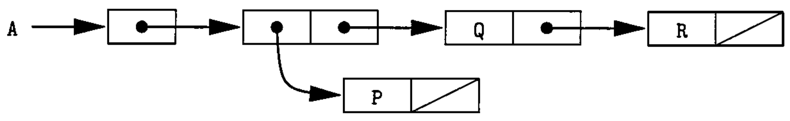
Notice that this assignment has changed the pattern of sharing in the structure so that now `Car[Car[Car[A]]]` and `Car[Cdr[Cdr[A]]]` actually share the same L-value (and hence also the same R-value). This is because the assignment statements only take a copy of the R-value of its right hand side, not a copy of all the information associated with it. In this respect, structures are similar to functions whose FVL is not copied on assignment.

Thus, as with functions, the R-value of a compound data structure gives access to all the information in the structure but does not contain it all. So that the distinction between fixed and free applies as much to structures as it does to functions.

3.7.4. Implementation. The discussion of R- and L-values of nodes has so far been quite general. I have indicated what information must be available, but in spite of giving diagrams I have not specified in any way how it should be represented. I do not propose to go into problems of implementation in any detail—in any case many of them are very machine dependent—but an outline of a possible scheme may help to clarify the concepts.

Suppose we have a machine with a word length which is a few bits longer than a single address. The R-value of a node will then be an address pointing to a small block of consecutive words, one for each component, containing the R-values of the components. An element requires for its R-value an address (e.g., the R-value of a node) and a marker to say which of the various possibilities is its dynamically current type. (There should be an escape mechanism in case there are too few bits available for the marker.) The allocation and control of storage for these nodes presents certain difficulties. A great deal of work has been done on this problem and workable systems have been devised. Unfortunately there is no time to discuss these here.

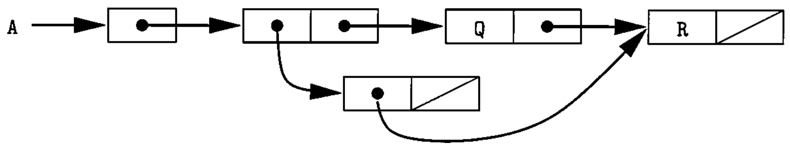
If we use an implementation of this sort for our example in the last section, we shall find that nodes of type Cons will fill two consecutive words. The 'puppet string' R-values can be replaced by the address of the first of these, so that we can redraw our diagram as



After the assignment

```
Car[Car[A]] := Cdr[Cdr[A]]
```

this becomes



3.7.5. Programming example. The following example shows the use of a recursively defined routine which has a structure as a parameter and calls it by reference (L-value). A tree sort takes place in two phases. During the first the items to be sorted are supplied in sequence as arguments to the routine AddtoTree. The effect is to build up a tree structure with an item and two branches at each node. The following node definitions define the necessary components.

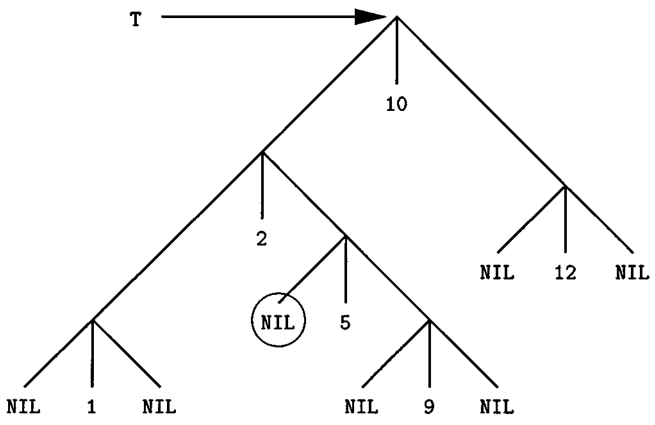
```
node Knot is Knot    : Pre
        with Knot    : Suc
        with Data    : Item

node Data is integer   Key
        with Body    : Rest
```

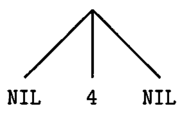
Here the key on which the sort is to be performed is an integer and the rest of the information is of type Body. The routine for the first phase is

```
rec AddtoTree[ref Knot : x, value Data : n] is
    § Test Null[x]
        If so do x := Knot[NIL,NIL,n]
        If not do AddtoTree[((Key[n] < Key[Item[x]]) → Pre[x],
                                                        Suc[x]),n]
        return §
```

The effect of this is to build up a tree where all the items accessible from the Pre (predecessor) branch of a Knot precede (i.e., have smaller keys) than the item at the Knot itself, and this in turn precedes all those which are accessible from the Suc (successor) branch.



The effect of AddtoTree(T,N) where N is a data-item whose Key is 4 would be to replace the circled NIL node by the node



where the central branch marked 4 stands for the entire data-item N.

The second phase of a tree sort forms a singularly elegant example of the use of a recursively defined routine. Its purpose is effectively to traverse the tree from left to right printing out the data-items at each Knot. The way the tree has been built up ensures that the items will be in ascending order of Keys.

We suppose that we have a routine PrintBody which will print information in a data-item in the required format. The following routine will then print out the entire tree.

```
rec PrintTree[Knot:x] is

 § Unless Null[x] do
     § PrintTree[Pre[x]]
       PrintBody[Rest[Item[x]]]
       PrintTree[Suc[x]] §
   return  §
```

3.7.6. Pointers. There is no reason why an R-value should not represent (or be) a location; such objects are known as *pointers*. Suppose X is a *real* variable with an L-value α. Then if P is an object whose R-value is α, we say the type of P is **real pointer** and that P ‘points to’ X. Notice that the type of a pointer includes the type of the thing it points to, so that pointers form an example of parametric type. (Arrays form another.) We could, for example, have another pointer Q which pointed to P; in this case Q would be of type **real pointer pointer**.

There are two basic (polymorphic) functions associated with pointers:

Follow[P] (also written ↓ P in CPL) calls its argument by *R*-value and produces as a result the *L*-value of the object pointed to. This is, apart from changes of representation, the same as its argument. Thus we have

L-value of Follow[P] = P

R-value of Follow[P] = Contents of P

The function Pointer[X] calls its argument by *L*-value and produces as a result an *R*-value which is a pointer to X.

```
Follow[Pointer[X]]
```

has the same *L*-value as X.

We can assign either to P or to Follow[P], but as their types are not the same we must be careful to distinguish which we mean.

```
P := Follow[Y]
```

will move the pointer P

```
↓ P := ↓ P + 2
```

will add 2 to the number P points to.

Pointers are useful for operating on structures and often allow the use of loops instead of recursive functions. (Whether this is an advantage or not may be a matter for discussion. With current machines and compilers loops are generally faster than recursion, but the program is sometimes harder to follow.) The following routine has the same effect as the first routine in the previous section. (It is not nearly so easy to turn the other recursive routine into a loop, although it can be done.)

```
AddtoTree'[ref Knot : x, value Data : n] is
 § let p = Pointer[x]
   until Null[↓p] do
      p := (Key[n] < Key[Item[↓p]]) → a Pointer[Pre[↓p]],
                                      Pointer[Suc[↓p]]
   ↓p := Knot[NIL,NIL,n]
   return §
```

3.7.7. Other forms of structure. Vectors and arrays are reasonably well understood. They are parametric types so that the type of an array includes its dimensionality (the number of its dimensions but not their size) and also the type of its elements. Thus unlike in nodes, all the elements of an array have to be of the same type, though their number may vary dynamically. It is convenient, though perhaps not really necessary, to regard an *n*-array (i.e., one with *n* dimensions) as a vector whose elements are $(n-1)$-arrays.

We can then regard the R-value of a vector as something rather similar to that of a node in that it gives access (or points to) the elements rather than containing them. Thus the assignment of a vector does not involve copying its elements.

Clearly if this is the case we need a system function `Copy` (or possibly `CopyVector`) which does produce a fresh copy.

There are many other possible parametric structure types which are less well understood. The following list is certainly incomplete.

List An ordered sequence of objects all of the same type. The number is dynamically variable.

Ntuple A fixed (manifest) number of objects all of the same type. This has many advantages for the implementer.

Set In the mathematical sense. An unordered collection of objects all of which are of the same type but different from each other. Operations on sets have been proposed for some languages. The lack of ordering presents considerable difficulty.

Bag or Coll This is a new sort of collection for which there is, as yet, no generally accepted name. It consists of an unordered collection of objects all of which are of the same type and differs from a set in that repetitions are allowed. (The name bag is derived from probability problems concerned with balls of various colours in a bag.) A bag is frequently the collection over which an iteration is required—e.g., when averaging.

There are also structures such as *rings* which cannot be ‘syntactically’ defined in the manner of nodes. They will probably have to be defined in terms of the primitive functions which operate on them or produce them.

It is easy enough to include any selection of these in a programming language, but the result would seem rather arbitrary. We still lack a convincing way of describing those and any other extensions to the sort of structures that a programmer may want to use.

4. Miscellaneous topics

In this section we take up a few points whose detailed discussion would have been out of place before.

4.1. Load-Update Pairs

A general L-value (location) has two important features: There is a function which gives the corresponding R-value (contents) and another which will update this. If the location is not simply addressable, it can therefore be represented by a structure with two components—a `Load` part and an `Update` part; these two can generally share a common FVL. Such an

L-value is known as a Load-Update Pair (LUP). We can now represent any location of type α by an element (in the sense of Section 3.7.2)

```
element  α Location is  α Address
                    or  α LUP

node α LUP is α Function[ ] : Load
            with  Routine [α: ∗] : Update
```

Note that these are parametrically polymorphic definitions. There is also a constraint on the components of a LUP that if `X` is an α `LUP` and `y` is of type α

```
y = value of  § Update[x][y]
            result is   Load[X] §
```

LUPs are of considerable practical value even when using machine code. A uniform system which tests a general location to see if it is addressable or not (in which case it is a LUP)—say by testing a single bit—can then use the appropriate machine instruction (e.g. `CDA` or `STO`) or apply the appropriate part of the LUP. This allows all parts of the machine to be treated in a uniform manner as if they were all addressable. In particular index registers, which may need loading by special instruction, can then be used much more freely.

Another interesting example of the use of a LUP is in dealing with the registers which set up the peripheral equipment. In some machines these registers can be set but not read by the hardware. Supervisory programs are therefore forced to keep a copy of their settings in normal store, and it is quite easy to fail to keep these two in step. If the L-value of the offending register is a LUP, and it is always referred to by this, the `Update` part can be made to change both the register and its copy, while the `Load` part reads from the copy.

The importance of this use of LUPs is that it reduces the number of ad hoc features of the machine and allows much greater uniformity by treatment. This in turn makes it easier for programmers at the machine code level to avoid oversights and other errors and, possibly more important, makes it easier to write the software programs dealing with these parts of the machine in a high level language and to compile them.

The disadvantage in current machines is that, roughly speaking, every indirect reference requires an extra test to see if the location is addressable. Although this may be unacceptable for reasons of space or time (a point of view which requires the support of much more convincing reasons than have yet been given), it would be a relatively insignificant extra complication to build a trap into the hardware for this test. It is the job of people investigating the fundamental concepts of programming to isolate the features such as this whose incorporation in the hardware of machine would allow or encourage the simplification of its software.

4.2. Macrogenerators

Throughout this course, I have adopted the point of view that programming languages are dealing with abstract objects (such as numbers or functions) and that the details of the way in which we represent these are of relatively secondary importance. It will not have escaped many readers that in the computing world, and even more so in the world of mathematicians today, this is an unfashionable if not heretical point of view. A much more conventional view is that a program is a symbol string (with the strong implication that it is nothing more), a programming language the set of rules for writing down local strings, and mathematics in general a set of rules for manipulating strings.

The outcome of this attitude is a macrogenerator whose function is to manipulate or generate symbol strings in programming languages without any regard to their semantic content. Typically such a macrogenerator produces 'code' in some language which is already implemented on the machine and whose detailed representation must be familiar to anyone writing further more definitions. It will be used to extend the power of the base language, although generally at the expense of syntactic convenience and often transparency, by adding new macrocommands.

This process should be compared with that of functional abstraction and the definition of functions and routines. Both aim to extend the power of the language by introducing now operations. Both put a rather severe limit on the syntactic freedom with which the extensions can be made.

The difference lies in the fact that macrogenerators deal with the symbols which represent the variables, values and other objects of concern to a program so that all their manipulation is performed before the final compiling. In other words all macrogeneration is manifest. Function and routine definitions on the other hand are concerned with the values themselves, not with the symbols which represent them and thus, in the first instance are dynamic (or latent) rather than manifest.

The distinction is blurred by the fact that the boundary between manifest and latent is not very clear cut, and also by the fact that it is possible by ingenuity and at the expense of clarity to do by a macrogenerator almost everything that can be done by a function definition and *vice versa*. However the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there. Each of these techniques of language extension should be used in its proper place.

Macrogeneration seems to be particularly valuable when a semantic extension of the language is required. If this is one which was not contemplated by the language designer the only alternative to trickery with macros is to rewrite the compiler—in effect to design a new language. This has normally been the situation with machine code and assembly languages and also to a large extend with operating systems. The best way to avoid spending all your time fighting the system (or language) is to use a macrogenerator and build up your own.

However with a more sophisticated language the need for a macrogenerator diminishes, and it is a fact that ALGOL systems on the whole use macrogenerators very rarely. It is, I believe, a proper aim for programming language designers to try to make the use of macrogenerators wholly unnecessary.

4.3. Formal semantics

Section 3.3 gives an outline of a possible method for formalising the semantics of programming languages. It is a development of an earlier proposal [8], but it is far from complete and cannot yet be regarded as adequate.

There are at present (Oct. 1967) only three examples of the formal description of the semantics of a real programming language, as opposed to those which deal with emasculated versions of languages with all the difficulties removed. These are the following:

(i) Landin's reduction of ALGOL to λ-expressions with the addition of assignments and jumps. This requires a special form of evaluating mechanism (which is, of course, a notional computer) to deal with the otherwise non-applicative parts of the language. The method is described in [6] and given in full in [9].

(ii) de Bakker [10] has published a formalisation of most of ALGOL based on an extension of Markov algorithms. This is an extreme example of treating the language as a symbol string. It requires no special machine except, of course, the symbol string manipulator.

(iii) A team at the IBM Laboratories in Vienna have published [12, 13] a description of PL/I which is based on an earlier evaluating mechanism for pure λ-expressions suggested by Landin [11] and the concept of a state vector for a machine suggested by McCarthy [14]. This method requires a special 'PL/I machine' whose properties and transition function are described. The whole description is very long and complex and it is hard to determine how much of this complexity is due to the method of semantic description and how much to the amorphous nature of PL/I.

The method suggested in Section 3.3 has more in common with the approach of Landin or the IBM team than it has with de Bakker's. It differs, however, in that the ultimative machine required (and all methods of describing semantics come to a machine ultimately) is in no way specialised. Its only requirement is that it should be able to evaluate pure λ-expressions. It achieves this result by explicitly bringing in the store of the computer in an abstract form, an operation which brings with it the unexpected bonus of being able to distinguish explicitly between manifest and latent properties. However until the whole of a real language has been described in these terms, it must remain as a proposal for a method, rather than a method to be recommended.

Notes

1. This is the CPL notation for a conditional expression which is similar to that used by LISP. In ALGOL the equivalent would be **if** `a > b` **then** `j` **else** `k`.
2. The ALGOL equivalent of this would have to be **if** `a > b` **then** `j := i` **else** `k := i`.
3. ALGOL 60 **call by name** Let `f` be an ALGOL procedure which calls a formal parameter `x` by name. Then a call for `f` with an actual parameter expression ε will have the same effect as forming a parameterless procedure $\lambda\,().\varepsilon$ and supplying this by value to a procedure `f*` which is derived from `f` by replacing every written occurrence of `x` in the body of `f` by `x()`. The notation $\lambda().\varepsilon$ denotes a parameterless procedure whose body is ε while `x()` denotes its application (to a null parameter list).
4. The only elementary operator to which this rule does not already apply is exponentiation. Thus, for example, if `a` and `b` are both integers $\mathtt{a}^{\mathtt{b}}$ will be an integer if $\mathtt{b} \geq 0$ and `a` real if $\mathtt{b} < 0$. If `a` and `b` are reals, the type of $\mathtt{a}^{\mathtt{b}}$ depends on the sign of `a` as well as that of `b`. In CPL this leads to a definition of $\mathtt{a} \uparrow \mathtt{b}$ which differs slightly in its domain from $\mathtt{a}^{\mathtt{b}}$.
5. By analogy with monadic, dyadic and polyadic for functions with one, two and many arguments. Functions with no arguments will be known as *anadic*. Unfortunately there appears to be no suitable Greek prefix meaning variable.

References

1. Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., and Strachey, C. The main features of CPL. *Comp. J.* **6** (1963) 134–143.
2. Buxton, J.N., Gray, J.C., and Park, D. CPL elementary programming manual, Edition II. Technical Report, Cambridge, 1966.
3. Strachey, C. (Ed.). CPL working papers. Technical Report, London and Cambridge Universities, 1966.
4. Quine, W.V. *Word and Object*. New York Technology Press and Wiley, 1960.
5. Schönfinkel, M. Über die Bausteine der mathematischen Logik. *Math. Ann.* **92** (1924) 305–316.
6. Landin, F.J. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*, T.B. Steel (Ed.). North Holland Publishing Company, Amsterdam, 1966, pp. 266–294.
7. Curry, H.B. and Feys, R. *Combinatory Logic*, Vol. 1, North Holland Publishing Company, Amsterdam, 1958.
8. Strachey, C. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming* T.B. Steel (Ed.). North Holland Publishing Company, Amsterdam, 1966, pp. 198–216.
9. Landin, P.J. A correspondence between ALGOL 60 and Church's Lambda notation. *Comm. ACM* **8** (1965) 89–101, 158–165.
10. de Bakker, J.W. *Mathematical Centre Tracts 16: Formal Definition of Programming Languages*. Mathematisch Centrum, Amsterdam, 1967.
11. Landin, P.J. The Mechanical Evaluation of Expressions. *Comp. J*. **6** (1964) 308–320.
12. PL/I—Definition Group of the Vienna Laboratory. Formal definition of PL/I. IBM Technical Report TR 25.071, 1966.
13. Alber, K. Syntactical description of PL/I text and its translation into abstract normal form. IBM Technical Report TR 25.074, 1967.
14. McCarthy, J. Problems in the theory of computation. In *Proc. IFIP Congress 1965*, Vol. 1, W.A. Kalenich (Ed.). Spartan Books, Washington, 1965, pp. 219–222.

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

$$\begin{array}{lll} \text{A5} & (r - y) + y \times (1 + q) & \\ & & = (r - y) + (y \times 1 + y \times q) \\ \text{A9} & & = (r - y) + (y + y \times q) \\ \text{A3} & & = ((r - y) + y) + y \times q \\ \text{A6} & & = r + y \times q \quad \text{provided } y \leqslant r \end{array}$$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$$\text{A10}_I \quad \neg \exists x \forall y \quad (y \leqslant x),$$

where all finite arithmetics satisfy:

$$\text{A10}_F \quad \forall x \quad (x \leqslant \max)$$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$$\text{A11}_S \quad \neg \exists x \quad (x = \max + 1) \qquad \text{(strict interpretation)}$$

$$\text{A11}_B \quad \max + 1 = \max \qquad \text{(firm boundary)}$$

$$\text{A11}_M \quad \max + 1 = 0 \qquad \text{(modulo arithmetic)}$$

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

TABLE I

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leqslant x \supset (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

TABLE II

1. Strict Interpretation

+	0	1	2	3	×	0	1	2	3
0	0	1	2	3	0	0	0	0	0
1	1	2	3	*	1	0	1	2	3
2	2	3	*	*	2	0	2	*	*
3	3	*	*	*	3	0	3	*	*

* nonexistent

2. Firm Boundary

+	0	1	2	3	×	0	1	2	3
0	0	1	2	3	0	0	0	0	0
1	1	2	3	3	1	0	1	2	3
2	2	3	3	3	2	0	2	3	3
3	3	3	3	3	3	0	3	3	3

3. Modulo Arithmetic

+	0	1	2	3	×	0	1	2	3
0	0	1	2	3	0	0	0	0	0
1	1	2	3	0	1	0	1	2	3
2	2	3	0	1	2	0	2	0	2
3	3	0	1	2	3	0	3	2	1

these properties will not necessarily obtain, unless the program is executed on an implementation which satisfies the chosen axiom.

3. Program Execution

As mentioned above, the purpose of this study is to provide a logical basis for proofs of the properties of a program. One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program. These assertions will usually not ascribe particular values to each variable, but will rather specify certain general properties of the values and the relationships holding between them. We use the normal notations of mathematical logic to express these assertions, and the familiar rules of operator precedence have been used wherever possible to improve legibility.

In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q) and a description of the result of its execution (R), we introduce a new notation:

$$P\,\{Q\}\,R.$$

This may be interpreted "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion." If there are no preconditions imposed, we write **true** $\{Q\}R$.[1]

The treatment given below is essentially due to Floyd [8] but is applied to texts rather than flowcharts.

3.1. Axiom of Assignment

Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic.

Consider the assignment statement:

$$x := f$$

where

x is an identifier for a simple variable;

f is an expression of a programming language without side effects, but possibly containing x.

Now any assertion $P(x)$ which is to be true of (the value of) x *after* the assignment is made must also have been true of (the value of) the expression f, taken *before* the assignment is made, i.e. with the old value of x. Thus if $P(x)$ is to be true after the assignment, then $P(f)$ must be true before the assignment. This fact may be expressed more formally:

D0 Axiom of Assignment

$\vdash P_0\{x := f\}\,P$

where

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting f for all occurrences of x.

It may be noticed that D0 is not really an axiom at all, but rather an axiom schema, describing an infinite set of axioms which share a common pattern. This pattern is described in purely syntactic terms, and it is easy to check whether any finite text conforms to the pattern, thereby qualifying as an axiom, which may validly appear in any line of a proof.

[1] If this can be proved in our formal system, we use the familiar logical symbol for theoremhood: $\vdash P\ \{Q\}\ R$

3.2. Rules of Consequence

In addition to axioms, a deductive science requires at least one rule of inference, which permits the deduction of new theorems from one or more axioms or theorems already proved. A rule of inference takes the form "If $\vdash X$ and $\vdash Y$ then $\vdash Z$", i.e. if assertions of the form X and Y have been proved as theorems, then Z also is thereby proved as a theorem. The simplest example of an inference rule states that if the execution of a program Q ensures the truth of the assertion R, then it also ensures the truth of every assertion logically implied by R. Also, if P is known to be a precondition for a program Q to produce result R, then so is any other assertion which logically implies P. These rules may be expressed more formally:

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$
If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

3.3. Rule of Composition

A program generally consists of a sequence of statements which are executed one after another. The statements may be separated by a semicolon or equivalent symbol denoting procedural composition: $(Q_1 ; Q_2 ; \cdots ; Q_n)$. In order to avoid the awkwardness of dots, it is possible to deal initially with only two statements $(Q_1 ; Q_2)$, since longer sequences can be reconstructed by nesting, thus $(Q_1 ; (Q_2 ; (\cdots (Q_{n-1} ; Q_n) \cdots)))$. The removal of the brackets of this nest may be regarded as convention based on the associativity of the ";-operator", in the same way as brackets are removed from an arithmetic expression $(t_1 + (t_2 + (\cdots (t_{n-1} + t_n) \cdots)))$.

The inference rule associated with composition states that if the proven result of the first part of a program is identical with the precondition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied.

In more formal terms:

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1 ; Q_2)\}R$

3.4. Rule of Iteration

The essential feature of a stored program computer is the ability to execute some portion of program (S) repeatedly until a condition (B) goes false. A simple way of expressing such an iteration is to adapt the Algol 60 **while** notation:

while B **do** S

In executing this statement, a computer first tests the condition B. If this is false, S is omitted, and execution of the loop is complete. Otherwise, S is executed and B is tested again. This action is repeated until B is found to be false. The reasoning which leads to a formulation of an inference rule for iteration is as follows. Suppose P to be an assertion which is always true on completion of S, provided that it is also true on initiation. Then obviously P will still be true after any number of iterations of the statement S (even no iterations). Furthermore, it is known that the controlling condition B is false when the iteration finally terminates. A slightly more powerful formulation is possible in light of the fact that B may be assumed to be true on initiation of S:

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{$**while** B **do** $S\}\neg B \wedge P$

3.5. Example

The axioms quoted above are sufficient to construct the proof of properties of simple programs, for example, a routine intended to find the quotient q and remainder r obtained on dividing x by y. All variables are assumed to range over a set of nonnegative integers conforming to the axioms listed in Table I. For simplicity we use the trivial but inefficient method of successive subtraction. The proposed program is:

```
((r := x;  q := 0);  while
                y ⩽ r do (r := r − y;  q := 1 + q))
```

An important property of this program is that when it terminates, we can recover the numerator x by adding to the remainder r the product of the divisor y and the quotient q (i.e. $x = r + y \times q$). Furthermore, the remainder is less than the divisor. These properties may be expressed formally:

$$\textbf{true}\ \{Q\}\ \neg y \leqslant r \wedge x = r + y \times q$$

where Q stands for the program displayed above. This expresses a necessary (but not sufficient) condition for the "correctness" of the program.

A formal proof of this theorem is given in Table III. Like all formal proofs, it is excessively tedious, and it would be fairly easy to introduce notational conventions which would significantly shorten it. An even more powerful method of reducing the tedium of formal proofs is to derive general rules for proof construction out of the simple rules accepted as postulates. These general rules would be shown to be valid by demonstrating how every theorem proved with their assistance could equally well (if more tediously) have been proved without. Once a powerful set of supplementary rules has been developed, a "formal proof" reduces to little more than an informal indication of how a formal proof could be constructed.

4. General Reservations

The axioms and rules of inference quoted in this paper have implicitly assumed the absence of side effects of the evaluation of expressions and conditions. In proving properties of programs expressed in a language permitting side effects, it would be necessary to prove their absence in each case before applying the appropriate proof technique. If the main purpose of a high level programming language is to assist in the construction and verification of correct programs, it is doubtful whether the use of functional notation to call procedures with side effects is a genuine advantage.

Another deficiency in the axioms and rules quoted above is that they give no basis for a proof that a program successfully terminates. Failure to terminate may be due to an infinite loop; or it may be due to violation of an implementation-defined limit, for example, the range of numeric operands, the size of storage, or an operating system time limit. Thus the notation "$P\{Q\}R$" should be interpreted "provided that the program successfully terminates, the properties of its results are described by R." It is fairly easy to adapt the axioms so that they cannot be used to predict the "results" of nonterminating programs; but the actual use of the axioms would now depend on knowledge of many implementation-dependent features, for example, the size and speed of the computer, the range of numbers, and the choice of overflow technique. Apart from proofs of the avoidance of infinite loops, it is probably better to prove the "conditional" correctness of a program and rely on an implementation to give a warning if it has had to abandon execution of the program as a result of violation of an implementation limit.

Finally it is necessary to list some of the areas which have not been covered: for example, real arithmetic, bit and character manipulation, complex arithmetic, fractional arithmetic, arrays, records, overlay definition, files, input/output, declarations, subroutines, parameters, recursion, and parallel execution. Even the characterization of integer arithmetic is far from complete. There does not appear to be any great difficulty in dealing with these points, provided that the programming language is kept simple. Areas which do present real difficulty are labels and jumps, pointers, and name parameters. Proofs of programs which made use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

TABLE III

Line number	Formal proof	Justification
1	$\textbf{true} \supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0 \{r := x\} x = r + y \times 0$	D0
3	$x = r + y \times 0 \{q := 0\}\ x = r + y \times q$	D0
4	$\textbf{true}\ \{r := x\}\ x = r + y \times 0$	D1 (1, 2)
5	$\textbf{true}\ \{r := x;\ \ q := 0\}\ x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x = (r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q) \{r := r-y\} x = r + y \times (1+q)$	D0
8	$x = r + y \times (1+q) \{q := 1+q\} x = r + y \times q$	D0
9	$x = (r-y) + y \times (1+q) \{r := r-y;\ q := 1+q\}\ x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r\ \{r := r-y;\ q := 1+q\}\ x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q\ \{\textbf{while}\ y \leqslant r\ \textbf{do}\ (r := r-y;\ \ q := 1+q)\}\ \neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\textbf{true}\ \{((r := x;\ \ q := 0);\ \ \textbf{while}\ y \leqslant r\ \textbf{do}\ (r := r-y;\ \ q := 1+q))\}\ \neg y \leqslant r \wedge x = r + y \times q$	D2 (5, 11)

Notes

1. The left hand column is used to number the lines, and the right hand column to justify each line, by appealing to an axiom, a lemma or a rule of inference applied to one or two previous lines, indicated in brackets. Neither of these columns is part of the formal proof. For example, line 2 is an instance of the axiom of assignment (D0); line 12 is obtained from lines 5 and 11 by application of the rule of composition (D2).
2. Lemma 1 may be proved from axioms A7 and A8.
3. Lemma 2 follows directly from the theorem proved in Sec. 2.

5. Proofs of Program Correctness

The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program, provided that the implementation of the programming language conforms to the axioms and rules which have been used in the proof. This fact itself might also be established by deductive reasoning, using an axiom set which describes the logical properties of the hardware circuits. When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.

The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error. At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out in particular cases and to modify it if the results produced do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. The time spent in this program testing is often more than half the time spent on the entire programming project; and with a realistic costing of machine time, two thirds (or more) of the cost of the project is involved in removing errors during this phase.

The cost of removing errors discovered after a program has gone into use is often greater, particularly in the case of items of computer manufacturer's software for which a large part of the expense is borne by the user. And finally, the cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus the practice of program proving is not only a theoretical pursuit, followed in the interests of academic respectability, but a serious recommendation for the reduction of the costs associated with programming error.

The practice of proving programs is likely to alleviate some of the other problems which afflict the computing world. For example, there is the problem of program documentation, which is essential, firstly, to inform a potential user of a subroutine how to use it and what it accomplishes, and secondly, to assist in further development when it becomes necessary to update a program to meet changing circumstances or to improve it in the light of increased knowledge. The most rigorous method of formulating the purpose of a subroutine, as well as the conditions of its proper use, is to make assertions about the values of variables before and after its execution. The proof of the correctness of these assertions can then be used as a lemma in the proof of any program which calls the subroutine. Thus, in a large program, the structure of the whole can be clearly mirrored in the structure of its proof. Furthermore, when it becomes necessary to modify a program, it will always be valid to replace any subroutine by another which satisfies the same criterion of correctness. Finally, when examining the detail of the algorithm, it seems probable that the proof will be helpful in explaining not only *what* is happening but *why*.

Another problem which can be solved, insofar as it is soluble, by the practice of program proofs is that of transferring programs from one design of computer to another. Even when written in a so-called machine-independent programming language, many large programs inadvertently take advantage of some machine-dependent property of a particular implementation, and unpleasant and expensive surprises can result when attempting to transfer it to another machine. However, presence of a machine-dependent feature will always be revealed in advance by the failure of an attempt to prove the program from machine-independent axioms. The programmer will then have the choice of formulating his algorithm in a machine-independent fashion, possibly with the help of environment enquiries; or if this involves too much effort or inefficiency, he can deliberately construct a machine-dependent program, and rely for his proof on some machine-dependent axiom, for example, one of the versions of A11 (Section 2). In the latter case, the axiom must be explicitly quoted as one of the preconditions of successful use of the program. The program can still, with complete confidence, be transferred to any other machine which happens to satisfy the same machine-dependent axiom; but if it becomes necessary to transfer it to an implementation which does not, then all the places where changes are required will be clearly annotated by the fact that the proof at that point appeals to the truth of the offending machine-dependent axiom.

Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased only at the price of simplicity.

6. Formal Language Definition

A high level programming language, such as ALGOL, FORTRAN, or COBOL, is usually intended to be implemented on a variety of computers of differing size, configuration, and design. It has been found a serious problem to define these languages with sufficient rigour to ensure compatibility among all implementors. Since the purpose of compatibility is to facilitate interchange of programs expressed in the language, one way to achieve this would be to insist that all implementations of the language shall "satisfy" the axioms and rules of inference which underlie proofs of the properties of programs expressed in the language, so that all predictions based on these proofs will be fulfilled, except in the event of hardware failure. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.

Apart from giving an immediate and possibly even provable criterion for the correctness of an implementation, the axiomatic technique for the definition of programming language semantics appears to be like the formal syntax of the ALGOL 60 report, in that it is sufficiently simple to be understood both by the implementor and by the reasonably sophisticated user of the language. It is only by bridging this widening communication gap in a single document (perhaps even provably consistent) that the maximum advantage can be obtained from a formal language definition.

Another of the great advantages of using an axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined*, for example, range of integers, accuracy of floating point, and choice of overflow technique. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs. Thus a programming language standard should consist of a set of axioms of universal applicability, together with a choice from a set of supplementary axioms describing the range of choices facing an implementor. An example of the use of axioms for this purpose was given in Section 2.

Another of the objectives of formal language definition is to assist in the design of better programming languages. The regularity, clarity, and ease of implementation of the ALGOL 60 syntax may at least in part be due to the use of an elegant formal technique for its definition. The use of axioms may lead to similar advantages in the area of "semantics," since it seems likely that a language which can

(Continued on p. 583)

by Lowe. In addition, we define $F(j) = \sum_{i=1}^{j-1} f(i)$ and write $\lfloor x \rfloor$ for the greatest integer not exceeding x.

In a packed list file, the bucket which contains the first element of list j will have its first

$$F(j) - C\lfloor F(j)/C \rfloor \qquad (1)$$

positions occupied by lists $j - 1, j - 2, \cdots$. For any practical file, when j is not a small integer, (1) behaves as a random variable uniformly distributed between 0 and C. In other words, the start of a list is independent of bucket boundaries. It is easy to see that the expected number of accesses required to retrieve list j is $f(j)/C + 1$.

Hence we have

$$T_r = t_s \left\{ \sum_{j=1}^{N} (f(j)/C + 1)p(j) \right\},$$
$$\therefore \quad T_r/t_s = 1 + \sum_{j=1}^{N} f(j)p(j)/C, \qquad (2)$$

since $\sum_{j=1}^{N} p(j) = 1$. Equation (2) corresponds to (11) in [1].

The assumptions on $f(j)$ and $p(j)$ in [1] may be substituted into (2). The first two assumptions yield

$$T_r/t_s = 1 + S/NC, \qquad (3)$$

and the third assumption, for large N, yields approximately

$$T_r/t_s = 1 + (lnN + \gamma)^{-2}\pi^2/6. \qquad (4)$$

These equations should be compared with the right-hand inequalities of (13) and (24) in [1].

Received March 1969; revised June 1969

REFERENCES

1. Lowe, Thomas C. The influence of data-base characteristics and usage on direct-access file organization. *J. ACM 15*, 4 (Oct. 1968), 535–548.

C. A. R. HOARE—*cont'd from page 580*

be described by a few "self-evident" axioms from which proofs will be relatively easy to construct will be preferable to a language with many obscure axioms which are difficult to apply in proofs. Furthermore, axioms enable the language designer to express his general *intentions* quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions. Finally, axioms can be formulated in a manner largely independent of each other, so that the designer can work freely on one axiom or group of axioms without fear of unexpected interaction effects with other parts of the language.

Acknowledgments. Many axiomatic treatments of computer programming [1, 2, 3] tackle the problem of proving the equivalence, rather than the correctness, of algorithms. Other approaches [4, 5] take recursive functions rather than programs as a starting point for the theory. The suggestion to use axioms for defining the primitive operations of a computer appears in [6, 7]. The importance of program proofs is clearly emphasized in [9], and an informal technique for providing them is described. The suggestion that the specification of proof techniques provides an adequate formal definition of a programming language first appears in [8]. The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be: (1) a suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined; (2) a comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for formal language definition.

However, the formal material presented here has only an expository status and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by others.

Received November, 1968; revised May, 1969

REFERENCES

1. Yanov, Yu I. Logical operator schemes. *Kybernetika 1*, (1958).
2. Igarashi, S. An axiomatic approach to equivalence problems of algorithms with applications. Ph.D. Thesis 1964. Rep. Compt. Centre, U. Tokyo, 1968, pp. 1–101.
3. de Bakker, J. W. Axiomatics of simple assignment statements. M.R. 94, Mathematisch Centrum, Amsterdam, June 1968.
4. McCarthy, J. Towards a mathematical theory of computation. Proc. IFIP Cong. 1962, North Holland Pub. Co., Amsterdam, 1963.
5. Burstall, R. Proving properties of programs by structural induction. Experimental Programming Reports: No. 17 DMIP, Edinburgh, Feb. 1968.
6. van Wijngaarden, A. Numerical analysis as an independent science. *BIT 6* (1966), 66–81.
7. Laski, J. Sets and other types. ALGOL Bull. 27, 1968.
8. Floyd, R. W. Assigning meanings to programs. Proc. Amer. Math. Soc. Symposia in Applied Mathematics, Vol. 19, pp. 19–31.
9. Naur, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310–316.

This paper appears in ECOOP '98, the 12th European Conference on Object-Oriented Programming, Brussels, Belgium, July 20-24, 1998, pp. 186-211.

Predicate Dispatching: A Unified Theory of Dispatch

Michael Ernst, Craig Kaplan, and Craig Chambers

Department of Computer Science and Engineering
University of Washington
Seattle, WA, USA 98195-2350
{mernst,csk,chambers}@cs.washington.edu
http://www.cs.washington.edu/research/projects/cecil/

Abstract. *Predicate dispatching* generalizes previous method dispatch mechanisms by permitting arbitrary predicates to control method applicability and by using logical implication between predicates as the overriding relationship. The method selected to handle a message send can depend not just on the classes of the arguments, as in ordinary object-oriented dispatch, but also on the classes of subcomponents, on an argument's state, and on relationships between objects. This simple mechanism subsumes and extends object-oriented single and multiple dispatch, ML-style pattern matching, predicate classes, and classifiers, which can all be regarded as syntactic sugar for predicate dispatching. This paper introduces predicate dispatching, gives motivating examples, and presents its static and dynamic semantics. An implementation of predicate dispatching is available.

1 Introduction

Many programming languages support some mechanism for dividing the body of a procedure into a set of cases, with a declarative mechanism for selecting the right case for each dynamic invocation of the procedure. Case selection can be broken down into tests for *applicability* (a case is a candidate for invocation if its guard is satisfied) and *overriding* (which selects one of the applicable cases for invocation).

Object-oriented languages use overloaded methods as the cases and generic functions (implicit or explicit) as the procedures. A method is applicable if the run-time class of the receiver argument is the same as or a subclass of the class on which the receiver is specialized. Multiple dispatching [BKK$^+$86,Cha92] enables testing the classes of all of the arguments. One method overrides another if its specializer classes are subclasses of the other's, using either lexicographic (CLOS [Ste90]) or pointwise (Cecil [Cha93a]) ordering.

Predicate classes [Cha93b] automatically classify an object of class A as an instance of virtual subclass B (a subclass of A) whenever B's predicate (an arbitrary expression typically testing the runtime state of an object) is true. This creation of virtual class hierarchies makes method dispatching applicable even in cases where the effective class of an object may change over time. Classifiers [HHM90b] and modes [Tai93] are similar mechanisms for reclassifying an object into one of a number of subclasses based on a case-statement-like test of arbitrary boolean conditions.

Pattern matching (as in ML [MTH90]) bases applicability tests on the run-time datatype constructor tags of the arguments and their subcomponents. As with classifiers and modes, textual ordering determines overriding. Some languages, such as Haskell [HJW+92], allow arbitrary boolean guards to accompany patterns, restricting applicability. Views [Wad87] extend pattern matching to abstract data types by enabling them to offer interfaces like various concrete datatypes.

Predicate dispatching integrates, generalizes, and provides a uniform interface to these similar but previously incomparable mechanisms. A method declaration specifies its applicability via a *predicate expression*, which is a logical formula over class tests (i.e., tests that an object is of a particular class or one of its subclasses) and arbitrary boolean-valued expressions from the underlying programming language. A method is applicable when its predicate expression evaluates to *true*. Method m_1 overrides method m_2 when m_1's predicate logically implies that of m_2; this relationship is computed at compile time. Static typechecking verifies that, for all possible combinations of arguments to a generic function, there is always a single most-specific applicable method. This ensures that there are no "message not understood" errors (called "match not exhaustive" in ML) or "message ambiguous" errors at run-time.

Predicate expressions capture the basic primitive mechanisms underlying a wide range of declarative dispatching mechanisms. Combining these primitives in an orthogonal and general manner enables new sorts of dispatching that are not expressible by previous dispatch mechanisms. Predicate dispatching preserves several desirable properties from its object-oriented heritage, including that methods can be declared in any order and that new methods can be added to existing generic functions without modifying the existing methods or clients; these properties are not shared by pattern-matching-based mechanisms.

Section 2 introduces the syntax, semantics, and use of predicate dispatching through a series of examples. Section 3 defines its dynamic and static semantics formally. Section 4 discusses predicate tautology testing, which is the key mechanism required by the dynamic and static semantics. Section 5 surveys related work. Section 6 concludes with a discussion of future directions for research.

2 Overview

This section demonstrates some of the capabilities of predicate dispatching by way of a series of examples. We incrementally present a high-level syntax which appears in full in Fig. 6; Fig. 1 lists supporting syntactic domains. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions.

E	$\in$ *expr*	The set of expressions in the underlying programming language
Body	$\in$ *method-body*	The set of method bodies in the underlying programming language
T	$\in$ *type*	The set of types in the underlying programming language
c	$\in$ *class-id*	The namespace of classes
m	$\in$ *method-id*	The namespace of methods and fields
f	$\in$ *field-id*	The namespace of methods and fields
p	$\in$ *pred-id*	The namespace of predicate abstractions
v, w	$\in$ *var-id*	The namespace of variables

Fig. 1. Syntactic domains and variables. Method and field names appear in the same namespace; the *method-id* or *field-id* name is chosen for clarity in the text.

Predicate dispatching is parameterized by the syntax and semantics of the host programming language in which predicate dispatching is embedded. The ideas of predicate dispatching are independent of the host language; this paper specifies only a predicate dispatching sublanguage, with *expr* the generic nonterminal for expressions in the host language. Types and signatures are used when the host language is statically typed and omitted when it is dynamically typed.

2.1 Dynamic dispatch

Each method implementation has a predicate expression which specifies when the method is applicable. Class tests are predicate expressions, as are negations, conjunctions, and disjunctions of predicate expressions.

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) **:** *type*	
method-decl	::=	**method** *method-id* (⟨ *formal-pattern* ⟩) [**when** *pred-expr*] *method-body*	
formal-pattern	::=	*var-id*	
pred-expr	::=	*expr* @ *class-id*	succeeds if *expr* evaluates to an instance of *class-id* or one of its subclasses
	|	**not** *pred-expr*	negation
	|	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	|	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)

Predicate expressions are evaluated in an environment with the method's formal arguments bound (see Sect. 3 for details). An omitted **when** clause indicates that the method handles all (type-correct) arguments.

Method signature declarations give the type signature shared by a family of method implementations in a generic function. A message send expression need examine only the corresponding method signature declaration to determine its type-correctness, while a set of overloaded method implementations must completely and unambiguously implement the corresponding signature in order to be type-correct.

Predicate dispatching can simulate both singly- and multiply-dispatched methods by *specializing* formal parameters on a class (via the "@*class-id*" syntax).

Specialization limits the applicability of a method to objects that are instances of the given class or one of its subclasses. More generally, predicate dispatching supports the construction of arbitrary conjunctions, disjunctions, and negations of class tests. The following example uses predicate dispatching to implement the `Zip` function which converts a pair of lists into a list of pairs:[1]

```
type List;
  class Cons subtypes List { head:Any, tail:List };
  class Nil subtypes List;

signature Zip(List, List):List;
method Zip(l1, l2) when l1@Cons and l2@Cons {
    return Cons(Pair(l1.head, l2.head), Zip(l1.tail, l2.tail)); }
method Zip(l1, l2) when l1@Nil or l2@Nil { return Nil; }
```

The first `Zip` method applies when both of its arguments are instances of `Cons` (or some subclass). The second `Zip` method uses disjunction to test whether either argument is an instance of `Nil` (or some subclass). The type checker can verify statically that the two implementations of `Zip` are mutually exclusive and exhaustive over all possible arguments that match the signature, ensuring that there will be no "message not understood" or "message ambiguous" errors at run-time, without requiring the cases to be put in any particular order.

There are several unsatisfactory alternatives to the use of implication to determine overriding relationships. ML-style pattern matching requires all cases to be written in one place and put in a particular total order, resolving ambiguities in favor of the first successfully matching pattern. Likewise, a lexicographic ordering for multimethods [Ste90] is error-prone and unnatural, and programmers are not warned of potential ambiguities. In a traditional (singly- or multiply-dispatched) object-oriented language without the ability to order cases, either the base case of `Zip` must be written as the default case for all pairs of `List` objects (unnaturally, and unsafely in the face of future additions of new subclasses of `List`), or *three* separate but identical base methods must be written: one for `Nil`$\times$`Any`, one for `Any`$\times$`Nil`, and a third for `Nil`$\times$`Nil` to resolve the ambiguity between the first two. In our experience with object-oriented languages (using a pointwise, not lexicographic, ordering), these triplicate base methods for binary messages occur frequently.

As a syntactic convenience, class tests can be written in the formal argument list:

formal-pattern ::= [*var-id*] [@ *class-id*]   like *var-id* @ *class-id* in *pred-expr*

The class name can be omitted if the argument is not dispatched upon, and the formal name can be omitted if the argument is not used elsewhere in the predicate or method body.

The first `Zip` method above could then be rewritten as

[1] `Any` is the top class, subclassed by all other classes, and `Pair` returns an object containing its two arguments.

```
  method Zip(l1@Cons, l2@Cons) {
    return Cons(Pair(l1.head, l2.head), Zip(l1.tail, l2.tail)); }
```

This form uses an implicit conjunction of class tests, like a multimethod.

2.2 Pattern matching

Predicates can test the run-time classes of components of an argument, just as pattern matching can query substructures, by suffixing the **@***class* test with a record-like list of field names and corresponding class tests; names can be bound to field contents at the same time.

$$\begin{array}{lcl} \textit{pred-expr} & ::= & \ldots \\ & | & \textit{expr} \ @ \ \textit{specializer} \\ \textit{specializer} & ::= & \textit{class-id} \ [\ \{ \ \langle \ \textit{field-pat} \ \rangle \ \} \ ] \\ \textit{field-pat} & ::= & \textit{field-id} \ [\ = \textit{var-id} \ ] \ [\ @ \ \textit{specializer} \ ] \end{array}$$

A *specializer* succeeds if all of the specified fields (or results of invoking methods named by *field-id*) satisfy their own *specializer*s, in which case the *var-id*s are bound to the field values or method results. As with *formal-pattern*, the formal name or specializer may be omitted.

Our syntax for pattern matching on records is analogous to that for creating a record: `{ x := 7, y := 22 }` creates a two-component record, binding the `x` field to 7 and the `y` field to 22, while `{ x = xval }` pattern-matches against a record containing an `x` field, binding the new variable `xval` to the contents of that field and ignoring any other fields that might be present. The similarity between the record construction and matching syntaxes follows ML. Our presentation syntax also uses curly braces in two other places: for record type specifiers (as in the declaration of the `Cons` class, above) and to delimit code blocks (as in the definitions of the `Zip` methods, above).

The following example, adapted from our implementation of an optimizing compiler, shows how a `ConstantFold` method can dispatch for binary operators whose arguments are constants and whose operator is integer addition:

```
type Expr;
  signature ConstantFold(Expr):Expr;
  -- default constant-fold optimization: do nothing
  method ConstantFold(e) { return e; }

  type AtomicExpr subtypes Expr;
    class VarRef subtypes AtomicExpr { ... };
    class IntConst subtypes AtomicExpr { value:int };
    ... -- other atomic expressions here

  type Binop;
    class IntPlus subtypes Binop { ... };
    class IntMul subtypes Binop { ... };
    ... -- other binary operators here
```

```
class BinopExpr subtypes Expr { op:Binop, arg1:Expr, arg2:Expr, ... };
  -- override default to constant-fold binops with constant arguments
  method ConstantFold(e@BinopExpr{ op@IntPlus, arg1@IntConst, arg2@IntConst }) {
    return new IntConst{ value := e.arg1.value + e.arg2.value }; }
... -- more similarly expressed cases for other binary and unary operators here
```

The ability in pattern matching to test for constants of built-in types is a simple extension of class tests. In a prototype-based language, @ operates over objects as well as classes, as in "*answer* @ 42".

As with pattern matching, testing the representation of components of an object makes sense when the object and the tested components together implement a single abstraction. We do not advocate using pattern matching to test components of objects in a way that crosses natural abstraction boundaries.

2.3 Boolean expressions

To increase the expressiveness of predicate dispatching, predicates may test arbitrary boolean expressions from the underlying programming language. Additionally, names may be bound to values, for use later in the predicate expressions and in the method body. Expressions from the underlying programming language that appear in predicate expressions should have no externally observable side effects.[2]

pred-expr ::= ...		
|	**test** *expr*	succeeds if *expr* evaluates to *true*
|	**let** *var-id* := *expr*	evaluate *expr* and bind *var-id* to its value; always succeeds

The following extension to the `ConstantFold` example illustrates these features. Recall that in `{ value=v }`, the left-hand side is a field name and the right-hand side is a variable being bound.

```
-- Handle case of adding zero to anything (but don't be ambiguous
-- with existing method for zero plus a constant).
method ConstantFold(
          e@BinopExpr{ op@IntPlus, arg1@IntConst{ value=v }, arg2=a2 })
  when test(v == 0) and not(a2@IntConst) {
  return a2; }
method ConstantFold(
          e@BinopExpr{ op@IntPlus, arg1=a1, arg2@IntConst{ value=v } })
  when test(v == 0) and not(a1@IntConst) {
  return a1; }

... -- other special cases for operations on 0,1 here
```

[2] We do not presently enforce this restriction, but there is no guarantee regarding in what order or how many times predicate expressions are evaluated.

2.4 Predicate abstractions

Named predicate abstractions can factor out recurring tests and give names to semantically meaningful concepts in the application domain. Named predicates abstract over both tests and variable bindings — the two capabilities of inline predicate expressions — by both succeeding or failing and returning a record-like set of bindings. These bindings resemble the fields of a record or class, and similar support is given to pattern matching against a subset of the results of a named predicate invocation. Predicate abstractions thus can act like views or virtual subclasses of some object (or tuple of objects), with the results of predicate abstractions acting like the virtual fields of the virtual class. If the properties of an object tested by a collection of predicates are mutable, the object may be given different virtual subclass bindings at different times in its life, providing the benefits of using classes to organize code even in situations where an object's "class" is not fixed.

```
pred-sig    ::= predsignature pred-id ( ⟨ type ⟩ )
                  [ return { ⟨ field-id : type ⟩ } ]
pred-decl   ::= predicate pred-id ( ⟨ formal-pattern ⟩ )
                  [ when pred-expr ] [ return { ⟨ field-id := expr ⟩ } ]
pred-expr   ::= ...
              | pred-id ( ⟨ expr ⟩ ) [ => { ⟨ field-pat ⟩ } ]
                                      test predicate abstraction
specializer ::= class-spec [ { ⟨ field-pat ⟩ } ]
class-spec  ::= class-id              expr @ class-id is a class test
              | pred-id               expr @ pred-id [ {...} ] is alternate syntax
                                        for pred-id(expr) [ =>{...} ]
```

A predicate abstraction takes a list of arguments and succeeds or fails as determined by its own predicate expression. A succeeding predicate abstraction invocation can expose bindings of names to values it computed during its evaluation, and the caller can retrieve any subset of the predicate abstraction's result bindings. Predicate signatures specify the type interface used in typechecking predicate abstraction callers and implementations. In this presentation, we prohibit recursive predicates.

Simple predicate abstractions can be used just like ordinary classes:

```
predicate on_x_axis(p@point)
  when (p@cartesianPoint and test(p.y == 0))
    or (p@polarPoint and (test(p.theta == 0) or test(p.theta == pi)));

method draw(p@point) { ... }        -- draw the point
method draw(p@on_x_axis) { ... }    -- use a contrasting color so point is visible
```

In the following example, taken from our compiler implementation, `CFG_2succ` is a control flow graph (CFG) node with two successors. Each successor is marked with whether it is a loop exit (information which, in our implementation, is dynamically maintained when the CFG is modified) and the innermost loop it does not exit. It is advantageous for an iterative dataflow algorithm to propagate values along the loop exit only after reaching a fixed point within the loop; such an algorithm would dispatch on the `LoopExit` predicate. Similarly, the algorithm could switch from iterative to non-iterative mode when exiting the outermost loop, as indicated by `TopLevelLoopExit`.

```
predsignature LoopExit(CFGnode)
  return { loop:CFGloop };
predicate LoopExit(n@CFG_2succ{ next_true = t, next_false = f })
  when test(t.is_loop_exit) or test(f.is_loop_exit)
  return { loop := outermost(t.containing_loop, f.containing_loop) };
predicate TopLevelLoopExit(n@LoopExit{ loop@TopLevelScope });
```

Only one definition per predicate abstraction is permitted; App. B relaxes this restriction.

Because object identity is not affected by these different views on an object, named predicate abstractions are more flexible than coercions in environments with side-effects. Additionally, a single object can be classified in multiple independent ways by different predicate abstractions without being forced to define all the possible conjunctions of independent predicates as explicit classes, relieving some of the problems associated with a mix-in style of class organization [HHM90b,HHM90a].

2.5 Classifiers

Classifiers [HHM90b] are a convenient syntax for imposing a linear ordering on a collection of predicates, ensuring mutual exclusion. They combine the state testing of predicate classes and the total ordering of pattern matching. An optional `otherwise` case, which executes if none of the predicates in the classifier evaluates to true, adds the guarantee of complete coverage. Multiple independent classifications of a particular class or object do not interfere with one another.

classifier-decl ::= **classify** (⟨ *formal-pattern* ⟩)
⟨ **as** *pred-id* **when** *pred-expr* [**return** { ⟨ *field-id* := *expr* ⟩ }] ⟩
[**as** *pred-id* **otherwise** [**return** { ⟨ *field-id* := *expr* ⟩ }]]

Here is an example of the use of classifiers:

```
class Window { ... }

classify(w@Window)
  as Iconified when test(w.iconified)
  as FullScreen when test(w.area() == RootWindow.area())
  as Big when test(w.area() > RootWindow.area()/2)
  as Small otherwise;

method move(w@FullScreen, x@int, y@int) { }      -- nothing to do
method move(w@Big, x@int, y@int) { ... }         -- move a wireframe outline
method move(w@Small, x@int, y@int) { ... }       -- move an opaque window
method move(w@Iconified, x@int, y@int) { ... }  -- modify icon coordinates

-- resize, maximize, and iconify similarly test these predicates
```

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) **:** *type*	
method-decl	::=	**method** *method-id* (⟨ *var-id* ⟩) **when** *pred-expr method-body*	
pred-sig	::=	**predsignature** *pred-id* (⟨ *type* ⟩) **return** **{** ⟨ *field-id* **:** *type* ⟩ **}**	
pred-decl	::=	**predicate** *pred-id* (⟨ *var-id* ⟩) **when** *pred-expr* **return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**	
$P, Q \in$ *pred-expr*	::=	*var-id* **@** *class-id*	succeeds if *var-id* is an instance of *class-id* or a subclass
	|	**test** *var-id*	succeeds if *var-id*'s value is *true*
	|	**let** *var-id* **:=** *expr*	evaluate *expr* and bind *var-id* to that value; always succeeds
	|	*pred-id* (⟨ *var-id* ⟩) **=>** **{** ⟨ *field-id* **=** *var-id* ⟩ **}**	test predicate abstraction
	|	**true**	always succeeds
	|	**not** *pred-expr*	negation
	|	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	|	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)

Fig. 2. Abstract syntax of the core language. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions. The text uses parentheses around *pred-exprs* to indicate order of operations. Each predicate may be defined only once (App. B relaxes this restriction), and recursive predicates are forbidden.

Classifiers introduce no new primitives, but provide syntactic support for a common programming idiom. To force the classification to be mutually exclusive, each case is transformed into a predicate which includes the negation of the disjunction of all previous predicates (for details, see App. A). Therefore, an object is classified by some case only when it cannot be classified by any earlier case.

3 Dynamic and static semantics

The rest of this paper formalizes the dynamic and static semantics of a core predicate dispatching sublanguage. Figure 2 presents the abstract syntax of the core sublanguage which is used throughout this section. Appendix A defines desugaring rules that translate the high-level syntax of Fig. 6 into the core syntax.

In the remainder of this paper, we assume that all variable names are distinct so that the semantic rules can ignore the details of avoiding variable capture.

3.1 Dynamic semantics

This section explains how to select the most-specific applicable method at each message send. This selection relies on two key tests on predicated methods: whether a method is applicable to a call, and whether one method overrides another.

$\alpha, \beta \in \mathit{value}$	Values in the underlying programming language
$b \in \{\mathit{true}, \mathit{false}\}$	Mathematical booleans
$K \in (\textit{var-id} \rightarrow \mathit{value})$ $\cup (\textit{pred-id} \rightarrow \textit{pred-decl})$	Environments mapping variables to values and predicate names to predicate declarations

$\mathrm{lookup}(v, K) \rightarrow \alpha$	Look up variable v in environment K, returning the value α.
$K[v := \alpha] \rightarrow K'$	Bind name v to value α in environment K, resulting in the new environment K'. Any existing binding for v is overridden.
$\mathrm{eval}(E, K) \rightarrow \alpha$	Evaluate expression E in environment K, returning the value α.
$\mathrm{instanceof}(\alpha, c) \rightarrow b$	Determine whether value α is an instance of c or a subclass of c.
$\mathrm{accept}(\alpha) \rightarrow b$	Coerce arbitrary program values to *true* or *false*, for use with **test**.

Fig. 3. Dynamic semantics domains and helper functions. Evaluation rules appear in Fig. 4. The host programming language supplies functions eval, instanceof, and accept.

A method is applicable if its predicate evaluates to *true*. Predicate evaluation also provides an extended environment in which the method's body is executed. Bindings created via **let** in a predicate may be used in a method body, predicate **return** clause, or the second conjunct of a conjunction whose first conjunct created the binding. Such bindings permit reuse of values without recomputation, as well as simplifying and clarifying the code. Figures 3 and 4 define the execution model of predicate evaluation.

Predicate dispatching considers one method m_1 to override another method m_2 exactly when m_1's predicate implies m_2's predicate and not vice versa. Section 4 describes how to compute the overriding relation, which can be performed at compile time.

Given the evaluation model for predicate expressions and the ability to compare predicate expressions for overriding, the execution of generic function invocations is straightforward. Suppose that generic function m is defined with the following cases:

$$\begin{array}{l}
\textbf{method}\, m(v_1, \ldots, v_n) \ \ \textbf{when}\, P_1 \ \mathit{Body}_1 \\
\textbf{method}\, m(v_1, \ldots, v_n) \ \ \textbf{when}\, P_2 \ \mathit{Body}_2 \\
\vdots \\
\textbf{method}\, m(v_1, \ldots, v_n) \ \ \textbf{when}\, P_k \ \mathit{Body}_k
\end{array}$$

To evaluate the invocation $m(E_1, \ldots, E_n)$ in the environment K, first obtain $\alpha_i = \mathrm{eval}(E_i, K)$ for all $i = 1, \ldots, n$. Then, for $j = 1, \ldots, k$, obtain a truth value b_j and a new environment K_j through $\langle P_j, K[v_1 := \alpha_1, \ldots, v_n := \alpha_n] \rangle \Rightarrow \langle b_j, K_j \rangle$, as in the predicate invocation rules of Fig. 4.[3]

Now let I be the set of integers i such that $b_i = \mathit{true}$, and find $i_0 \in I$ such that P_{i_0} overrides all others in $\{P_i\}_{i \in I}$. The result of evaluating $m(E_1, \ldots, E_n)$

[3] Since we assume that all variable names are distinct and disallow lexically nested predicate abstractions, we can safely use the dynamic environment at the call site instead of preserving the static environment at the predicate abstraction's definition point.

$$\frac{}{\langle \textbf{true}, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\text{lookup}(v, K) = \alpha \qquad \text{instanceof}(\alpha, c) = b}{\langle v \mathbin{@} c, K\rangle \Rightarrow \langle b, K\rangle}$$

$$\frac{\text{lookup}(v, K) = \alpha \qquad \text{accept}(\alpha) = b}{\langle \textbf{test}\ \ v, K\rangle \Rightarrow \langle b, K\rangle}$$

$$\frac{\text{eval}(E, K) = \alpha \qquad K[v := \alpha] = K'}{\langle \textbf{let}\ \ v \mathbin{\textbf{:=}} E, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}$$

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} \quad \text{lookup}(v_i, K) = \alpha_i \\ \text{lookup}(p, K) = \textbf{predicate}\, p(v'_1, \ldots, v'_n)\, \textbf{when}\ \ P\, \textbf{return}\ \ \{f_1 \mathbin{\textbf{:=}} w'_1, \ldots, f_m \mathbin{\textbf{:=}} w'_m, \ldots\} \\ \langle P, K[v'_1 := \alpha_1, \ldots, v'_n := \alpha_n]\rangle \Rightarrow \langle \mathit{false}, K'\rangle\end{array}}{\langle p(v_1, \ldots, v_n) \mathbin{\textbf{=>}} \{f_1 = w_1, \ldots, f_m = w_m\}, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} \quad \text{lookup}(v_i, K) = \alpha_i \\ \text{lookup}(p, K) = \textbf{predicate}\, p(v'_1, \ldots, v'_n)\, \textbf{when}\ \ P\, \textbf{return}\ \ \{f_1 \mathbin{\textbf{:=}} w'_1, \ldots, f_m \mathbin{\textbf{:=}} w'_m, \ldots\} \\ \langle P, K[v'_1 := \alpha_1, \ldots, v'_n := \alpha_n]\rangle \Rightarrow \langle \mathit{true}, K'\rangle \\ \forall i \in \{1, \ldots, m\} \quad \text{lookup}(w'_i, K') = \beta_i \\ K[w_1 := \beta_1, \ldots, w_m := \beta_m] = K'' \qquad\qquad (*)\end{array}}{\langle p(v_1, \ldots, v_n) \mathbin{\textbf{=>}} \{f_1 = w_1, \ldots, f_m = w_m\}, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle b, K'\rangle}{\langle \textbf{not}\, P, K\rangle \Rightarrow \langle \neg b, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle}{\langle P\, \textbf{and}\, Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle \qquad \langle Q, K'\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P\, \textbf{and}\, Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle \qquad \langle Q, K'\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P\, \textbf{and}\, Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}{\langle P\, \textbf{or}\, Q, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P\, \textbf{or}\, Q, K\rangle \Rightarrow \langle \mathit{true}, K\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P\, \textbf{or}\, Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

Fig. 4. Dynamic semantics evaluation rules. Domains and helper functions appear in Fig. 3. We say $\langle P, K\rangle \Rightarrow \langle b, K'\rangle$ when the predicate P evaluates in the environment K to the boolean result b, producing the new environment K'. If the result b is *false*, then the resulting environment K' is ignored. Bindings do not escape from **not** or **or** constructs; App. B relaxes the latter restriction. The starred hypothesis uses K, not K', to construct the result environment K'' because only the bindings specified in the **return** clause, not all bindings in the predicate's **when** clause, are exposed at the call site.

is then the result of evaluating $Body_{i_0}$ in the environment K_{i_0}, so that variables bound in the predicate can be referred to in the body. If no such i_0 exists, then an exception is raised: a "message not understood" error if I is empty, or a "message ambiguous" error if there is no unique most specific element of I.

An implementation can make a number of improvements to this base algorithm. Here we briefly mention just a few such optimizations. First, common subexpression elimination over predicate expressions can limit the computation done in evaluating guards. Second, precomputed implication relationships can prevent the necessity for evaluating every predicate expression. If a more specific one is true, then the less specific one is certain to be satisfied; however, such satisfaction is irrelevant since the more specific predicate will be chosen. Third, clauses and methods can be reordered to succeed or fail more quickly, as in some Prolog implementations [Zel93].

3.2 Static semantics and typechecking

The operational model of predicate dispatch described in Sect. 3.1 can raise a run-time exception at a message send if no method is applicable or if no applicable method overrides all the others. We extend the typechecking rules of the underlying language to guarantee that no such exception occurs.

Figure 5 presents the static semantic domains, helper functions, and typechecking rules for the core predicate dispatching sublanguage. The return type for a predicate invocation is an unordered record. Bindings do not escape from **not** or **or** constructs (App. B makes bindings on both sides of a **or** disjunct visible outside the disjunct).

We can separate typechecking into two parts: *client-side*, which handles all checking of expressions in the underlying language and uses method signatures to typecheck message sends, and *implementation-side*, which checks method and predicate implementations against their corresponding signatures. Only implementation-side checking is affected by predicate dispatching.

Implementation-side typechecking must guarantee *completeness* and *uniqueness*. Completeness guarantees that no "message not understood" error is raised: for every possible set of arguments at each call site, some method is applicable. Let P_m be the disjunction of the predicates of all of m's implementations, and let P_s be a predicate expressing the set of argument classes that conform to the types in the method signature. (See below for the details of predicate P_s; a class c conforms to a type T if every object which is an instance of that class has type T or a subtype of T.) If P_s implies P_m, then some method is always applicable. Uniqueness guarantees that no "message ambiguous" error is raised: for no possible set of arguments at any call site are there multiple most-specific methods. Uniqueness is guaranteed if, for each pair of predicates P and Q attached to two different implementations, either P and Q are disjoint (so their associated methods can never be simultaneously applicable) or one of the predicates implies the other (so one of the methods overrides the other). Section 4 presents implication and disjointness tests over predicate expressions.

$T \le T'$	Type T is a subtype of T'.
conformant-type(T, c)	Return the most-specific (with respect to the subtyping partial order) type T' such that every subclass c' of c that conforms to T also conforms to T'. This helper function is supplied by the underlying programming language.
$\Gamma + \Gamma' = \Gamma''$	Overriding extension of typing environments. For each $v \in \mathrm{dom}(\Gamma')$, if $\Gamma' \models v : T'$, then $\Gamma'' \models v : T'$; for each $v \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma')$, if $\Gamma \models v : T$, then $\Gamma'' \models v : T$.

$$\frac{}{\Gamma \vdash \textbf{signature } m(T_1, \ldots, T_n) \textbf{ : } T_r \Rightarrow \Gamma + \{m : (T_1, \ldots, T_n) \rightarrow T_r\}}$$

$$\frac{\begin{array}{c}\Gamma \models m : (T_1, \ldots, T_n) \rightarrow T_r \\ \Gamma + \{v_1 : T_1, \ldots, v_n : T_n\} \vdash P \Rightarrow \Gamma' \qquad \Gamma' \models \mathit{Body} : T_b \qquad T_b \le T_r\end{array}}{\Gamma \vdash \textbf{method } m(v_1, \ldots, v_n) \textbf{ when } P \ \mathit{Body} \Rightarrow \Gamma}$$

$$\frac{}{\begin{array}{l}\langle \Gamma, \textbf{predsignature } p(T_1, \ldots, T_n) \textbf{ return } \{f_1 \textbf{ : } T_1^r, \ldots, f_m \textbf{ : } T_m^r\}\rangle \\ \quad \Rightarrow \Gamma + \{p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r\}\ \}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \models p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r, \ldots\} \\ \Gamma + \{v_1 : T_1, \ldots, v_n : T_n\} \vdash P \Rightarrow \Gamma' \\ \forall i \in \{1, \ldots, m\} \quad \Gamma' \models v_i' : T_i' \ \wedge \ T_i' \le T_i^r\end{array}}{\Gamma \vdash \textbf{predicate } p(v_1, \ldots, v_n) \textbf{ when } P \textbf{ return } \{f_1 \textbf{ := } v_1', \ldots, f_m \textbf{ := } v_m'\} \Rightarrow \Gamma}$$

$$\frac{}{\Gamma \vdash \textbf{true} \Rightarrow \Gamma}$$

$$\frac{\Gamma \models v : T \qquad \mathit{conformant\text{-}type}(c, T) = T'}{\Gamma \vdash v \ @\ c \Rightarrow \Gamma + \{v : T'\}}$$

$$\frac{\Gamma \models v : \mathit{Bool}}{\Gamma \vdash \textbf{test } \ v \Rightarrow \Gamma}$$

$$\frac{\Gamma \models \mathit{expr} : T}{\Gamma \vdash \textbf{let } \ v \textbf{ := } \mathit{expr} \Rightarrow \Gamma + \{v : T\}}$$

$$\frac{\begin{array}{c}\Gamma \models p : (T_1, \ldots, T_n) \rightarrow \{f_1 : T_1^r, \ldots, f_m : T_m^r, \ldots\} \\ \Gamma \models v_1 : T_1' \quad \ldots \quad \Gamma \models v_n : T_n' \qquad T_1' \le T_1 \quad \ldots \quad T_n' \le T_n\end{array}}{\Gamma \vdash p(v_1, \ldots, v_n) \textbf{ => } \{f_1 = v_1', \ldots, f_m = v_m'\} \Rightarrow \Gamma + \{v_1' : T_1^r, \ldots, v_m' : T_m^r\}}$$

$$\frac{\Gamma \vdash P \Rightarrow \Gamma'}{\Gamma \vdash \textbf{not } P \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash P_2 \Rightarrow \Gamma''}{\Gamma \vdash P_1 \textbf{ and } P_2 \Rightarrow \Gamma''}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma \vdash P_2 \Rightarrow \Gamma''}{\Gamma \vdash P_1 \textbf{ or } P_2 \Rightarrow \Gamma}$$

Fig. 5. Typechecking rules. The hypothesis $\Gamma \models E : T$ indicates that typechecking in typing environment Γ assigns type T to expression E. The judgment $\Gamma \vdash P \Rightarrow \Gamma'$ represents extension of typechecking environments: given type environment Γ, P typechecks and produces new typechecking environment Γ'.

Completeness checking requires a predicate P_s that expresses the set of tuples of values $v_1, \ldots, v_n$ conforming to some signature's argument types $T_1, \ldots, T_n$; this predicate depends on the host language's model of classes and typing. If classes and types are the same, and all classes are concrete, then the corresponding predicate is simply v_1 @ T_1 **and** ... **and** v_n @ T_n. If abstract classes are allowed, then each v_i @ T_i is replaced with v_i @ T_{i1} **or** ... **or** v_i @ T_{im}, where the T_{ij} are the top concrete subclasses of T_i. If inheritance and subtyping are separate notions, then the predicates become more complex.

Our typechecking need not test that methods conform to signatures, unlike previous work on typechecking multimethods [CL95]. In predicate dispatching, a method's formal argument has two distinct types: the "external" type derived from the signature declaration, and the possibly finer "internal" type guaranteed by successful evaluation of the method's predicate. The individual @ tests narrow the type of the tested value to the most-specific type to which all classes passing the test conform, in a host-language-specific manner, using *conformant-type*. The *conformant-type* function replaces the more complicated conformance test of earlier work.

4 Comparing predicate expressions

The static and dynamic semantics of predicate dispatching require compile-time tests of implication between predicates to determine the method overriding relationship. The static semantics also requires tests of completeness and uniqueness to ensure the absence of "message not understood" errors and "message ambiguous" errors, respectively. All of these tests reduce to tautology tests over predicates. Method m_1 with predicate p_1 overrides method m_2 with predicate p_2 iff p_1 implies p_2 — that is, if (**not** p_1) **or** p_2 is true. A set of methods is complete if the disjunction of their predicates is true. Uniqueness for a set of methods requires that for any pair of methods, either one's predicate overrides the other's, or the two predicates are logically exclusive. Two formulas are mutually exclusive exactly if one implies the negation of the other.

Section 4.1 presents a tautology test over predicate expressions which is simple, sound, and complete up to equivalence of arbitrary program expressions in **test** constructs, which we treat as black boxes. Because determining logical tautology is NP-complete, in the worst case an algorithm takes exponential time in the size of the predicate expressions. For object-oriented dispatch, this is the number of arguments to a method (a small constant). Simple optimizations (Sect. 4.2) make the tests fast in many practical situations. This cost is incurred only at compile time; at run time, precomputed overriding relations among methods are simply looked up.

We treat expressions from the underlying programming language as black boxes (but do identify those whose canonicalizations are structurally identical). Tests involving the run-time values of arbitrary host language expressions are undecidable. The algorithm presented here also does not address recursive predicates. While we have a set of heuristics that succeed in many common, practical cases, we do not yet have a complete, sound, and efficient algorithm.

4.1 The base algorithm

The base algorithm for testing predicate tautology has three components. First, the predicate expression is canonicalized to macro-expand predicate abstractions, eliminate variable bindings, and use canonical names for formal arguments. This transformation prevents different names for the same value from being considered distinct. Second, implication relations are computed among the atomic predicates (for instance, x `@ int` implies x `@ num`). Finally, the canonicalized predicate is tested for every assignment of atomic predicates to truth values which is consistent with the atomic predicate implications. The predicate is a tautology iff evaluating it in every consistent truth assignment yields *true*.

Canonicalization Canonicalization performs the following transformations:

– Expand predicate calls inline, replacing the **=>** clause by a series of **let** bindings.
– Replace **let**-bound variables by the expressions to which they are bound, and replace **let** expressions by **true**.
– Canonically rename formal parameters according to their position in the formal list.

After canonicalization, each predicate expression is a logical formula over the following atoms with connectives **and**, **or**, and **not**.

```
pred-atom ::= true
           |  test expr
           |  expr @ class-id
```

Canonicalized predicates are a compile-time construct used only for predicate comparison; they are never executed. Canonicalized predicates bind no variables, and they use only global variables and formal parameters.

In the worst case, canonicalization exponentially blows up expression sizes. For instance, in

$$\textbf{let}\, x_1 = x{+}x \,\textbf{and}\,\textbf{let}\, x_2 = x_1{+}x_1 \,\textbf{and}\,\textbf{let}\, x_3 = x_2{+}x_2 \,\textbf{and}\ldots\textbf{and}\,\textbf{test}\ x_n = y\ ,$$

the final x_n is replaced by an expression containing 2^n instances of x. Inline expansion of predicate abstractions similarly contributes to this blowup. As with ML typechecking [KM89], which is exponential in the worst case but linear in practice, we anticipate that predicates leading to exponential behavior will be rare.

In what follows, we consider two expressions identical if, after canonicalization, they have the same abstract syntax tree.

Omitting the canonicalization step prevents some equivalent expressions from being recognized as such, but does not prevent the remainder of the algorithm from succeeding when results are named and reused rather than the computation repeated.

Truth assignment checking We present a simple exponential-time algorithm to check logical tautology; because the problem is NP-complete, any algorithm takes exponential time in the worst case. Let there be n distinct predicate atoms in the predicate; there are 2^n different truth assignments for those atoms. Not all of those truth assignments are consistent with the implications over predicate atoms: for instance, it is not sensible to set `a @ int` to *true* but `a @ num` to *false*, because `a @ int` implies `a @ num`. If every consistent truth assignment satisfies the predicate, then the predicate is a tautology. Each check of a single truth assignment takes time linear in the size of the predicate expression, for a total time of $O(n2^n)$.

The following rules specify implication over (possibly negated) canonical predicate atoms.

$$\begin{array}{rll}
E_1 \mathbin{@} c_1 \Rightarrow E_2 \mathbin{@} c_2 & \text{iff} & (E_1 \equiv E_2) \text{ and } (c_1 \text{ is a subclass of } c_2) \\
E_1 \mathbin{@} c_1 \Rightarrow \mathbf{not}(E_2 \mathbin{@} c_2) & \text{iff} & (E_1 \equiv E_2) \text{ and } (c_1 \text{ is disjoint from } c_2) \\
a_1 \Rightarrow a_2 & \text{iff} & \mathbf{not}\, a_2 \Rightarrow \mathbf{not}\, a_1 \\
a_1 \Rightarrow \mathbf{not}\, a_2 & \text{iff} & a_2 \Rightarrow \mathbf{not}\, a_1
\end{array}$$

Two classes are disjoint if they have no common descendant, and $\mathbf{not\,not}\, a = a$.

4.2 Optimizations

The worst-case exponential-time cost to check predicate tautology need not prevent its use in practice. Satisfiability is checked only at compile time. When computing overriding relationships, the predicates tend to be small (linear in the number of arguments to a method). We present heuristics that reduce the costs even further.

Logical simplification — such as eliminating uses of **true**, **false**, a **and not** a, and a **or not** a, and replacing **not not** a by a — can be performed as part of canonicalization to reduce the size of predicate expressions.

Unrelated atomic predicates can be treated separately. To determine whether **method** $m_1(f_1@c_1, f_2@c_2)\{...\}$ overrides **method** $m_1(f_1@c_3, f_2@c_4)\{...\}$ it is sufficient to independently determine the relationship between c_1 and c_3 and that between c_2 and c_4. Two tests with a smaller exponent replace one with a larger one, substantially reducing the overall cost. This technique always solves ordinary single and multiple dispatching overriding in time constant and linear in the number of formals, respectively, by examining each formal position independently. The technique also applies to more complicated cases, by examining subsets of formal parameters which appear together in tests from the underlying programming language.

It is not always necessary to completely expand predicate abstraction calls as part of canonicalization. If relations between predicate abstractions or other predicate expressions are known, then the tautology test can use them directly. As one example, different cases of a classifier are mutually exclusive by definition.

The side conditions on atomic predicate values (their implication relationships) usually prevent the need to check all 2^n different truth assignments for a predicate containing n atomic predicates. When `a @ int` is set to *true*, then all truth assignments which set `a @ num` to *false* can be skipped without further consideration.

Finally, it may be possible to achieve faster results in some cases by recasting the tautology test. Rather than attempting to prove that every truth assignment satisfies a predicate expression, it may be advantageous to search for a single truth assignment that satisfies its negation.

5 Related work

5.1 Object-oriented approaches

In the model of predicate dispatching, traditional object-oriented dispatching translates to either a single class test on the receiver argument or, for multiple dispatching, a conjunction of class tests over several arguments. Full predicate dispatching additionally enables testing arbitrary boolean expressions from the underlying language, accessing and naming subcomponents of the arguments, performing tests over multiple arguments, and arbitrarily combining tests via conjunction, disjunction, and negation. Also, named predicate abstractions effectively introduce new virtual classes and corresponding subclassing links into the program inheritance hierarchy. Predicate dispatching preserves the ability in object-oriented languages to statically determine when one method overrides another and when no message lookup error can occur. Singly-dispatched object-oriented languages have efficient method lookup algorithms and separate typechecking, which depend crucially on the absence of any separate modules that dispatch on other argument positions. Multiply-dispatched object-oriented languages have more challenging problems in implementation [KR89,CTK94,AGS94] and typechecking [CL95], and predicate dispatching in its unrestricted form shares these challenges.

Predicate classes [Cha93b] are an earlier extension of object-oriented dispatching to include arbitrary boolean predicates. A predicate class which inherits from class A and has an associated predicate expression *guard* would be modeled as a named predicate abstraction that tests @A **and** *guard*. Predicate dispatching is more general, for example by being able to define predicates over multiple arguments. Predicate dispatching exploits the structure of **and**, **or**, and **not** to automatically determine when no message lookup error can occur, while typechecking of predicate classes relies on uncheckable user assertions about the relations between the predicate classes' guard expressions.

Classifiers in Kea [HHM90b,HHM90a,MHH91] let an instance of a class be dynamically reclassified as being of a subclass. A classifier for a class is composed of a sequence of predicate/subclass pairs, with an object of the input class automatically classified as being of the subclass with the first successful predicate. Because the sequence of predicates is totally ordered and the first successful predicate takes precedence over all later predicates, a classifier provides a concise syntax for a set of mutually exclusive, exhaustive predicate abstractions. Predicate abstractions are more general than classifiers in many of the ways discussed above, but they also provide syntactic support for this important idiom. Kea is a purely functional language, so classifiers do not need to consider the semantics of reclassifying objects when the values of predicates change; predicate dispatching addresses this issue by (conceptually) performing reclassification as needed as part of message dispatching.

Modes [Tai93] are another mechanism for adding dynamic reclassification of a class into a subclass. Unlike predicate classes and classifiers, the modes of a class are not first-class subclasses but rather internal components of a class that cannot be extended externally and that cannot exploit inheritance to factor shared code. Mode reselection can be done either explicitly at the end of each method or implicitly after each assignment using a declaratively specified classification.

5.2 Pattern matching approaches

Predicate dispatching supports many of the facilities found in pattern matching as in ML [MTH90] and Haskell [HJW+92], including tests over arbitrary nested structure, binding of names to subcomponents, and arbitrary boolean guard expressions. Predicate dispatching additionally supports inheritance (its class tests are more general than datatype constructor patterns), disjunctions and negations of tests and conjunctions of tests on the same object, and named predicate abstractions to factor out common patterns of tests and to offer conditional views of objects extended with virtual fields. The patterns in a function are totally ordered, while predicate dispatching computes a partial order over predicates and warns when two patterns might be ambiguous. Finally, new methods can be added to existing generic functions without changing any existing code, while new patterns can be added to a function only by modifying it.

Views [Wad87] extend pattern matching to abstract data types by allowing an abstract data type to offer a number of views of itself as a concrete datatype, over which pattern matching is defined. Predicate dispatching supports "pattern matching" over the results of methods (by **let**-binding their results to names and then testing those names, just as field contents are bound and tested), and those methods can serve as accessor functions to a virtual view of the object, for instance `rho` and `theta` methods presenting a polar view of a cartesian point. Views must be isomorphisms, which enables equational reasoning over them; by contrast, named predicate abstractions provide conditional views of an object without requiring the presence of both in and out views.

Pizza [OW97] supports both algebraic datatypes (and associated pattern matching) and object-oriented dispatching, but the two mechanisms are largely distinct. The authors argue that datatypes are good for fixed numbers of representations with extensible operations, while classes are good for a fixed set of operations with extensible representations. By integrating pattern matching and dispatching, including multimethods, predicate dispatching achieves extensibility in both dimensions along with the syntactic convenience of pattern matching. Predicate dispatching faces more difficult implementation and separate typechecking challenges with the shift to multimethod-like dispatching.

6 Conclusions

Many language features express the concept of selecting a most-specific applicable method from a collection of candidates, including object-oriented dispatch, pattern matching, views, predicate classes, and classifiers. Predicate dispatching integrates and generalizes these mechanisms in a single framework, based on a core language of boolean expressions over class tests and arbitrary expressions, explicit binding forms to generalize features of pattern matching, and named predicate abstractions with result bindings. By providing a single integrated mechanism, programs can take advantage of various styles of dispatch and even combine them to create applicability conditions that were previously either impossible or inconvenient to express.

We have implemented predicate dispatching in the context of Dubious, a simple core multiply-dispatched object-oriented programming language. The implementation supports all the examples presented in this paper, although for clarity this paper uses a slightly different presentation syntax. The implementation supports the full core language of Sect. 3 and many of the syntactic sugars of App. A. This implementation was helpful in verifying our base design. We expect that it will also provide insight into the advantages and disadvantages of programming with predicate dispatching, as well as help us to evaluate optimization strategies. The implementation is available from `http://www.cs.washington.edu/research/projects/cecil/www/Gud/`.

So far, we have focused on developing the static and dynamic semantics for predicate dispatching. Two unresolved practical issues that we will address in the future are efficient implementation techniques and separate typechecking support for predicate dispatching. We anticipate that efficient implementations of unrestricted predicate dispatching will build upon work on efficient implementation of multimethod dispatching and on predicate classes. In addition, static analyses that factor a collection of predicates to avoid redundant tests and side-effect analyses that determine when predicates need not be re-evaluated appear to be promising lines for future research. Similarly, separate typechecking of collections of predicated methods will build upon current work to develop modular and incremental methods for typechecking multimethods [CL95].

Acknowledgments

Todd Millstein, Vassily Litvinov, Wilson Hsieh, David Grove, and the anonymous referees made helpful comments on a draft of this paper. This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), an IBM graduate fellowship, an FCAR graduate scholarship, and gifts from Sun Microsystems, IBM, Xerox PARC, Pure Software, and Edison Design Group.

References

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings OOPSLA '94*, pages 244–258, Portland, OR, October 1994.

[BKK$^+$86] Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA '86*, pages 17–29, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Cha93a] Craig Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Cha93b] Craig Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[CL95] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CTK94] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient dynamic look-up strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 408–431, Bologna, Italy, July 1994. Springer-Verlag.

[HHM90a] J. Hamer, J.G. Hosking, and W.B. Mugridge. A method for integrating classification within an object-oriented environment. Technical Report Auckland Computer Science Report No. 48, Department of Computer Science, University of Auckland, October 1990.

[HHM90b] J.G. Hosking, J. Hamer, and W.B. Mugridge. Integrating functional and object-oriented programming. In *Technology of Object-Oriented Languages and Systems TOOLS 3*, pages 345–355, Sydney, 1990.

[HJW$^+$92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[KM89] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 105–115, Austin, TX, USA, January 1989. ACM Press.

[KR89] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. Technical Report SSL 89-95, Xerox PARC Systems Sciences Laboratory, 1989.

[MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 307–324, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[Ste90] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1990. Second edition.

[Tai93] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, pages 25–32, June 1993.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, Germany, January 1987.

[Zel93] John M. Zelle. Learning search-control heuristics for logic programs: Applications tospeed-up learning and languageacquisitions. Technical Report AI93-200, University of Texas, Austin, May 1, 1993.

A Desugaring rules

The following rewrite rules desugar the high-level syntax of Fig. 6 into the core abstract syntax of Fig. 2. The rules are grouped by their intention, such as providing names for arbitrary expressions or breaking down compound predicate abstractions.

For brevity, we omit the rewrite rules which introduce defaults for omitted optional program fragments: dummy variables for pattern variables, "`@Any`" specializers, empty field pattern sets in specializers, and "**when true**" and "**return { }**" clauses. Additional rules may be introduced to simplify the resulting formula, such as converting "v @ `Any`" to "**true**" and performing logical simplification.

For brevity, we use $\bigwedge_{i=1}^{n}\{P_i\}$ to stand for the conjunction of the terms: P_1 **and** . . . **and** P_n. When $n = 0$, $\bigwedge_{i=1}^{n}\{P_i\}$ stands for **true**. Variables v' and v'_i are new variables which do not appear elsewhere in the program. Ceiling braces $\lceil\cdot\rceil$ surround (potentially) sugared expressions; application of the rewrite rules eliminates those braces.

A.1 Declarations

These rules move specializers from formal lists into **when** clauses.

$$\lceil\textbf{method } m(v_1 \mathbin{@} S_1, \ldots, v_n \mathbin{@} S_n) \ \textbf{ when } \ P \ \mathit{Body}\rceil$$
$$\Longrightarrow \textbf{method } m(v_1, \ldots, v_n) \ \textbf{ when } \ \textstyle\bigwedge_{i=1}^{n}\{\lceil v_i \mathbin{@} S_i\rceil\} \textbf{ and } \lceil P\rceil \ \ \mathit{Body}$$
$$\lceil\textbf{predicate } p(v_1 \mathbin{@} S_1, \ldots, v_n \mathbin{@} S_n) \ \textbf{ when } \ P \ \textbf{ return } \ \{f_1 := E_1, \ldots, f_m := E_m\}\rceil$$
$$\Longrightarrow \textbf{predicate } p(v_1, \ldots, v_n)$$
$$\textbf{when } \ \textstyle\bigwedge_{i=1}^{n}\{\lceil v_i \mathbin{@} S_i\rceil\} \textbf{ and } \ \lceil P\rceil \, \lceil\textbf{return } \{f_1 := E_1, \ldots, f_m := E_m\}\rceil$$

method-sig	::=	**signature** *method-id* (⟨ *type* ⟩) **:** *type*	
method-decl	::=	**method** *method-id* (⟨ *formal-pattern* ⟩)	
		[**when** *pred-expr*] *method-body*	
pred-sig	::=	**predsignature** *pred-id* (⟨ *type* ⟩)	
		[**return** **{** ⟨ *field-id* **:** *type* ⟩ **}**]	
pred-decl	::=	**predicate** *pred-id* (⟨ *formal-pattern* ⟩) [**when** *pred-expr*]	
		[**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**]	
classifier-decl	::=	**classify** (⟨ *formal-pattern* ⟩)	
		⟨ **as** *pred-id* **when** *pred-expr* [**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**] ⟩	
		[**as** *pred-id* **otherwise** [**return** **{** ⟨ *field-id* **:=** *expr* ⟩ **}**]]	
$P, Q \in$ *pred-expr*	::=	*expr* **@** *specializer*	succeeds if *expr* evaluates to a value that satisfies *specializer*
	|	**test** *expr*	succeeds if *expr* evaluates to *true*
	|	**let** *var-id* **:=** *expr*	evaluate *expr* and bind *var-id* to its value; always succeeds
	|	*pred-id* (⟨ *expr* ⟩) [**=>** **{** ⟨ *field-pat* ⟩ **}**]	test predicate abstraction
	|	**true**	always succeeds
	|	**false**	never succeeds
	|	**not** *pred-expr*	negation
	|	*pred-expr* **and** *pred-expr*	conjunction (short-circuited)
	|	*pred-expr* **or** *pred-expr*	disjunction (short-circuited)
formal-pattern	::=	[*var-id*] [**@** *specializer*]	like *var-id***@***specializer* in *pred-expr*
$F \in$ *field-pat*	::=	*field-id* [**=** *var-id*] [**@** *specializer*]	
$S \in$ *specializer*	::=	*class-spec* [**{** ⟨ *field-pat* ⟩ **}**]	
$C \in$ *class-spec*	::=	*class-id*	*expr* **@** *class-id* is a class test
	|	*pred-id*	*expr* **@** *pred-id* [**{** ... **}**] is sugar for *pred-id*(*expr*) [**=>{** ... **}**]
	|	**not** *class-spec*	succeeds if *class-spec* does not
	|	*class-spec* **&** *class-spec*	succeeds if both *class-spec*s do
	|	*class-spec* **|** *class-spec*	succeeds if either *class-spec* does

Fig. 6. Full extended abstract syntax for predicate dispatching. The syntax is as presented incrementally in Sect. 2, with the addition of the **true** and **false** predicate expressions and the **not**, **&**, and **|** class specializers. Words and symbols in **boldface** represent terminals. Angle brackets denote zero or more comma-separated repetitions of an item. Square brackets contain optional expressions. Each predicate may be defined only once (App. B relaxes this restriction), and recursive predicates are forbidden.

A.2 Naming of non-variable expressions

The core language permits arbitrary expressions only in **let** bindings and uses variable references elsewhere. These rules introduce **let** bindings and are intended to fire only once (alternately, only if one of the E expressions is not a mere variable reference), lest the @ and predicate application rules cause an infinite loop in desugaring.

$$\lceil E \mathbin{@} S \rceil \Longrightarrow \textbf{let}\ v' := E\ \textbf{and}\ \lceil v' \mathbin{@} S \rceil$$

$$\lceil \textbf{test}\ E \rceil \Longrightarrow \textbf{let}\ v' := E\ \textbf{and}\ \textbf{test}\ v'$$
$$\lceil p(E_1, \ldots, E_n)\ \texttt{=>}\ \{F_1, \ldots, F_m\} \rceil$$
$$\Longrightarrow \bigwedge_{i=1}^{n} \{\textbf{let}\ v_i' := E_i\}\ \textbf{and}\ \lceil p(v_1', \ldots, v_n')\ \texttt{=>}\ \{F_1, \ldots, F_m\} \rceil$$
$$\lceil \textbf{return}\ \{f_1 := E_1, \ldots, f_m := E_m\} \rceil$$
$$\Longrightarrow \textbf{and} \bigwedge_{i=1}^{m} \{\textbf{let}\ v_i' := E_i\}\ \textbf{return}\ \{f_1 := v_1', \ldots, f_m := v_m'\}$$

A.3 Compound predicate expressions

These rules show how to desugar **false** and compound predicate expressions.

$$\lceil \textbf{false} \rceil \Longrightarrow \textbf{not true}$$
$$\lceil \textbf{not}\ P \rceil \Longrightarrow \textbf{not}\ \lceil P \rceil$$
$$\lceil P_1\ \textbf{and}\ P_2 \rceil \Longrightarrow \lceil P_1 \rceil\ \textbf{and}\ \lceil P_2 \rceil$$
$$\lceil P_1\ \textbf{or}\ P_2 \rceil \Longrightarrow \lceil P_1 \rceil\ \textbf{or}\ \lceil P_2 \rceil$$

A.4 Field bindings

Pattern matching permits arbitrarily nested tests and simultaneous matching on fields of objects, fields of predicate results, and results of arbitrary method invocations. These rules separate these varieties of record patterns and flatten tests.

We introduce the concept of a class specializer *generating* a field. A class name generates the fields in the class; a predicate name generates the fields in the predicate's **return** clause; a conjunction generates the fields generated by either of its conjuncts; and a disjunction generates the fields generated by both of its disjuncts.

If F_i is generated by $C \neq c$ for $1 \leq i \leq m < n$:

$$\lceil v\ @\ C\ \{F_1, \ldots, F_m, \ldots, F_n\} \rceil$$
$$\Longrightarrow \lceil v\ @\ C\ \{F_1, \ldots, F_m\} \rceil\ \textbf{and}\ \lceil v\ @\ \texttt{Any}\ \{F_{m+1}, \ldots, F_n\} \rceil$$
$$\lceil v\ @\ c\ \{f_1 = v_1\ @\ S_1, \ldots, f_n = v_n\ @\ S_n\} \rceil$$
$$\Longrightarrow v\ @\ c\ \textbf{and}\ \bigwedge_{i=1}^{n} \{\textbf{let}\ v_i := v.f_i\ \textbf{and}\ \lceil v_i\ @\ S_i \rceil\}$$
$$\lceil v\ @\ p\ \{f_1 = v_1\ @\ S_1, \ldots, f_n = v_n\ @\ S_n\} \rceil$$
$$\Longrightarrow p(v)\ \texttt{=>}\ \{f_1 = v_1, \ldots, f_n = v_n\}\ \textbf{and}\ \bigwedge_{i=1}^{n} \{\lceil v'\ @\ S_i \rceil\}$$

A.5 Compound predicate abstractions

These rules simplify compound predicate abstractions.

$$\lceil v\ @\ \textbf{not}\ C\{F_1, \ldots, F_m\} \rceil \Longrightarrow \textbf{not}\ \lceil v\ @\ C\{F_1, \ldots, F_m\} \rceil$$

If F_i, $m+1 \leq i \leq n$, is generated by C_2 (& rule only):

$$\lceil v\ @\ C_1\ \&\ C_2\{F_1, \ldots, F_m, \ldots, F_n\} \rceil \Longrightarrow \lceil v\ @\ C_1\{F_1, \ldots, F_m\} \rceil\ \textbf{and}\ \lceil v\ @\ C_2\{F_{m+1}, \ldots, F_n\} \rceil$$
$$\lceil v\ @\ C_1\ |\ C_2\{F_1, \ldots, F_m\} \rceil \Longrightarrow \lceil v\ @\ C_1\{F_1, \ldots, F_m\} \rceil\ \textbf{or}\ \lceil v\ @\ C_2\{F_1, \ldots, F_m\} \rceil$$

A.6 Classifiers

Sequential ordering over classifier cases is enforced by creating extra predicates d_i such that d_i is true if any c_j, $j \leq i$, is true. Each c_i is true only if d_{i-1} is not (that is, no previous conjunct was true).

$$\left\lceil\begin{array}{l}\textbf{classify}(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m)\\ \quad \textbf{as } c_1 \textbf{ when } P_1 \textbf{ return } \{f_{1,1} := v'_{1,1}, \ldots, f_{1,m_1} := v'_{1,m_1}\}\\ \quad \vdots \\ \quad \textbf{as } c_n \textbf{ when } P_n \textbf{ return } \{f_{n,1} := v'_{n,1}, \ldots, f_{n,m_n} := v'_{n,m_n}\}\\ \quad \textbf{as } c_{n+1} \textbf{ otherwise return } \{f_{n+1,1} := v'_{n+1,1}, \ldots, f_{n+1,m_{n+1}} := v'_{n+1,m_{n+1}}\}\end{array}\right\rceil$$

$$\Longrightarrow \left\lceil\begin{array}{l}\textbf{predicate } c_1(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_1\\ \qquad \textbf{return } \{f_{1,1} := v'_{1,1}, \ldots, f_{1,m_1} := v'_{1,m_1}\};\end{array}\right\rceil$$

$$\lceil \textbf{predicate } d_1(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_1; \rceil$$

$$\left\lceil\begin{array}{l}\textbf{predicate } c_2(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_2 \textbf{ and not } d_1(v_1, \ldots, v_m)\\ \qquad \textbf{return } \{f_{2,1} := v'_{2,1}, \ldots, f_{2,m_2} := v'_{2,m_2}\};\end{array}\right\rceil$$

$$\lceil \textbf{predicate } d_2(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } d_1(v_1, \ldots, v_m) \textbf{ or } P_2; \rceil$$

$$\vdots$$

$$\left\lceil\begin{array}{l}\textbf{predicate } c_n(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } P_n \textbf{ and not } d_{n-1}(v_1, \ldots, v_m)\\ \qquad \textbf{return } \{f_{n,1} := v'_{n,1}, \ldots, f_{n,m_n} := v'_{n,m_n}\};\end{array}\right\rceil$$

$$\lceil \textbf{predicate } d_n(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when } d_{n-1}(v_1, \ldots, v_m) \textbf{ or } P_n; \rceil$$

$$\left\lceil\begin{array}{l}\textbf{predicate } c_{n+1}(v_1 \mathbin{@} S_1, \ldots, v_m \mathbin{@} S_m) \textbf{ when not } d_n(v_1, \ldots, v_m)\\ \qquad \textbf{return } \{f_{n+1,1} := v'_{n+1,1}, \ldots, f_{n+1,m_{n+1}} := v'_{n+1,m_{n+1}}\};\end{array}\right\rceil$$

B Bindings escaping "or"

In the static and dynamic semantics presented in Sect. 3, bindings never escape from **or** predicate expressions. Relaxing this constraint provides extra convenience to the programmer and permits more values to be reused rather than recomputed. It is also equivalent to permitting overloaded predicates or multiple predicate definitions — so far we have permitted only a single definition of each predicate. With appropriate variable renaming, multiple predicate definitions that rely on a dispatching-like mechanism to select the most specific applicable method can be converted into uses of **or**, and vice versa.

For example, the two `ConstantFold` methods of Sect. 2.3 can be combined into a single method. Eliminating code duplication is a prime goal of object-oriented programming, but the previous version repeated the body twice. Use of a helper method would unnecessarily separate the dispatching conditions from the code being executed, though a helper predicate could reduce code duplication in the predicate expression.

```
 -- handle case of adding zero to a non-constant
 method ConstantFold(e@BinopExpr{ op@IntPlus, arg1=a1, arg2=a2 })
   when (a1@IntConst{ value=v } and test(v == 0)
         and not(a2@IntConst) and let res := a2)
     or (a2@IntConst{ value=v } and test(v == 0)
         and not(a1@IntConst) and let res := a1) {
   ... -- increment counter, or do other common work here
   return res; }
```

As another example, the **LoopExit** example of Sect. 2.4 can be extended to present a view which indicates which branch of the **CFG_2succ** is the loop exit and which the backward branch. When performing iterative dataflow, this is the only information of interest, and in our current implementation (which uses predicate classes [Cha93b]) we generally recompute this information after discovering that an object is a **LoopExit**. Presenting a view which includes this information directly would improve the code's readability and efficiency.

```
predsignature LoopExit(CFGnode)
  return { loop:CFGloop, next_loop:CFGedge, next_exit:CFGedge };
predicate LoopExit(n@CFG_2succ{ next_true: t, next_false: f })
  when (test(t.is_loop_exit) and let nl := t and let ne := f)
    or (test(f.is_loop_exit) and let nl := f and let ne := t)
  return { loop := nl.containing_loop, next_loop := nl, next_exit := ne };
```

Permitting bindings which appear on both sides of **or** to escape requires the following changes to the dynamic semantics of Fig. 4. (The third rule is unchanged from Fig. 4 but included here for completeness.)

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}{\langle P \ \mathbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{true}, K'\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}{\langle P \ \mathbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{true}, K''\rangle}$$

$$\frac{\langle P, K\rangle \Rightarrow \langle \mathit{false}, K'\rangle \qquad \langle Q, K\rangle \Rightarrow \langle \mathit{false}, K''\rangle}{\langle P \ \mathbf{or}\ Q, K\rangle \Rightarrow \langle \mathit{false}, K\rangle}$$

These execution rules do not reflect that only the bindings appearing on both sides, not all those appearing on the succeeding side, should escape; however, the typechecking rules below guarantee that only the appropriate variables are referenced.

The static semantics of Fig. 5 are modified to add two helper functions and replace a typechecking rule:

$T \sqcup T' = T''$	Least upper bound over types. T'' is the least common supertype of T and T'.
$\sqcup_{\mathrm{env}}(\Gamma, \Gamma') = \Gamma''$	Pointwise lub over typing environments. For each $v \in \mathrm{dom}(\Gamma'') = \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, if $\Gamma \models v : T$ and $\Gamma' \models v : T'$, then $\Gamma'' \models v : T \sqcup T'$.

$$\frac{\Gamma \vdash P_1 \Rightarrow \Gamma' \qquad \Gamma \vdash P_2 \Rightarrow \Gamma'' \qquad \sqcup_{\text{env}}(\Gamma', \Gamma'') = \Gamma'''}{\Gamma \vdash P_1 \ \mathbf{or}\ P_2 \Rightarrow \Gamma'''}$$

Finally, canonicalization must account for the new semantics of **or**. In order to permit replacement of variables by their values, we introduce a new compile-time-only ternary conditional operator **?:** for each variable bound on both sides of the predicate. The first argument is the predicate expression on the left-hand side of the **or** expression; the second and third arguments are the variable's values on each side of the **or**.

Canonicalizing this new **?:** expression requires ordering the tests canonically; any ordering will do. This may necessitate duplication of some expressions, such as transforming $b \mathbin{?} e_1 : (a \mathbin{?} e_2 : e_3)$ into $a \mathbin{?} (b \mathbin{?} e_1 : e_2) : (b \mathbin{?} e_1 : e_3)$ so that those two expressions are not considered distinct. With these two modifications, the tautology test is once again sound and complete.

To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991
© 1991 Kluwer Academic Publishers - Manufactured in The Netherlands

Organizing Programs Without Classes[*]

DAVID UNGAR[†]
CRAIG CHAMBERS
BAY-WEI CHANG
URS HÖLZLE (*self@self.stanford.edu*)

Computer Systems Laboratory, Stanford University, Stanford, California 94305

Abstract. All organizational functions carried out by classes can be accomplished in a simple and natural way by object inheritance in classless languages, with no need for special mechanisms. A single model—dividing types into prototypes and traits—supports sharing of behavior and extending or replacing representations. A natural extension, dynamic object inheritance, can model behavioral modes. Object inheritance can also be used to provide structured name spaces for well-known objects. Classless languages can even express "class-based" encapsulation. These stylized uses of object inheritance become instantly recognizable idioms, and extend the repertory of organizing principles to cover a wider range of programs.

1 Introduction

Recently, several researchers have proposed object models based on prototypes and delegation instead of classes and static inheritance [2, 9, 11, 14, 15, 18]. These proposals have concentrated on explaining how prototype-based languages allow more flexible arrangements of objects. Although such flexibility is certainly desirable, many have felt that large prototype-based systems would be very difficult to manage because of the lack of organizational structure normally provided by classes.

Organizing a large object-oriented system requires several capabilities. Foremost among these is the ability to *share* implementation and state among the instances of a data type and among related data types. The ability to define strict interfaces to data types that hide and protect implementation is also useful when organizing

*This work has been generously supported by National Science Foundation Presidential Young Investigator Grant # CCR-8657631, and by Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

large systems. Finally, the ability to use global names to refer to data types and to categorize large name spaces into structured parts for easier browsing are important for managing the huge number of objects that exist in a large object-oriented system.

In this paper we argue that programs in languages without classes are able to accomplish these tasks just as well as programs in class-based languages. In particular, we show that:

- all organizational functions carried out by classes can be accomplished in a very natural and simple way by classless languages,
- these organizational functions can be expressed using objects and inheritance, with no need for special mechanisms or an extralingual layer of data structures,
- the additional flexibility of prototype-based languages is a natural extension of the possibilities provided by class-based systems, and finally,
- exploiting this additional flexibility need not lead to unstructured programs.

The ideas presented here are based on the lessons we learned as we found ways to organize code in SELF, a dynamically-typed prototype-based language [3, 4, 10, 18]. Accordingly, we will illustrate the ideas using examples in SELF, but the ideas could be applied as well to other classless languages providing similar inheritance models.

2 Sharing

Programming in an object-oriented language largely revolves around specifying sharing relationships: what code is shared among the instances of a data type and what code is shared among similar data types.

2.1 Intra-Type Sharing: Classes and Traits Objects

The principal activity in object-based programming is defining new data types. To define a simple data type, the programmer needs to specify the state and behavior that are specific to each instance of the data type and the state and behavior that are common to (*shared by*) all instances of the type. For example, one way to define a simple polygon data type is to specify that each polygon instance contains a list of vertices and that all polygons share an operation to draw themselves.

In a typical class-based language, the class object defines a set of methods and (class) variables that are shared by all instances of the class, and a set of (instance) variables that are specific to each instance. For example, the polygon data type could be implemented by a class `Polygon` that defines a `draw` method and specifies that its instances have a single instance variable named `vertices`; the `Polygon` metaclass would contain a `new` method to create new polygon instances

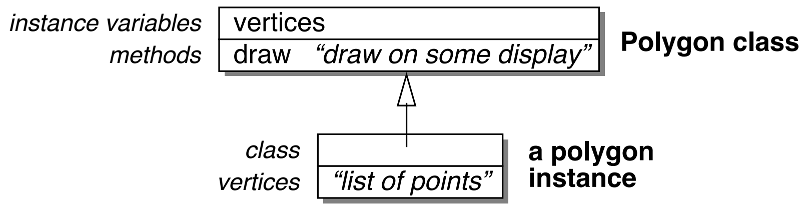


Figure 1a. Data types in a class-based language.

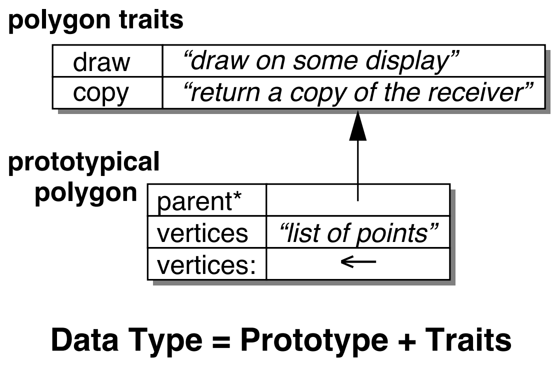

polygon traits
draw "draw on some display"
copy "return a copy of the receiver"
prototypical polygon
parent*
vertices "list of points"
vertices: ←
Data Type = Prototype + Traits

Figure 1b. Data types in a classless language.

(see Figure 1a). To initialize a new instance's list of vertices, the `Polygon` class could define a wrapper method named `vertices:` that just assigned its argument to the `vertices` instance variable. This wrapper method is required in languages like Smalltalk-80[1] [7] that limit access to an object's instance variables to the object itself.

In a classless language, the polygon data type is defined similarly. A prototypical `polygon` object is created as the first instance of the polygon type (see Figure 1b). This object contains three slots: an assignable data slot named `vertices`, the corresponding `vertices:` assignment slot,[2] and a constant *parent* slot[3] pointing to another object that contains a `draw` method and a `copy` method. The

[1]Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

[2]Data slots in SELF may be *assignable* or *constant*. Data slots are assignable by virtue of being associated with an *assignment slot* that changes the data slot's contents when invoked. The assignment slot's name is constructed by appending a colon to the data slot's name.

[3]Parent slots are indicated in SELF syntax by asterisks following the slot name. Parent slots in the figures of this paper are therefore indicated with asterisks.

`vertices` slot in this prototype is initialized to a convenient default list of vertices (e.g., a list of three points defining a triangle), making it usable as is and thus serving as a programming example.

New polygons are created by sending the `copy` message to an existing polygon (such as the prototypical polygon), which first *clones* (shallow-copies) the receiver polygon and then copies the internal vertex list. Since the prototype's slots contain default values, clones of the prototype are automatically initialized with these values as well. In particular, the same parent object is shared by each new polygon, providing the common behavior for all polygons in the system. We call these shared parent objects *traits objects*. Traits objects in a classless language provide the same sharing capability as classes, and just as in a class-based language, making changes to the behavior of all instances of a type is simple since the common behavior is factored out into a single shared object.

In general, data types may be defined in a classless language by dividing the definition of the type into two objects: the prototypical instance of the type and the shared traits object. The prototype defines the instance-specific aspects of the type, such as the representation of the type, while the traits object defines common aspects of all instances of the type. No special language features need to be added to support traits objects—a traits object is a regular object shared by all instances of the type using normal object inheritance. Since traits objects are regular objects, they may contain assignable data slots which are then shared by all instances of the data type, providing the equivalent of class variables.

Classless languages actually gain some descriptive power over class-based languages by dividing the implementation of a data type into two separate objects. If the data type is a *concrete* type (i.e. if instances of the data type will be created), then both the traits object and the initial prototype object are defined. If, however, the type is *abstract*, existing simply to define reusable behavior shared by other types, then no prototypical instance need be defined. Alternately, if there is only ever *one* instance of a particular data type, such as with unique objects like `nil`, `true`, and `false`, then the traits object need not be separated from the object at all. Traditional class-based languages implicitly specify both the shared behavior and the format of the class' instances; without extra language mechanisms they cannot distinguish between concrete, abstract, and singleton data types, with a corresponding loss of descriptive and organizational power.

2.2 Inter-Type Sharing: Subclasses and Refinements

Object-oriented languages with inheritance support *differential programming*, allowing new data types to be defined as differences from existing data types. The implementor of a new data type may specify that the type is equivalent to a combination of existing types, possibly with some additions and/or changes. For example, a filled polygon type might be identical to the polygon type, except that drawing filled polygons is different from drawing unfilled polygons, and a filled polygon instance needs extra state to hold its fill pattern.

In typical class-based languages, a class may be defined as a subclass of other classes. The methods of the new class are the union of the methods of the superclasses, possibly with some methods added or changed, and the instance variables of the new class are the union of the instance variables of the superclasses, possibly with some instance variables added. For example, filled polygons could be implemented by a `FilledPolygon` class that is a subclass of the `Polygon` class (see Figure 2a). The `FilledPolygon` class overrides the `draw` method, and specifies an additional instance variable named `fillPattern` that all `FilledPolygon` instances will have; the `vertices` instance variable is automatically provided since `FilledPolygon` is a subclass of `Polygon`. To initialize a new instance's fill pattern, the `FilledPolygon` class could define a wrapper method named `fillPattern:` that assigned its argument to the `fillPattern` instance variable.

Filled polygons are defined similarly in a language without classes. A new filled polygon traits object is created as a *refinement* (child) of the existing polygon traits object (see Figure 2b). This traits object defines its own `draw` method. To complete the definition of the new data type, a prototypical `filledPolygon` object is created that inherits from the filled polygon traits object. This object could contain both a `vertices` data slot and a `fillPattern` data slot, plus their corresponding assignment slots. (We will revise this representation to avoid unnecessary repetition of the `vertices` data slot in the next subsection.)

In general, a new data type in a classless language may be defined in terms of existing data types simply by *refining* the traits objects implementing the existing data types with a new traits object that is a child of the existing traits objects. Object inheritance is used to specify the refinement relationships, without needing extra language features.

2.3 Representation Sharing: Instance Variable Extension and Data Parents

When defining a data type as an extension of some pre-existing data types, frequently the instance-specific information of the existing data type should be combined with some extra information particular to the new data type, to construct the instance-specific information of the new data type. For example, a filled polygon instance needs both the polygon information (the list of vertices) plus the new filled-polygon-specific information (the fill pattern). Ideally, the new data type wouldn't need to repeat the instance-specific information it inherits from the existing data types, but instead share the information; this would enhance the malleability of the resulting system, since changing one data type's representation causes all data types that inherit from the changed data type to be updated automatically.

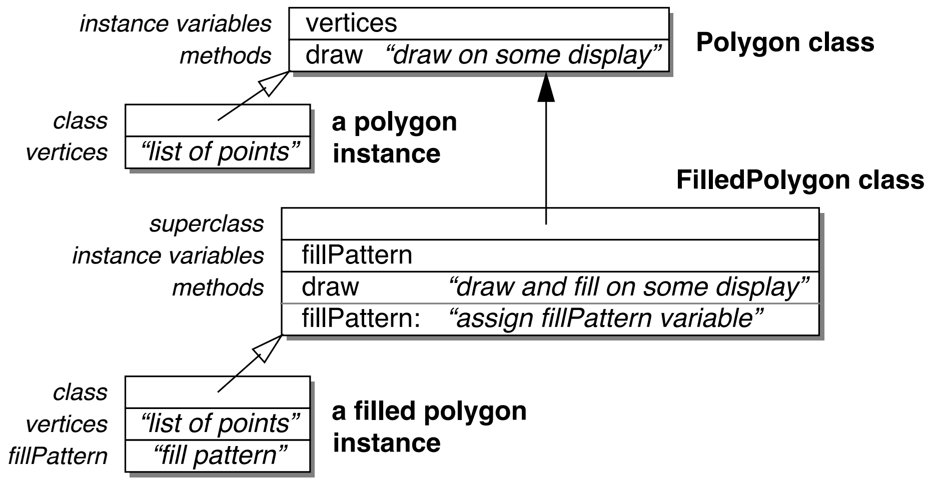


Figure 2a. Differential programming in a class-based language: subclassing, with implicit representation extension.

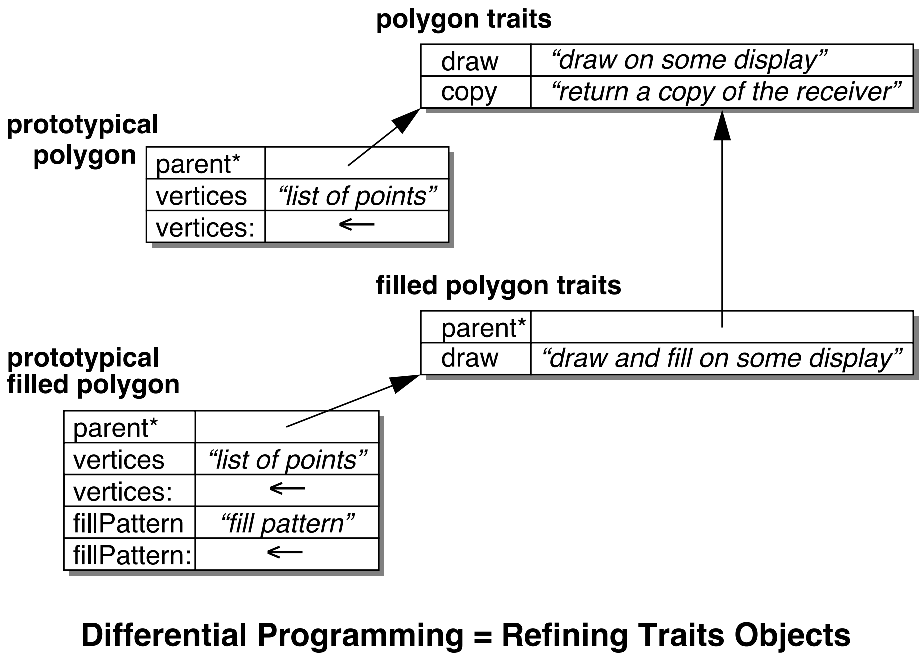

polygon traits
draw “draw on some display”
copy “return a copy of the receiver”
prototypical polygon
parent*
vertices “list of points”
vertices: ←
filled polygon traits
parent*
draw “draw and fill on some display”
prototypical filled polygon
parent*
vertices “list of points”
vertices: ←
fillPattern “fill pattern”
fillPattern: ←
Differential Programming = Refining Traits Objects

Figure 2b. Differential programming in a classless language.

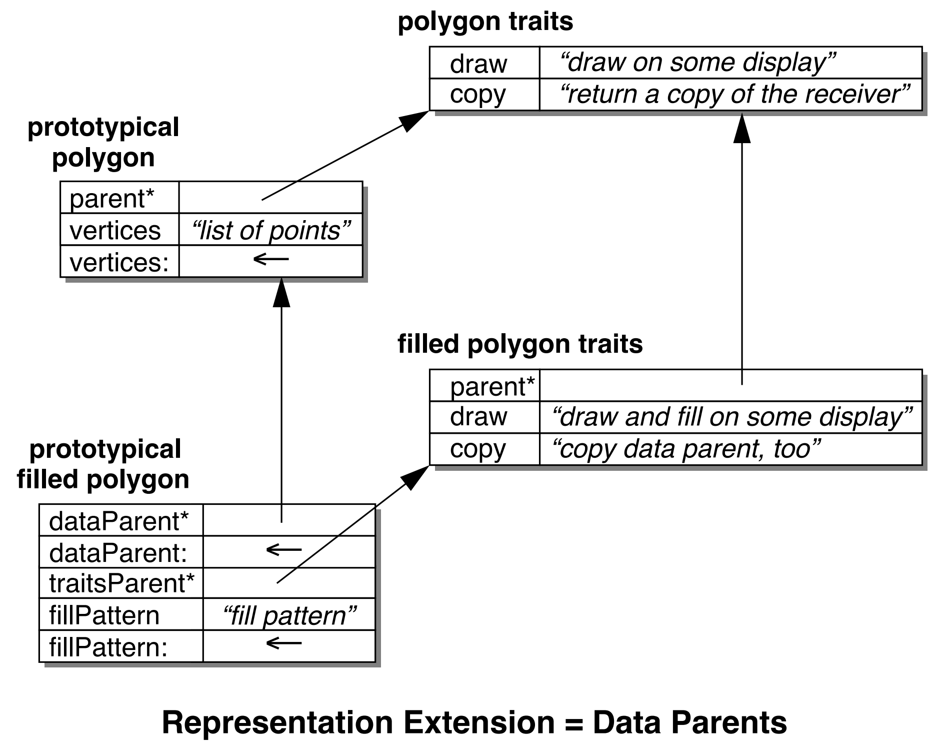


Figure 3. Representation extension in a classless language.

Class-based languages do this well. When a subclass is defined, it automatically inherits the instance variable lists from its superclasses; any instance variables specified in the subclass are interpreted as *extensions* of the superclasses' instance variables. This feature is illustrated by the `FilledPolygon` example class that extends the instance variables of the `Polygon` superclass with a `fillPattern` instance variable (see Figure 2a).

In a classless language with multiple inheritance we can provide similar functionality using *data parents*. Instead of manually repeating the data slot declarations of the prototypes of the parent data types, as was done in the implementation of filled polygons in Figure 2b, the new prototype may *share* the representation of its parent data types by inheriting from them. Thus, a better way to implement filled polygons is to define the `filledPolygon` prototype as a child of both the `filledPolygon` traits object *and* the `polygon` prototype object (see Figure 3). A new `copy` method is defined in the `filledPolygon` traits object to copy both the receiver filled polygon *and* the data parent polygon object, so that each instance of the filled polygon data type is implemented with two objects, one containing the instance's fill pattern and another containing its list of vertices.

Data parents explicitly implement the representation extension mechanism implicit in traditional class-based languages. Since the data parent objects are parents, data slots defined in the data parent are transparently accessed as if they were defined in the receiver object without defining explicit forwarding methods. By relying only on the ability to inherit state and to initialize a new object's parents to computed values, no special language mechanisms are needed to concatenate representations.

A problem with class-based representation extension surfaces in languages with multiple inheritance. If two superclasses define instance variables with the same name, does the subclass contain *two* different instance variables or are the superclasses' instance variables merged into *one* shared instance variable in the subclass? For some programming situations, it may be correct to keep two different instance variables; for other situations, it may be necessary to share a single instance variable. Different class-based languages that support multiple inheritance answer this difficult question differently; some languages, like C++ [16, 17], provide the programmer the option of doing either, at some cost in extra language complexity.

Classless languages don't face this dilemma. Since the prototypical instance of the data type is defined explicitly, the programmer has complete control over each type's representation. If the new type should contain only one version of the data slot, then the prototype just contains that one data slot. If several versions need to be maintained, one per parent data type, then data parents may be used to keep the versions of data slots with the same name.[4]

2.4 Beyond Representation Sharing

Class-based languages automatically extend the representation of a subclass to include its superclasses' instance variables. However, this automatic extension may not always be desired. For example, an application might want to define a rectangle data type as a subtype of the polygon data type. The representation of the rectangle might be four numbers (instead of a list of four vertices), and the draw routine could be optimized for this special case.

Most class-based languages cannot define such a `Rectangle` class as a subclass of the `Polygon` class because the `Rectangle` class would be extended automatically with the `Polygon` class' `vertices` instance variable. To fix this problem, an additional `AbstractPolygon` class (with no instance variables) must be defined as the common superclass of both `Polygon` and `Rectangle`; the behavior common to all polygons would then be moved from the concrete

[4]SELF includes a message lookup rule (the sender path tiebreaker rule) that automatically disambiguates internal accesses to these data slots.

`Polygon` class to the `AbstractPolygon` class (see Figure 4a). But this then creates another problem: the code for abstract polygons can no longer access the `vertices` instance variable, even for `Polygon` instances. Only instances of `Polygon` and its subclasses know about the `vertices` instance variable. One possible solution would be to define wrapper methods to access the `Polygon` class' `vertices` instance variable from within the `AbstractPolygon` class; the `Rectangle` class would define a method to construct a list of vertices from its four numeric instance variables.

To avoid any problems with altering the representation of a class in a subclass, only *leaf* classes should be concrete and define instance variables. All other non-leaf classes should be abstract, defining no instance variables, and their code should be written to invoke wrapper methods instead of explicit variable accesses. This programming style would support reuse of code while still allowing the representation of a subclass to be different from the representation of a superclass. But it would sacrifice the ability to share representation information by concatenating the instance variables of a class' ancestors, and it would require the definition and use of wrapper methods to access the instance variables. Thus, programs would be more awkward to write and modify.

Prototype-based languages can change the representation of refinements easily. In the rectangle example, the prototypical `rectangle` object contains four data slots and a parent slot pointing to the rectangle traits object, but doesn't include any data parent slots (see Figure 4b). By not including a data parent to the prototypical `polygon` object, the implementation is explicitly deciding not to base the representation of rectangles on the representation of polygons.

The rectangle traits object overrides the polygon traits object's `draw` method with one that is tuned to drawing rectangles using the representation specific to rectangles. To preserve compatibility with polygons, the rectangle traits object defines a `vertices` method to construct a list of vertices from the four numbers that define a rectangle. This is particularly convenient in SELF since a `vertices` message sent in a method in the polygon traits object would either access the `vertices` data slot of a polygon receiver or invoke the `vertices` method of a rectangle receiver, with no extra wrapper methods needed for the `vertices` data slot or modifications to the invoking methods. This convenience is afforded by SELF's uniform use of messages to access both state and behavior, and could be adopted by other classless and class-based languages to achieve similar flexibility. Trellis/Owl [12, 13], a class-based language, also accesses instance variables using messages and is able to change the representation of a subclass by overriding the instance variables inherited from its superclasses with methods defined in the subclass.

An implementation of a data type in a classless language can specify whether to extend the parent types' representations when forming the new type's representation by either including data parents that refer to some of the parent types' repre-

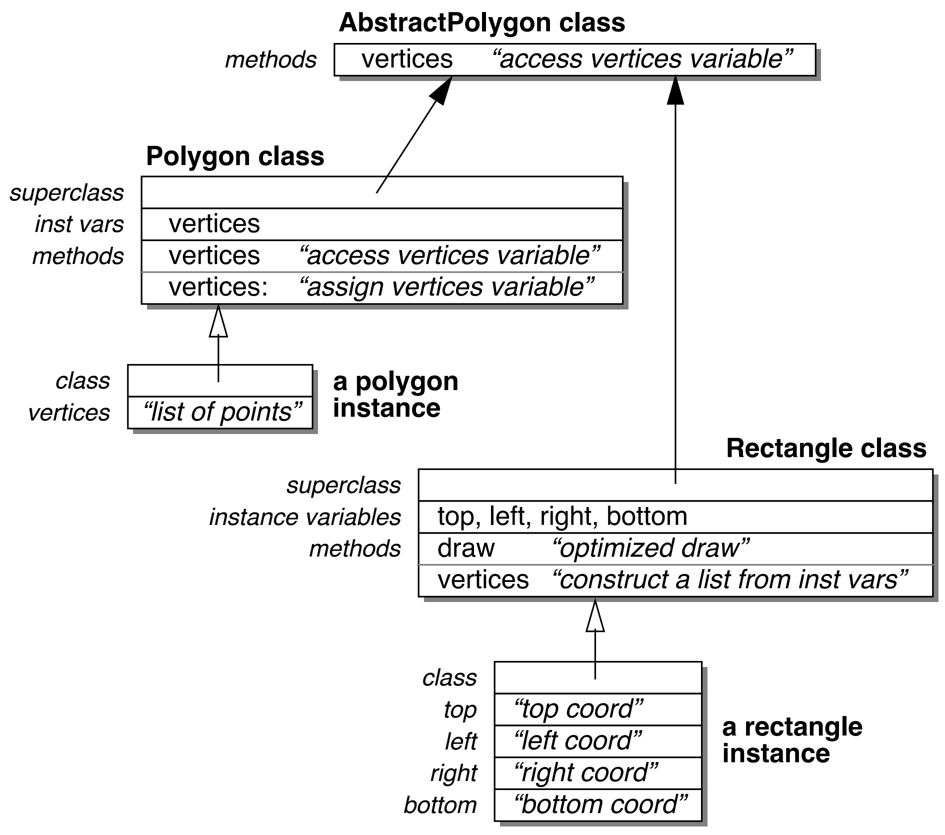


Figure 4a. Representation modification in a class-based language.

sentations (as in the filled polygon example) or not (as in the rectangle example). Both are natural and structured programming styles fostered by classless languages. Class-based languages typically have a much more difficult time handling cases that differ from strict representation extension. As mentioned above, Trellis/Owl is one notable exception. Languages with powerful metaclass facilities, such as CLOS [1], are able to define metaclasses for subclasses that do not inherit the instance variables of their superclasses, but this solution is much more complex and probably more verbose than the simple solution in classless languages.

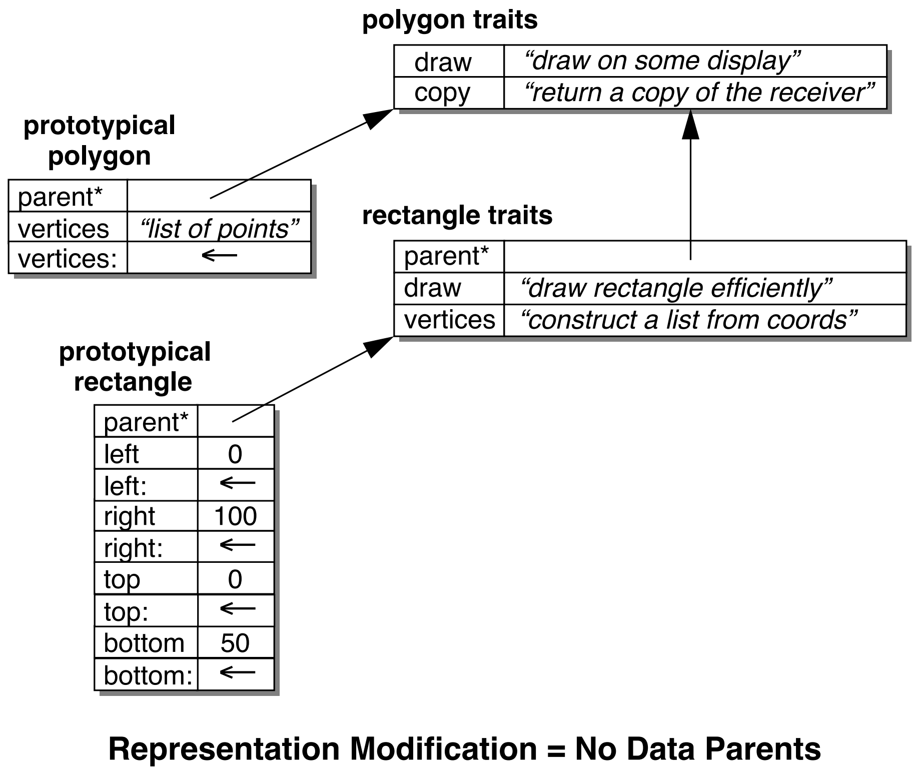


Figure 4b. Representation modification in a classless language.

2.5 Dynamic Behavior Changes: Changing An Instance's Class and Dynamic Inheritance

Sometimes the behavior of an instance of a data type can be divided into several different "modes" of behavior or implementation, with the state of the instance determining the mode of behavior. For example, a boxed polygon (using straight lines) has very different drawing methods than a smoothed polygon (using splines). In many situations, the distinction in "behavior" may be completely internal to the implementation of the data type, reflecting different ways of representing the instance depending on the current and past states of the object. A self-reorganizing collection might use radically different representations depending on recent access patterns, such as whether insertion has been more or less frequent than indexing, even though the external interface to the collection remains unchanged.

One common way of capturing different behavioral modes is to include a flag instance variable defining the behavior mode, and testing the flag at the beginning of each method that depends on the behavior mode. This obscures the code for each behavior mode, merging all behavior modes into shared methods that are sprinkled with if-tests and case-statements. This code is analogous to programs simulating object-oriented method dispatching: if-tests and case-statements are used to determine the type of the receiver of a "message." Not surprisingly, flag tests for behavior modes suffer from the same problems as flag tests for receiver types: it is hard to add new behavior modes without modifying lots of code, it is error-prone to write, and it is difficult to understand a particular mode since its code is intermixed with code for other behavior modes.

A better way of implementing behavior modes is to define each mode as its own special subtype of the general data type, and use method dispatching and inheritance to eliminate the flag tests. For example, the collection data type could be refined into an empty collection data type and a non-empty collection data type, using inheritance to relate the three types [8]. However, the behavior mode of an instance may change as its state changes: an empty collection becomes non-empty if an element is added to it. This would correspond in a class-based language to changing an object's class dynamically, and in a prototype-based language to changing an object's parent dynamically.

Most class-based languages do not allow an object to change its class, and those that do face hard problems. Since the class of an object implicitly specifies its representation, what happens to an object that changes its class to one that specifies a different representation? An object could be restricted to change its class only to those that have identical representations, but this wouldn't allow different behavior modes to have different representations.

Classless languages, on the other hand, can be naturally extended to handle dynamically-changing behavior modes by allowing an object's parents to change at run-time; an object can inherit from different behavior mode traits objects depending on its state. If the representations of the behavior modes differ, data parents can be used for behavior-mode-specific data slots; changing the behavior mode would then require changing both the traits parent and the data parent (or simply having the behavior mode data parent inherit directly from the behavior mode traits object and changing just the data parent). In SELF this *dynamic inheritance* comes for free with the basic object model. Since any data slot may be a parent slot, and any data slot may have a corresponding assignment slot, any parent slot may be assignable; an object's parents are changed simply by assigning to them.

In the polygon example, the boxed `draw` method would be the same as the `draw` method defined before in the polygon traits object; the smooth `draw` method would treat the vertices of the polygon as the spline's control points. The `polygon` prototype's parent slot would be assignable and alternate between the boxed polygon traits object and the smooth polygon traits object (see Figure 5).

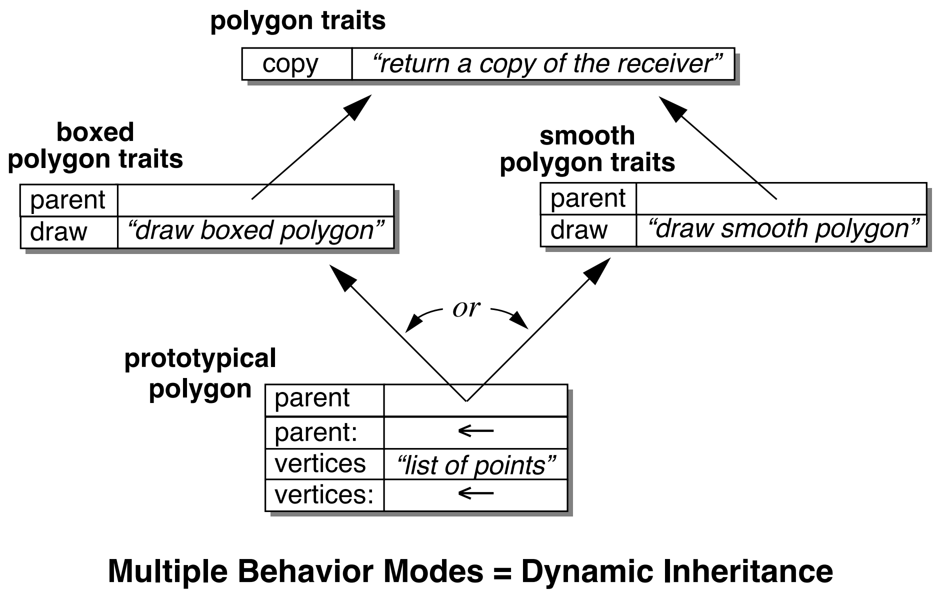


Figure 5. Multiple behavior modes in a classless language.

Behavior modes are naturally implemented in classless languages by using dynamic inheritance to choose from a small set of parents. This style of programming does not compromise the structure of the system; on the contrary, it can make the structure and organization of the system *clearer* by separating out the various modes of behavior. In contrast, the close coupling between a class and its representation prevent class-based languages from being extended naturally to handle behavior modes.

3 Encapsulation

Languages with user-defined data types usually provide a means for a data type to hide some of its attributes from other types. This encapsulation may be used to specify an external interface to an abstraction that should be unaffected by internal implementation changes or improvements, isolating the dependencies between a data type and its clients. Encapsulation may also be used to protect the local state of an implementation of a data type from external alterations that might violate an implementation invariant. Encapsulation thus can improve the structure and organization of the system as a whole by identifying public interfaces that should remain unaffected by implementation changes and allowing an implementation to preserve its internal invariants.

Existing encapsulation models are based on either objects or types. In languages with object-based encapsulation, such as the Smalltalk, Trellis/Owl, and Eiffel, the only accessible private members are the receiver's. In languages with type-based encapsulation, such as C++, the private members of *any* instance of the type are accessible from methods defined in the type.[5] Type-based encapsulation is significantly more flexible, supporting binary methods that need access to the private data of their arguments and initialization methods that need access to initialize the private state of newly created objects. With only receiver-based encapsulation, these situations require that initialization methods and wrapper methods be in the external public interface to the type, largely defeating the purpose of encapsulation in the first place.

Since classless languages have no explicit classes or types, it would appear that type-based encapsulation would be impossible to support, severely weakening any encapsulation provided by the language. Perhaps surprisingly, SELF's visibility rules *do* support a form of type-based encapsulation [5]. A method may access the private slots of any of its descendants or ancestors, so that a method defined in a traits object may access the private slots of all "instances" of the trait (i.e. clones of prototypes inheriting from the traits object), just as methods defined in a C++ class may access the private members of all instances of the class and its subclasses. In effect, the traits object itself defines a "type," with all descendant objects considered members of the type.

For example, in the polygon example before, the `polygon` prototype object's `vertices:` slot could be declared to be a private slot. This would prevent outside objects from modifying a polygon's list of vertices, but would allow the `copy` method defined in the `polygon` traits object to send the `vertices:` method to the new copied polygon object, since that new object is a descendant of the `polygon` traits object. Similarly, the assignment slots for rectangle objects could also be marked private to prevent unwanted external modification.

Both class-based and prototype-based languages may provide features for encapsulation, even for type-based encapsulation. These features are more dependent on the individual languages than whether the language includes classes or not.

4 Naming and Categorizing

Any system must be structured so that programs can name well-known objects and data types and so that programmers can find objects and types. Objects and

[5]Eiffel includes selective export clauses that allow object-based encapsulation to be extended to type-based encapsulation for particular members.

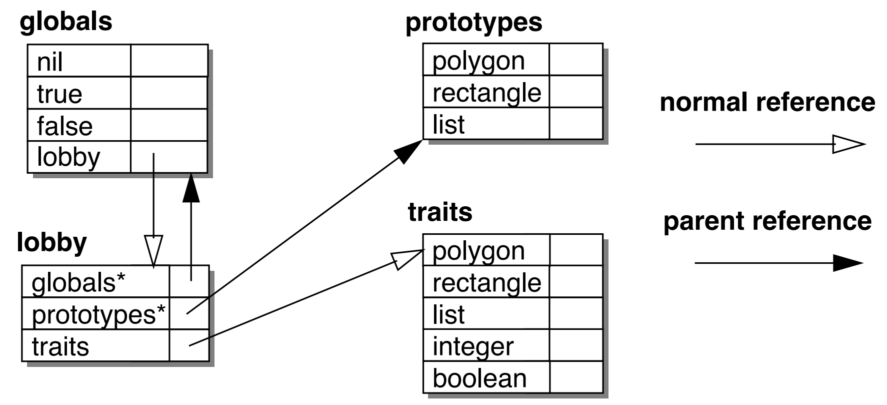


Figure 6. Name spaces are used for global references.

object inheritance support these tasks without explicit support from classes or extralingual environment structures.

4.1 Naming Objects: Global Variables and Name Spaces

Programs need to refer to well-known objects from many different places in the system. For example, a data type may need to be referenced from many places in order to create new instances of the type or to define subtypes. Most class-based languages associate a unique name with each class which may be uttered anywhere in the program to refer to the class; normal instance objects have no explicit names. In a classless language, prototypes and traits objects need to be globally accessible (to clone new objects and to define new refining traits objects), but since these objects are implemented by regular objects, they have no internal names.

In classless languages normal object inheritance may be used to define *name space objects* whose sole function is to provide names for well-known objects. The name of an object in a name space is simply the name of the slot that refers to the object. Any object that inherits the name space object may refer to well-known objects defined by the name space by *sending a message to itself* that accesses the appropriate slot of the name space object.[6] The scope of a name space is the set of objects that inherit from it. The designers of Eiffel encourage a similar strategy to handle shared, possibly global constants, although different language mechanisms are used to handle other global names like class names.

[6]In SELF, this approach is just as concise as global variables because state (e.g. well-known objects) may be accessed using messages without defining wrapper functions, and because messages sent to *self* are written with the *self* keyword omitted. Thus `polygon` is really a message sent to *self* that accesses data; this is just as concise as a global variable access.

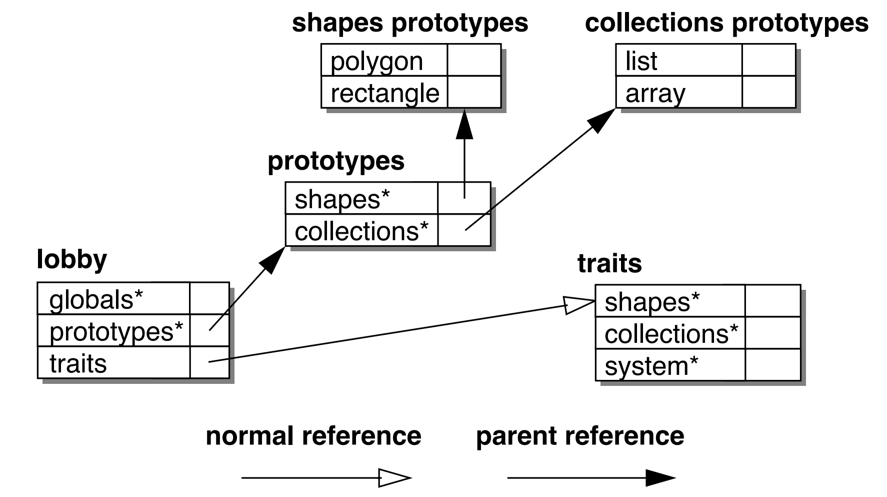


Figure 7. Categorizing name spaces.

Figure 6 illustrates name spaces with part of the inheritance graph in the SELF system. The `lobby` object is the "root" of the inheritance graph, since most objects inherit from it and expressions typed in at the SELF prompt are evaluated with the `lobby` as *self*. The `prototypes` parent object is therefore inherited by most objects and so provides succinct names for the prototypes of the standard data types. The `traits` object contains slots naming the traits objects in the system, typically using the same name as the name for the data type's prototypical instance. For example, the expression `polygon` names the polygon prototype, and the expression `traits polygon` names the polygon traits object. In the first case, since the prototypes name space object is inherited via the lobby, `polygon` yields the contents of the polygon slot in that name space object; in the second case, sending `traits` to the lobby gives the traits name space object, and sending `polygon` to that object gives the polygon traits object.

4.2 Organizing Names: Categories

Large flat name spaces for globals are convenient for programs, but awkward for programmers. Many systems provide features to help organize these name spaces into smaller *categories* of names that break down the name spaces into digestible chunks. For example, the Smalltalk-80 environment [6] supports a two-level structure for browsing classes, dividing up classes into *class categories*.

Classless systems using name space objects can be similarly broken down into categories by subdividing name spaces into multiple parents. For example, the

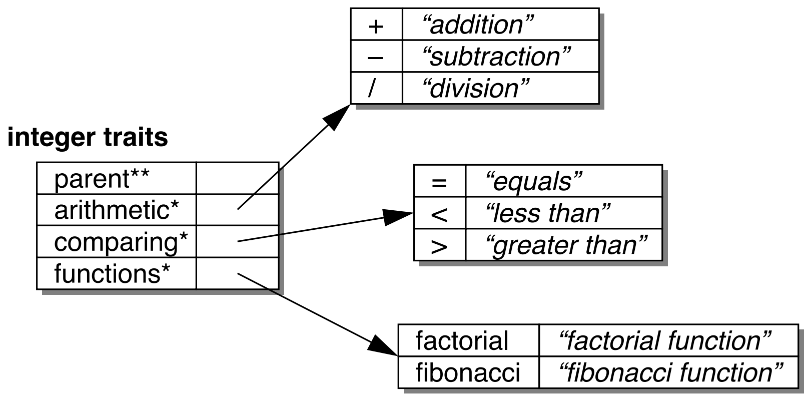


Figure 8. Categorizing traits objects.

`prototypes` name space object could be broken down into several name space subobjects, one for each kind of prototype. The original `prototypes` name space object contains parent slots referring to the name space subobjects; the name of each slot is the name of the category.

These *composite name spaces* behave just like a flat name space from the point of view of the program referring to global objects, since the categories are parents of the original name space object. (For example, the message `polygon` will still yield the prototypical polygon, even when the name space has been broken up into the categories as in Figure 7.) However, since the name spaces are actually structured into multiple objects, the programmer may browse them (using the facilities available for browsing objects) and use both the slot names and the object structure to locate objects of interest and to understand the organization of the system. Composite name spaces may have any number of levels of structure, and need not be balanced (some categories may be subcategorized while others are not). A single object may be categorized in several different ways simultaneously simply by defining slots in multiple categories that all refer to the object. This flexibility is a natural consequence of using normal objects for categorization.

Global variables are not the only name spaces that need to be broken up for programmers. Individual data types are a sort of name space for methods, and these name spaces may be large enough to require their own categorization. The Smalltalk-80 environment again provides a two-level structure for organizing the methods within a class into *method categories*.

For classless languages, the same techniques for organizing large name space objects may be applied to organize large traits objects. Each traits object may refer to parent subobjects that define some category of the slots of the traits object; the name of each parent slot is the name of that subobject category (see Figure 8). Again these *composite traits objects* extend to any number of levels of structure.

4.3 Extensional vs. Intensional Names and Categorization

By using name space objects and message passing to access global objects, an object's "name" becomes the sequence of message sends needed to reach it. We call this an *extensional name*, since it is derived from the structure of the system. Languages with internal class names, on the other hand, have *intensional names*, since classes are given explicit names by the programmer that may not be related to the structure of the system. Similarly, categorizing name spaces and traits objects using the object structure is *extensional categorization*, while using browser data structures to describe the categorization of classes and methods is *intensional categorization*.

Extensional names have a number of advantages over intensional names:

- No extra language or environment features are needed to support extensional names or categories.
- Extensional names have additional interpretations as *expressions that evaluate to the named object* (and so may be used within a program to access the object) and as *paths to reach the named object* (and so may be used in the browser to navigate to the object).
- The data structures defining intensional names for programmers can become inconsistent with the global variable names used by programs. For example, the internal names for classes and the data structures used by the environment to find a class' subclasses can become incorrect if the global variable referring to the class is renamed or if the inheritance hierarchy is changed without updating the browser's data structures. No such inconsistency can exist with extensional names, since they are derived from the actual structure of the system.

The only restriction associated with extensional names is that they must be legal expressions in the language (since an object's name must be described in the object structure). This restriction has not been a problem in our system, and we feel that the advantages of extensional naming over intensional naming are much more important.

5 Conclusion

Classes are not necessary for structure since objects themselves can provide it: traits objects provide behavior-sharing facilities for their instances and refinements, encapsulation mechanisms can provide type-based encapsulation without needing explicit types or classes, and structured name space objects provide names for programs to use and for people to browse. Traits objects and name space objects are no different than other objects, but their stylized use becomes an idiom that is instantly recognizable by the programmer. Languages without classes can structure programs as well as languages with classes.

Additionally, certain properties of traditional class-based systems conspire to hinder some kinds of useful structures that are handled naturally by classless systems. Since a class implicitly extends its superclasses' representations, it is hard to define a subclass that *alters* the representation defined by its superclasses. Classless languages define a type's representation explicitly using prototype objects, and so are able to implement both representation extension and representation alteration naturally. Because the representation of an object in a class-based system is so tied up with the object's class, it is difficult to implement dynamic behavior modes. Classless languages may use dynamic inheritance in a structured way to implement these behavior modes as a natural extension of static inheritance. Languages without classes can structure many programs better than languages with classes.

References

1. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification. Published as *SIGPLAN Notices*, 23, 9 (1988).
2. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.
3. Chambers, C., and Ungar, D. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.
4. Chambers, C., Ungar, D., and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 49-70. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
5. Chambers, C., Ungar, D., Chang, B., and Hölzle, U. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. To be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
6. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA (1984).

7. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).

8. LaLonde, W. R. Designing Families of Data Types Using Exemplars. In *ACM Transactions on Programming Languages and Systems*, 11, 2 (1989) 212-248.

9. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.

10. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

11. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.

12. Schaffert, C., Cooper, T., and Wilpolt, C. *Trellis Object-Based Environment: Language Reference Manual, Version 1.1*. DEC-TR-372, Digital Equipment Corp., Hudson, MA (1985).

13. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.

14. Stein, L. A. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 138-146.

15. Stein, L. A., Lieberman, H., and Ungar, D. A Shared View of Sharing: The Treaty of Orlando. In Kim, W., and Lochovosky, F., editors, *Object-Oriented Concepts, Applications, and Databases*, Addison-Wesley, Reading, MA (1988).

16. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA (1986).

17. Stroustrup, B. The Evolution of C++: 1985 to 1987. In *USENIX C++ Workshop Proceedings* (1987) 1-21.

18. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

Abstract

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data.*

1 Introduction

The "software crisis" was first identified in 1968 [NR69, p70] and in the intervening decades has deepened rather than abated. The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of *state* and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are *code volume*, and explicit concern with the *flow of control* through the system.

The classical ways to approach the difficulty of state include object-oriented programming which tightly couples state together with related behaviour, and functional programming which — in its pure form — eschews state and side-effects all together. These approaches each suffer from various (and differing) problems when applied to traditional large-scale systems.

We argue that it is possible to take useful ideas from both and that — when combined with some ideas from the relational database world —

this approach offers significant potential for simplifying the construction of large-scale software systems.

The paper is divided into two halves. In the first half we focus on complexity. In section 2 we look at complexity in general and justify our assertion that it is at the root of the crisis, then we look at how we currently attempt to understand systems in section 3. In section 4 we look at the causes of complexity (i.e. things which make understanding difficult) before discussing the classical approaches to managing these complexity causes in section 5. In section 6 we define what we mean by "accidental" and "essential" and then in section 7 we give recommendations for alternative ways of addressing the causes of complexity — with an emphasis on avoidance of the problems rather than coping with them.

In the second half of the paper we consider in more detail a possible approach that follows our recommended strategy. We start with a review of the relational model in section 8 and give an overview of the potential approach in section 9. In section 10 we give a brief example of how the approach might be used.

Finally we contrast our approach with others in section 11 and then give conclusions in section 12.

2 Complexity

In his classic paper — "No Silver Bullet" Brooks[Bro86] identified four properties of software systems which make building software hard: Complexity, Conformity, Changeability and Invisibility. Of these we believe that Complexity is the *only* significant one — the others can either be *classified as* forms of complexity, or be seen as problematic solely *because* of the complexity in the system.

Complexity is *the* root cause of the vast majority of problems with software today. Unreliability, late delivery, lack of security — often even poor performance in large-scale systems can all be seen as deriving ultimately from unmanageable complexity. The primary status of complexity as *the* major cause of these other problems comes simply from the fact that being able to *understand* a system is a prerequisite for avoiding all of them, and of course it is this which complexity destroys.

The relevance of complexity is widely recognised. As Dijkstra said [Dij97, EWD1243]:

> "...we have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making..."

. . . and the Economist devoted a whole article to software complexity [Eco04] — noting that by some estimates software problems cost the American economy $59 billion annually.

Being able to think and reason about our systems (particularly the effects of changes to those systems) is of *crucial* importance. The dangers of complexity and the importance of simplicity in this regard have also been a popular topic in ACM Turing award lectures. In his 1990 lecture Corbato said [Cor91]:

> "The general problem with ambitious systems is complexity.", "...it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties"

In 1977 Backus [Bac78] talked about the "complexities and weaknesses" of traditional languages and noted:

> "there is a desperate need for a powerful methodology to help us think about programs. ... conventional languages create unnecessary confusion in the way we think about programs"

Finally, in his Turing award speech in 1980 Hoare [Hoa81] observed:

> "...there is one quality that cannot be purchased... — and that is reliability. The price of reliability is the pursuit of the utmost simplicity"

and

> "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult."

This is the unfortunate truth:

Simplicity is *Hard*

. . . but the purpose of this paper is to give some cause for optimism.

One final point is that the type of complexity we are discussing in this paper is that which makes large systems *hard to understand*. It is this that causes *us* to expend huge resources in *creating and maintaining* such systems. This type of complexity has nothing to do with complexity theory — the branch of computer science which studies the resources consumed *by a machine executing* a program. The two are completely unrelated — it is a straightforward matter to write a small program in a few lines which is incredibly simple (in our sense) and yet is of the highest complexity class (in the complexity theory sense). From this point on we shall only discuss complexity of the first kind.

We shall look at what we consider to be the major common causes of complexity (things which make understanding difficult) after first discussing exactly how we normally attempt to *understand* systems.

3 Approaches to Understanding

We argued above that the danger of complexity came from its impact on our attempts to *understand* a system. Because of this, it is helpful to consider the mechanisms that are commonly used to try to understand systems. We can then later consider the impact that potential causes of complexity have on these approaches. There are two widely-used approaches to understanding systems (or components of systems):

Testing This is attempting to understand a system from the outside — as a "black box". Conclusions about the system are drawn on the basis of observations about how it behaves in certain specific situations. Testing may be performed either by human or by machine. The former is more common for whole-system testing, the latter more common for individual component testing.

Informal Reasoning This is attempting to understand the system by examining it from the inside. The hope is that by using the extra information available, a more accurate understanding can be gained.

Of the two informal reasoning is the most important by far. This is because — as we shall see below — there are inherent limits to what can be achieved by testing, and because informal reasoning (by virtue of being an inherent part of the development process) is *always* used. The other justification is that improvements in informal reasoning will lead to *less errors being created* whilst all that improvements in testing can do is to lead to *more errors being detected*. As Dijkstra said in his Turing award speech [Dij72, EWD340]:

> "Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with."

and as O'Keefe (who also stressed the importance of "*understanding your problem*" and that "*Elegance is not optional*") said [O'K90]:

> "Our response to mistakes should be to look for ways that we can avoid making them, not to blame the nature of things."

The key problem with testing is that a test (of any kind) that uses one particular set of inputs tells you *nothing at all* about the behaviour of the system or component when it is given a different set of inputs. The huge number of different possible inputs usually rules out the possibility of testing them all, hence the unavoidable concern with testing will always be — *have you performed the* right *tests?*. The only certain answer you will ever get to this question is an answer in the negative — when the system breaks. Again, as Dijkstra observed [Dij71, EWD303]:

> "testing is hopelessly inadequate....(it) can be used very effectively to show the presence of bugs but never to show their absence."

We agree with Dijkstra. *Rely* on testing at your peril.

This is *not* to say that testing has no use. The bottom line is that *all* ways of attempting to understand a system have their limitations (and this includes both *informal reasoning* — which is limited in scope, imprecise and hence prone to error — as well as *formal reasoning* — which is dependent upon the accuracy of a specification). Because of these limitations it may often be prudent to employ both testing *and* reasoning together.

It is precisely *because* of the limitations of all these approaches that *simplicity* is vital. When considered next to testing and reasoning, simplicity is more important than either. Given a stark choice between investment in testing and investment in simplicity, the latter may often be the better choice because it will facilitate *all* future attempts to understand the system — attempts of any kind.

4 Causes of Complexity

In any non-trivial system there is some complexity inherent in the problem that needs to be solved. In real large systems however, we regularly encounter complexity whose status as "inherent in the problem" is open to some doubt. We now consider some of these causes of complexity.

4.1 Complexity caused by State

Anyone who has ever telephoned a support desk for a software system and been told to "try it again", or "reload the document", or "restart the program", or "reboot your computer" or "re-install the program" or even "re-install the operating system and then the program" has direct experience of the problems that *state*[1] causes for writing reliable, understandable software.

The reason these quotes will sound familiar to many people is that they are often suggested because they are often successful in resolving the problem. The reason that they are often successful in resolving the problem is that many systems have errors in their handling of state. The reason that many of these errors exist is that the presence of state makes programs hard to *understand*. It makes them complex.

When it comes to state, we are in broad agreement with Brooks' sentiment when he says [Bro86]:

> "From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability"

— we agree with this, but believe that it is the presence of many possible states which gives rise to the complexity in the first place, and:

> "computers. . . have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do."

4.1.1 Impact of State on Testing

The severity of the impact of *state* on testing noted by Brooks is hard to over-emphasise. State affects all types of testing — from system-level testing (where the tester will be at the mercy of the same problems as the hapless user just mentioned) through to component-level or unit testing. The key problem is that a test (of any kind) on a system or component that is in one particular *state* tells you *nothing at all* about the behaviour of that system or component when it happens to be in another *state*.

The common approach to testing a stateful system (either at the component or system levels) is to start it up such that it is in some kind of "clean" or "initial" (albeit mostly *hidden*) state, perform the desired tests using the

[1]By "state" we mean *mutable state* specifically — i.e. excluding things like (immutable) single-assignment variables which are provided by logic programming languages for example

test inputs and then rely upon the (often in the case of bugs ill-founded) assumption that the system would perform the same way — *regardless of its hidden internal state* — every time the test is run with those inputs.

In essence, this approach is simply sweeping the problem of state under the carpet. The reasons that this is done are firstly because it is often possible to get away with this approach and more crucially because there isn't really any alternative when you're testing a stateful system with a complicated internal hidden state.

The difficulty of course is that it's not *always* possible to "get away with it" — if some sequence of events (inputs) can cause the system to "get into a bad *state*" (specifically an internal hidden state which was *different* from the one in which the test was performed) then things can and do go wrong. This is what is happening to the hypothetical support-desk caller discussed at the beginning of this section. The proposed remedies are all attempts to force the system back into a "good internal state".

This problem (that a test in one *state* tells you *nothing at all* about the system in a different *state*) is a direct parallel to one of the fundamental problems with testing discussed above — namely that testing for one *set of inputs* tells you *nothing at all* about the behaviour with a different *set of inputs*. In fact the problem caused by state is typically worse — particularly when testing large chunks of a system — simply because even though the number of possible *inputs* may be very large, the number of possible *states* the system can be in is often even larger.

These two similar problems — one intrinsic to testing, the other coming from *state* — combine together *horribly*. Each introduces a huge amount of uncertainty, and we are left with very little about which we *can* be certain if the system/component under scrutiny is of a stateful nature.

4.1.2 Impact of State on Informal Reasoning

In addition to causing problems for understanding a system from the outside, state also hinders the developer who must attempt to *reason* (most commonly on an informal basis) about the expected behaviour of the system "from the inside".

The mental processes which are used to do this informal reasoning often revolve around a case-by-case mental simulation of behaviour: "if this variable is in this state, then this will happen — which is correct — otherwise that will happen — which is also correct". As the number of states — and hence the number of possible scenarios that must be considered — grows, the effectiveness of this mental approach buckles almost as quickly as testing (it does achieve some advantage through abstraction over sets of similar values which can be seen to be treated identically).

One of the issues (that affects both testing and reasoning) is the exponential rate at which the number of possible states grows — for every single *bit* of state that we add we *double* the total number of possible states. Another issue — which is a particular problem for informal reasoning — is *contamination*.

Consider a system to be made up of procedures, some of which are stateful and others which aren't. We have already discussed the difficulties of understanding the bits which are stateful, but what we would hope is that the procedures which aren't themselves stateful would be more easy to comprehend. Alas, this is largely not the case. If the procedure in question (which is itself stateless) makes use of any other procedure which *is* stateful — *even indirectly* — then all bets are off, our procedure becomes *contaminated* and we can only understand it in the context of state. If we try to do anything else we will again run the risk of all the classic state-derived problems discussed above. As has been said, the problem with state is that "*when you let the nose of the camel into the tent, the rest of him tends to follow*".

As a result of all the above reasons it is our belief that the single biggest remaining cause of complexity in most contemporary large systems is *state*, and the more we can do to *limit* and *manage* state, the better.

4.2 Complexity caused by Control

Control is basically about the *order* in which things happen.

The problem with control is that very often we do not want to have to be concerned with this. Obviously — given that we want to construct a real system in which things will actually *happen* — at some point order is going to be relevant to someone, but there are significant risks in concerning ourselves with this issue unnecessarily.

Most traditional programming languages *do* force a concern with ordering — most often the ordering in which things will *happen* is controlled by the order in which the statements of the programming language are written in the textual form of the program. This order is then modified by explicit branching instructions (possibly with conditions attached), and subroutines are normally provided which will be invoked in an implicit stack.

Of course a variety of evaluation orders is possible, but there is little variation in this regard amongst widespread languages.

The difficulty is that when control is an implicit part of the language (as it almost always is), then every single piece of program must be understood in that context — even when (as is often the case) the programmer may wish to say nothing about this. When a programmer is forced (through use of a language with implicit control flow) to specify the control, he or she is being forced to specify an aspect of *how* the system should work rather than simply *what* is desired. Effectively they are being forced to *over-specify* the problem. Consider the simple pseudo-code below:

```
a := b + 3
c := d + 2
e := f * 4
```

In this case it is clear that the programmer has no concern at all with the order in which (i.e. *how*) these things eventually happen. The programmer is only interested in specifying a *relationship* between certain values, but has been forced to say more than this by choosing an arbitrary control flow. Often in cases such as this a compiler may go to lengths to establish that such a requirement (ordering) — which the programmer has been forced to make because of the semantics of the language — can be safely ignored.

In simple cases like the above the issue is often given little consideration, but it is important to realise that two completely unnecessary things are happening — first an artificial ordering is being imposed, and then further work is done to remove it.

This seemingly innocuous occurrence can actually significantly complicate the process of informal reasoning. This is because the person reading the code above must effectively duplicate the work of the hypothetical compiler — they must (by virtue of the definition of the language semantics) start with the assumption that the ordering specified *is* significant, and then by further inspection determine that it is not (in cases less trivial than the one above determining this can become very difficult). The problem here is that mistakes in this determination can lead to the introduction of very subtle and hard-to-find bugs.

It is important to note that the problem is not in the *text* of the program above — after all that does have to be written down in some order — it is solely in the *semantics* of the hypothetical imperative language we have assumed. It *is* possible to consider the exact same program text as being a valid program in a language whose semantics did not define a run-time sequencing based upon textual ordering within the program[2].

[2]Indeed early versions of the Oz language (with *implicit* concurrency at the statement level) were somewhat of this kind [vRH04, p809].

Having considered the impact of control on *informal reasoning*, we now look at a second control-related problem, concurrency, which affects *testing* as well.

Like basic control such as branching, but as opposed to sequencing, concurrency is normally specified *explicitly* in most languages. The most common model is "shared-state concurrency" in which specification for explicit synchronization is provided. The impacts that this has for informal reasoning are well known, and the difficulty comes from adding further to the *number of scenarios* that must mentally be considered as the program is read. (In this respect the problem is similar to that of state which also adds to the number of scenarios for mental consideration as noted above).

Concurrency also affects testing, for in this case, we can no longer even be assured of result consistency when repeating tests on a system — even if we somehow ensure a consistent starting state. Running a test in the presence of concurrency with a known initial state and set of inputs tells you *nothing at all* about what will happen the next time you run that very same test with the very same inputs and the very same starting state... and things can't really get any worse than that.

4.3 Complexity caused by Code Volume

The final cause of complexity that we want to examine in any detail is sheer code volume.

This cause is basically in many ways a secondary effect — much code is simply concerned with managing *state* or specifying *control*. Because of this we shall often not mention code volume explicitly. It is however worth brief independent attention for at least two reasons — firstly because it is the easiest form of complexity to *measure*, and secondly because it interacts badly with the other causes of complexity and this is important to consider.

Brooks noted [Bro86]:

> "Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increase with size"

We basically agree that *in most current systems* this is true (we disagree with the word "essential" as already noted) — i.e. in most systems complexity definitely *does* exhibit nonlinear increase with size (of the code). This nonlinearity in turn means that it's vital to reduce the amount of code to an absolute minimum.

We also want to draw attention to one of Dijkstra's [Dij72, EWD340] thoughts on this subject:

> "It has been suggested that there is some kind of law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true. . . . I tend to the assumption — up till now not disproved by experience — that by suitable application of our powers of abstraction, the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length."

We agree with this — it is the reason for our "in most current systems" caveat above. We believe that — with the effective management of the two major complexity causes which we have discussed, state and control — it becomes *far* less clear that complexity increases with code volume in a non-linear way.

4.4 Other causes of complexity

Finally there are other causes, for example: duplicated code, code which is never actually used ("dead code"), **unnecessary abstraction**[3], missed abstraction, poor modularity, poor documentation. . .

All of these other causes come down to the following three (inter-related) principles:

Complexity breeds complexity There are a whole set of *secondary* causes of complexity. This covers all complexity introduced *as a result of* not being able to clearly understand a system. *Duplication* is a prime example of this — if (due to state, control or code volume) it is not clear that functionality already exists, or it is too *complex* to understand whether what exists does exactly what is required, there will be a strong tendency to duplicate. This is particularly true in the presence of time pressures.

Simplicity is *Hard* This was noted above — significant *effort* can be required to achieve simplicity. The first solution is often not the most simple, particularly if there is existing complexity, or time pressure. Simplicity can only be attained if it is recognised, sought and prized.

[3]Particularly unnecessary *data* abstraction. We examine an argument that this is actually *most* data abstraction in section 9.2.4.

Power corrupts What we mean by this is that, in the absence of language-enforced guarantees (i.e. restrictions on the *power* of the language) mistakes (and abuses) *will* happen. This is the reason that garbage collection is good — the *power* of manual memory management is removed. Exactly the same principle applies to *state* — another kind of *power*. In this case it means that we need to be very wary of any language that even *permits* state, regardless of how much it discourages its use (obvious examples are ML and Scheme). The bottom line is that the more *powerful* a language (i.e. the more that is *possible* within the language), the harder it is to *understand* systems constructed in it.

Some of these causes are of a human-nature, others due to environmental issues, but all can — we believe — be greatly alleviated by focusing on effective management of the complexity causes discussed in sections 4.1–4.3.

5 Classical approaches to managing complexity

The different classical approaches to managing complexity can perhaps best be understood by looking at how programming languages of each of the three major styles (imperative, functional, logic) approach the issue. (We take object-oriented languages as a commonly used example of the imperative style).

5.1 Object-Orientation

Object-orientation — whilst being a very broadly applied term (encompassing everything from Java-style class-based to Self-style prototype-based languages, from single-dispatch to CLOS-style multiple dispatch languages, and from traditional passive objects to the active / actor styles) — is essentially an imperative approach to programming. It has evolved as the dominant method of general software development for traditional (von-Neumann) computers, and many of its characteristics spring from a desire to facilitate von-Neumann style (i.e. state-based) computation.

5.1.1 State

In most forms of object-oriented programming (OOP) an object is seen as consisting of some state together with a set of procedures for accessing and manipulating that state.

This is essentially similar to the (earlier) idea of an *abstract data type (ADT)* and is one of the primary strengths of the OOP approach when compared with less structured imperative styles. In the OOP context this is referred to as the idea of *encapsulation*, and it allows the programmer to enforce integrity constraints over an object's state by regulating access to that state through the access procedures ("methods").

One problem with this is that, if several of the access procedures access or manipulate the same bit of state, then there may be several places where a given constraint must be enforced (these different access procedures may or may not be within the same file depending on the specific language and whether features, such as inheritance, are in use). Another major problem[4] is that encapsulation-based integrity constraint enforcement is strongly biased toward single-object constraints and it is awkward to enforce more complicated constraints involving multiple objects with this approach (for one thing it becomes unclear where such multiple-object constraints should reside).

Identity and State

There is one other intrinsic aspect of OOP which is intimately bound up with the issue of state, and that is the concept of *object identity.*

In OOP, each object is seen as being a uniquely identifiable entity regardless of its attributes. This is known as *intensional* identity (in contrast with *extensional* identity in which things are considered the same if their attributes are the same). As Baker observed [Bak93]:

> In a sense, object identity can be considered to be a rejection of the "relational algebra" view of the world in which two objects can only be distinguished through differing attributes.

Object identity *does* make sense when objects are used to provide a (mutable) stateful abstraction — because two distinct stateful objects can be mutated to contain different state even if their attributes (the contained state) happen initially to be the same.

However, in other situations where mutability is *not* required (such as — say — the need to represent a simple numeric value), the OOP approach is forced to adopt techniques such as the creation of "Value Objects", and an

[4]this particular problem doesn't really apply to object-oriented languages (such as CLOS) which are based upon generic functions — but they don't have the same concept of encapsulation.

attempt is made to de-emphasise the original intensional concept of *object identity* and re-introduce extensional identity. In these cases it is common to start using custom access procedures (methods) to determine whether two objects are equivalent in some other, domain-specific sense. (One risk — aside from the extra code volume required to support this — is that there can no longer be any guarantee that such domain-specific equivalence concepts conform to the standard idea of an equivalence relation — for example there is not necessarily any guarantee of transitivity).

The intrinsic concept of *object identity* stems directly from the use of state, and is (being part of the paradigm itself) unavoidable. This additional concept of identity adds complexity to the task of reasoning about systems developed in the OOP style (it is necessary to switch mentally between the two equivalence concepts — serious errors can result from confusion between the two).

State in OOP

The bottom line is that all forms of OOP rely on state (contained within objects) and in general all behaviour is affected by this state. As a result of this, OOP suffers directly from the problems associated with state described above, and as such we believe that it does not provide an adequate foundation for avoiding complexity.

5.1.2 Control

Most OOP languages offer standard sequential control flow, and many offer explicit classical "shared-state concurrency" mechanisms together with all the standard complexity problems that these can cause. One slight variation is that actor-style languages use the "message-passing" model of concurrency — they associate threads of control with individual objects and messages are passed between these. This can lead to easier informal reasoning in some cases, but the use of actor-style languages is not widespread.

5.1.3 Summary — OOP

Conventional imperative and object-oriented programs suffer greatly from both state-derived and control-derived complexity.

5.2 Functional Programming

Whilst OOP developed out of a desire to offer improved ways of managing and dealing with the classic stateful von-Neumann architecture, functional programming has its roots in the completely stateless lambda calculus of Church (we are ignoring the even simpler functional systems based on combinatory logic). The untyped lambda calculus is known to be equivalent in power to the standard stateful abstraction of computation — the Turing machine.

5.2.1 State

Modern functional programming languages are often classified as 'pure' — those such as Haskell[PJ^{+}03] which shun state and side-effects completely, and 'impure' — those such as ML which, whilst advocating the avoidance of state and side-effects in general, do permit their use. Where not explicitly mentioned we shall generally be considering functional programming in its pure form.

The primary strength of functional programming is that by avoiding state (and side-effects) the entire system gains the property of *referential transparency* — which implies that when supplied with a given set of arguments a function will *always* return exactly the same result (speaking loosely we could say that it will always behave in the same way). Everything which can possibly affect the result in any way is always immediately visible in the actual parameters.

It is this cast iron guarantee of *referential transparency* that obliterates one of the two crucial weaknesses of testing as discussed above. As a result, even though the other weakness of testing remains (testing for one set of inputs says *nothing at all* about behaviour with another set of inputs), testing does become far more effective if a system has been developed in a functional style.

By avoiding state functional programming also avoids all of the other state-related weaknesses discussed above, so — for example — informal reasoning also becomes much more effective.

5.2.2 Control

Most functional languages specify implicit (left-to-right) sequencing (of calculation of function arguments) and hence they face many of the same issues mentioned above. Functional languages do derive one slight benefit when it comes to control because they encourage a more abstract use of control using functionals (such as `fold` / `map`) rather than explicit looping.

There are also concurrent versions of many functional languages, and the fact that state is generally avoided can give benefits in this area (for example in a pure functional language it will always be safe to evaluate all arguments to a function in parallel).

5.2.3 Kinds of State

In most of this paper when we refer to "state" what we really mean is *mutable state*.

In languages which do not support (or discourage) mutable state it is common to achieve somewhat similar effects by means of passing extra parameters to procedures (functions). Consider a procedure which performs some internal stateful computation and returns a result — perhaps the procedure implements a counter and returns an incremented value each time it is called:

```
procedure int getNextCounter()
  // 'counter' is declared and initialized elsewhere in the code
  counter := counter + 1
  return counter
```

The way that this is typically implemented in a basic functional programming language is to replace the stateful procedure which took no arguments and returned one result with a function which takes one argument and returns a pair of values as a result.

```
function (int,int) getNextCounter(int oldCounter)
  let int result = oldCounter + 1
  let int newCounter = oldCounter + 1
  return (newCounter, result)
```

There is then an obligation upon the caller of the function to make sure that the next time the `getNextCounter` function gets called it is supplied with the `newCounter` returned from the previous invocation.Effectively what is happening is that the *mutable* state that was hidden inside the `getNextCounter` procedure is replaced by an extra parameter on both the input and output of the `getNextCounter` function. This extra parameter is not *mutable* in any way (the entity which is referred to by `oldCounter` is a different *value* each time the function is called).

As we have discussed, the functional version of this program *is* referentially transparent, and the imperative version is not (hence the caller of the `getNextCounter` *procedure* has no idea what may influence the result he gets — it could in principle be dependent upon many, many different hidden mutable variables — but the caller of the `getNextCounter` *function* can instantly see exactly that the result can depend only on the single value supplied to the function).

Despite this, the fact is that we are using functional values to simulate state. There is *in principle* nothing to stop functional programs from passing a single extra parameter into and out of every single function in the entire system. If this extra parameter were a collection (compound value) of some kind then it could be used to *simulate* an arbitrarily large set of mutable variables. In effect this approach recreates a single pool of global variables — hence, even though referential transparency is maintained, ease of reasoning is lost (we still know that each function is dependent only upon its arguments, but one of them has become so large *and contains irrelevant values* that the benefit of this knowledge as an aid to understanding is almost nothing). This is however an extreme example and does not detract from the general power of the functional approach.

It is worth noting in passing that — even though it would be no substitute for a *guarantee* of referential transparency — there is no reason why the functional style of programming cannot be adopted in stateful languages (i.e. imperative as well as impure functional ones). More generally, we would argue that — whatever the language being used — there are large benefits to be had from avoiding hidden, implicit, mutable state.

5.2.4 State and Modularity

It is sometimes argued (e.g. [vRH04, p315]) that state is important because it permits a particular kind of modularity. This is certainly true. Working within a stateful framework it is possible to add state to any component without adjusting the components which invoke it. Working within a functional framework the same effect can only be achieved by adjusting every single component that invokes it to carry the additional information around (as with the `getNextCounter` function above).

There is a fundamental trade off between the two approaches. In the functional approach (when trying to achieve state-like results) you are forced to make changes to every part of the program that could be affected (adding the relevant extra parameter), in the stateful you are not.

But what this means is that in a functional program *you can always tell* *exactly what will control the outcome of a procedure (i.e. function)* simply by looking at the arguments supplied where it is invoked. In a stateful program this property (again a consequence of *referential transparency*) is completely destroyed, you can never tell what will control the outcome, and *potentially* have to look at every single piece of code in the *entire system* to determine this information.

The trade-off is between *complexity* (with the ability to take a shortcut when making some specific types of change) and *simplicity* (with *huge* improvements in both testing and reasoning). As with the discipline of (static) typing, it is trading a one-off up-front cost for continuing future gains and safety ("one-off" because each piece of code is written once but is read, reasoned about and tested on a continuing basis).

A further problem with the modularity argument is that some examples — such as the use of procedure (function) invocation counts for debugging / performance-tuning purposes — seem to be better addressed within the supporting infrastructure / language, rather than within the system itself (we prefer to advocate a clear separation between such administrative/diagnostic information and the core logic of the system).

Still, the fact remains that such arguments have been insufficient to result in widespread adoption of functional programming. We must therefore conclude that the main weakness of functional programming is the flip side of its main strength — namely that problems arise when (as is often the case) the system to be built must maintain state of some kind.

The question inevitably arises of whether we can find some way to "have our cake and eat it". One potential approach is the elegant system of monads used by Haskell [Wad95]. This does basically allow you to avoid the problem described above, but it can very easily be abused to create a stateful, side-effecting sub-language (and hence re-introduce all the problems we are seeking to avoid) inside Haskell — albeit one that is marked by its type. Again, despite their huge strengths, monads have as yet been insufficient to give rise to widespread adoption of functional techniques.

5.2.5 Summary — Functional Programming

Functional programming goes a long way towards avoiding the problems of state-derived complexity. This has very significant benefits for testing (avoiding what is normally one of testing's biggest weaknesses) as well as for reasoning.

5.3 Logic Programming

Together with functional programming, logic programming is considered to be a *declarative* style of programming because the emphasis is on specifying *what* needs to be done rather than exactly *how* to do it. Also as with functional programming — and in contrast with OOP — its principles and the way of thinking encouraged do *not* derive from the stateful von-Neumann architecture.

Pure logic programming is the approach of doing nothing more than making statements *about* the problem (and desired solutions). This is done by stating a set of *axioms* which *describe* the problem and the attributes required of something for it to be considered a solution. The ideal of logic programming is that there should be an infrastructure which can take the raw axioms and use them to find or check solutions. All solutions are formal logical consequences of the axioms supplied, and "running" the system is equivalent to the construction of a formal *proof* of each solution.

The seminal "logic programming" language was Prolog. Prolog is best seen as a *pure logical core* (pure Prolog) with various *extra-logical*[5] extensions. Pure Prolog is close to the ideals of logic programming, but there are important differences. Every pure Prolog program can be "read" in two ways — either as a *pure set of logical axioms* (i.e. assertions about the problem domain — this is the pure logic programming reading), or *operationally* — as a sequence of commands which are applied (in a particular order) to determine whether a goal can be deduced (from the axioms). This second reading corresponds to the actual way that pure Prolog will make use of the axioms when it tries to prove goals. It is worth noting that a single Prolog program can be both correct when read in the first way, and incorrect (for example due to non-termination) when read in the second.

It is for this reason that Prolog falls short of the ideals of logic programming. Specifically it is necessary to be concerned with the operational interpretation of the program whilst writing the axioms.

5.3.1 State

Pure logic programming makes no use of mutable state, and for this reason profits from the same advantages in understandability that accrue to pure functional programming. Many languages based on the paradigm do however provide some stateful mechanisms. In the extra-logical part of Prolog

[5]We are using the term here to cover *everything* apart from the pure core of Prolog — for example we include what are sometimes referred to as the *meta-logical* features

for example there are facilities for adjusting the program itself by adding new axioms for example. Other languages such as Oz (which has its roots in logic programming but has been extended to become "multi-paradigm") provide mutable state in a traditional way — similar to the way it is provided by *impure* functional languages.

All of these approaches to state sacrifice referential transparency and hence potentially suffer from the same drawbacks as imperative languages in this regard. The one advantage that all these impure non-von-Neumann derived languages can claim is that — whilst state is permitted its use is generally discouraged (which is in stark contrast to the stateful von-Neumann world). Still, without purity there are no guarantees and all the same state-related problems can sometimes occur.

5.3.2 Control

In the case of pure Prolog the language specifies both an *implicit* ordering for the processing of sub-goals (left to right), and also an *implicit* ordering of clause application (top down) — these basically correspond to an operational commitment to *process* the program in the same order as it is read textually (in a depth first manner). This means that some particular ways of writing down the program can lead to non-termination, and — when combined with the fact that some extra-logical features of the language permit side-effects — leads inevitably to the standard difficulty for informal reasoning caused by control flow. (Note that these reasoning difficulties do *not* arise in ideal world of logic programming where there simply is no specified control — as distinct from in pure Prolog programming where there is).

As for Prolog's other extra-logical features, some of them further widen the gap between the language and logic programming in its ideal form. One example of this is the provision of "cuts" which offer *explicit* restriction of control flow. These explicit restrictions are intertwined with the pure logic component of the system and inevitably have an adverse affect on attempts to reason about the program (misunderstandings of the effects of cuts are recognised to be a major source of bugs in Prolog programs [SS94, p190]).

It is worth noting that some more modern languages of the logic programming family offer more flexibility over control than the implicit depth-first search used by Prolog. One example would be Oz which offers the ability to program specific control strategies which can then be applied to different problems as desired. This is a very useful feature because it allows significant *explicit* control flexibility to be specified *separately* from the main program (i.e. without contaminating it through the addition of control complexity).

5.3.3 Summary — Logic Programming

One of the most interesting things about logic programming is that (despite the limitations of some actual logic-based languages) it offers the tantalising promise of the ability to escape from the complexity problems caused by control.

6 Accidents and Essence

Brooks defined difficulties of "essence" as those inherent in the nature of software and classified the rest as "accidents".

We shall basically use the terms in the same sense — but prefer to start by considering the complexity of the problem itself before software has even entered the picture. Hence we define the following two types of complexity:

Essential Complexity is inherent in, and the essence of, the *problem* (as seen by the *users*).

Accidental Complexity is all the rest — complexity with which the development team would not have to deal in the ideal world (e.g. complexity arising from performance issues and from suboptimal language and infrastructure).

Note that the definition of *essential* is deliberately more strict than common usage. Specifically when we use the term *essential* we will mean strictly essential *to the users' problem* (as opposed to — perhaps — essential *to some specific, implemented, system*, or even — essential *to software in general*). For example — according to the terminology we shall use in this paper — bits, bytes, transistors, electricity and computers themselves are *not* in any way essential (because they have nothing to do with the users' problem).

Also, the term "accident" is more commonly used with the connotation of "mishap". Here (as with Brooks) we use it in the more general sense of "something non-essential which is present".

In order to justify these two definitions we start by considering the role of a software development team — namely to produce (using some given language and infrastructure) and maintain a software system which serves the purposes of its users. The complexity in which we are interested is the complexity involved in this task, and it is this which we seek to classify as accidental or essential. We hence see essential complexity as "the complexity with which the team will *have* to be concerned, even in the ideal world".

Note that the "have to" part of this observation is critical — if there is any *possible* way that the team could produce a system that the users will consider correct *without* having to be concerned with a given type of complexity then that complexity is not essential.

Given that in the real world not all *possible* ways are practical, the implication is that any real development *will* need to contend with *some* accidental complexity. The definition does not seek to deny this — merely to identify its secondary nature.

Ultimately (as we shall see below in section 7) our definition is equivalent to saying that what is essential to the team is what the *users* have to be concerned with. This is because in the ideal world we would be using language and infrastructure which would let us express the users' problem directly without having to express anything else — and this is how we arrive at the definitions given above.

The argument might be presented that in the *ideal* world we could just find infrastructure which already solves the users' problem completely. Whilst it is possible to imagine that someone has done the work already, it is not particularly enlightening — it may be best to consider an implicit restriction that the hypothetical language and infrastructure be general purpose and domain-neutral.

One implication of this definition is that if the user doesn't even *know what something is* (e.g. a thread pool or a loop counter — to pick two arbitrary examples) then it cannot possibly be essential by our definition (we are assuming of course — alas possibly with some optimism — that the users do in fact know and understand the problem that they want solved).

Brooks asserts [Bro86] (and others such as Booch agree [Boo91]) that "The complexity of software is an essential property, not an accidental one". This would suggest that the majority (at least) of the complexity that we find in contemporary large systems is of the essential type.

We disagree. Complexity itself is not an inherent (or essential) property of software (it is perfectly possible to write software which is simple and yet is still software), and further, much complexity that we do see in existing software is not essential (to the problem). When it comes to accidental and essential complexity we firmly believe that the former exists and that the goal of software engineering must be both to eliminate as much of it as possible, and to assist with the latter.

Because of this it is vital that we carefully scrutinize *accidental complexity*. We now attempt to classify occurrences of complexity as either accidental or essential.

7 Recommended General Approach

Given that our main recommendations revolve around trying to *avoid* as much accidental complexity as possible, we now need to look at *which* bits of the complexity must be considered accidental and which essential.

We shall answer this by considering exactly what complexity could not *possibly* be avoided even in the ideal world (this is basically how we *define* essential). We then follow this up with a look at just how realistic this ideal world really is before finally giving some recommendations.

7.1 Ideal World

In the ideal world we are not concerned with performance, and our language and infrastructure provide all the general support we desire. It is against this background that we are going to examine *state* and *control*. Specifically, we are going to identify state as *accidental state* if we can omit it in this ideal world, and the same applies to control.

Even in the ideal world we need to start somewhere, and it seems reasonable to assume that we need to start with a set of *informal requirements* from the prospective users.

Our next observation is that because we ultimately need something to *happen* — i.e. we are going to need to have our system processed mechanically (on a computer) — we are going to need *formality*. We are going to need to derive formal requirements from the informal ones.

So, taken together, this means that even in the ideal world we have:

$$\text{Informal requirements} \rightarrow \text{Formal requirements}$$

Note that given that we're aiming for simplicity, it is crucial that the formalisation be done without adding any *accidental* aspects at all. Specifically this means that in the ideal world, formalisation must be done with *no view to execution whatsoever*. The *sole* concern when producing the formal requirements must be to ensure that there is no *relevant*[6] ambiguity in the informal requirements (i.e. that it has no omissions).

So, having produced the formalised requirements, what should the next step be? Given that we are considering the ideal world, it is not unreasonable

[6]We include the word "relevant" here because in many cases there may be many possible acceptable solutions — and in such cases the requirements can be ambiguous in that regard, however that is not considered to be a "relevant" ambiguity, i.e. it does not correspond to an erroneous *omission* from the requirements.

to assume that the next step is simply to *execute* these formal requirements directly on our underlying general purpose infrastructure.[7]

This state of affairs is *absolute* simplicity — it does not seem conceivable that we can do any better than this even in an ideal world.

It is interesting to note that effectively what we have just described is in fact the very *essence* of *declarative programming* — i.e. that you need only specify *what* you require, not *how* it must be achieved.

We now consider the implications of this "ideal" approach for the causes of complexity discussed above.

7.1.1 State in the ideal world

Our main aim for state in the ideal world is to get rid of it — i.e. we are hoping that most state will turn out to be *accidental state*.

We start from the perspective of the users' informal requirements. These will mention data of various kinds — some of which can give rise to *state* — and it is these kinds which we now classify.

All data will either be provided directly to the system (*input*) or *derived*. Additionally, derived data is either *immutable* (if the data is intended only for display) or *mutable* (if explicit reference is made within the requirements to the ability of users to update that data).

All data mentioned in the users' informal requirements is of concern to the users, and is as such *essential*. The fact that all such data is *essential* does *not* however mean that it will all unavoidably correspond to *essential state*. It may well be possible to avoid storing some such data, instead dealing with it in some other essential aspect of the system (such as the *logic*) — this is the case with derived data, as we shall see. In cases where this is possible the data corresponds to *accidental state*.

Input Data

Data which is provided directly (input) will have to have been included in the informal requirements and as such is deemed *essential*. There are basically two cases:

- There is (according to the requirements) a possibility that the system may be required to refer to the data in the future.

- There is no such possibility.

[7]In the presence of *irrelevant* ambiguities this will mean that the infrastructure must *choose* one of the possibilities, or perhaps even provide all possible solutions

In the first case, even in the ideal world, the system must retain the data and as such it corresponds to *essential state*.

In the second case (which will most often happen when the input is designed simply to cause some side-effect) the data need not be maintained at all.

Essential Derived Data — Immutable

Data of this kind can always be re-derived (from the input data — i.e. from the *essential state*) whenever required. As a result we do *not* need to store it in the ideal world (we just re-derive it when it is required) and it is clearly *accidental state*.

Essential Derived Data — Mutable

As with immutable essential derived data, this can be excluded (and the data re-derived on demand) and hence corresponds to *accidental state*.

Mutability of derived data makes sense only where the function (logic) used to derive the data has an inverse (otherwise — given its mutability — the data cannot be considered *derived* on an ongoing basis, and it is effectively *input*). An inverse often exists where the derived data represents simple restructurings of the input data. In this situation modifications to the data can simply be treated identically to the corresponding modifications to the existing *essential state*.

Accidental Derived Data

State which is *derived* but *not* in the users' requirements is also *accidental state*. Consider the following imperative pseudo-code:

```
procedure int doCalculation(int y)
  // 'subsidaryCalcCache' is declared and initialized
  // elsewhere in the code
  if (subsidaryCalcCache.contains(y) == false) {
    subsidaryCalcCache.y := slowSubsidaryCalculation(y)
  }
  return 3 * (4 + subsidaryCalcCache.y)
```

The above use of state in the `doCalculation` procedure seems to be unnecessary (in the ideal world), and hence of the *accidental* variety. We

Data Essentiality	Data Type	Data Mutability	Classification
Essential	Input	-	Essential State
Essential	Derived	Immutable	Accidental State
Essential	Derived	Mutable	Accidental State
Accidental	Derived	-	Accidental State

Table 1: Data and State

cannot actually be sure without knowing whether and how the `subsidary-CalcCache` is used elsewhere in the program, but for this example we shall assume that there are no other uses aside from initialization. The above procedure is thus equivalent to:

```
procedure int doCalculation(int y)
  return 3 * (4 + slowSubsidaryCalculation(y))
```

It is almost certain that this use of state would *not* have been part of the users' informal requirements. It is also *derived*. Hence, it is quite clear that we can eliminate it completely from our ideal world, and that hence it *is* accidental.

Summary — State in the ideal world

For our ideal approach to state, we largely follow the example of functional programming which shows how mutable state can be avoided. We need to remember though that:

1. even in the ideal world we *are* going to have *some* essential state — as we have just established

2. pure functional programs can effectively simulate accidental state in the same way that they can simulate essential state (using techniques such as the one discussed above in section 5.2.3) — we obviously want to avoid this in the ideal world.

The data type classifications are summarized in Table 1. Wherever the table shows data as corresponding to *accidental state* it means that it can be excluded from the ideal world (by re-deriving the data as required).

The obvious implication of the above is that there are large amounts of *accidental state* in typical systems. In fact, it is our belief that the vast majority of state (as encountered in typical contemporary systems) simply isn't needed (in this ideal world). Because of this, and the huge complexity which state can cause, the ideal world removes *all* non-essential state. There is no other state at all. No caches, no stores of derived calculations of any kind. One effect of this is that *all* the state in the system is *visible* to the user of (or person testing) the system (because inputs can reasonably be expected to be visible in ways which internal cached state normally is not).

7.1.2 Control in the ideal world

Whereas we have seen that some state *is* essential, control generally can be completely omitted from the ideal world and as such is considered entirely *accidental*. It typically won't be mentioned in the informal requirements and hence should not appear in the formal requirements (because these are derived with *no view to execution*).

What do we mean by this? Clearly if the program is ever to run, *some* control will be needed somewhere because things will have to happen in some order — but this should no more be our concern than the fact that the chances are some electricity will be needed somewhere. The important thing is that we (as developers of the system) should not have to worry about the control flow in the system. Specifically the *results* of the system should be independent of the actual control mechanism which is finally used.

These are precisely the lessons which logic programming teaches us, and because of this we would like to take the lead for our ideal approach to control from logic programming which shows that control can be separated completely.

It is worth noting that because typically the informal requirements will not mention concurrency, that too is normally of an accidental nature. In an ideal world we can assume that finite (stateless) computations take zero time[8] and as such it is immaterial to a user whether they happen in sequence or in parallel.

7.1.3 Summary

In the ideal world we have been able to avoid large amounts of complexity — both state and control. As a result, it is clear that a lot of complexity is *accidental*. This gives us hope that it may be possible to significantly reduce the complexity of *real* large systems. The question is — how close is it possible to get to the ideal world in the real one?

[8]this assumption is generally known as the "synchrony hypothesis"

7.2 Theoretical and Practical Limitations

The real world is not of course ideal. In this section we examine a few of the assumptions made in the section 7.1 and see where they break down.

As already noted, our vision of an ideal world is similar in many ways to the vision of *declarative programming* that lies behind functional and logic programming.

Unfortunately we have seen that functional and logic programming ultimately had to confront both state and control. We should note that the reasons for having to confront each are slightly different. State is required simply because most systems do have some state as part of their true essence. Control generally *is* accidental (the users normally are not concerned about it at all) but the ability to restrict and influence it is often required from a practical point of view. Additionally practical (e.g. efficiency) concerns will often dictate the use of some accidental state.

These observations give some indication of where we can expect to encounter difficulties.

7.2.1 Formal Specification Languages

First of all, we want to consider two problems (one of a theoretical kind, the other practical) that arise in connection with the ideal-world *formal requirements*.

In that section we discussed the need for *formal requirements* derived directly from the *informal requirements*. We observed that in the ideal world we would like to be able to execute the formal requirements without first having to translate them into some other language.

The phrase "formal requirements" is basically synonymous with "formal specification", so what effectively we're saying would be ideal are *executable specifications*. Indeed both the declarative programming paradigms discussed above (functional programming and logic programming) have been proposed as approaches for executable specifications.

Before we consider the problems with executing them, we want to comment that *the way in which* the ideal world formal specifications were derived — *directly* from the users' informal requirements — was critical. Formal specifications can be derived in various other ways (some of which risk the introduction of accidental complexity), and can be of various different kinds.

Traditionally formal specification has been categorized into two main camps:

Property-based approaches focus (in a declarative manner) on *what* is required rather than *how* the requirements should be achieved. These approaches include the *algebraic* (*equational axiomatic semantics*) approaches such as Larch and OBJ.

Model-based (or State-based) approaches construct a potential model for the system (often a stateful model) and specify how that model must behave. These approaches (which include Z and VDM) can hence be used to specify how a stateful, imperative language solution must behave to satisfy the requirements. (We discussed the weaknesses of stateful imperative languages in section 5).

The first problem that we want to discuss in this section is the more theoretical one. Arguments (which focus more on the model-based approaches) have been put forward *against* the concept of executable specifications [HJ89]. The main objection is that requiring a specification language to be executable can directly restrict its expressiveness (for example when specifying requirements for a variable x it may be desirable to assert something like $\neg\exists y|f(y,x)$ which clearly has no direct operational interpretation).

In response to this objection, we would say firstly that in our experience a requirement for this kind of expressivity does not seem to be common in many problem domains. Secondly it would seem sensible that where such specifications *do* occur they should be maintained in their natural form but supplemented with a *separate* operational component. Indeed in this situation it would not seem too unreasonable to consider the required operational component to be accidental in nature (of course the reality is that in cases like this the boundary between what is accidental and essential, what is reasonable to hope for in an "ideal" world, becomes less clear). Some specification languages address this issue by having an executable subset.

Finally, it is the *property-based* approaches that seem to have the greatest similarity to what we have in mind when we talk about *executable specifications* in the ideal world. It certainly *is* possible to execute algebraic specifications — deriving an operational semantics by choosing a direction for each of the equational axioms.[9]

In summary, the first problem is that consideration of specification languages highlights the (theoretically) fuzzy boundary between what is essential and what is accidental — specifically it challenges the validity of our definition of *essential* (which we identified closely with requirements from the *users*) by observing that it is possible to specify things which are *not*

[9]Care must be taken that the resulting reduction rules are *confluent* and *terminating*.

directly executable. For the reasons given above (and in section 6) we think that — from the practical point of view — our definition is still viable, import and justified.

The second problem is of a more practical nature — namely that even when specifications *are* directly executable, this can be impractical for efficiency reasons. Our response to this is that whilst it is undoubtedly true, we believe that it is very important (for understanding and hence for avoiding complexity) not to lose the distinction we have defined between what is *accidental* and *essential*. As a result, this means that we will *require* some accidental components as we shall see in section 7.2.3.

7.2.2 Ease of Expression

There is one final practical problem that we want to consider — even though we believe it is fairly rare in most application domains. In section 7.1.1 we argued that immutable, derived data would correspond to *accidental state* and could be omitted (because the *logic* of the system could always be used to derive the data on-demand).

Whilst this is true, there are occasionally situations where the ideal world approach (of having no accidental state, and using on-demand derivation) does not give rise to the most natural modelling of the problem.

One possible situation of this kind is for derived data which is dependent upon *both* a whole series of user inputs over time, *and* its own previous values. In such cases it can be advantageous[10] to *maintain* the *accidental state* even in the ideal world.

An example of this would be the derived data representing the position state of a computer-controlled opponent in an interactive game — it is at all times *derivable* by a function of both all prior user movements and the initial starting positions,[11] but this is not the way it is most naturally expressed.

7.2.3 Required Accidental Complexity

We have seen two possible reasons why in practice — even with optimal language and infrastructure — we may *require* complexity which strictly is *accidental*. These reasons are:

Performance making use of accidental state and control can be required for efficiency — as we saw in the second problem of section 7.2.1.

[10]because it can make the *logic* easier to express — as we shall see in section 7.3.2

[11]We are implicitly considering *time* as an additional input.

Ease of Expression making use of accidental state can be the most natural way to express logic in some cases — as we saw in section 7.2.2.

Of the two, we believe that *performance* will be the most common.

It is of course vital to be aware that as soon as we re-introduce this accidental complexity, we are again becoming exposed to the dangers discussed in sections 4.1 and 4.2. Specifically we can see that if we add in *accidental state* which has to be managed explicitly by the logic of the system, then we become at risk of the possibility of the system entering an *inconsistent state* (or "bad state") due to errors in that explicit logic. This is a very serious concern, and is one that we address in our recommendations below.

7.3 Recommendations

We believe that — despite the existence of required accidental complexity — it *is* possible to retain most of the simplicity of the ideal world (section 7.1) in the real one. We now look at how this might be achievable.

Our recommendations for dealing with complexity (as exemplified by both state and control) can be summed up as:

- Avoid
- Separate

Specifically the overriding aim must be to *avoid* state and control where they are not absolutely and truly essential.

The recommendation of avoidance is however tempered by the acknowledgement that there will sometimes be complexity that either is truly essential (section 7.1.1) or, whilst not *truly* essential, is useful from a practical point of view (section 7.2.3). Such complexity must be separated out from the rest of the system — and this gives us our second recommendation.

There is nothing particularly profound in these recommendations, but they are worth stating because they are emphatically *not* the way most software is developed today. It is the fact that current established practice does *not* use these as central overriding principles for software development that leads directly to the complexity that we see everywhere, and as already argued, it is that complexity which leads to the software crisis[12].

In addition to not being profound, the principles behind these recommendations are not really new. In fact, in a classic 1979 paper Kowalski

[12]There *is* some limited similarity between our goal of "Separate" and the goal of *separation of concerns* as promoted by proponents of Aspect Oriented Programming — but as we shall see in section 7.3.2, exactly what is meant by separation is critical.

(co-inventor of Prolog) argued in exactly this direction [Kow79]. The title of his paper was the equation:

$$\text{“}Algorithm = Logic + Control\text{”}$$

...and this separation that he advocated is close to the heart of what we're recommending.

7.3.1 Required Accidental Complexity

In section 7.2.3 we noted two possible reasons for requiring accidental complexity (even in the presence of optimal language and infrastructure). We now consider the most appropriate way of handling each.

Performance

We have seen that there are many serious risks which arise from accidental complexity — particularly when introduced in an undisciplined manner. To mitigate these risks we take two defensive measures.

The first is with regard to the risks of explicit management of accidental state (which we have argued is actually the *majority* of state). The recommendation here is that we completely *avoid* explicit management of the accidental state — instead we should restrict ourselves to simply *declaring* what accidental state should be used, and leave it to a completely separate infrastructure (on which our system will eventually run) to maintain. This is reasonable because the infrastructure can make use of the (separate) system logic which specifies how accidental data must be derived.

By doing this we eliminate any risk of state inconsistency (bugs in the infrastructure aside of course). Indeed, as we shall see (in section 7.3.2), from the point of view of the *logic* of the system, we can effectively forget that the *accidental state* even exists. More specific examples of this approach are given in the second half of this paper.

The other defensive action we take is “Separate”. We examine separation after first looking at the other possible reason for requiring accidental complexity.

Ease of Expression

This problem (see section 7.2.2) fundamentally arises when derived (i.e. *accidental*) state offers the most natural way to express parts of the logic of the system.

Complexity	Type	Recommendation
Essential Logic		Separate
Essential Complexity	State	Separate
Accidental Useful Complexity	State / Control	Separate
Accidental Useless Complexity	State / Control	Avoid

Table 2: Types of complexity within a system

The difficulty then arises that this requirement (to *use* the *accidental state* in a fairly direct manner inside the system logic) clashes with the goal of *separation* that we have just discussed. This very *separation* is *critical* when it comes to avoiding complexity, so we do not want to sacrifice it for this (probably fairly rare) situation.

Instead what we recommend is that, in cases where it really is the only natural thing to do, we should *pretend* that the *accidental state* is really *essential state* for the purposes of the *separation* discussed below. One straightforward way to do this is to make use of an external component which observes the derived data in question and creates the illusion of the *user* typing that same (derived, *accidental*) data back in as input data (we touch on this issue again in section 9.1.4).

7.3.2 Separation and the relationship between the components

In the above we deliberately glossed over *exactly* what we meant by our second recommendation: "Separate". This is because it actually encompasses two things.

The first thing that we're doing is to advocate separating out *all* complexity of any kind from the pure logic of the system (which — having nothing to do with either state or control — we're not really considering part of the complexity). This could be referred to as the *logic / state* split (although of course state is just one aspect of complexity — albeit the main one).

The second is that we're further dividing the complexity which we do retain into accidental and essential.This could be referred to as the *accidental / essential* split. These two splits can more clearly be seen by considering the Table 2. (N.B. We do not consider there to be any essential control).

The essential bits correspond to the requirements in the ideal world of section 7.1 — i.e. we are recommending that the formal requirements adopt the *logic / state* split.

The top three rows of the table correspond to components which we expect to exist in most practical systems (some systems may not actually require any essential state, but we include it here for generality). i.e. These are the three things which will need to be specified (in terms of a given underlying language and infrastructure) by the development team.

"Separate" is basically advocating clean distinction between all three of these components. It is additionally advocating a split between the state and control components of the "Useful" Accidental Complexity — but this distinction is less important than the others.

One implication of this overall structure is that the system (essential + accidental but useful) should still function *completely correctly* if the "accidental but useful" bits are removed (leaving only the two *essential* components) — albeit possibly unacceptably slowly. As Kowalski (who — writing in a Prolog-context — was not really considering any essential state) says:

> "The logic component determines the meaning . . . whereas the control component only affects its efficiency".

A consequence of *separation* is that the separately specified components will each be of a *very* different nature, and as a result it may be ideal to use *different languages* for each. These languages would each be oriented (i.e. *restricted*) to their specific goal — there is no sense in having control specification primitives in a language for specifying state. This notion of *restricting the power* of the individual languages is an important one — the weaker the language, the more simple it is to reason about. This has something in common with the ideas behind "Domain Specific Languages" — one exception being that the domains in question are of a fairly abstract nature and combine to form a general-purpose platform.

The vital importance of separation comes simply from the fact that it is separation that allows us to "*restrict the power*" of each of the components independently. The restricted power of the respective languages with which each component is expressed facilitates reasoning about them individually. The very fact that the three are separated from each other facilitates reasoning about them as a whole (e.g. you do not have to think about accidental state at all when you are working on the essential logic of your system[13]).

Figure 1 shows the same three expected components of a system in a different way (compare with Table 2). Each box in the diagram corresponds to some aspect of the system which will *need to be specified* by the development team. Specifically, it will be necessary to specify what the essential

[13]indeed it should be perfectly possible for different users of the same essential system to employ different accidental components — each designed for their particular needs

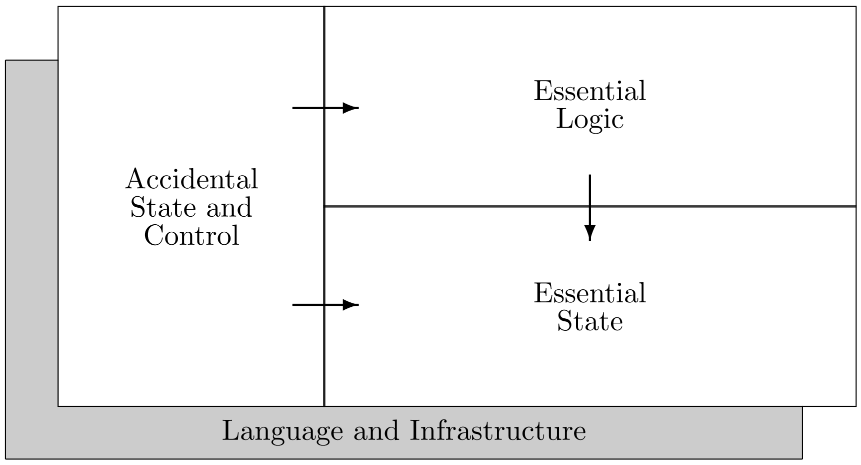


Figure 1: Recommended Architecture (arrows show static references)

state can be, what must always be logically true, and finally what accidental use can be made of state and control (typically for performance reasons).

The differing nature of what is specified by each of the components leads naturally to certain relationships between them, to *restrictions* on the ways in which they can or cannot refer to each other. These restrictions are absolute, and because of this provide a huge aid to understanding the different components of the system independently.

Essential State This can be seen as the foundation of the system. The specification of the required state is completely self-contained — it can *make no reference* to *either* of the other parts which must be specified. One implication of this is that changes to the essential state specification itself may require changes in both the other specifications, but changes in either of the other specifications may never require changes to the specification of essential state.

Essential Logic This is in some ways the "heart" of the system — it expresses what is sometimes termed the "business" logic. This logic expresses — in terms of the state — what must be true. It does *not* say anything about how, when, or why the state might change dynamically — indeed it wouldn't make *sense* for the logic to be able to change the state in any way.

Changes to the essential state specification may require changes to the logic specification, and changes to the logic specification may require changes to the specification for accidental state and control. The logic specification will *make no reference* to *any* part of the accidental specification. Changes in the accidental specification can hence never require any change to the essential logic.

Accidental State and Control This (by virtue of its accidental nature) is conceptually the least important part of the system. Changes to it can *never* affect the other specifications (because neither of them make any reference to any part of it), but changes to *either* of the others may require changes here.

Together the goals of *avoid* and *separate* give us reason to hope that we may well be able to retain much of the simplicity of the ideal world in the real one.

7.4 Summary

This first part of the paper has done two main things. It has given arguments for the overriding danger of complexity, and it has given some hope that much of the complexity may be avoided or controlled.

The key difference between what we are advocating and existing approaches (as embodied by the various styles of programming language) is a high level *separation* into three components — each specified in a different language[14]. It is this *separation* which allows us to *restrict* the power of each individual component, and it is this use of *restricted* languages which is vital in making the overall system easier to comprehend (as we argued in section 4.4 — *power corrupts*).

Doing this separation when building a system may not be easy, but we believe that for any large system it will be *significantly* less difficult than dealing with the complexity that arises otherwise.

It is hard to overstate the dangers of complexity. If it is not controlled it spreads. The *only* way to escape this risk is to place the goals of *avoid* and *separate* at the top of the design objectives for a system. It is not sufficient simply to pay heed to these two objectives — it is crucial that they be the *overriding* consideration. This is because complexity breeds complexity and one or two early "compromises" can spell complexity disaster in the long run.

It is worth noting in particular the risks of "designing for performance". The dangers of "premature optimisation" are as real as ever — there can be no comparison between the difficulty of improving the performance of a

[14]or different subsets of the same language, *provided* it is possible to *forcibly restrict* each component to the relevant subset.

slow system designed for simplicity and that of removing complexity from a complex system which was designed to be fast (and quite possibly isn't even that because of myriad inefficiencies hiding within its complexity).

In the second half of this paper we shall consider a possible approach based on these recommendations.

8 The Relational Model

The relational model [Cod70] has — despite its origins — nothing *intrinsically* to do with databases. Rather it is an elegant approach to *structuring* data, a means for *manipulating* such data, and a mechanism for maintaining *integrity* and consistency of state. These features are applicable to state and data in any context.

In addition to these three broad areas [Cod79, section 2.1], [Dat04, p109], a fourth strength of the relational model is its insistence on a clear separation between the logical and physical layers of the system. This means that the concerns of designing a logical model (minimizing the complexity) are addressed separately from the concerns of designing an efficient physical storage model and mapping between that and the logical model[15]. This principle is called *data independence* and is a crucial part of the relational model [Cod70, section 1.1].

We see the relational model as having the following four aspects:

Structure the use of *relations* as the means for representing all data

Manipulation a means to specify derived data

Integrity a means to specify certain inviolable restrictions on the data

Data Independence a clear separation is enforced between the logical data and its physical representation

We will look briefly at each of these aspects. [Dat04] provides a more thorough overview of the relational model.

As a final comment, it is widely recognised that SQL (of any version) — despite its widespread use — is *not* an accurate reflection of the relational model [Cod90, p371, Serious flaws in SQL], [Dat04, p xxiv] so the reader is warned against equating the two.

[15]Unfortunately most contemporary DBMSs are somewhat limited in the degree of flexibility permitted by the physical/logical mapping. This has the unhappy result that physical performance concerns can invade the logical design even though *avoiding* exactly this was one of Codd's most important original goals.

8.1 Structure

8.1.1 Relations

As mentioned above, relations provide the sole means for structuring data in the relational model. A relation is best seen as a homogeneous *set of records*, each record itself consisting of a heterogeneous set of uniquely named *attributes* (this is slightly different from the general mathematical definition of a relation as a set of tuples whose components are identified by position rather than name).

Implications of this definition include the fact that — by virtue of being a set — a relation can contain no duplicates, and it has no ordering. Both of these restrictions are in contrast with the common usage of the word *table* which can obviously contain duplicate rows (and column names), and — by virtue of being a visual entity on a page — inevitably has both an ordering of its rows and of its columns.

Relations can be of the following kinds:

Base Relations are those which are stored directly

Derived Relations (also known as *Views*) are those which are defined in terms of other relations (base or derived) — see section 8.2

Following Date [Dat04] it is useful to think of a relation as being a single (albeit compound) *value*, and to consider any mutable state not as a "mutable relation" but rather as a *variable* which at any time can contain a particular relation *value*. Date calls these variables *relation variables* or *relvars*, leading to the terms *base relvar* and *derived relvar*, and we shall use this terminology later. (Note however that our definition of relation is slightly different from his in that — following standard static typing practice — we do not consider the type to be part of the value).

8.1.2 Structuring benefits of Relations — Access path independence

The idea of structuring data using relations is appealing because no *subjective,* up-front decisions need to be made about the *access paths* that will later be used to query and process the data.

To understand what is meant by *access path*, let us consider a simple example. Suppose we are trying to represent information about employees and the departments in which they work. A system in which choosing the structure for the data involves setting up "routes" between data instances (such as *from* a particular employee *to* a particular department) is *access path dependent*.

The two main data structuring approaches which preceded the relational model (the network and hierarchical models) were both access path dependent in this way. For example, in the hierarchical model a subjective choice would be forced early on as to whether departments would form the top level (with each department "containing" its employees) or the other way round (with employees "containing" their departments). The choice made would impact all future use of the data. If the first alternative was selected, then users of the data would find it easy to retrieve all employees within a given department (following the access path), but they would find it harder to retrieve the department of a given employee (and would have to use some other technique corresponding to a search of all departments). If the second alternative was selected then the problem was simply reversed.

The network model alleviated the problem to some degree by allowing multiple access paths between data instances (so the choice could be made to provide both an access path from department to employee and an access path from employee to department). The problem of course is that it is impossible to predict in advance what all the future required access paths will be, and because of this there will always be a disparity between:

Primary retrieval requirements which were foreseen, and can be satisfied simply by following the provided access paths

Secondary retrieval requirements which were either unforeseen, or at least not specially supported, and hence can only be satisfied by some alternative mechanism such as search

The ability of the relational model to *avoid* access paths completely was one of the primary reasons for its success over the network and hierarchical models.

It is also interesting to consider briefly what is involved when taking an object-oriented (OOP) approach to our example. We can choose between the following options:

- Give Employee objects a reference to their Department
- Give Department objects a set (or array) of references to their Employees
- Both of the above

If we choose the third option, then we at best expose ourselves to extra work in maintaining the redundant references, and at worst expose ourselves to bugs.

There are disturbing similarities between the data structuring approaches of OOP and XML on the one hand and the network and hierarchical models on the other.

A final advantage of using relations for the structure — in contrast with approaches such as Chen's ER-modelling [Che76] — is that no distinction is made between *entities* and *relationships*. (Using such a distinction can be problematic because whether something is an *entity* or a *relationship* can be a very subjective question).

8.2 Manipulation

Codd introduced two different mechanisms for expressing the manipulation aspects of the relational model — the relational calculus and the relational algebra. They are formally equivalent (in that expressions in each can be converted into equivalent expressions in the other), and we shall only consider the algebra.

The relational algebra (which is now normally considered in a slightly different form from the one used originally by Codd) consists of the following eight operations:

Restrict is a unary operation which allows the selection of a subset of the records in a relation according to some desired criteria

Project is a unary operation which creates a new relation corresponding to the old relation with various attributes removed from the records

Product is a binary operation corresponding to the cartesian product of mathematics

Union is a binary operation which creates a relation consisting of all records in either argument relation

Intersection is a binary operation which creates a relation consisting of all records in both argument relations

Difference is a binary operation which creates a relation consisting of all records in the first but not the second argument relation

Join is a binary operation which constructs all possible records that result from matching identical attributes of the records of the argument relations

Divide is a ternary operation which returns all records of the first argument which occur in the second argument associated with *each* record of the third argument

One significant benefit of this manipulation language (aside from its simplicity) is that it has the property of *closure* — that all operands and results are of the same kind (relations) — hence the operations can be nested in arbitrary ways (indeed this property is inherent in any single-sorted algebra).

8.3 Integrity

Integrity in the relational model is maintained simply by specifying — in a purely declarative way — a set of constraints which must hold at all times.

Any infrastructure implementing the relational model must ensure that these constraints always hold — specifically attempts to modify the state which would result in violation of the constraints must be either rejected outright or restricted to operate within the bounds of the constraints.

The most common types of constraint are those identifying *candidate* or *primary* keys and *foreign* keys. Constraints may in fact be arbitrarily complex, involve multiple relations, and be constructed from either the relational calculus or the relational algebra.

Finally, many commercially available DBMSs provide *imperative* mechanisms such as triggers for maintaining integrity — such mechanisms suffer from control-flow concerns (see section 4.2) and are *not* considered to be part of the relational model.

8.4 Data Independence

Data independence is the principle of separating the logical model from the physical storage representation, and was one of the original motivations for the relational model.

It is interesting to note that the *data independence* principle is in fact a very close parallel to the *accidental / essential* split recommended above (section 7.3.2). This is one of several reasons that motivate the adoption of the relational model in Functional Relational Programming (see section 9 below).

8.5 Extensions

The relational algebra — whilst flexible — is a restrictive language in computational terms (it is not Turing-complete) and is normally augmented in various ways when used in practice. Common extensions include:

General computation capabilities for example simple arithmetical operations, possibly along with user-defined computations.

Aggregate operators such as `MAX`, `MIN`, `COUNT`, `SUM`, etc.

Grouping and Summarization capabilities to allow for easy application of aggregate operations to relations

Renaming capabilities the ability to generate derived relations by changing attribute names

9 Functional Relational Programming

The approach of functional relational programming (FRP[16]) derives its name from the fact that the *essential* components of the system (the logic and the essential state) are based upon functional programming and the relational model (see Figure 2).

FRP is currently a purely hypothetical[17] approach to system architecture that has not in any way been proven in practice. It is however based firmly on principles from other areas (the relational model, functional and logic programming) which *have* been widely proven.

In FRP all *essential state* takes the form of relations, and the *essential logic* is expressed using relational algebra extended with (pure) user-defined[18] functions.

The primary, overriding goal behind the FRP architecture (and indeed this whole paper) is of course *elimination of complexity.*

[16]Not to be confused with functional *reactive* programming [EH97] which does in fact have some similarities to this approach, but has no intrinsic focus on relations or the relational model

[17]Aside from token experimental implementations of FRP infrastructures created by the authors.

[18]By *user-defined* we mean *specific to this particular FRP system* (as opposed to pre-provided by an underlying infrastructure).

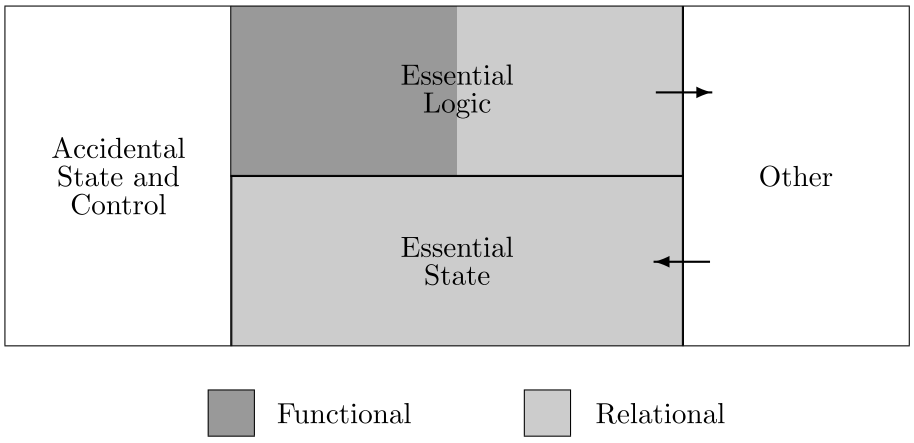


Figure 2: The components of an FRP system (infrastructure not shown, arrows show dynamic data flow)

9.1 Architecture

We describe the architecture of an FRP system by first looking at *what must be specified* for each of the components when constructing a system in this manner. Then we look at what infrastructure needs to be available in order to be able to construct systems in this fashion.

In accordance with the first half of this paper, FRP recommends that the system be constructed from *separate* specifications for each of the following components:

Essential State A Relational definition of the stateful components of the system

Essential Logic Derived-relation definitions, integrity constraints and (pure) functions

Accidental State and Control A declarative specification of a set of performance optimizations for the system

Other A specification of the required interfaces to the outside world (user and system interfaces)

Speaking somewhat loosely, the first two components can be seen as corresponding to “State” and “Behaviour” respectively, whilst the third concentrates on “Performance”. In contrast with the object-oriented approach, FRP emphasises a clear *separation* of state and behaviour[19].

9.1.1 Essential State (“State”)

This component consists solely of a specification of the essential state for the system in terms of base relvars[20] (in FRP *all* state is stored solely in terms of relations — there are no exceptions to this). Specifically it is the *names* and *types* of the base relvars that are specified here, not their actual contents. The contents of the relvars (i.e. the relations themselves) will of course be crucial when the system is used, but here we are discussing only the *static* structure of the system.

In accordance with section 7.1.1, FRP strongly encourages that data be treated as essential state *only*[21] when it has been *input directly by a user*.[22]

9.1.2 Essential Logic (“Behaviour”)

The essential logic comprises both functional and algebraic (relational) parts. The main (in the sense that it provides the overall structure for the component) part is the relational one, and consists of derived-relvar names and definitions. These definitions consist of applications of the relational algebra operators to other relvars (both derived relvars and the base relvars which make up the essential state).

In addition to the relational algebra, the definitions can make use of an arbitrary set of pure user-defined functions which make up the functional part of the essential logic.

Finally (in accordance with the standard relational model) the logic specifies a set of *integrity constraints* — boolean expressions which must hold at all times. (These can include everything from simple foreign key constraints to complicated multiple-relvar requirements making use of user-defined functions). Any attempt to modify the essential state (see section 9.1.4) will always be subject to these integrity constraints.

Much of the standard theory of relational database design can obviously be used as a guide for the relational parts of these two *essential* components. For example, normalization of the relvars will allow consistent updates (see

[19]Equally, traditional OOP pays little attention to the accidental / essential split which was also discussed in section 7.3.2.

[20]see section 8.1.1 for a definition of this term.

[21]aside from the *ease of expression* issue discussed in section 9.1.4.

[22]Other systems connected electronically are considered equivalent to users inputting data for these purposes.

section 9.1.4) to the state to be more easily expressed. Note that — assuming no integrity constraints have been accidentally omitted — normalization does *not* in any way help to preserve the integrity of our relvars — that is after all what the constraints do, and if the constraints are not violated (and the infrastructure must always ensure this) then the relvars have integrity by definition. What *is* true is that more normalized designs do impose *implicit* restrictions, and this can reduce the number of (explicit) integrity constraints that must be specified.

Having raised the issue of design, it is vital to note that absolutely *NO* consideration is paid to performance issues of any kind here. Concepts such as "denormalization for performance" make absolutely no sense in this context because they contain the implicit assumption that the physical storage used will closely mirror the relational structure which is being specified here. This is absolutely not the case (it is only the *accidental state and control* component — see below — which is concerned with efficiency of storage structures).

9.1.3 Accidental State and Control ("Performance")

This component fundamentally consists of a series of isolated (in the sense that they cannot refer to each other in any way) performance "hints". These hints — which should be declarative in nature — are intended to provide guidance to the infrastructure responsible for running the FRP system.

On the *state* side, this component is concerned with both *accidental state* itself and *accidental aspects of* state. Firstly, it provides a means to specify *what state* (of the accidental variety) should exist. Secondly it provides (if desired) a means to specify *what physical storage mechanisms* should be used for storing state (of both kinds) — i.e. the *accidental* aspects of storage. This second aspect is the *flexible mapping* providing physical / logical *data independence* as required by the relational model (section 8).

An example of the first kind of state-related hint might be a simple directive that a particular derived-relvar should actually be stored (rather than continually recalculated), so that it is always quickly available.

An example of the second kind of state-related hint might be that an infrequently used subset of the attributes of a particular relvar (either derived or base) should be stored separately for performance reasons. The use of indices or other custom storage strategies would also be examples of this second kind of state-related hint. The exact types of hint available here will depend entirely on what is provided by the underlying infrastructure.

On the *control* side, recommendations for parallel evaluation of derived-relvars might be given. Also, declarative hints could be given about whether the derived relvars should be computed eagerly (as soon as the essential state changes), lazily (when the infrastructure is forced to provide them), or some combination of different policies for different relvars.

All hints are incapable of referring to each other, but do refer to the relevant (essential, base and derived) relvars by name.

9.1.4 Other (Interfacing)

The primary consideration not addressed by the above is that of *interfacing with the outside world*.

Specifically, all input must be converted into relational assignments (which replace the old relvar values in the *essential state* with new ones), and all output (and side-effects) must be driven from changes to the values of relvars (primarily derived relvars).

The exact nature of this task is likely to be highly application-dependent, but we can say that there will probably be a requirement for a series of *feeder* (or *input*) and *observer* (or *output*) components. These may well be defined, at least partially, in a traditional, imperative way if custom interfacing is required. There will be cases when it is necessary for a given interfacing component to act in both capacities (if for example a message must be observed and sent to another system, then a response received, recorded and fed back in).

The expectation is that all of these components will be of a *minimal* nature — performing only the necessary translations to and from relations.

Feeders

Feeders are components which convert input into relational assignments — i.e. cause changes to the *essential state*. In order to be able to cause these state changes, *feeders* will need to specify them in some form of *state manipulation language* provided by the infrastructure. At a minimum, this language can consist of just a relational assignment command which assigns to a relvar a whole new relation value in its entirety:

```
relvar := newRelationValue
```

The infrastructure which eventually runs the FRP system will ensure that the command respects the integrity constraints[23] — either by rejecting

[23]In fact one implication of this is that it is in fact necessary for the assignment command to support multiple *simultaneous* assignment of several distinct relation values to several

non-conformant commands, or possibly in some cases by *modifying* them to ensure conformance.

Observers

Observers are components which generate output in response to changes which they observe in the values of the (derived) relvars. At a minimum, *observers* will only need to specify the *name* of the relvar which they wish to observe. The infrastructure which runs the system will ensure that the observer is invoked (with the new relation value) whenever it changes. In this way *observers* act both as what are sometimes called *live-queries* and also as *triggers*.

Despite this the intention is *not* for *observers* to be used as a substitute for true integrity constraints. Specifically, hybrid *feeders / observers* should *not* act as triggers which directly update the *essential state* (this would by definition be creating *derived* and hence *accidental* state). The only (occasional) exceptions to this should be of the *ease of expression* kind discussed in sections 7.2.2 and 7.3.1.

Summary

The most complicated scenario when interfacing the core relational system with the outside world is likely to come when the interfacing requires highly structured input or output (this is most likely to occur when interfacing with other systems rather than with people).

In this situation, the *feeders* or *observers* are forced to convert between structured data and flat relations[24].

9.1.5 Infrastructure

In several places above we have referred to the "infrastructure which runs the FRP system". The *FRP system* is the specification — comprising of the four components above, the *infrastructure* is what is needed to execute this specification (by interpretation, compilation or some mixture).

distinct relvars — this is to avoid temporary inconsistencies which could otherwise occur with integrity constraints that involved multiple relvars.

[24]Some systems — for example the Kleisli system used in bio-informatics [Won00] — seek to avoid this conversion by providing support for more complex structures such as nested relations. We believe that the simplicity gained from having flat relations throughout the system is worth the effort sometimes involved at the system edges (section 9.2.4 describes some of the rationale behind this).

The different components of an FRP system lead to different requirements on the infrastructure which is going to support them.

Infrastructure for Essential State

1. some means of storing and retrieving data in the form of relations assigned to named relvars

2. a state manipulation language which allows the stored relvars to be updated (within the bounds of the integrity constraints)

3. optionally (depending on the exact range of FRP systems which the infrastructure is intended to support) secondary (e.g. disk-based) storage in addition to the primary (in memory) storage

4. a base set of generally useful types (typically integer, boolean, string, date etc)

Infrastructure for Essential Logic

1. a means to evaluate relational expressions

2. a base set of generally useful functions (for things such as basic arithmetic etc)

3. a language to allow specification (and evaluation) of the user-defined functions in the FRP system. (It does not have to be a functional language, but the infrastructure must only allow it to be used in a functional way)

4. optionally a means of type inference (this will also require a mechanism for declaring the types of the user-defined functions in the FRP system)

5. a means to express and enforce integrity constraints

Infrastructure for Accidental State and Control

1. a means to specify which *derived* relvars should actually be stored, along with the ability to store such relvars *and ensure that the stored values are accurately up-to-date at all times*

2. a *flexible* means to specify physical storage mechanisms to be used by a relvar. This is a vital part of the infrastructure — without it the infrastructure must store relvars in a way which closely mirrors their

logical (essential) definitions, and that inevitably leads to accidental (performance) concerns corrupting the essential parts of the system. Specifically, procedures such as *normalization* or "*de-normalization*" at the logical (essential) level should have no intrinsic performance implications because of the presence of this mechanism.

Infrastructure for Feeders and Observers

The minimum requirement on the infrastructure (specifically on the state manipulation language) from *feeders* is for it to be able to process relational assignment commands (containing complete new relation values) and reject them if necessary. Practical extensions that could be useful include the ability to accept commands which specify new relvar values in terms of their previous values — typically in the form of INSERT / UPDATE / DELETE commands.

The minimum requirement on the infrastructure from *observers* is for it to be able to supply the new value of a relvar whenever it changes. Practical extensions that could be useful are the ability to provide deltas, throttling and coalescing capabilities (if the *observers* are viewed as *live query*-handlers, then these extensions represent potential *query meta-data*).

Another possible extension is the ability to observe general relational expressions rather than just relvars from the essential logic (this is not a significant extension as it is basically equivalent to a short-term addition to the essential logic's set of derived relvars — the only difference being that the expression in question would be anonymous).

Finally, the ability to access arbitrary *historical* relvar values would obviously be a useful extension in some scenarios.

Summary

If a system is to be based upon the FRP architecture it will be necessary either to obtain an FRP infrastructure from a third party, or to develop one with existing tools and techniques.

Currently of course no real FRP infrastructures exist and so at present the choice is clear. However, even in the presence of third party infrastructures there may in fact be compelling reasons for large systems to adopt the custom route. Firstly, the effort involved in doing so need not be huge[25], and

[25] A prototype implementation of the essential state and essential logic infrastructure — the most significant parts — was developed in a mere 1500 lines of Scheme. In fact this prototype supported not only the relational algebra but also some temporal extensions.

secondly the custom approach leads to the ability to tailor the *hints* available (for use in the *accidental state and control* component) to the exact requirements of the application domain.

Finally, it is of course perfectly possible to develop an FRP infrastructure in *any* general purpose language — be it object-oriented, functional or logic.

9.2 Benefits of this approach

FRP follows the guidelines of *avoid* and *separate* as recommended in section 7 and hence gains all the benefits which derive from that. We now examine how FRP helps to avoid complexity from the common causes.

9.2.1 Benefits for State

The architecture is explicitly designed to *avoid* useless accidental state, and to *avoid* even the *possibility* of an FRP system ever getting into a "bad state".

Specifically derived state is not normally stored (is not treated as essential state). In normal circumstances[26] hybrid feeders/observers *never* feed back in the exact same data which they observed — they only ever feed in some externally generated input or response. *So long as this principle is observed* errors in the logic of the system can never cause it to get into a "bad state" — the only thing required to fix such errors[27] is to correct the logic, there is no need to perform an exhaustive search through and correction of the essential state. This also means that (aside from errors in the *infrastructure*) the system can *never* require "restarting" / "rebooting" etc.

When it comes to *separation*, the architecture clearly exhibits both the *logic / state* split and the *accidental / essential* split recommended in section 7. An example of what this means is that *you do not have to* think about *any accidental state when concentrating on the logic of your system.* In fact, you do not really have to think about the *essential state* as being state either — from the point of view of the logic, the *essential state* is seen as *constant.*

Furthermore, the *functional* component (of the logic) has *no* access to any state at all (even the essential state) — it is totally referentially trans-

The effort involved in this is insignificant when compared to the hundreds of man-years often involved in large systems.

[26]The exception might be in the kind of highly interactive scenario considered in sections 7.2.2 and 7.3.1

[27]We're talking here solely about fixing the system itself — of course FRP can't guarantee that errors in the logic won't escape and affect the real world via *observers*!

parent, can only access what is supplied in the function arguments, and hence offers *hugely* better prospects for testing (as mentioned earlier in section 4.1.1).

Additionally, there are major advantages gained from adopting a relational representation of data — specifically, there is no introduction of subjective bias into the data, no concern with data access paths. This is in contrast with approaches such as OOP or XML (as we saw in section 8.1.2).

Finally, integrity constraints provide big benefits for maintaining consistency of state in a *declarative* manner:

> *The fact that we can impose the integrity constraints of our system in a* purely declarative *manner (without requiring triggers or worse, methods / procedures) is one of the key benefits of the FRP approach. It means that the addition of new constraints increases the complexity of the system only* linearly *because the constraints do not — indeed cannot — interact in any way at all. (Constraints can make use of user-defined functions — but they have no way of referring to other constraints). This is in stark contrast with more imperative approaches such as object oriented programming where interaction between methods causes the complexity to grow at a far greater rate.*

Furthermore, the declarative nature of the integrity constraints opens the door to the possibility of a suitably sophisticated infrastructure making use of them for performance reasons (to give a trivial example, there is no need to compute the relational `intersection` of two relvars at all if it can be established that their integrity constraints are mutually exclusive — because then the result is guaranteed to be empty). This type of optimisation is just not possible if the integrity is maintained in an imperative way.

9.2.2 Benefits for Control

Control is *avoided* completely in the relational component which constitutes the top level of the essential logic. In FRP this logic consists simply of a set of equations (equating derived relvars with the relations calculated by their expressions) which have no intrinsic ordering or control flow at all.

FRP also avoids any *explicit* parallelism in the essential components but provides for the possibility of *separated* accidental control should that be required.

An *infrastructure* which supports FRP may well make use of *implicit* parallelism to improve its performance — but this shouldn't be the concern of anyone other than the implementor of the *infrastructure* — certainly it is not the concern of someone developing an FRP *system*.

A final advantage (which isn't particularly related to control) is that the uniform nature of the representation of data as relations makes it much easier to create distributed implementations of an FRP infrastructure should that be required (e.g. there are no pointers or other access paths to maintain).

9.2.3 Benefits for Code Volume

FRP addresses this in two ways. The first is that a sharp focus on true essentials and *avoiding* useless accidental complexity inevitably leads to less code.

The second way is that the FRP approach reduces the harm that large volumes of code cause through its use of *separation* (see section 4.3).

9.2.4 Benefits for Data Abstraction

Data Abstraction is something which we have only mentioned in passing (in section 4.4) so far. By *data abstraction* we basically mean the creation of compound data types and the use of the corresponding compound values (whose internal contents are *hidden*).

We believe that in many cases, un-needed data abstraction actually represents another common (and serious) cause of complexity. This is for two reasons:

Subjectivity Firstly the grouping of data items together into larger compound data abstractions is an inherently *subjective* business (Ungar and Smith discuss this problem in the context of Self in [SU96]). Groupings which make sense for one purpose will inevitably differ from those most natural for other uses, yet the presence of pre-existing data abstractions all too easily leads to *inappropriate* reuse.

Data Hiding Secondly, large and heavily structured data abstractions can seriously erode the benefits of referential transparency (section 5.2.1) in exactly the manner of the extreme example discussed in section 5.2.3. This problem occurs both because data abstractions will often cause un-needed, irrelevant data to be supplied to a function, and because the data which *does* get used (and hence influences the result of a function) is *hidden* at the function call site. This hidden and excessive data leads to problems for testing as well as informal reasoning in ways very similar to state (see section 4.1).

One of the primary strengths of the relational model (inherited by FRP) is that it involves only minimal commitment to any subjective groupings (basically just the structure chosen for the base relations), and this commitment has only minimal impact on the rest of the system. Derived relvars offer a straightforward way for different (application-specific) groupings to be used *alongside* the base groupings. The benefits in terms of subjectivity are closely related to the benefits of access path independence (section 8.1.2).

FRP also offers benefits in the area of data hiding, simply by discouraging it. Specifically, FRP offers no support for nested relations or for creating product types (as we shall see in section 9.3).

9.2.5 Other Benefits

The previous sections considered the benefits offered by FRP for minimizing complexity. Other potential benefits include performance (as mentioned briefly under section 9.2.1) and the possibility that development teams themselves could be organised around the different components — for example one team could focus on the accidental aspects of the system, one on the essential aspects, one on the interfacing, and another on providing the infrastructure.

9.3 Types

A final comment is that — in addition to a fairly typical set of standard types — FRP provides a limited ability to define new user types for use in the *essential state* and *essential logic* components.

Specifically it permits the creation of disjoint union types (sometimes known as "enumeration" types) but does *not* permit the creation of new product types (types with multiple subsidiary components). This is because (as mentioned above) we have a strong desire to *avoid* any unnecessary data abstraction.

Finally, it probably makes sense for infrastructures to provide *type inference* for the *essential logic*. Interesting work in this area has been carried out in the Machiavelli system [OB88].

10 Example of an FRP system

We now examine a simple example FRP system. The system is designed to support an estate agency (real estate) business. It will keep track of properties which are being sold, offers which are made on the properties, decisions made on the offers by the owners, and commission fees earnt by the individual agency employees from their successful sales. The example should serve to highlight the declarative nature of the components of an FRP system.

To keep things simple, this system operates under some restrictions:

1. Sales only — no rentals / lettings
2. People only have one home, and the owners reside at the property they are selling
3. Rooms are perfectly rectangular
4. Offer acceptance is binding (ie an accepted offer constitutes a sale)

The example will use syntax from a hypothetical FRP infrastructure (which supports not only the relational algebra but also some of the common extensions from section 8.5) — typewriter font is used for this.

10.1 User-defined Types

The example system makes use of a small number of custom types (see section 9.3), some of which are just aliases for types provided by the infrastructure:

```
def alias address :  string
def alias agent :  string
def alias name :  string
def alias price :  double
def enum roomType :  KITCHEN | BATHROOM | LIVING_ROOM
def enum priceBand :  LOW | MED | HIGH | PREMIUM
def enum areaCode :  CITY | SUBURBAN | RURAL
def enum speedBand :  VERY_FAST | FAST | MEDIUM | SLOW |
VERY_SLOW
```

10.2 Essential State

The essential state (see section 9.1.1) consists of the definitions of the *types* of the base relvars (the types of the attributes are shown in *italics*).

```
def relvar Property :: {address:address price:price
     photo:filename agent:agent dateRegistered:date}
```

```
def relvar Offer :: {address:address offerPrice:price
     offerDate:date bidderName:name bidderAddress:address}

def relvar Decision :: {address:address offerDate:date
     bidderName:name bidderAddress:address decisionDate:date
     accepted:bool}

def relvar Room :: {address:address roomName:string
     width:double breadth:double type:roomType}

def relvar Floor :: {address:address roomName:string
     floor:int}

def relvar Commission :: {priceBand:priceBand
     areaCode:areaCode saleSpeed:speedBand commission:double}
```

The example makes use of six base relations, most of which are self-explanatory.

The Property relation stores all properties sold or for-sale. As will be seen in section 10.3.3, properties are uniquely identified by their *address*. The *price* is the desired sale price, the *agent* is the agency employee responsible for selling the Property, and the *dateRegistered* is the date that the Property was registred for sale with the agency.

The Offer relation records the history of all offers ever made. The *address* represents the Property on which the Offer is being made (by the *bidderName* who lives at *bidderAddress*). The *offerDate* attribute records the date when the offer was made, and the *offerPrice* records the price offered. Offers are uniquely identified by an (*address*, *offerDate*, *bidderName*, *bidderAddress*) combination.

The Decision relation records the decisions made by the owner on the Offers that have been made. The Offer in question is identified by the (*address*, *offerDate*, *bidderName*, *bidderAddress*) attributes, and the date and outcome of the decision are recorded by (*decisionDate* and *accepted*).

The Room relation records information (*width*, *breadth*, *type*) about the rooms that exist at each Property. The Property is of course represented by the *address*. One point worthy of note (because it's slightly artificial) is that an assumption is made that every Room in each Property has a unique (within the scope of that Property) *roomName*. This is necessary because many properties may have more than one room of a given *type* (and size).

The Floor relation records which *floor* each Room (*roomName*, *address*) is on.

Finally, the Commission relation stores *commission* fees that can be earned by the agency employees. The commission fees are assigned on the basis of sale *price*s divided into different *priceBand*s, Property *address*es categorized into *areaCode*s and ratings of the *saleSpeed*. (The decision has been made to represent commission rates as a base relation — rather than as a function — so that the commission fees can be queried and easily adjusted).

10.3 Essential Logic

This is the heart of the system (see section 9.1.2) and corresponds to the "business logic".

10.3.1 Functions

We do not give the actual function definitions here, we just describe their operation informally. In reality we would supply the function definitions in terms of some language provided by the infrastructure.

`priceBandForPrice` Converts a price into a *priceBand* (which will be used in the commission calculations)

`areaCodeForAddress` Converts an address into an *areaCode*

`datesToSpeedBand` Converts a pair of dates into a *speedBand* (reflecting the speed of sale after taking into account the time of year)

10.3.2 Derived Relations

There are thirteen derived relations in the system. These can be very loosely classified as *internal* or *external* according to whether their main purpose is simply to facilitate the definition of other derived relations (and constraints) or to provide information to the users. We consider the definition and purpose of each in turn.

As an aid to understanding, the types of the derived relations are shown in comments (delimited by `/*` and `*/`). In reality these types would be derived (or checked) by an infrastructure-provided type inference mechanism.

Internal

The ten *internal* derived relations exist mainly to help with the later definition of the three *external* ones.

```
/* RoomInfo :: {address:address roomName:string width:double
                breadth:double type:roomType roomSize:double} */
RoomInfo = extend(Room, (roomSize = width*breadth))
```

The RoomInfo derived relation simply extends the Room base relation with an extra attribute *roomSize* which gives the area of each room.

```
/* Acceptance :: {address:address offerDate:date bidderName:name
                  bidderAddress:address decisionDate:date} */
Acceptance = project_away(restrict(Decision | accepted == true),
                          accepted)
```

The Acceptance derived relation simply selects the positive entries from the Decision base relation, and then strips away the *accepted* attribute (the `project_away` operation is the dual of the `project` operation — it removes the listed attributes rather than keeping them).

```
/* Rejection :: {address:address offerDate:date bidderName:name
                 bidderAddress:address decisionDate:date} */
Rejection = project_away(restrict(Decision | accepted == false),
                         accepted)
```

The Rejection derived relation simply selects the negative decisions and removes the *accepted* attribute.

```
/* PropertyInfo :: {address:address price:price photo:filename
                    agent:agent dateRegistered:date
                    priceBand:priceBand areaCode:areaCode
                    numberOfRooms:int squareFeet:double} */
PropertyInfo =
extend(Property,
       (priceBand = priceBandForPrice(price)),
       (areaCode = areaCodeForAddress(address)),
       (numberOfRooms = count(restrict(RoomInfo |
                                       address == address))),
       (squareFeet = sum(roomSize, restrict(RoomInfo |
                                            address == address))))
```

The PropertyInfo derived relation extends the Property base relation with four new attributes. The first — called *priceBand* — indicates which of the estate agency's price bands the property is in. The price band of the final sale price will affect the commission derived by the agent for selling the property. The *areaCode* attribute indicates the area code, which also affects the commission an agent may earn. The *numberOfRooms* is calculated by counting the number of rooms (actually the number of entries in the Room-Info derived relation at the corresponding address), and the *squareFeet* is computed by summing up the relevant *roomSize*s.

```
/* CurrentOffer :: {address:address offerPrice:price
                    offerDate:date bidderName:name
                    bidderAddress:address} */
CurrentOffer =
summarize(Offer,
          project(Offer, address bidderName bidderAddress),
          quota(offerDate,1))
```

The purpose of the CurrentOffer derived relation is to filter out old offers which have been superceded by newer ones (e.g. if the bidder has submitted a revised — higher or lower — offer, then we are no longer interested in older offers they may have made on the same property).

The definition summarizes the Offer base relation, taking the most recent (ie the single greatest *offerDate*) offer made by each bidder on a property (ie per unique *address*, *bidderName*, *bidderAddress* combination). Because both *bidderName* and *bidderAddress* are included, the system supports the (admittedly unusual) possibility of different people living in the same place (*bidderAddress*) submitting different offers on the same property (*address*).

```
/* RawSales :: {address:address offerPrice:price
                decisionDate:date agent:agent
                dateRegistered:date} */
RawSales =
project_away(join(Acceptance,
                  join(CurrentOffer,
                       project(Property, address agent
                                         dateRegistered))),
             offerDate bidderName bidderAddress)
```

For the purposes of this example, sales are seen as corresponding directly to accepted offers. As a result the definition of the RawSales relation is in terms of the Acceptance relation. These accepted offers are augmented (joined) with the CurrentOffer information (which includes the agreed *offerPrice*) and with information (*agent*, *dateRegistered*) from the Property relation.

```
/* SoldProperty :: {address:address} */
SoldProperty = project(RawSales, address)
```

The SoldProperty relation simply contains the *address* of all Properties on which a sale has been agreed (ie the properties in the RawSales relation).

```
/* UnsoldProperty :: {address:address} */
UnsoldProperty = minus(project(Property, address), SoldProperty)
```

The UnsoldProperty is obviously just the Property which is not SoldProperty (i.e. all Property addresses minus the SoldProperty addresses).

```
/* SalesInfo :: {address:address agent:agent areaCode:areaCode
                 saleSpeed:speedBand priceBand:priceBand} */
SalesInfo =
project(extend(RawSales,
               (areaCode = areaCodeForAddress(address)),
               (saleSpeed = datesToSpeedBand(dateRegistered,
                                             decisionDate)),
               (priceBand = priceBandForPrice(offerPrice))),
             address agent areaCode saleSpeed priceBand)
```

The SalesInfo relation is based on the RawSales relation, but extends it with *areaCode*, *saleSpeed* and *priceBand* information by calling the three relevant functions.

```
/* SalesCommissions :: {address:address agent:agent
                        commission:double} */
SalesCommissions =
project(join(SalesInfo, Commission),
        address agent commission)
```

The SalesCommissions which are due to the agents are derived simply by joining together the SalesInfo with the Commission base relation. This gives the amount of *commission* due to each *agent* on each Property (represented by *address*).

External

Having now defined all the *internal* derived relations, we are now in a position to define the *external* derived relations — these are the ones which will be of most direct interest to the users of the system.

```
/* OpenOffers :: {address:address offerPrice:price
                  offerDate:date bidderName:name
                  bidderAddress:address} */
OpenOffers =
join(CurrentOffer,
     minus(project_away(CurrentOffer, offerPrice),
           project_away(Decision, accepted decisionDate)))
```

The OpenOffers relation gives details of the CurrentOffers on which the owner has not yet made a Decision. This is calculated by joining the CurrentOffer information (which includes *offerPrice*) with those CurrentOffers (excluding the price information) that do not have corresponding Decisions. `project_away` is used here because `minus` requires its arguments to be of the same type.

```
/* PropertyForWebSite :: {address:address price:price
                          photo:filename numberOfRooms:int
                          squareFeet:double} */
PropertyForWebSite = project( join(UnsoldProperty, PropertyInfo),
                              address price photo
                              numberOfRooms squareFeet )
```

The business wants to display the information from PropertyInfo on their external website. However, they only want to show unsold property (this is achieved simply by a `join`), and they only want to show a subset of the attributes (this is achieved with a `project`).

```
/* CommissionDue :: {agent:agent totalCommission:double} */
CommissionDue =
project(summarize(SalesCommissions,
                  project(SalesCommissions, agent),
                  totalCommission = sum(commission)),
        agent totalCommission)
```

Finally, the total commission due to each agent is calculated by simply summing up the *commission* attribute of the SalesCommissions relation on a per *agent* basis to give the *totalCommission* attribute.

10.3.3 Integrity

Integrity constraints are given in the form of relational algebra or relational calculus expressions. As already noted, our hypothetical FRP infrastructure provides common relational algebra extensions (see section 8.5). It also provides special syntax for *candidate* and *foreign* key constraints. (This syntax is effectively just a shorthand for the underlying algebra or calculus expression).

We consider the standard (key) constraints first:

```
candidate key Property = (address)
candidate key Offer = (address, offerDate,
                       bidderName, bidderAddress)
candidate key Decision = (address, offerDate,
                          bidderName, bidderAddress)
candidate key Room = (address, roomName)
candidate key Floor = (address, roomName)
candidate key Commision = (priceBand, areaCode, saleSpeed)

foreign key Offer (address) in Property
foreign key Decision (address, offerDate,
                      bidderName, bidderAddress) in Offer
foreign key Room (address) in Property
foreign key Floor (address) in Property
```

There are also some slightly more interesting, domain-specific constraints. The first insists that all properties must have at least one room:

```
count(restrict(PropertyInfo | numberOfRooms < 1)) == 0
```

The next ensures that people cannot submit bids on their own property (owners are assumed to be residing at the property they are selling):

```
count(restrict(Offer | bidderAddress == address)) == 0
```

This constraint prohibits the submission of any Offers on a property (*address*) after a sale has happened (i.e. after an Acceptance has occurred for the *address*):

```
count(restrict(join(Offer,
                    project(Acceptance, address decisionDate))
               | offerDate > decisionDate)) == 0
```

The next constraint ensures that there are never more than 50 properties advertised on the website in the `PREMIUM` price band:

```
count(restrict(extend(PropertyForWebSite,
                      (priceBand = priceBandForPrice(price)))
               | priceBand == PREMIUM)) < 50
```

This is an interesting constraint because it depends (directly as it happens) on a user-defined function (`priceBandForPrice`). One implication of this is that changes to function definitions (as well as changes to *essential state*) could — if unchecked — cause the system to violate its constraints. No FRP infrastructure can allow this.

Fortunately there are two straightforward approaches to solving this. The first is that the infrastructure could treat function definitions as data (essential state) and apply the same kind of modification checks. The alternative is that it could refuse to run a system with a new function version which causes existing data to be considered invalid. In this latter case manual state changes would be required to restore integrity and to allow the system became operational again.

Finally, no single bidder can submit more than 10 offers (over time) on a single Property. This constraint works by first computing the number of offers made by each bidder (*bidderName*, *bidderAddress*) on each Property (*address*), and ensuring that this is never more than 10:

```
count(restrict(summarize(Offer,
                         project(Offer, address bidderName
                                        bidderAddress),
                         numberOfOffers = count())
               | numberOfOffers > 10)) == 0
```

Once the system is deployed, the FRP infrastructure will reject any state modification attempts which would violate any of these integrity constraints.

10.4 Accidental State and Control

The *accidental state and control* component of an FRP system consists solely of a set of declarations which represent performance hints for the infrastructure (see section 9.1.3). In this example the *accidental state and control* is a set of three hint declarations.

```
declare store PropertyInfo
```

This declaration is simply a hint to the infrastructure to request that the PropertyInfo derived relation is actually stored (ie cached) rather than continually recalculated.

```
declare store shared Room Floor
```

This hint instructs the infrastructure to *denormalize* the Room and Floor relations into a single shared storage structure. (Note that because we are able to express this as part of the *accidental state and control* we have not been forced to compromise the *essential* parts of our system which still treat Room and Floor separately).

```
declare store separate Property (photo)
```

This hint instructs the infrastructure to store the *photo* attribute of the Property relation separately from its other attributes (because it is not frequently used).

These three hints have all focused on *state* (PropertyInfo is *accidental state*, and the other two declarations are concerned with *accidental* aspects *of* state). Larger systems would probably also include *accidental control* specifications for performance reasons.

10.5 Other

The *feeders* and *observers* for this system would be fairly simple — *feeding* user input into Decisions, Offers etc., and directly *observing* and displaying the various derived relations as output (e.g. OpenOffers, PropertyForWebSite and CommisionDue).

Because of this it is reasonable to expect that the *feeders* and *observers* would require no custom coding at all, but could instead be specified in a completely declarative fashion.

One extension which might require a custom *observer* would be a requirement to connect CommissionDue into an external payroll system.

11 Related Work

FRP draws some influence from the ideas of [DD00]. In contrast with this work however, FRP is aimed at general purpose, large-scale *application* programming. Additionally FRP focuses on a *separate*, *functional*, sub-language and has different ideas about the use of types. Finally the *accidental* component of FRP has a broader range than the physical / logical mapping of traditional DBMSs.

There are also some similarities to Backus' Applicative State Transition systems [Bac78], and to the Aldat project at McGill [Mer85] which investigated general purpose applications of relational algebra.

12 Conclusions

We have argued that *complexity* causes more problems in large software systems than anything else. We have also argued that it *can* be tamed — but only through a concerted effort to *avoid* it where possible, and to *separate* it where not. Specifically we have argued that a system can usefully be *separated* into three main parts: the *essential state*, the *essential logic*, and the *accidental state and control*.

We believe that taking these principles and applying them to the top level of a system design — effectively using different specialised *languages* for the different components — can offer more in terms simplicity than can the unstructured adoption of any single general language (be it imperative, logic or functional). In making this argument we briefly surveyed each of the common programming paradigms, paying some attention to the weaknesses of object-orientation as a particular example of an imperative approach.

In cases (such as existing large systems) where this *separation* cannot be directly applied we believe the focus should be on avoiding state, avoiding explicit control where possible, and striving at all costs to *get rid of code.*

So, what is the way out of the tar pit? What is the silver bullet? . . . it may not be FRP, but we believe there can be no doubt that it is *simplicity.*

References

[Bac78] John W. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[Bak93] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *Journal of Object-Oriented Programming*, 4(4):2–27, October 1993.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[Bro86] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Information Processing 1986, Proceedings of the Tenth World Computing Conference*, H.-J. Kugler, ed.: 1069–76. Reprinted in IEEE Computer, 20(4):10-19, April 1987, and in Brooks, The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Chapter 16, Addison-Wesley, 1995.

[Che76] P. P. Chen. “The Entity-Relationship Model”. *ACM Trans. on Database Systems (TODS)*, 1:9–36, 1976.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comm. ACM*, 13(6):377–387, June 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Sys.*, 4(4):397, December 1979.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.

[Cor91] Fernando J. Corbató. On building systems that will fail. *Commun. ACM*, 34(9):72–81, 1991.

[Dat04] C. J. Date. *An Introduction to Database Systems*. Addison Wesley, 8th edition, 2004.

[DD00] Hugh Darwen and C. J. Date. *Foundation for Future Database Systems: The Third Manifesto*. Addison-Wesley, 2nd edition, 2000.

[Dij71] Edsger W. Dijkstra. On the reliability of programs. circulated privately, 1971.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Dij97] Dijkstra. The tide, not the waves. In Peter J. Denning and Robert M. Metcalfe, editors, *Beyond Calculation: The Next Fifty Years of Computing, Copernicus, 1997*. 1997.

[Eco04] Managing complexity. *The Economist*, 373(8403):89–91, 2004.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices*, pages 263–273, New York, June 9–11 1997. ACM Press.

[HJ89] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, November 1989.

[Hoa81] C. A. R. Hoare. The emperor's old clothes. *Commun. ACM*, 24(2):75–83, 1981.

[Kow79] Robert A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[Mer85] T. H. Merrett. Persistence and Aldat. In *Data Types and Persistence (Appin)*, pages 173–188, 1985.

[NR69] P. Naur and B. Randell. Software engineering report of a conference sponsored by the NATO science committee Garmisch Germany 7th-11th October 1968, January 01 1969.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 174–183, New York, NY, 1988. ACM.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, 1990.

[PJ$^+$03] Simon Peyton Jones et al., editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[SU96] Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[vRH04] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[Won00] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program*, 10(1):19–56, 2000.

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge, Mass. [*]

April 1960

1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

[*]Putting this paper in LaTeXpartly supported by ARPA (ONR) grant N00014-94-1-0775 to Stanford University where John McCarthy has been since 1962. Copied with minor notational changes from *CACM*, April 1960. If you want the exact typography, look there. Current address, John McCarthy, Computer Science Department, Stanford, CA 94305, (email: jmc@cs.stanford.edu), (URL: http://www-formal.stanford.edu/jmc/)

In this article, we first describe a formalism for defining functions recursively. We believe this formalism has advantages both as a programming language and as a vehicle for developing a theory of computation. Next, we describe S-expressions and S-functions, give some examples, and then describe the universal S-function *apply* which plays the theoretical role of a universal Turing machine and the practical role of an interpreter. Then we describe the representation of S-expressions in the memory of the IBM 704 by list structures similar to those used by Newell, Shaw and Simon [2], and the representation of S-functions by program. Then we mention the main features of the LISP programming system for the IBM 704. Next comes another way of describing computations with symbolic expressions, and finally we give a recursive function interpretation of flow charts.

We hope to describe some of the symbolic computations for which LISP has been used in another paper, and also to give elsewhere some applications of our recursive function formalism to mathematical logic and to the problem of mechanical theorem proving.

2 Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of conditional expression is believed to be new[1], and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. *Partial Functions.* A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. *Propositional Expressions and Predicates.* A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\neg$ ("not"). Typical propositional expressions are:

[1]reference Kleene

$$x < y$$

$$(x < y) \wedge (b = c)$$

$$\text{x is prime}$$

A predicate is a function whose range consists of the truth values T and F.

c. *Conditional Expressions.* The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function |x| is usually defined in words. Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1 otherwise if p_2 then e_2, $\cdots$, otherwise if p_n then e_n," or "p_1 yields $e_1, \cdots, p_n$ yields e_n." [2]

We now give the rules for determining whether the value of

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

is defined, and if so what its value is. Examine the p's from left to right. If a p whose value is T is encountered before any p whose value is undefined is encountered then the value of the conditional expression is the value of the corresponding e (if this is defined). If any undefined p is encountered before

[2] I sent a proposal for conditional expressions to a *CACM* forum on what should be included in Algol 60. Because the item was short, the editor demoted it to a letter to the editor, for which *CACM* subsequently apologized. The notation given here was rejected for Algol 60, because it had been decided that no new mathematical notation should be allowed in Algol 60, and everything new had to be English. The **if** . . . **then** . . . **else** that Algol 60 adopted was suggested by John Backus.

a true p, or if all p's are false, or if the e corresponding to the first true p is undefined, then the value of the conditional expression is undefined. We now give examples.

$$(1 < 2 \to 4, 1 > 2 \to 3) = 4$$

$$(2 < 1 \to 4, 2 > 1 \to 3, 2 > 1 \to 2) = 3$$

$$(2 < 1 \to 4, T \to 3) = 3$$

$$(2 < 1 \to \frac{0}{0}, T \to 3) = 3$$

$$(2 < 1 \to 3, T \to \frac{0}{0}) \text{ is undefined}$$

$$(2 < 1 \to 3, 4 < 1 \to 4) \text{ is undefined}$$

Some of the simplest applications of conditional expressions are in giving such definitions as

$$|x| = (x < 0 \to -x, T \to x)$$

$$\delta_{ij} = (i = j \to 1, T \to 0)$$

$$sgn(x) = (x < 0 \to -1, x = 0 \to 0, T \to 1)$$

d. *Recursive Function Definitions.* By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

$$n! = (n = 0 \to 1, T \to n \cdot (n-1)!)$$

When we use this formula to evaluate 0! we get the answer 1; because of the way in which the value of a conditional expression was defined, the meaningless expression 0 · (0 - 1)! does not arise. The evaluation of 2! according to this definition proceeds as follows:

$$\begin{aligned}
2! &= (2 = 0 \rightarrow 1, T \rightarrow 2 \cdot (2-1)!) \\
&= 2 \cdot 1! \\
&= 2 \cdot (1 = 0 \rightarrow 1T \rightarrow \cdot(1-1)!) \\
&= 2 \cdot 1 \cdot 0! \\
&= 2 \cdot 1 \cdot (0 = 0 \rightarrow 1, T \rightarrow 0 \cdot (0-1)!) \\
&= 2 \cdot 1 \cdot 1 \\
&= 2
\end{aligned}$$

We now give two other applications of recursive function definitions. The greatest common divisor, gcd(m,n), of two positive integers m and n is computed by means of the Euclidean algorithm. This algorithm is expressed by the recursive function definition:

$$gcd(m,n) = (m > n \rightarrow gcd(n,m), rem(n,m) = 0 \rightarrow m, T \rightarrow gcd(rem(n,m), m))$$

where $rem(n,m)$ denotes the remainder left when n is divided by m.

The Newtonian algorithm for obtaining an approximate square root of a number a, starting with an initial approximation x and requiring that an acceptable approximation y satisfy $|y^2 - a| < \epsilon$, may be written as

sqrt(a, x, ϵ)

$$= (|x^2 - a| < \epsilon \rightarrow \text{x,T} \rightarrow \text{sqrt (a, } \tfrac{1}{2}(x + \tfrac{a}{x}), \epsilon))$$

The simultaneous recursive definition of several functions is also possible, and we shall use such definitions if they are required.

There is no guarantee that the computation determined by a recursive definition will ever terminate and, for example, an attempt to compute n! from our definition will only succeed if n is a non-negative integer. If the computation does not terminate, the function must be regarded as undefined for the given arguments.

The propositional connectives themselves can be defined by conditional expressions. We write

$$\begin{aligned}
p \wedge q &= (p \to q, T \to F) \\
p \vee q &= (p \to T, T \to q) \\
\neg p &= (p \to F, T \to T) \\
p \supset q &= (p \to q, T \to T)
\end{aligned}$$

It is readily seen that the right-hand sides of the equations have the correct truth tables. If we consider situations in which p or q may be undefined, the connectives $\wedge$ and $\vee$ are seen to be noncommutative. For example if p is false and q is undefined, we see that according to the definitions given above $p \wedge q$ is false, but $q \wedge p$ is undefined. For our applications this noncommutativity is desirable, since $p \wedge q$ is computed by first computing p, and if p is false q is not computed. If the computation for p does not terminate, we never get around to computing q. We shall use propositional connectives in this sense hereafter.

e. *Functions and Forms.* It is usual in mathematics—outside of mathematical logic—to use the word "function" imprecisely and to apply it to forms such as $y^2 + x$. Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially, is given by Church [3].

Let f be an expression that stands for a function of two integer variables. It should make sense to write $f(3, 4)$ and the value of this expression should be determined. The expression $y^2 + x$ does not meet this requirement; $y^2 + x(3, 4)$ is not a conventional notation, and if we attempted to define it we would be uncertain whether its value would turn out to be 13 or 19. Church calls an expression like $y^2 + x$, a form. A form can be converted into a function if we can determine the correspondence between the variables occurring in the form and the ordered list of arguments of the desired function. This is accomplished by Church's λ-notation.

If $\mathcal{E}$ is a form in variables $x_1, \cdots, x_n$, then $\lambda((x_1, \cdots, x_n), \mathcal{E})$ will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables $x_1, \cdots, x_n$ in that order in $\mathcal{E}$ and evaluating the resulting expression. For example, $\lambda((x, y), y^2 + x)$ is a function of two variables, and $\lambda((x, y), y^2 + x)(3, 4) = 19$.

The variables occurring in the list of variables of a λ-expression are dummy or bound, like variables of integration in a definite integral. That is, we may change the names of the bound variables in a function expression without changing the value of the expression, provided that we make the same change for each occurrence of the variable and do not make two variables the same that previously were different. Thus $\lambda((x,y),y^2+x),\lambda((u,v),v^2+u)$ and $\lambda((y,x),x^2+y)$ denote the same function.

We shall frequently use expressions in which some of the variables are bound by λ's and others are not. Such an expression may be regarded as defining a function with parameters. The unbound variables are called free variables.

An adequate notation that distinguishes functions from forms allows an unambiguous treatment of functions of functions. It would involve too much of a digression to give examples here, but we shall use functions with functions as arguments later in this report.

Difficulties arise in combining functions described by λ-expressions, or by any other notation involving variables, because different bound variables may be represented by the same symbol. This is called collision of bound variables. There is a notation involving operators that are called combinators for combining functions without the use of variables. Unfortunately, the combinatory expressions for interesting combinations of functions tend to be lengthy and unreadable.

f. *Expressions for Recursive Functions.* The λ-notation is inadequate for naming functions defined recursively. For example, using λ's, we can convert the definition

$$\mathrm{sqrt}(a,x,\epsilon)=(|x^2-a|<\epsilon\rightarrow x,T\rightarrow\mathrm{sqrt}(a,\frac{1}{2}(x+\frac{a}{x}),\epsilon))$$

into

$$\mathrm{sqrt}=\lambda((a,x,\epsilon),(|x^2-a|<\epsilon\rightarrow x,T\rightarrow\mathrm{sqrt}(a,\frac{1}{2}(x+\frac{a}{x}),\epsilon))),$$

but the right-hand side cannot serve as an expression for the function because there would be nothing to indicate that the reference to *sqrt* within the expression stood for the expression as a whole.

In order to be able to write expressions for recursive functions, we introduce another notation. $label(a,\mathcal{E})$ denotes the expression $\mathcal{E}$, provided that occurrences of a within $\mathcal{E}$ are to be interpreted as referring to the expression as a whole. Thus we can write

$$\text{label}(\text{sqrt}, \lambda((a, x, \epsilon), (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \tfrac{1}{2}(x + \tfrac{a}{x}), \epsilon))))$$

as a name for our sqrt function.

The symbol a in label $(a, \mathcal{E})$ is also bound, that is, it may be altered systematically without changing the meaning of the expression. It behaves differently from a variable bound by a λ, however.

3 Recursive Functions of Symbolic Expressions

We shall first define a class of symbolic expressions in terms of ordered pairs and lists. Then we shall define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples. We shall then show how these functions themselves can be expressed as symbolic expressions, and we shall define a universal function *apply* that allows us to compute from the expression for a given function its value for given arguments. Finally, we shall define some functions with functions as arguments and give some useful examples.

a. *A Class of Symbolic Expressions.* We shall now define the S-expressions (S stands for symbolic). They are formed by using the special characters

$$\cdot$$
$$)$$
$$($$

and an infinite set of distinguishable atomic symbols. For atomic symbols, we shall use strings of capital Latin letters and digits with single imbedded blanks.[3] Examples of atomic symbols are

$$\begin{array}{l} A \\ ABA \\ APPLE\ PIE\ NUMBER\ 3 \end{array}$$

There is a twofold reason for departing from the usual mathematical practice of using single letters for atomic symbols. First, computer programs frequently require hundreds of distinguishable symbols that must be formed from the 47 characters that are printable by the IBM 704 computer. Second, it is convenient to allow English words and phrases to stand for atomic entities for mnemonic reasons. The symbols are atomic in the sense that any substructure they may have as sequences of characters is ignored. We assume only that different symbols can be distinguished. S-expressions are then defined as follows:

1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.

Examples of S-expressions are

$$\begin{array}{l} AB \\ (A \cdot B) \\ ((AB \cdot C) \cdot D) \end{array}$$

An S-expression is then simply an ordered pair, the terms of which may be atomic symbols or simpler S-expressions. We can can represent a list of arbitrary length in terms of S-expressions as follows. The list

$$(m_1, m_2, \cdots, m_n)$$

is represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))$$

Here NIL is an atomic symbol used to terminate lists. Since many of the symbolic expressions with which we deal are conveniently expressed as lists, we shall introduce a list notation to abbreviate certain S-expressions. We have

[3] 1995 remark: Imbedded blanks could be allowed within symbols, because lists were then written with commas between elements.

l. (m) stands for (m $\cdot NIL$).
2. $(m_1, \cdots, m_n)$ stands for $(m_1 \cdot (\cdots (m_n \cdot NIL) \cdots))$.
3. $(m_1, \cdots, m_n \cdot x)$ stands for $(m_1 \cdot (\cdots (m_n \cdot x) \cdots))$.

Subexpressions can be similarly abbreviated. Some examples of these abbreviations are
$((AB, C), D)$ for $((AB \cdot (C \cdot NIL)) \cdot (D \cdot NIL))$
$((A, B), C, D \cdot E)$ for $((A \cdot (B \cdot NIL)) \cdot (C \cdot (D \cdot E)))$

Since we regard the expressions with commas as abbreviations for those not involving commas, we shall refer to them all as S-expressions.

b. *Functions of S-expressions and the Expressions That Represent Them.* We now define a class of functions of S-expressions. The expressions representing these functions are written in a conventional functional notation. However, in order to clearly distinguish the expressions representing functions from S-expressions, we shall use sequences of lower-case letters for function names and variables ranging over the set of S-expressions. We also use brackets and semicolons, instead of parentheses and commas, for denoting the application of functions to their arguments. Thus we write

$$car[x]$$
$$car[cons[(A \cdot B); x]]$$

In these M-expressions (meta-expressions) any S-expression that occur stand for themselves.

c. *The Elementary S-functions and Predicates.* We introduce the following functions and predicates:
1. atom. atom[x] has the value of T or F according to whether x is an atomic symbol. Thus

atom [X] = T
atom [(X · A)] = F

2. eq. eq [x;y] is defined if and only if both x and y are atomic. eq [x; y] = T if x and y are the same symbol, and eq [x; y] = F otherwise. Thus

eq [X; X] = T
eq [X; A] = F
eq [X; (X · A)] is undefined.

3. car. car[x] is defined if and only if x is not atomic. car $[(e_1 \cdot e_2)] = e_1$. Thus car [X] is undefined.

car $[(X \cdot A)]$ = X
car $[((X \cdot A) \cdot Y)] = (X \cdot A)$

4. cdr. cdr [x] is also defined when x is not atomic. We have cdr $[(e_1 \cdot e_2)] = e_2$. Thus cdr [X] is undefined.

cdr $[(X \cdot A)]$ = A cdr $[((X \cdot A) \cdot Y)]$ = Y

5. cons. cons [x; y] is defined for any x and y. We have cons $[e_1; e_2] = (e_1 \cdot e_2)$. Thus

cons [X; A] = (X A)
cons $[(X \cdot A); Y] = ((X \cdot A)Y)$

car, cdr, and cons are easily seen to satisfy the relations

car [cons [x; y]] = x
cdr [cons [x; y]] = y
cons [car [x]; cdr [x]] = x, provided that x is not atomic.

The names "car" and "cons" will come to have mnemonic significance only when we discuss the representation of the system in the computer. Compositions of car and cdr give the subexpressions of a given expression in a given position. Compositions of cons form expressions of a given structure out of parts. The class of functions which can be formed in this way is quite limited and not very interesting.

d. *Recursive S-functions.* We get a much larger class of functions (in fact, all computable functions) when we allow ourselves to form new functions of S-expressions by conditional expressions and recursive definition. We now give some examples of functions that are definable in this way.

1. ff$[x]$. The value of ff$[x]$ is the first atomic symbol of the S-expression x with the parentheses ignored. Thus

$$\text{ff}[((A \cdot B) \cdot C)] = A$$

We have

$$\text{ff}[x] = [atom[x] \to x; T \to \text{ff}[car[x]]]$$

We now trace in detail the steps in the evaluation of

ff [((A · B) · C)]:
ff [((A · B) · C)]

$$= \quad [atom[((A \cdot B) \cdot C)] \to ((A \cdot B) \cdot C); T \to \text{ff}[car[((A \cdot B)C \cdot)]]]$$

$$= \quad [F \to ((A \cdot B) \cdot C); T \to \text{ff}[car[((A \cdot B) \cdot C)]]]$$

$$= \quad [T \to \text{ff}[car[((A \cdot B) \cdot C)]]]$$

$$= \quad \text{ff}[car[((A \cdot B) \cdot C)]]$$

$$= \quad \text{ff}[(A \cdot B)]$$

$$= \quad [atom[(A \cdot B)] \to (A \cdot B); T \to \text{ff}[car[(A \cdot B)]]]$$

$$= \quad [F \to (A \cdot B); T \to \text{ff}[car[(A \cdot B)]]]$$

$$= \quad [T \to \text{ff}[car[(A \cdot B)]]]$$

$$= \quad \text{ff}[car[(A \cdot B)]]$$

$$= \quad \text{ff}[A]$$

$$= \quad [atom[A] \rightarrow A; T \rightarrow \text{ff}[car[A]]]$$

$$= \quad [T \rightarrow A; T \rightarrow \text{ff}[car[A]]]$$

$$= \quad A$$

2. subst $[x; y; z]$. This function gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. It is defined by

subst [x; y; z] = [atom [z] → [eq [z; y] → x; T → z];
T → cons [subst [x; y; car [z]]; subst [x; y; cdr [z]]]]

As an example, we have

$$subst[(X \cdot A); B; ((A \cdot B) \cdot C)] = ((A \cdot (X \cdot A)) \cdot C)$$

3. equal [x; y]. This is a predicate that has the value T if x and y are the same S-expression, and has the value F otherwise. We have

equal [x; y] = [atom [x] ∧ atom [y] ∧ eq [x; y]]

∨[¬ atom [x] ∧¬ atom [y] ∧ equal [car [x]; car [y]]

∧ equal [cdr [x]; cdr [y]]]

It is convenient to see how the elementary functions look in the abbreviated list notation. The reader will easily verify that

(i) $car[(m_1, m_2, \cdots, m_n)] = m_1$
(ii) $cdr[(m_s, m_2, \cdots, m_n)] = (m_2, \cdots, m_n)$
(iii) $cdr[(m)] = NIL$
(iv) $cons[m_1; (m_2, \cdots, m_n)] = (m_1, m_2, \cdots, m_n)$
(v) $cons[m; NIL] = (m)$

We define

$null[x] = atom[x] \wedge eq[x; NIL]$
This predicate is useful in dealing with lists.

Compositions of car and cdr arise so frequently that many expressions can be written more concisely if we abbreviate

$cadr[x]$ for $car[cdr[x]]$,
$caddr[x]$ for $car[cdr[cdr[x]]]$, etc.

Another useful abbreviation is to write list $[e_1; e_2; \cdots; e_n]$
for $cons[e_1; cons[e_2; \cdots; cons[e_n; NIL] \cdots]]$.

This function gives the list, $(e_1, \cdots, e_n)$, as a function of its elements.

The following functions are useful when S-expressions are regarded as lists.

1. append [x;y].

append [x; y] = [null[x] → y; T → cons [car [x]; append [cdr [x]; y]]]

An example is

append [(A, B); (C, D, E)] = (A, B, C, D, E)

2. among [x;y]. This predicate is true if the S-expression x occurs among the elements of the list y. We have

$$among[x; y] = \neg null[y] \wedge [equal[x; car[y]] \vee among[x; cdr[y]]]$$

3. pair [x;y]. This function gives the list of pairs of corresponding elements of the lists x and y. We have

$$pair[x; y] = [null[x] \wedge null[y] \rightarrow NIL; \neg atom[x] \wedge \neg atom[y] \rightarrow cons[list[car[x]; car[y]]; pair[cdr[x]; cdr[y]]]$$

An example is

$$pair[(A, B, C); (X, (Y, Z), U)] = ((A, X), (B, (Y, Z)), (C, U)).$$

4. assoc [x;y]. If y is a list of the form $((u_1, v_1), \cdots, (u_n, v_n))$ and x is one of the u's, then assoc $[x; y]$ is the corresponding v. We have

$$assoc[x; y] = eq[caar[y]; x] \rightarrow cadar[y]; T \rightarrow assoc[x; cdr[y]]]$$

An example is

$$assoc[X; ((W, (A, B)), (X, (C, D)), (Y, (E, F)))] = (C, D).$$

5. *sublis*$[x; y]$. Here x is assumed to have the form of a list of pairs $((u_1, v_1), \cdots, (u_n, v_n))$, where the u's are atomic, and y may be any S-expression. The value of *sublis*$[x; y]$ is the result of substituting each v for the corresponding u in y. In order to define sublis, we first define an auxiliary function. We have

$$sub2[x; z] = [null[x] \rightarrow z; eq[caar[x]; z] \rightarrow cadar[x]; T \rightarrow sub2[cdr[x]; z]]$$

and

$$sublis[x; y] = [atom[y] \rightarrow sub2[x; y]; T \rightarrow cons[sublis[x; car[y]]; sublis[x; cdr[y]]]$$

We have

sublis [((X, (A, B)), (Y, (B, C))); (A, X · Y)] = (A, (A, B), B, C)

e. *Representation of S-Functions by S-Expressions.* S-functions have been described by M-expressions. We now give a rule for translating M-expressions into S-expressions, in order to be able to use S-functions for making certain computations with S-functions and for answering certain questions about S-functions.

The translation is determined by the following rules in rich we denote the translation of an M-expression $\mathcal{E}$ by $\mathcal{E}^*$.

1. If $\mathcal{E}$ is an S-expression $\mathcal{E}^*$ is (QUOTE, $\mathcal{E}$).

2. Variables and function names that were represented by strings of lowercase letters are translated to the corresponding strings of the corresponding uppercase letters. Thus car* is CAR, and subst* is SUBST.

3. A form $f[e_1; \cdots; e_n]$ is translated to $(f^*, e_1^* \cdots, e_n^*)$. Thus cons [car [x]; cdr [x]]* is (CONS, (CAR, X), (CDR, X)).

4. $\{[p_1 \rightarrow e_1; \cdots; p_n \rightarrow e_n]\}^*$ is (COND, $(p_1^*, e_1^*), \cdots, (p_n^* \cdot e_n^*)$).

5. $\{\lambda[[x_1; \cdots; x_n]; \mathcal{E}]\}^*$ is (LAMBDA, $(x_1^*, \cdots, x_n^*), \mathcal{E}^*$).

6. $\{label[a; \mathcal{E}]\}^*$ is (LABEL, a*, $\mathcal{E}^*$).

With these conventions the substitution function whose M-expression is label [subst; λ [[x; y; z]; [atom [z] $\rightarrow$ [eq [y; z] $\rightarrow$ x; T $\rightarrow$ z]; T $\rightarrow$ cons [subst [x; y; car [z]]; subst [x; y; cdr [z]]]]]] has the S-expression

(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND ((ATOM, Z), (COND, (EQ, Y, Z), X), ((QUOTE, T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y, (CAR Z)), (SUBST, X, Y, (CDR, Z)))))))

This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular. If more characters were available on the computer, it could be improved considerably.[4]

f. *The Universal S-Function apply.* There is an S-function *apply* with the property that if f is an S-expression for an S-function f' and $args$ is a list of arguments of the form $(arg_1, \cdots, arg_n)$, where $arg_1, \cdots, arg_n$ are arbitrary S-expressions, then $apply[f; args]$ and $f'[arg_1; \cdots; arg_n]$ are defined for the same values of $arg_1, \cdots, arg_n$, and are equal when defined. For example,

$$\lambda[[x; y]; cons[car[x]; y]][(A, B); (C, D)]$$

$$= apply[(LAMBDA, (X, Y), (CONS, (CAR, X), Y)); ((A, B), (C, D))] = (A, C, D)$$

The S-function *apply* is defined by

$$apply[f; args] = eval[cons[f; appq[args]]; NIL],$$

where

$$appq[m] = [null[m] \rightarrow NIL; T \rightarrow cons[list[QUOTE; car[m]]; appq[cdr[m]]]]$$

and

eval[e; a] = [

[4]1995: More characters were made available on SAIL and later on the Lisp machines. Alas, the world went back to inferior character sets again—though not as far back as when this paper was written in early 1959.

atom [e] → assoc [e; a];

atom [car [e]] → [

eq [car [e]; QUOTE] → cadr [e];

eq [car [e]; ATOM] → atom [eval [cadr [e]; a]];

eq [car [e]; EQ] → [eval [cadr [e]; a] = eval [caddr [e]; a]];

eq [car [e]; COND] → evcon [cdr [e]; a];

eq [car [e]; CAR] → car [eval [cadr [e]; a]];

eq [car [e]; CDR] → cdr [eval [cadr [e]; a]];

eq [car [e]; CONS] → cons [eval [cadr [e]; a]; eval [caddr [e];

a]]; T → eval [cons [assoc [car [e]; a];

evlis [cdr [e]; a]]; a]];

eq [caar [e]; LABEL] → eval [cons [caddar [e]; cdr [e]];

cons [list [cadar [e]; car [e]; a]];

eq [caar [e]; LAMBDA] → eval [caddar [e];

append [pair [cadar [e]; evlis [cdr [e]; a]; a]]]

and

$$evcon[c; a] = [eval[caar[c]; a] \rightarrow eval[cadar[c]; a]; T \rightarrow evcon[cdr[c]; a]]$$

and

$$evlis[m; a] = [null[m] \rightarrow NIL; T \rightarrow cons[eval[car[m]; a]; evlis[cdr[m]; a]]]$$

We now explain a number of points about these definitions. [5]

1. *apply* itself forms an expression representing the value of the function applied to the arguments, and puts the work of evaluating this expression onto a function *eval*. It uses *appq* to put quotes around each of the arguments, so that *eval* will regard them as standing for themselves.

2. $eval[e; a]$ has two arguments, an expression e to be evaluated, and a list of pairs a. The first item of each pair is an atomic symbol, and the second is the expression for which the symbol stands.

3. If the expression to be evaluated is atomic, eval evaluates whatever is paired with it first on the list a.

4. If e is not atomic but $car[e]$ is atomic, then the expression has one of the forms $(QUOTE, e)$ or $(ATOM, e)$ or (EQ, e_1, e_2) or $(COND, (p_1, e_1), \cdots, (p_n, e_n))$, or (CAR, e) or (CDR, e) or $(CONS, e_1, e_2)$ or $(f, e_1, \cdots, e_n)$ where f is an atomic symbol.

In the case $(QUOTE, e)$ the expression e, itself, is taken. In the case of $(ATOM, e)$ or (CAR, e) or (CDR, e) the expression e is evaluated and the appropriate function taken. In the case of (EQ, e_1, e_2) or $(CONS, e_1, e_2)$ two expressions have to be evaluated. In the case of $(COND, (p_1, e_1), \cdots (p_n, e_n))$ the p's have to be evaluated in order until a true p is found, and then the corresponding e must be evaluated. This is accomplished by *evcon*. Finally, in the case of $(f, e_1, \cdots, e_n)$ we evaluate the expression that results from replacing f in this expression by whatever it is paired with in the list a.

5. The evaluation of $((LABEL, f, \mathcal{E}), e_1, \cdots, e_n)$ is accomplished by evaluating $(\mathcal{E}, e_1, \cdots, e_n)$ with the pairing $(f, (LABEL, f, \mathcal{E}))$ put on the front of the previous list a of pairs.

6. Finally, the evaluation of $((LAMBDA, (x_1, \cdots, x_n), \mathcal{E}), e_1, \cdots e_n)$ is accomplished by evaluating $\mathcal{E}$ with the list of pairs $((x_1, e_1), \cdots, ((x_n, e_n))$ put on the front of the previous list a.

The list a could be eliminated, and LAMBDA and LABEL expressions evaluated by substituting the arguments for the variables in the expressions $\mathcal{E}$. Unfortunately, difficulties involving collisions of bound variables arise, but they are avoided by using the list a.

[5]1995: This version isn't quite right. A comparison of this and other versions of *eval* including what was actually implemented (and debugged) is given in "The Influence of the Designer on the Design" by Herbert Stoyan and included in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Vladimir Lifschitz (ed.), Academic Press, 1991

Calculating the values of functions by using *apply* is an activity better suited to electronic computers than to people. As an illustration, however, we now give some of the steps for calculating
apply [(LABEL, FF, (LAMBDA, (X), (COND, (ATOM, X), X), ((QUOTE, T),(FF, (CAR, X))))));((A· B))] = A
The first argument is the S-expression that represents the function ff defined in section 3d. We shall abbreviate it by using the letter ϕ. We have

apply [ϕ; ((A·B))]
= eval [((LABEL, FF, ψ), (QUOTE, (A·B))); NIL]

where ψ is the part of ϕ beginning (LAMBDA

= eval[((LAMBDA, (X), ω), (QUOTE, (A·B)));((FF, ϕ))]

where ω is the part of ψ beginning (COND

= eval [(COND, $(\pi_1, \epsilon_1), (\pi_2, \epsilon_2)$); ((X, (QUOTE, (A·B))), (FF, ϕ))]

Denoting ((X, (QUOTE, (A·B))), (FF, ϕ)) by a, we obtain

= evcon [((π_1, ϵ_1), (π_2, ϵ_2)); a]

This involves eval [$\pi_1; a$]

= eval [(ATOM, X); a]

= atom [eval [X; a]]

= atom [eval [assoc [X; ((X, (QUOTE, (A·B))), (FF,ϕ))];a]]

= atom [eval [(QUOTE, (A·B)); a]]

= atom [(A·B)],

= F

Our main calulation continues with

apply [ϕ; ((A·B))]

= evcon $[((\pi_2, \epsilon_2,)); a]$,

which involves eval $[\pi_2; a]$ = eval [(QUOTE, T); a] = T

Our main calculation again continues with

apply [ϕ; ((A·B))]

= eval $[\epsilon_2; a]$

= eval [(FF, (CAR, X));a]

= eval [Cons [ϕ; evlis [((CAR, X)); a]]; a]

Evaluating evlis [((CAR, X)); a] involves

eval [(CAR, X); a]

= car [eval [X; a]]

= car [(A·B)], where we took steps from the earlier computation of atom [eval [X; a]] = A,

and so evlis [((CAR, X)); a] then becomes

list [list [QUOTE; A]] = ((QUOTE, A)),

and our main quantity becomes

= eval [(ϕ, (QUOTE, A)); a]

The subsequent steps are made as in the beginning of the calculation. The LABEL and LAMBDA cause new pairs to be added to a, which gives a new list of pairs a_1. The π_1 term of the conditional eval [(ATOM, X); a_1] has the value T because X is paired with (QUOTE, A) first in a_1, rather than with (QUOTE, (A·B)) as in a.

Therefore we end up with eval [X; a_1] from the *evcon*, and this is just A.

g. *Functions with Functions as Arguments.* There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions. One such function is $maplist[x; f]$ with an S-expression argument x and an argument f that is a function from S-expressions to S-expressions. We define

$$maplist[x; f] = [null[x] \rightarrow NIL; T \rightarrow cons[f[x]; maplist[cdr[x]; f]]]$$

The usefulness of *maplist* is illustrated by formulas for the partial derivative with respect to x of expressions involving sums and products of x and other variables. The S-expressions that we shall differentiate are formed as follows.

1. An atomic symbol is an allowed expression.

2. If $e_1, e_2, \cdots, e_n$ are allowed expressions, (PLUS, $e_1, \cdots, e_n$) and (TIMES, $e_1, \cdots, e_n$) are also, and represent the sum and product, respectively, of $e_1, \cdots, e_n$.

This is, essentially, the Polish notation for functions, except that the inclusion of parentheses and commas allows functions of variable numbers of arguments. An example of an allowed expression is (TIMES, X, (PLUS, X, A), Y), the conventional algebraic notation for which is X(X + A)Y.

Our differentiation formula, which gives the derivative of y with respect to x, is

diff [y; x] = [atom [y] → [eq [y; x] → ONE; T → ZERO]; eq [car [Y]; PLUS] → cons [PLUS; maplist [cdr [y]; λ[[z]; diff [car [z]; x]]]]; eq[car [y]; TIMES] → cons[PLUS; maplist[cdr[y]; λ[[z]; cons [TIMES; maplist[cdr [y]; λ[[w]; ¬ eq [z; w] → car [w]; T → diff [car [[w]; x]]]]]]]

The derivative of the expression (TIMES, X, (PLUS, X, A), Y), as computed by this formula, is

(PLUS, (TIMES, ONE, (PLUS, X, A), Y), (TIMES, X, (PLUS, ONE, ZERO), Y), (TIMES, X, (PLUS, X, A), ZERO))

Besides *maplist*, another useful function with functional arguments is *search*, which is defined as

$$search[x; p; f; u] = [null[x] \rightarrow u; p[x] \rightarrow f[x]; T \rightarrow search[cdr[x]; p; f; u]$$

The function *search* is used to search a list for an element that has the property p, and if such an element is found, f of that element is taken. If there is no such element, the function u of no arguments is computed.

4 The LISP Programming System

The LISP programming system is a system for using the IBM 704 computer to compute with symbolic information in the form of S-expressions. It has been or will be used for the following purposes:

l. Writing a compiler to compile LISP programs into machine language.

2. Writing a program to check proofs in a class of formal logical systems.

3. Writing programs for formal differentiation and integration.

4. Writing programs to realize various algorithms for generating proofs in predicate calculus.

5. Making certain engineering calculations whose results are formulas rather than numbers.

6. Programming the Advice Taker system.

The basis of the system is a way of writing computer programs to evaluate S-functions. This will be described in the following sections.

In addition to the facilities for describing S-functions, there are facilities for using S-functions in programs written as sequences of statements along the lines of FORTRAN (4) or ALGOL (5). These features will not be described in this article.

a. *Representation of S-Expressions by List Structure.* A list structure is a collection of computer words arranged as in figure 1a or 1b. Each word of the list structure is represented by one of the subdivided rectangles in the figure. The $left$ box of a rectangle represents the *address* field of the word and the *right* box represents the *decrement* field. An arrow from a box to another rectangle means that the field corresponding to the box contains the location of the word corresponding to the other rectangle.

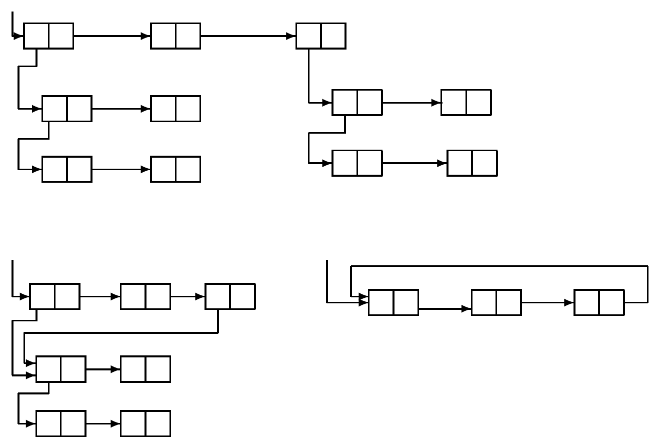


Fig. 1

It is permitted for a substructure to occur in more than one place in a list structure, as in figure 1b, but it is not permitted for a structure to have cycles, as in figure 1c. An atomic symbol is represented in the computer by a list structure of special form called the *association list* of the symbol. The address field of the first word contains a special constant which enables the program to tell that this word represents an atomic symbol. We shall describe association lists in section 4b.

An S-expression x that is not atomic is represented by a word, the address and decrement parts of which contain the locations of the subexpressions $car[x]$ and $cdr[x]$, respectively. If we use the symbols A, B, etc. to denote the locations of the association list of these symbols, then the S-expression $((A \cdot B) \cdot (C \cdot (E \cdot F)))$ is represented by the list structure a of figure 2. Turning to the list form of S-expressions, we see that the S-expression $(A, (B, C), D)$, which is an abbreviation for $(A \cdot ((B \cdot (C \cdot NIL)) \cdot (D \cdot NIL)))$, is represented by the list structure of figure 2b.

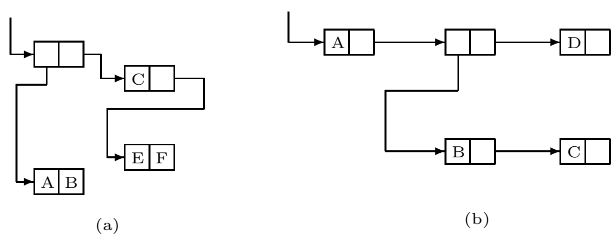


Figure 2

When a list structure is regarded as representing a list, we see that each term of the list occupies the address part of a word, the decrement part of which *points* to the word containing the next term, while the last word has NIL in its decrement.

An expression that has a given subexpression occurring more than once can be represented in more than one way. Whether the list structure for the subexpression is or is not repeated depends upon the history of the program. Whether or not a subexpression is repeated will make no difference in the results of a program as they appear outside the machine, although it will affect the time and storage requirements. For example, the S-expression ((A·B)·(A·B)) can be represented by either the list structure of figure 3a or 3b.

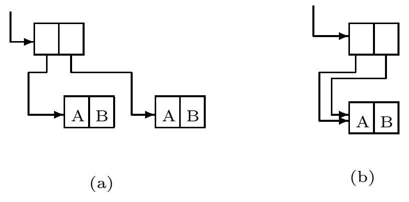


Figure 3

The prohibition against circular list structures is essentially a prohibition against an expression being a subexpression of itself. Such an expression could not exist on paper in a world with our topology. Circular list structures would have some advantages in the machine, for example, for representing recursive functions, but difficulties in printing them, and in certain other operations, make it seem advisable not to use them for the present.

The advantages of list structures for the storage of symbolic expressions are:

1. The size and even the number of expressions with which the program will have to deal cannot be predicted in advance. Therefore, it is difficult to arrange blocks of storage of fixed length to contain them.

2. Registers can be put back on the free-storage list when they are no longer needed. Even one register returned to the list is of value, but if expressions are stored linearly, it is difficult to make use of blocks of registers of odd sizes that may become available.

3. An expression that occurs as a subexpression of several expressions need be represented in storage only once.

b. *Association Lists*[6] . In the LISP programming system we put more in the association list of a symbol than is required by the mathematical system described in the previous sections. In fact, any information that we desire to associate with the symbol may be put on the association list. This information may include: the *print name*, that is, the string of letters and digits which represents the symbol outside the machine; a numerical value if the symbol represents a number; another S-expression if the symbol, in some way, serves as a name for it; or the location of a routine if the symbol represents a function for which there is a machine-language subroutine. All this implies that in the machine system there are more primitive entities than have been described in the sections on the mathematical system.

For the present, we shall only describe how *print names* are represented on association lists so that in reading or printing the program can establish a correspondence between information on punched cards, magnetic tape or printed page and the list structure inside the machine. The association list of the symbol DIFFERENTIATE has a segment of the form shown in figure 4. Here *pname* is a symbol that indicates that the structure for the print name of the symbol whose association list this is hanging from the next word on the association list. In the second row of the figure we have a list of three words. The address part of each of these words points to a Word containing

[6]1995: These were later called property lists.

six 6-bit characters. The last word is filled out with a 6-bit combination that does not represent a character printable by the computer. (Recall that the IBM 7O4 has a 36-bit word and that printable characters are each represented by 6 bits.) The presence of the words with character information means that the association lists do not themselves represent S-expressions, and that only some of the functions for dealing with S-expressions make sense within an association list.

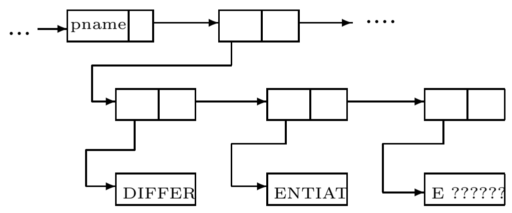


Figure 4

c. *Free-Storage List.* At any given time only a part of the memory reserved for list structures will actually be in use for storing S-expressions. The remaining registers (in our system the number, initially, is approximately 15,000) are arranged in a single list called the *free-storage list.* A certain register, FREE, in the program contains the location of the first register in this list. When a word is required to form some additional list structure, the first word on the *free-storage list* is taken and the number in register FREE is changed to become the location of the second word on the free-storage list. No provision need be made for the user to program the return of registers to the free-storage list.

This return takes place automatically, approximately as follows (it is necessary to give a simplified description of this process in this report): There is a fixed set of base registers in the program which contains the locations of list structures that are accessible to the program. Of course, because list structures branch, an arbitrary number of registers may be involved. Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of *car* and *cdr* operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a $car - cdr$ chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program; hence its contents are no longer of interest, and so we would like to have it back on the free-storage list. This comes about in the following way.

Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation[7] cycle starts.

First, the program finds all registers accessible from the base registers and makes their signs negative. This is accomplished by starting from each of the base registers and changing the sign of every register that can be reached from it by a $car - cdr$ chain. If the program encounters a register in this process which already has a negative sign, it assumes that this register has already been reached.

After all of the accessible registers have had their signs changed, the program goes through the area of memory reserved for the storage of list structures and puts all the registers whose signs were not changed in the previous step back on the free-storage list, and makes the signs of the accessible registers positive again.

This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists. Its efficiency depends upon not coming close to exhausting the available memory with accessible lists. This is because the reclamation process requires several seconds to execute, and therefore must result in the addition of at least several thousand registers to the free-storage list if the program is not to spend most of its time in reclamation.

d. *Elementary S-Functions in the Computer.* We shall now describe the computer representations of *atom*, $=$, *car*, *cdr*, and *cons*. An S-expression is communicated to the program that represents a function as the location of the word representing it, and the programs give S-expression answers in the same form.

atom. As stated above, a word representing an atomic symbol has a special

[7]We already called this process "garbage collection", but I guess I chickened out of using it in the paper—or else the Research Laboratory of Electronics grammar ladies wouldn't let me.

constant in its address part: *atom* is programmed as an open subroutine that tests this part. Unless the M-expression *atom*$[e]$ occurs as a condition in a conditional expression, the symbol T or F is generated as the result of the test. In case of a conditional expression, a conditional transfer is used and the symbol T or F is not generated.

eq. The program for *eq*$[e; f]$ involves testing for the numerical equality of the locations of the words. This works because each atomic symbol has only one association list. As with *atom*, the result is either a conditional transfer or one of the symbols T or F.

car. Computing *car*$[x]$ involves getting the contents of the address part of register x. This is essentially accomplished by the single instruction CLA 0, i, where the argument is in index register, and the result appears in the address part of the accumulator. (We take the view that the places from which a function takes its arguments and into which it puts its results are prescribed in the definition of the function, and it is the responsibility of the programmer or the compiler to insert the required datamoving instructions to get the results of one calculation in position for the next.) ("car" is a mnemonic for "contents of the address part of register.")

cdr. *cdr* is handled in the same way as *car*, except that the result appears in the decrement part of the accumulator ("cdr" stands for "contents of the decrement part of register.")

cons. The value of *cons*$[x; y]$ must be the location of a register that has x and y in its address and decrement parts, respectively. There may not be such a register in the computer and, even if there were, it would be time-consuming to find it. Actually, what we do is to take the first available register from the *free-storage list*, put x and y in the address and decrement parts, respectively, and make the value of the function the location of the register taken. ("cons" is an abbreviation for "construct.")

It is the subroutine for *cons* that initiates the reclamation when the *free-storage list* is exhausted. In the version of the system that is used at present *cons* is represented by a closed subroutine. In the compiled version, *cons* is open.

e. *Representation of S-Functions by Programs.* The compilation of functions that are compositions of *car*, *cdr*, and *cons*, either by hand or by a compiler program, is straightforward. Conditional expressions give no trouble except that they must be so compiled that only the p's and e's that are required are computed. However, problems arise in the compilation of recursive functions.

In general (we shall discuss an exception), the routine for a recursive function uses itself as a subroutine. For example, the program for *subst*$[x; y; z]$ uses itself as a subroutine to evaluate the result of substituting into the subexpressions *car*$[z]$ and *cdr*$[z]$. While *subst*$[x; y;$ *cdr*$[z]]$ is being evaluated, the result of the previous evaluation of *subst*$[x; y;$ *car*$[z]]$ must be saved in a temporary storage register. However, *subst* may need the same register for evaluating *subst*$[x; y;$ *cdr*$[z]]$. This possible conflict is resolved by the SAVE and UNSAVE routines that use the *public push-down list* [8]. The SAVE routine is entered at the beginning of the routine for the recursive function with a request to save a given set of consecutive registers. A block of registers called the *public push-down list* is reserved for this purpose. The SAVE routine has an index that tells it how many registers in the push-down list are already in use. It moves the contents of the registers which are to be saved to the first unused registers in the push-down list, advances the index of the list, and returns to the program from which control came. This program may then freely use these registers for temporary storage. Before the routine exits it uses UNSAVE, which restores the contents of the temporary registers from the push-down list and moves back the index of this list. The result of these conventions is described, in programming terminology, by saying that the recursive subroutine is transparent to the temporary storage registers.

f. *Status of the LISP Programming System* (February 1960). A variant of the function apply described in section 5f has been translated into a program APPLY for the IBM 704. Since this routine can compute values of S-functions given their descriptions as S-expressions and their arguments, it serves as an interpreter for the LISP programming language which describes computation processes in this way.

The program APPLY has been imbedded in the LISP programming system which has the following features:

1. The programmer may define any number of S-functions by S-expressions. these functions may refer to each other or to certain S-functions represented by machine language program.

2. The values of defined functions may be computed.

3. S-expressions may be read and printed (directly or via magnetic tape).

[8] 1995: now called a stack

4. Some error diagnostic and selective tracing facilities are included.

5. The programmer may have selected S-functions compiled into machine language programs put into the core memory. Values of compiled functions are computed about 60 times as fast as they would if interpreted. Compilation is fast enough so that it is not necessary to punch compiled program for future use.

6. A "program feature" allows programs containing assignment and **go to** statements in the style of ALGOL.

7. Computation with floating point numbers is possible in the system, but this is inefficient.

8. A programmer's manual is being prepared. The LISP programming system is appropriate for computations where the data can conveniently be represented as symbolic expressions allowing expressions of the same kind as subexpressions. A version of the system for the IBM 709 is being prepared.

5 Another Formalism for Functions of Symbolic Expressions

There are a number of ways of defining functions of symbolic expressions which are quite similar to the system we have adopted. Each of them involves three basic functions, conditional expressions, and recursive function definitions, but the class of expressions corresponding to S-expressions is different, and so are the precise definitions of the functions. We shall describe one of these variants called linear LISP.

The L-expressions are defined as follows:

1. A finite list of characters is admitted.

2. Any string of admitted characters in an L-expression. This includes the null string denoted by Λ.

There are three functions of strings:

1. $first[x]$ is the first character of the string x.

$first[\Lambda]$ is undefined. For example: $first[ABC] = A$

2. $rest[x]$ is the string of characters which remains when the first character of the string is deleted.

$rest[\Lambda]$ is undefined. For example: $rest[ABC] = BC$.

3. $combine[x; y]$ is the string formed by prefixing the character x to the string y. For example: $combine[A; BC] = ABC$

There are three predicates on strings:
1. $char[x]$, x is a single character.
2. $null[x]$, x is the null string.
3. $x = y$, defined for x and y characters.

The advantage of linear LISP is that no characters are given special roles, as are parentheses, dots, and commas in LISP. This permits computations with all expressions that can be written linearly. The disadvantage of linear LISP is that the extraction of subexpressions is a fairly involved, rather than an elementary, operation. It is not hard to write, in linear LISP, functions that correspond to the basic functions of LISP, so that, mathematically, linear LISP includes LISP. This turns out to be the most convenient way of programming, in linear LISP, the more complicated manipulations. However, if the functions are to be represented by computer routines, LISP is essentially faster.

6 Flowcharts and Recursion

Since both the usual form of computer program and recursive function definitions are universal computationally, it is interesting to display the relation between them. The translation of recursive symbolic functions into computer programs was the subject of the rest of this report. In this section we show how to go the other way, at least in principle.

The state of the machine at any time during a computation is given by the values of a number of variables. Let these variables be combined into a vector ξ. Consider a program block with one entrance and one exit. It defines and is essentially defined by a certain function f that takes one machine configuration into another, that is, f has the form $\xi' = f(\xi)$. Let us call f the associated function of the program block. Now let a number of such blocks be combined into a program by decision elements π that decide after each block is completed which block will be entered next. Nevertheless, let the whole program still have one entrance and one exit.

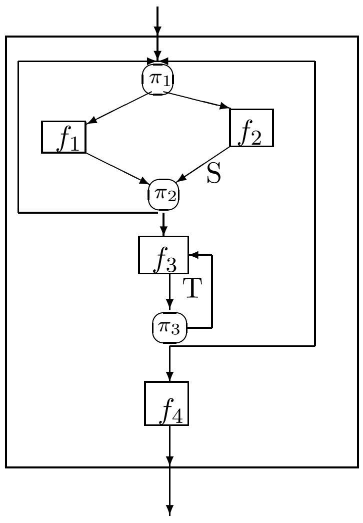


Figure 5

We give as an example the flowcart of figure 5. Let us describe the function $r[\xi]$ that gives the transformation of the vector ξ between entrance and exit of the whole block. We shall define it in conjunction with the functions s(ξ), and t[ξ], which give the transformations that ξ undergoes between the points S and T, respectively, and the exit. We have

$$\begin{aligned} r[\xi] &= [\pi_1 1[\xi] \to S[f_1[\xi]]; T \to S[f_2[\xi]]] \\ S[\xi] &= [\pi_2 1[\xi] \to r[\xi]; T \to t[f_3[\xi]]] \\ t[\xi] &= [\pi 3I[\xi] \to f_4[\xi]; \pi_3 2[\xi] \to r[\xi]; T \to t[f_3[\xi]]] \end{aligned}$$

Given a flowchart with a single entrance and a single exit, it is easy to write down the recursive function that gives the transformation of the state vector from entrance to exit in terms of the corresponding functions for the computation blocks and the predicates of the branch. In general, we proceed as follows.

In figure 6, let β be an n-way branch point, and let $f_1, \cdots, f_n$ be the computations leading to branch points $\beta_1, \beta_2, \cdots, \beta_n$. Let ϕ be the function that transforms ξ between β and the exit of the chart, and let $\phi_1, \cdots, \phi_n$ be the corresponding functions for $\beta_1, \cdots, \beta_n$. We then write

$$\phi[\xi] = [p_1[\xi] \to \phi_1[f_1[\xi]]; \cdots; p_n[\xi] \to \phi_n[\xi]]]$$

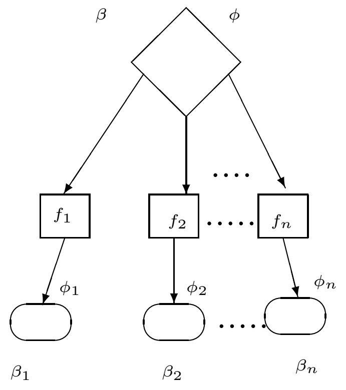


Figure 6

7 Acknowledgments

The inadequacy of the λ-notation for naming recursive functions was noticed by N. Rochester, and he discovered an alternative to the solution involving *label* which has been used here. The form of subroutine for *cons* which permits its composition with other functions was invented, in connection with another programming system, by C. Gerberick and H. L. Gelernter, of IBM Corporation. The LlSP programming system was developed by a group including R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, J. McCarthy, D. Park, S. Russell.

The group was supported by the M.I.T. Computation Center, and by the M.I.T. Research Laboratory of Electronics (which is supported in part by the the U.S. Army (Signal Corps), the U.S. Air Force (Office of Scientific Research, Air Research and Development Command), and the U.S. Navy (Office of Naval Research)). The author also wishes to acknowledge the personal financial support of the Alfred P. Sloan Foundation.

REFERENCES

1. J. McCARTHY, Programs with common sense, Paper presented at the Symposium on the Mechanization of Thought Processes, National Physical Laboratory, Teddington, England, Nov. 24-27, 1958. (Published in Proceedings of the Symposium by H. M. Stationery Office).

2. A. NEWELL AND J. C. SHAW, Programming the logic theory machine, Proc. Western Joint Computer Conference, Feb. 1957.

3. A. CHURCH, *The Calculi of Lambda-Conversion* (Princeton University Press, Princeton, N. J., 1941).

4. FORTRAN Programmer's Reference Manual, IBM Corporation, New York, Oct. 15, 1956.

5. A. J. PERLIS AND K. SAMELS0N, International algebraic language, Preliminary Report, *Comm. Assoc. Comp. Mach.*, Dec. 1958.

Operating Systems   R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.

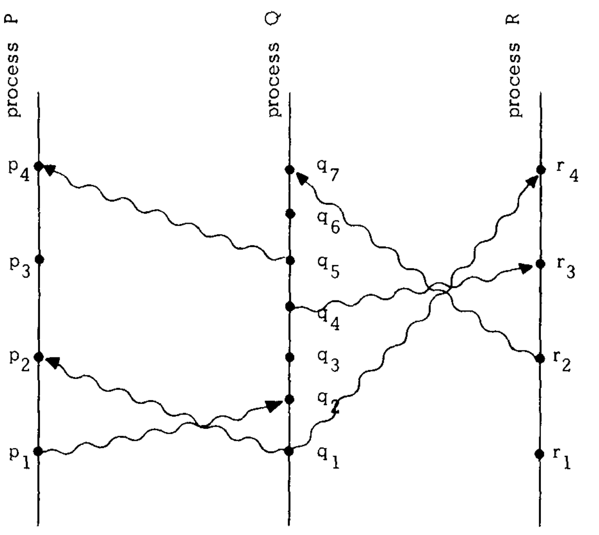
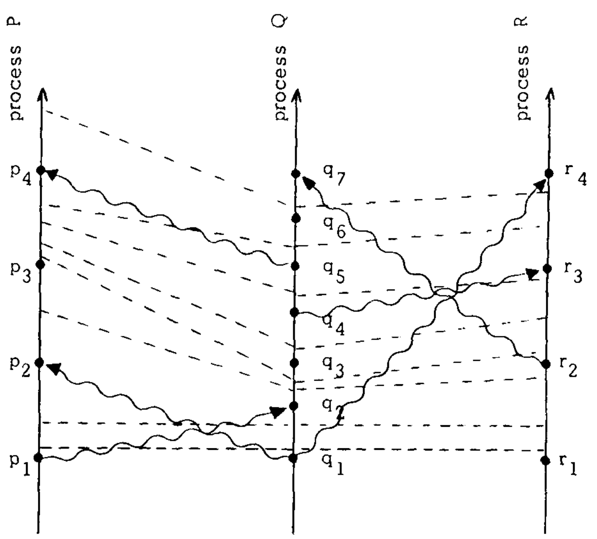


Fig. 2.



event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.[1] It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the "happened before" relation, denoted by "$\rightarrow$", as follows.

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a. (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a "space-time diagram" such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.[2] It is easy to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure 1.

Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3, process P cannot know what process Q did at q_3 until it receives the message at p_4. (Before event p_4, P could at most know what Q was *planning* to do at q_3.)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

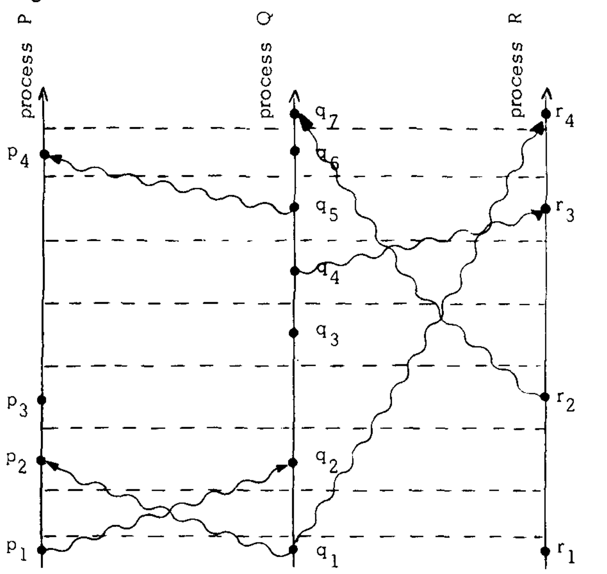
Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a \rangle$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C\langle b \rangle$, where $C\langle b \rangle = C_j\langle b \rangle$ if b is an event in process P_j. For now, we make no assumption about the relation of the numbers $C_i\langle a \rangle$ to physical time, so we can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

[1] The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process' message-receiving events.

[2] Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

Fig. 3.



We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event a occurs before another event b, then a should happen at an earlier time than b. We state this condition more formally as follows.

Clock Condition. For any events a, b:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1, p_2 and p_3 are both concurrent with q_3, so this would mean that they both must occur at the same time as q_3, which would contradict the Clock Condition because $p_2 \rightarrow p_3$.

It is easy to see from our definition of the relation "$\rightarrow$" that the Clock Condition is satisfied if the following two conditions hold.

C1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$.

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

Let us consider the clocks in terms of a space-time diagram. We imagine that a process' clock "ticks" through every number, with the ticks occurring between the process' events. For example, if a and b are consecutive events in process P_i with $C_i\langle a\rangle = 4$ and $C_i\langle b\rangle = 7$, then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed "tick line" through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and condition C2 means that every message line must cross a tick line. From the pictorial meaning of $\rightarrow$, it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process P_i's clock is represented by a register C_i, so that $C_i\langle a\rangle$ is the value contained by C_i during the event a. The value of C_i will change between events, so changing C_i does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

IR1. Each process P_i increments C_i between any two successive events.

To meet condition C2, we require that each message m contain a *timestamp* T_m which equals the time at which the message was sent. Upon receiving a message timestamped T_m, a process must advance its clock to be later than T_m. More precisely, we have the following rule.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

In IR2(b) we consider the event which represents the receipt of the message m to occur after the setting of C_j. (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times at which they occur. To break ties, we use any arbitrary total ordering $<$ of the processes. More precisely, we define a relation $\Rightarrow$ as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either (i) $C_i\langle a\rangle < C_j\langle b\rangle$ or (ii) $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i < P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, the relation $\Rightarrow$ is a way of completing the "happened before" partial ordering to a total ordering.[3]

The ordering $\Rightarrow$ depends upon the system of clocks C_i, and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations $\Rightarrow$. Given any total ordering relation $\Rightarrow$ which extends $\rightarrow$, there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering $\rightarrow$ which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.[4] Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let P_0 be the scheduling process. Suppose P_1 sends a request to P_0 and then sends a message to P_2. Upon receiving the latter message, P_2 sends a request to P_0. It is possible for P_2's request to reach P_0 before P_1's request does. Condition II is then violated if P_2's request is granted first.

To solve the problem, we implement a system of clocks with rules IR1 and IR2, and use them to define a total ordering $\Rightarrow$ of all events. This provides a total ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes' operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message T_0:P_0 *requests resource*, where P_0 is the process initially granted the resource and T_0 is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.
2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]
3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.
4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.
5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m.[6]

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

It is easy to verify that the algorithm defined by these rules satisfies conditions I–III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that P_i has learned about all requests which preceded its current

[3] The ordering $<$ establishes a priority among the processes. If a "fairer" method is desired, then $<$ can be made a function of the clock value. For example, if $C_i\langle a\rangle = C_j\langle b\rangle$ and $j < i$, then we can let $a \Rightarrow b$ if $j < C_i\langle a\rangle \bmod N \leq i$, and $b \Rightarrow a$ otherwise; where N is the total number of processes.

[4] The term "eventually" should be made precise, but that would require too long a diversion from our main topic.

[5] This acknowledgment message need not be sent if P_j has already sent a message to P_i timestamped later than T_m.

[6] If $P_i < P_j$, then P_i need only have received a message timestamped $\geq T_m$ from P_j.

request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering $\Rightarrow$ extends the partial ordering $\rightarrow$. Rule 2 guarantees that after P_i requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set C of possible commands, a set S of possible states, and a function $\mathbf{e}$: $C \times S \rightarrow S$. The relation $\mathbf{e}(C, S) = S'$ means that executing the command C with the machine in state S causes the machine state to change to S'. In our example, the set C consists of all the commands P_i *requests resource* and P_i *releases resource*, and the state consists of a queue of waiting *request* commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.[7]

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation $\Rightarrow$), so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

[7] If each process does not strictly alternate *request* and *release* commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering $\Rightarrow$. This permits the following type of "anomalous behavior." Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A, and then telephones a friend in another city to have him issue a request B on a different computer B. It is quite possible for request B to receive a lower timestamp and be ordered before request A. This can happen because the system has no way of knowing that A actually preceded B, since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let $\mathcal{S}$ be the set of all system events. Let us introduce a set of events which contains the events in $\mathcal{S}$ together with all other relevant external events, such as the phone calls in our example. Let $\twoheadrightarrow$ denote the "happened before" relation for $\underline{\mathcal{S}}$. In our example, we had A $\twoheadrightarrow$ B, but A $\not\twoheadrightarrow$ B. It is obvious that no algorithm based entirely upon events in $\mathcal{S}$, and which does not relate those events in any way with the other events in $\underline{\mathcal{S}}$, can guarantee that request A is ordered before request B.

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering $\twoheadrightarrow$. In our example, the person issuing request A could receive the timestamp T_A of that request from the system. When issuing request B, his friend could specify that B be given a timestamp later than T_A. This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

Strong Clock Condition. For any events a, b in $\mathcal{S}$:
if $a \twoheadrightarrow b$ then $C\langle a \rangle < C\langle b \rangle$.

This is stronger than the ordinary Clock Condition because $\twoheadrightarrow$ is a stronger relation than $\rightarrow$. It is not in general satisfied by our logical clocks.

Let us identify $\underline{\mathcal{S}}$ with some set of "real" events in physical space-time, and let $\twoheadrightarrow$ be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let $C_i(t)$ denote the reading of the clock C_i at physical time t.[8] For mathematical con-

[8] We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then $C_i(t)$ must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

venience, we assume that the clocks run continuously rather than in discrete "ticks." (A discrete clock can be thought of as a continuous one in which there is an error of up to $\frac{1}{2}$ "tick" in reading it.) More precisely, we assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset. Then $dC_i(t)/dt$ represents the rate at which the clock is running at time t.

In order for the clock C_i to be a true physical clock, it must run at approximately the correct rate. That is, we must have $dC_i(t)/dt \approx 1$ for all t. More precisely, we will assume that the following condition is satisfied:

PC1. There exists a constant $\kappa \ll 1$ such that for all i: $|dC_i(t)/dt - 1| < \kappa$.

For typical crystal controlled clocks, $\kappa \leq 10^{-6}$.

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j, and t. More precisely, there must be a sufficiently small constant ϵ so that the following condition holds:

PC2. For all i, j: $|C_i(t) - C_j(t)| < \epsilon$.

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than ϵ.

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure that PC2 always holds. First, however, let us examine how small κ and ϵ must be to prevent anomalous behavior. We must insure that the system $\mathcal{S}$ of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when a and b are events in $\mathcal{S}$ with $a \not\rightarrow b$. Hence, we need only consider events occurring in different processes.

Let μ be a number such that if event a occurs at physical time t and event b in another process satisfies $a \rightarrow b$, then b occurs later than physical time $t + \mu$. In other words, μ is less than the shortest transmission time for interprocess messages. We can always choose μ equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in $\mathcal{S}$ are transmitted, μ could be significantly larger.

To avoid anomalous behavior, we must make sure that for any i, j, and t: $C_i(t + \mu) - C_j(t) > 0$. Combining this with PC1 and 2 allows us to relate the required smallness of κ and ϵ to the value of μ as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$. Using PC2, it is then easy to deduce that $C_i(t + \mu) - C_j(t) > 0$ if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let m be a message which is sent at physical time t and received at time t'. We define $\nu_m = t' - t$ to be the *total delay* of the message m. This delay will, of course, not be known to the process which receives m. However, we assume that the receiving process knows some *minimum delay* $\mu_m \geq 0$ such that $\mu_m \leq \nu_m$. We call $\xi_m = \nu_m - \mu_m$ the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

IR1'. For each i, if P_i does not receive a message at physical time t, then C_i is differentiable at t and $dC_i(t)/dt > 0$.

IR2'. (a) If P_i sends a message m at physical time t, then m contains a timestamp $T_m = C_i(t)$. (b) Upon receiving a message m at time t', process P_j sets $C_j(t')$ equal to maximum $(C_j(t' - 0), T_m + \mu_m)$.[9]

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process P_i to process P_j represents a communication line over which messages are sent directly from P_i to P_j. We say that a message is sent over this arc every τ seconds if for any t, P_i sends at least one message to P_j between physical times t and $t + \tau$. The *diameter* of the directed graph is the smallest number d such that for any pair of distinct processes P_j, P_k, there is a path from P_j to P_k having at most d arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

THEOREM. Assume a strongly connected graph of processes with diameter d which always obeys rules IR1' and IR2'. Assume that for any message m, $\mu_m \leq \mu$ for some constant μ, and that for all $t \geq t_0$: (a) PC1 holds. (b) There are constants τ and ξ such that every τ seconds a message with an unpredictable delay less than ξ is sent over every arc. Then PC2 is satisfied with $\epsilon \approx d(2\kappa\tau + \xi)$ for all $t \gtrsim t_0 + \tau d$, where the approximations assume $\mu + \xi \ll \tau$.

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an intro-

[9] $C_j(t' - 0) = \lim_{\delta \to 0} C_j(t' - |\delta|)$.

duction to the subject. The methods described in the literature are useful for estimating the message delays μ_m and for adjusting the clock frequencies dC_i/dt (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

Conclusion

We have seen that the concept of "happening before" defines an invariant partial ordering of the events in a distributed multiprocess system. We described an algorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system's users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

Appendix

Proof of the Theorem

For any i and t, let us define C_i^t to be a clock which is set equal to C_i at time t and runs at the same rate as C_i, but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt]dt \qquad (1)$$

for all $t' \ge t$. Note that

$$C_i(t') \ge C_i^t(t') \text{ for all } t' \ge t. \qquad (2)$$

Suppose process P_1 at time t_1 sends a message to process P_2 which is received at time t_2 with an unpredictable delay $\le \xi$, where $t_0 \le t_1 \le t_2$. Then for all $t \ge t_2$ we have:

$$\begin{aligned} C_2^{t_2}(t) &\ge C_2^{t_2}(t_2) + (1-\kappa)(t - t_2) && \text{[by (1) and PC1]} \\ &\ge C_1(t_1) + \mu_m + (1-\kappa)(t-t_2) && \text{[by IR2}'\text{ (b)]} \\ &= C_1(t_1) + (1-\kappa)(t-t_1) - [(t_2-t_1)-\mu_m] + \kappa(t_2-t_1) \\ &\ge C_1(t_1) + (1-\kappa)(t-t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all $t \ge t_2$ we have:

$$C_2^{t_2}(t) \ge C_1(t_1) + (1-\kappa)(t-t_1) - \xi. \qquad (3)$$

Now suppose that for $i = 1, \dots, n$ we have $t_i \le t_i' < t_{i+1}$, $t_0 \le t_1$, and that at time t_i' process P_i sends a message to process P_{i+1} which is received at time t_{i+1} with an unpredictable delay less than ξ. Then repeated application of the inequality (3) yields the following result for $t \ge t_{n+1}$.

$$C_{n+1}^{t_{n+1}}(t) \ge C_1(t_1') + (1-\kappa)(t - t_1') - n\xi. \qquad (4)$$

From PC1, IR1′ and 2′ we deduce that

$$C_1(t_1') \ge C_1(t_1) + (1-\kappa)(t_1' - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \ge C_1(t_1) + (1-\kappa)(t-t_1) - n\xi \qquad (5)$$

for $t \ge t_{n+1}$.

For any two processes P and P′, we can find a sequence of processes $P = P_0, P_1, \dots, P_{n+1} = P'$, $n \le d$, with communication arcs from each P_i to P_{i+1}. By hypothesis (b) we can find times t_i, t_i' with $t_i' - t_i \le \tau$ and $t_{i+1} - t_i' \le \nu$, where $\nu = \mu + \xi$. Hence, an inequality of the form (5) holds with $n \le d$ whenever $t \ge t_1 + d(\tau + \nu)$. For any i, j and any t, t_1 with $t_1 \ge t_0$ and $t \ge t_1 + d(\tau + \nu)$ we therefore have:

$$C_i(t) \ge C_j(t_1) + (1-\kappa)(t-t_1) - d\xi. \qquad (6)$$

Now let m be any message timestamped T_m, and suppose it is sent at time t and received at time t'. We pretend that m has a clock C_m which runs at a constant rate such that $C_m(t) = t_m$ and $C_m(t') = t_m + \mu_m$. Then $\mu_m \le t' - t$ implies that $dC_m/dt \le 1$. Rule IR2′ (b) simply sets $C_j(t')$ to *maximum* $(C_j(t'-0), C_m(t'))$. Hence, clocks are reset only by setting them equal to other clocks.

For any time $t_x \ge t_0 + \mu/(1-\kappa)$, let C_x be the clock having the largest value at time t_x. Since all clocks run at a rate less than $1 + \kappa$, we have for all i and all $t \ge t_x$:

$$C_i(t) \le C_x(t_x) + (1+\kappa)(t-t_x). \qquad (7)$$

We now consider the following two cases: (i) C_x is the clock C_q of process P_q. (ii) C_x is the clock C_m of a message sent at time t_1 by process P_q. In case (i), (7) simply becomes

$$C_i(t) \le C_q(t_x) + (1+\kappa)(t-t_x). \qquad (8i)$$

In case (ii), since $C_m(t_1) = C_q(t_1)$ and $dC_m/dt \le 1$, we have

$$C_x(t_x) \le C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \le C_q(t_1) + (1+\kappa)(t-t_1). \qquad (8ii)$$

Since $t_x \ge t_0 + \mu/(1-\kappa)$, we get

$$\begin{aligned} C_q(t_x - \mu/(1-\kappa)) &\le C_q(t_x) - \mu && \text{[by PC1]} \\ &\le C_m(t_x) - \mu && \text{[by choice of } m\text{]} \\ &\le C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && [\mu_m \le \mu,\ t_x - t_1 \le \nu_m] \\ &= T_m && \text{[by definition of } C_m\text{]} \\ &= C_q(t_1) && \text{[by IR2}'\text{(a)].} \end{aligned}$$

Hence, $C_q(t_x - \mu/(1-\kappa)) \le C_q(t_1)$, so $t_x - t_1 \le \mu/(1-\kappa)$ and thus $t_1 \ge t_0$.

Letting $t_1 = t_x$ in case (i), we can combine (8i) and (8ii) to deduce that for any t, t_x with $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$ there is a process P_q and a time t_1 with $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$ such that for all i:

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \tag{9}$$

Choosing t and t_x with $t \geq t_x + d(\tau + \nu)$, we can combine (6) and (9) to conclude that there exists a t_1 and a process P_q such that for all i:

$$\begin{aligned} C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi \leq C_i(t) \\ \leq C_q(t_1) + (1 + \kappa)(t - t_1) \end{aligned} \tag{10}$$

Letting $t = t_x + d(\tau + \nu)$, we get

$$d(\tau + \nu) \leq t - t_1 \leq d(\tau + \nu) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$\begin{aligned} C_q(t_1) + (t - t_1) - \kappa d(\tau + \nu) - d\xi \leq C_i(t) \leq C_q(t_1) \\ + (t - t_1) + \kappa[d(\tau + \nu) + \mu/(1 - \kappa)] \end{aligned} \tag{11}$$

Using the hypotheses that $\kappa \ll 1$ and $\mu \leq \nu \ll \tau$, we can rewrite (11) as the following approximate inequality.

$$\begin{aligned} C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) \lesssim C_i(t) \\ \lesssim C_q(t_1) + (t - t_1) + d\kappa\tau. \end{aligned} \tag{12}$$

Since this holds for all i, we get

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi),$$

and this holds for all $t \gtrsim t_0 + d\tau$. □

Note that relation (11) of the proof yields an exact upper bound for $|C_i(t) - C_j(t)|$ in case the assumption $\mu + \xi \ll \tau$ is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than $2d(\mu + \xi)$ seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than ξ.

Acknowledgment. The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

References

1. Schwartz, J.T. *Relativity in Illustrations.* New York U. Press, New York, 1962.
2. Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
3. Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks.*
4. Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605–624.

Programming Languages — J. J. Horning, Editor

Shallow Binding in Lisp 1.5

Henry G. Baker, Jr.
Massachusetts Institute of Technology

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in Lisp 1.5 is presented in which context-switching is an environment tree transformation called rerooting. Rerooting is completely general and reversible, and is optional in the sense that a Lisp 1.5 interpreter will operate correctly whether or not rerooting is invoked on every context change. Since rerooting leaves assoc [*v*, *a*] invariant, for all variables *v* and all environments *a*, the programmer can have access to a rerooting primitive, shallow[], which gives him dynamic control over whether accesses are shallow or deep, and which affects only the speed of execution of a program, not its semantics. In addition, multiple processes can be active in the same environment structure, so long as rerooting is an indivisible operation. Finally, the concept of rerooting is shown to combine the concept of shallow binding in Lisp with Dijkstra's display for Algol and hence is a general model for shallow binding.

Key Words and Phrases: Lisp 1.5, environment trees, FUNARG's, shallow binding, deep binding, multiprogramming, Algol display
CR Categories: 4.13, 4.22, 4.32

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

Author's present address: Computer Science Department, University of Rochester, Rochester, NY 14627.

From "Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley, 1990, pp 17–42.

Why Functional Programming Matters

John Hughes
The University, Glasgow

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

1 Introduction

This paper is an attempt to demonstrate to the larger community of (non-functional) programmers the significance of functional programming, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are.

Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. All of these functions are much like ordinary mathematical functions, and in this paper they will be

[1]An earlier version of this paper appeared in the *The Computer Journal*, 32(2):98–107, April 1989. Copyright belongs to The British Computer Society, who grant permission to copy for educational purposes only without fee provided the copies are not made for direct commercial advantage and this BCS copyright notice appears.

defined by ordinary equations. We are following Turner's language Miranda[4][2] here, but the notation should be readable without specific knowledge of this.

The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant — since no side-effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa — that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.

Such a catalogue of "advantages" is all very well, but one must not be surprised if outsiders don't take it too seriously. It says a lot about what functional programming isn't (it has no assignment, no side effects, no flow of control) but not much about what it is. The functional programmer sounds rather like a mediæval monk, denying himself the pleasures of life in the hope that it will make him virtuous. To those more interested in material benefits, these "advantages" are totally unconvincing.

Functional programmers argue that there *are* great material benefits — that a functional programmer is an order of magnitude more productive than his or her conventional counterpart, because functional programs are an order of magnitude shorter. Yet why should this be? The only faintly plausible reason one can suggest on the basis of these "advantages" is that conventional programs consist of 90% assignment statements, and in functional programs these can be omitted! This is plainly ridiculous. If omitting assignment statements brought such enormous benefits then FORTRAN programmers would have been doing it for twenty years. It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be.

Even a functional programmer should be dissatisfied with these so-called advantages, because they give no help in exploiting the power of functional languages. One cannot write a program that is particularly lacking in assignment statements, or particularly referentially transparent. There is no yardstick of program quality here, and therefore no ideal to aim at.

Clearly this characterization of functional programming is inadequate. We must find something to put in its place — something that not only explains the power of functional programming but also gives a clear indication of what the functional programmer should strive towards.

[2]Miranda is a trademark of Research Software Ltd.

2 An Analogy with Structured Programming

It's helpful to draw an analogy between functional and structured programming. In the past, the characteristics and advantages of structured programming have been summed up more or less as follows. Structured programs contain no *goto* statements. Blocks in a structured program do not have multiple entries or exits. Structured programs are more tractable mathematically than their unstructured counterparts. These "advantages" of structured programming are very similar in spirit to the "advantages" of functional programming we discussed earlier. They are essentially negative statements, and have led to much fruitless argument about "essential *goto*s" and so on.

With the benefit of hindsight, it's clear that these properties of structured programs, although helpful, do not go to the heart of the matter. The most important difference between structured and unstructured programs is that structured programs are designed in a modular way. Modular design brings with it great productivity improvements. First of all, small modules can be coded quickly and easily. Second, general-purpose modules can be reused, leading to faster development of subsequent programs. Third, the modules of a program can be tested independently, helping to reduce the time spent debugging.

The absence of *goto*s, and so on, has very little to do with this. It helps with "programming in the small", whereas modular design helps with "programming in the large". Thus one can enjoy the benefits of structured programming in FORTRAN or assembly language, even if it is a little more work.

It is now generally accepted that modular design is the key to successful programming, and recent languages such as MODULA-II [6] and Ada [5] include features specifically designed to help improve modularity. However, there is a very important point that is often missed. When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation help only with clerical details — they can never make a great contribution to modularization.

We shall argue in the remainder of this paper that functional languages provide two new, very important kinds of glue. We shall give some examples of programs that can be modularized in new ways and can thereby be simplified. This is the key to functional programming's power — it allows improved modularization. It is also the goal for which functional programmers must strive — smaller and simpler and more general modules, glued together with the new glues we shall describe.

3 Gluing Functions Together

The first of the two new kinds of glue enables simple functions to be glued together to make more complex ones. It can be illustrated with a simple list-processing problem — adding the elements of a list. We can define lists[3] by

$$\mathit{listof} * ::= \mathit{Nil} \;\mid\; \mathit{Cons} * (\mathit{listof} *)$$

which means that a list of $*$s (whatever $*$ is) is either *Nil*, representing a list with no elements, or a *Cons* of a $*$ and another list of $*$s. A *Cons* represents a list whose first element is the $*$ and whose second and subsequent elements are the elements of the other list of $*$s. Here $*$ may stand for any type — for example, if $*$ is "integer" then the definition says that a list of integers is either empty or a *Cons* of an integer and another list of integers. Following normal practice, we will write down lists simply by enclosing their elements in square brackets, rather than by writing *Cons*es and *Nil*s explicitly. This is simply a shorthand for notational convenience. For example,

$[\,]$	means	Nil
$[1]$	means	$\mathit{Cons}\;1\;\mathit{Nil}$
$[1,2,3]$	means	$\mathit{Cons}\;1\;(\mathit{Cons}\;2\;(\mathit{Cons}\;3\;\mathit{Nil}))$

The elements of a list can be added by a recursive function *sum*. The function *sum* must be defined for two kinds of argument: an empty list (*Nil*), and a *Cons*. Since the sum of no numbers is zero, we define

$$\mathit{sum}\;\mathit{Nil} = 0$$

and since the sum of a *Cons* can be calculated by adding the first element of the list to the sum of the others, we can define

$$\mathit{sum}\;(\mathit{Cons}\;n\;\mathit{list}) = \;\mathit{num} + \mathit{sum}\;\mathit{list}$$

Examining this definition, we see that only the boxed parts below are specific to computing a sum.

$$\begin{aligned}&\mathit{sum}\;\mathit{Nil} \;= \boxed{0}\\ &\mathit{sum}\;(\mathit{Cons}\;n\;\mathit{list}) = n \boxed{+} \mathit{sum}\;\mathit{list}\end{aligned}$$

This means that the computation of a sum can be modularized by gluing together a general recursive pattern and the boxed parts. This recursive pattern is conventionally called *foldr* and so *sum* can be expressed as

$$\mathit{sum} \;=\; \mathit{foldr}\;(+)\;0$$

[3]In Miranda, lists can also be defined using the built-in constructor (:), but the notation used here is equally valid.

The definition of *foldr* can be derived just by parameterizing the definition of *sum*, giving

$$\begin{aligned}(\mathit{foldr}\ f\ x)\ \mathit{Nil}\ &= x\\ (\mathit{foldr}\ f\ x)\ (\mathit{Cons}\ a\ l\,) &= f\ a\ ((\mathit{foldr}\ f\ x)\ l\,)\end{aligned}$$

Here we have written brackets around $(\mathit{foldr}\ f\ x)$ to make it clear that it replaces *sum*. Conventionally the brackets are omitted, and so $((\mathit{foldr}\ f\ x)\ l)$ is written as $(\mathit{foldr}\ f\ x\ l)$. A function of three arguments such as *foldr*, applied to only two, is taken to be a function of the one remaining argument, and in general, a function of n arguments applied to only m of them $(m < n)$ is taken to be a function of the $n - m$ remaining ones. We will follow this convention in future.

Having modularized *sum* in this way, we can reap benefits by reusing the parts. The most interesting part is *foldr*, which can be used to write down a function for multiplying together the elements of a list with no further programming:

$$\mathit{product}\ =\ \mathit{foldr}\ (*)\ 1$$

It can also be used to test whether any of a list of booleans is true

$$\mathit{anytrue}\ =\ \mathit{foldr}\ (\vee)\ \mathit{False}$$

or whether they are all true

$$\mathit{alltrue}\ =\ \mathit{foldr}\ (\wedge)\ \mathit{True}$$

One way to understand $(\mathit{foldr}\ f\ a)$ is as a function that replaces all occurrences of *Cons* in a list by f, and all occurrences of *Nil* by a. Taking the list $[1, 2, 3]$ as an example, since this means

$$\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}\,))$$

then $(\mathit{foldr}\ (+)\ 0)$ converts it into

$$(+)\ 1\ ((+)\ 2\ ((+)\ 3\ 0)) = 6$$

and $(\mathit{foldr}\ (*)\ 1)$ converts it into

$$(*)\ 1\ ((*)\ 2\ ((*)\ 3\ 1)) = 6$$

Now it's obvious that $(\mathit{foldr}\ \mathit{Cons}\ \mathit{Nil}\,)$ just copies a list. Since one list can be appended to another by *Cons*ing its elements onto the front, we find

$$\mathit{append}\ a\ b =\ \mathit{foldr}\ \mathit{Cons}\ b\ a$$

As an example,

$$\begin{aligned}\mathit{append}\ [1, 2]\ [3, 4] &= \mathit{foldr}\ \mathit{Cons}\ [3, 4]\ [1, 2]\\ &= \mathit{foldr}\ \mathit{Cons}\ [3, 4]\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ \mathit{Nil}\,))\\ &= \mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ [3, 4]))\\ &\qquad \text{(replacing } \mathit{Cons} \text{ by } \mathit{Cons} \text{ and } \mathit{Nil} \text{ by } [3, 4])\\ &= [1, 2, 3, 4]\end{aligned}$$

We can count the number of elements in a list using the function *length*, defined by

$$\begin{aligned} \mathit{length} &= \mathit{foldr}\ \mathit{count}\ 0 \\ \mathit{count}\ a\ n &= n+1 \end{aligned}$$

because *count* increments 0 as many times as there are *Cons*es. A function that doubles all the elements of a list could be written as

$$\mathit{doubleall} = \mathit{foldr}\ \mathit{doubleandcons}\ \mathit{Nil}$$

where

$$\mathit{doubleandcons}\ n\ \mathit{list} = \mathit{Cons}\ (2*n)\ \mathit{list}$$

The function *doubleandcons* can be modularized even further, first into

$$\mathit{doubleandcons} = \mathit{fandcons}\ \mathit{double}$$

where

$$\begin{aligned} &\mathit{double}\ n = 2*n \\ &\mathit{fandcons}\ f\ \mathit{el}\ \mathit{list} = \mathit{Cons}\ (f\ \mathit{el})\ \mathit{list} \end{aligned}$$

and then by

$$\mathit{fandcons}\ f = \mathit{Cons}\ .\ f$$

where "." (function composition, a standard operator) is defined by

$$(f\ .\ g)\ h = f\ (g\ h)$$

We can see that the new definition of *fandcons* is correct by applying it to some arguments:

$$\begin{aligned} \mathit{fandcons}\ f\ \mathit{el} &= (\mathit{Cons}\ .\ f)\ \mathit{el} \\ &= \mathit{Cons}\ (f\ \mathit{el}) \end{aligned}$$

so

$$\mathit{fandcons}\ f\ \mathit{el}\ \mathit{list} = \mathit{Cons}\ (f\ \mathit{el})\ \mathit{list}$$

The final version is

$$\mathit{doubleall} = \mathit{foldr}\ (\mathit{Cons}\ .\ \mathit{double})\ \mathit{Nil}$$

With one further modularization we arrive at

$$\begin{aligned} &\mathit{doubleall} = \mathit{map}\ \mathit{double} \\ &\mathit{map}\ f = \mathit{foldr}\ (\mathit{Cons}\ .\ f)\ \mathit{Nil} \end{aligned}$$

where *map* — another generally useful function — applies any function f to all the elements of a list.

We can even write a function to add all the elements of a matrix, represented as a list of lists. It is

$$summatrix \ = \ sum \ . \ map \ sum$$

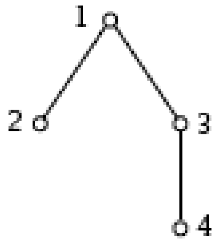
The function *map sum* uses *sum* to add up all the rows, and then the leftmost *sum* adds up the row totals to get the sum of the whole matrix.

These examples should be enough to convince the reader that a little modularization can go a long way. By modularizing a simple function (*sum*) as a combination of a "higher-order function" and some simple arguments, we have arrived at a part ($foldr$) that can be used to write many other functions on lists with no more programming effort.

We do not need to stop with functions on lists. As another example, consider the datatype of ordered labeled trees, defined by

$$treeof \ * ::= Node \ * \ \ (listof \ (treeof \ *))$$

This definition says that a tree of $*$s is a node, with a label which is a $*$, and a list of subtrees which are also trees of $*$s. For example, the tree



would be represented by

```
Node 1
     (Cons (Node 2 Nil)
           (Cons (Node 3
                       (Cons (Node 4 Nil) Nil))
                 Nil))
```

Instead of considering an example and abstracting a higher-order function from it, we will go straight to a function *foldtree* analogous to *foldr*. Recall that *foldr* took two arguments: something to replace *Cons* with and something to replace *Nil* with. Since trees are built using *Node*, *Cons*, and *Nil*, *foldtree* must take three arguments — something to replace each of these with. Therefore we define

```
foldtree f g a (Node label subtrees) =
        f label (foldtree f g a subtrees)
foldtree f g a (Cons subtree rest) =
        g (foldtree f g a subtree) (foldtree f g a rest)
foldtree f g a Nil  = a
```

Many interesting functions can be defined by gluing *foldtree* and other functions together. For example, all the labels in a tree of numbers can be added together using

$$sumtree \;=\; foldtree\,(+)\,(+)\,0$$

Taking the tree we wrote down earlier as an example, *sumtree* gives

```
(+)  1
        ((+) ((+) 2 0)
                ((+) ((+) 3
                                ((+) ((+) 4 0) 0))
                        0))
= 10
```

A list of all the labels in a tree can be computed using

$$labels \;=\; foldtree\; Cons\; append\; Nil$$

The same example gives

```
Cons 1
        (append  (Cons 2  Nil)
                      (append (Cons 3
                                           (append (Cons 4 Nil) Nil))
                                  Nil))
  = [1, 2, 3, 4]
```

Finally, one can define a function analogous to *map* which applies a function *f* to all the labels in a tree:

$$maptree\; f = \; foldtree\,(Node\;.\;f\,)\; Cons\; Nil$$

All this can be achieved because functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts — a general higher-order function and some particular specializing functions. Once defined, such higher-order functions allow many operations to be programmed very easily. Whenever a new datatype is defined, higher-order functions should be written for processing it. This makes manipulating the datatype easy, and it also localizes knowledge about the details of its representation. The best analogy with conventional programming is with extensible languages — in effect, the programming language can be extended with new control structures whenever desired.

4 Gluing Programs Together

The other new kind of glue that functional languages provide enables whole programs to be glued together. Recall that a complete functional program is just a function from its input to its output. If f and g are such programs, then $(g \;.\; f\,)$ is a program that, when applied to its input, computes

$$g\ (f \text{ input})$$

The program f computes its output, which is used as the input to program g. This might be implemented conventionally by storing the output from f in a temporary file. The problem with this is that the temporary file might occupy so much memory that it is impractical to glue the programs together in this way. Functional languages provide a solution to this problem. The two programs f and g are run together in strict synchronization. Program f is started only when g tries to read some input, and runs only for long enough to deliver the output g is trying to read. Then f is suspended and g is run until it tries to read another input. As an added bonus, if g terminates without reading all of f 's output, then f is aborted. Program f can even be a nonterminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies — a powerful modularization.

Since this method of evaluation runs f as little as possible, it is called "lazy evaluation". It makes it practical to modularize a program as a generator that constructs a large number of possible answers, and a selector that chooses the appropriate one. While some other systems allow programs to be run together in this manner, only functional languages (and not even all of them) use lazy evaluation uniformly for every function call, allowing any part of a program to be modularized in this way. Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.

We have described lazy evaluation in the context of functional languages, but surely so useful a feature should be added to nonfunctional languages — or should it? Can lazy evaluation and side-effects coexist? Unfortunately, they cannot: Adding lazy evaluation to an imperative notation is not actually impossible, but the combination would make the programmer's life harder, rather than easier. Because lazy evaluation's power depends on the programmer giving up any direct control over the order in which the parts of a program are executed, it would make programming with side effects rather difficult, because predicting in what order —or even whether— they might take place would require knowing a lot about the context in which they are embedded. Such global interdependence would defeat the very modularity that —in functional languages— lazy evaluation is designed to enhance.

4.1 Newton-Raphson Square Roots

We will illustrate the power of lazy evaluation by programming some numerical algorithms. First of all, consider the Newton-Raphson algorithm for finding square roots. This algorithm computes the square root of a number n by starting from an initial approximation $a0$ and computing better and better ones using the rule

$$a_{i+1} = (a_i + n/a_i)/2$$

If the approximations converge to some limit a, then

$$a = (a + n/a)/2$$

so

$$\begin{aligned} 2a &= a + n/a \\ a &= n/a \\ a * a &= n \\ a &= \sqrt{n} \end{aligned}$$

In fact the approximations converge rapidly to a limit. Square root programs take a tolerance (*eps*) and stop when two successive approximations differ by less than *eps*.

The algorithm is usually programmed more or less as follows:

```
C   N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
      X = A0
      Y = A0 + 2. * EPS
C   Y'S VALUE DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
100   IF ABS(X-Y).LE.EPS GOTO 200
      Y = X
      X = (X + ZN/X) / 2.
      GOTO 100
200   CONTINUE
C   THE SQUARE ROOT OF ZN IS NOW IN X.
```

This program is indivisible in conventional languages. We will express it in a more modular form using lazy evaluation and then show some other uses to which the parts may be put.

Since the Newton-Raphson algorithm computes a sequence of approximations it is natural to represent this explicitly in the program by a list of approximations. Each approximation is derived from the previous one by the function

$$\mathit{next}\ n\ x = (x + n/x)/2$$

so ($\mathit{next}\ n$) is the function mapping one approximation onto the next. Calling this function f, the sequence of approximations is

$$[a0,\ f\ a0,\ f\ (f\ a0),\ f\ (f\ (f\ a0)),\ \ldots]$$

We can define a function to compute this:

$$\mathit{repeat}\ f\ a = \ \mathit{Cons}\, a\ (\mathit{repeat}\ f\ (f\ a))$$

so that the list of approximations can be computed by

$$\textit{repeat}\ (\textit{next}\ n)\ a0$$

The function *repeat* is an example of a function with an "infinite" output — but it doesn't matter, because no more approximations will actually be computed than the rest of the program requires. The infinity is only potential: All it means is that any number of approximations can be computed if required; *repeat* itself places no limit.

The remainder of a square root finder is a function *within*, which takes a tolerance and a list of approximations and looks down the list for two successive approximations that differ by no more than the given tolerance. It can be defined by

$$\begin{aligned}&\textit{within eps}\ (\textit{Cons}\ a\ (\textit{Cons}\ b\ \textit{rest}))\\ &\qquad = b, &&\textbf{if } \textit{abs}\ (a-b) \leq \textit{eps}\\ &\qquad = \textit{within eps}\ (\textit{Cons}\ b\ \textit{rest}), &&\textbf{otherwise}\end{aligned}$$

Putting the parts together, we have

$$\textit{sqrt}\ a0\ \textit{eps}\ n = \textit{within eps}\ (\textit{repeat}\ (\textit{next}\ n)\ a0)$$

Now that we have the parts of a square root finder, we can try combining them in different ways. One modification we might wish to make is to wait for the ratio between successive approximations to approach 1, rather than for the difference to approach 0. This is more appropriate for very small numbers (when the difference between successive approximations is small to start with) and for very large ones (when rounding error could be much larger than the tolerance). It is only necessary to define a replacement for *within*:

$$\begin{aligned}&\textit{relative eps}\ (\textit{Cons}\ a\ (\textit{Cons}\ b\ \textit{rest}))\\ &\qquad = b, &&\textbf{if } \textit{abs}\ (a/b-1) \leq \textit{eps}\\ &\qquad = \textit{relative eps}\ (\textit{Cons}\ b\ \textit{rest}), &&\textbf{otherwise}\end{aligned}$$

Now a new version of *sqrt* can be defined by

$$\textit{relativesqrt}\ a0\ \textit{eps}\ n = \textit{relative eps}\ (\textit{repeat}\ (\textit{next}\ n)\ a0)$$

It is not necessary to rewrite the part that generates approximations.

4.2 Numerical Differentiation

We have reused the sequence of approximations to a square root. Of course, it is also possible to reuse *within* and *relative* with any numerical algorithm that generates a sequence of approximations. We will do so in a numerical differentiation algorithm.

The result of differentiating a function at a point is the slope of the function's graph at that point. It can be estimated quite easily by evaluating the function at the given point and at another point nearby and computing the slope of a straight line between the two points. This assumes that if the two points are close enough together, then the graph of the function will not curve much in between. This gives the definition

$$\mathit{easydiff}\ f\ x\ h = (f(x+h) - f\ x)/h$$

In order to get a good approximation the value of h should be very small. Unfortunately, if h is too small then the two values $f(x+h)$ and $f(x)$ are very close together, and so the rounding error in the subtraction may swamp the result. How can the right value of h be chosen? One solution to this dilemma is to compute a sequence of approximations with smaller and smaller values of h, starting with a reasonably large one. Such a sequence should converge to the value of the derivative, but will become hopelessly inaccurate eventually due to rounding error. If ($\mathit{within\ eps}$) is used to select the first approximation that is accurate enough, then the risk of rounding error affecting the result can be much reduced. We need a function to compute the sequence:

$$\begin{aligned}&\mathit{differentiate}\ h0\ f\ x = \ \mathit{map}\ (\mathit{easydiff}\ f\ x)\ (\mathit{repeat}\ \mathit{halve}\ h0)\\ &\mathit{halve}\ x = x/2\end{aligned}$$

Here $h0$ is the initial value of h, and successive values are obtained by repeated halving. Given this function, the derivative at any point can be computed by

$$\mathit{within}\ \mathit{eps}\ (\mathit{differentiate}\ h0\ f\ x)$$

Even this solution is not very satisfactory because the sequence of approximations converges fairly slowly. A little simple mathematics can help here. The elements of the sequence can be expressed as

$$\text{the right answer } + \text{ an error term involving } h$$

and it can be shown theoretically that the error term is roughly proportional to a power of h, so that it gets smaller as h gets smaller. Let the right answer be A, and let the error term be $B \times h^n$. Since each approximation is computed using a value of h twice that used for the next one, any two successive approximations can be expressed as

$$a_i = A + B \times 2^n \times h^n$$

and

$$a_{i+1} = A + B \times h^n$$

Now the error term can be eliminated. We conclude

$$A = \frac{a_{n+1} \times 2^n - a_n}{2^n - 1}$$

Of course, since the error term is only roughly a power of h this conclusion is also approximate, but it is a much better approximation. This improvement can be applied to all successive pairs of approximations using the function

$$\begin{aligned}&\mathit{elimerror}\ n\ (\mathit{Cons}\,a\ (\mathit{Cons}\,b\,\mathit{rest}))\\ &=\ \mathit{Cons}\ ((b * (2\text{\^{}}n) - a)/(2\text{\^{}}n - 1))\ (\mathit{elimerror}\ n\ (\mathit{Cons}\,b\,\mathit{rest}))\end{aligned}$$

Eliminating error terms from a sequence of approximations yields another sequence, which converges much more rapidly.

One problem remains before we can use *elimerror* — we have to know the right value of n. This is difficult to predict in general but is easy to measure. It's not difficult to show that the following function estimates it correctly, but we won't include the proof here:

$$\begin{aligned}&\mathit{order}\ (\mathit{Cons}\,a\ (\mathit{Cons}\,b\ (\mathit{Cons}\,c\,\mathit{rest})))\\ &\qquad =\ \mathit{round}\ (\mathit{log}2\ ((a-c)/(b-c)-1))\\ &\mathit{round}\ x = x \text{ rounded to the nearest integer}\\ &\mathit{log}2\ x =\ \text{the logarithm of } x \text{ to the base 2}\end{aligned}$$

Now a general function to improve a sequence of approximations can be defined:

$$\mathit{improve}\ s =\ \mathit{elimerror}\ (\mathit{order}\ s)\ s$$

The derivative of a function f can be computed more efficiently using *improve*, as follows:

$$\mathit{within}\ \mathit{eps}\ (\mathit{improve}\ (\mathit{differentiate}\ h0\ f\ x))$$

The function *improve* works only on sequences of approximations that are computed using a parameter h, which is halved for each successive approximation. However, if it is applied to such a sequence its result is also such a sequence! This means that a sequence of approximations can be improved more than once. A different error term is eliminated each time, and the resulting sequences converge faster and faster. Hence one could compute a derivative very efficiently using

$$\mathit{within}\ \mathit{eps}\ (\mathit{improve}\ (\mathit{improve}\ (\mathit{improve}\ (\mathit{differentiate}\ h0\ f\ x))))$$

In numerical analysts' terms, this is likely to be a fourth-order method, and it gives an accurate result very quickly. One could even define

$$\begin{aligned}&\mathit{super}\ s =\ \mathit{map}\ \mathit{second}\ (\mathit{repeat}\ \mathit{improve}\ s)\\ &\mathit{second}\ (\mathit{Cons}\,a\ (\mathit{Cons}\,b\,\mathit{rest})) = b\end{aligned}$$

which uses *repeat improve* to get a sequence of more and more improved sequences of approximations and constructs a new sequence of approximations by taking the second approximation from each of the improved sequences (it turns out that the second one is the best one to take — it is more accurate than the first and doesn't require any extra work to compute). This algorithm is really very sophisticated — it uses a better and better numerical method as more and more approximations are computed. One could compute derivatives very efficiently indeed with the program:

$$\mathit{within}\ \mathit{eps}\ (\mathit{super}\ (\mathit{differentiate}\ h0\ f\ x))$$

This is probably a case of using a sledgehammer to crack a nut, but the point is that even an algorithm as sophisticated as *super* is easily expressed when modularized using lazy evaluation.

4.3 Numerical Integration

The last example we will discuss in this section is numerical integration. The problem may be stated very simply: Given a real-valued function f of one real argument, and two points a and b, estimate the area under the curve that f describes between the points. The easiest way to estimate the area is to assume that f is nearly a straight line, in which case the area would be

$$\mathit{easyintegrate}\ f\ a\ b = (f\ a + f\ b) * (b - a)/2$$

Unfortunately this estimate is likely to be very inaccurate unless a and b are close together. A better estimate can be made by dividing the interval from a to b in two, estimating the area on each half, and adding the results. We can define a sequence of better and better approximations to the value of the integral by using the formula above for the first approximation, and then adding together better and better approximations to the integrals on each half to calculate the others. This sequence is computed by the function

$$\begin{aligned}
\mathit{integrate}\ f\ a\ b\ &= \mathit{Cons}\ (\mathit{easyintegrate}\ f\ a\ b) \\
&\qquad\qquad (\mathit{map}\ \mathit{addpair}\ (\mathit{zip}2\ (\mathit{integrate}\ f\ a\ \mathit{mid}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{integrate}\ f\ \mathit{mid}\ b))) \\
\textbf{where}\ \mathit{mid}\ &= (a + b)/2
\end{aligned}$$

The function *zip*2 is another standard list-processing function. It takes two lists and returns a list of pairs, each pair consisting of corresponding elements of the two lists. Thus the first pair consists of the first element of the first list and the first element of the second, and so on. We can define *zip*2 by

$$\mathit{zip}2\ (\mathit{Cons}\ a\ s)\ (\mathit{Cons}\ b\ t) = \ \mathit{Cons}\ (a, b)\ (\mathit{zip}2\ s\ t)$$

In *integrate*, *zip*2 computes a list of pairs of corresponding approximations to the integrals on the two subintervals, and *map addpair* adds the elements of the pairs together to give a list of approximations to the original integral.

Actually, this version of *integrate* is rather inefficient because it continually recomputes values of f. As written, *easyintegrate* evaluates f at a and at b, and then the recursive calls of *integrate* re-evaluate each of these. Also, $(f\ mid)$ is evaluated in each recursive call. It is therefore preferable to use the following version, which never recomputes a value of f:

$$\begin{aligned}
&\mathit{integrate}\ f\ a\ b = \mathit{integ}\ f\ a\ b\ (f\ a)\ (f\ b)\\
&\mathit{integ}\ f\ a\ b\ \mathit{fa}\ \mathit{fb} = \mathit{Cons}\ \ ((\mathit{fa} + \mathit{fb}) * (b - a)/2)\\
&\qquad\qquad\qquad\qquad \mathit{map}\ \mathit{addpair}(\mathit{zip}2\ \ (\mathit{integ}\ f\ a\ m\ \mathit{fa}\ \mathit{fm})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{integ}\ f\ m\ b\ \mathit{fm}\ \mathit{fb})))\\
&\qquad\qquad\qquad \textbf{where}\ \ m = (a + b)/2\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{fm} = f\ m
\end{aligned}$$

The function *integrate* computes an infinite list of better and better approximations to the integral, just as *differentiate* did in the section above. One can therefore just write down integration routines that integrate to any required accuracy, as in

$$\begin{aligned}
&\mathit{within}\ \mathit{eps}\ (\mathit{integrate}\ f\ a\ b)\\
&\mathit{relative}\ \mathit{eps}\ (\mathit{integrate}\ f\ a\ b)
\end{aligned}$$

This integration algorithm suffers from the same disadvantage as the first differentiation algorithm in the preceding subsection — it converges rather slowly. Once again, it can be improved. The first approximation in the sequence is computed (by *easyintegrate*) using only two points, with a separation of $b - a$. The second approximation also uses the midpoint, so that the separation between neighboring points is only $(b - a)/2$. The third approximation uses this method on each half-interval, so the separation between neighboring points is only $(b - a)/4$. Clearly the separation between neighboring points is halved between each approximation and the next. Taking this separation as h, the sequence is a candidate for improvement using the function *improve* defined in the preceding section. Therefore we can now write down quickly converging sequences of approximations to integrals, for example,

$$\mathit{super}\ (\mathit{integrate}\ \mathit{sin}\ 0\ 4)$$

and

$$\begin{aligned}
&\mathit{improve}\ (\mathit{integrate}\ f\ 0\ 1)\\
&\textbf{where}\ f\ x = 1/(1 + x * x)
\end{aligned}$$

(This latter sequence is an eighth-order method for computing $\pi/4$. The second approximation, which requires only five evaluations of f to compute, is correct to five decimal places.)

In this section we have taken a number of numerical algorithms and programmed them functionally, using lazy evaluation as glue to stick their parts together. Thanks to this, we have been able to modularize them in new ways, into generally useful functions such as *within*, *relative*, and *improve*. By combining these parts in various ways we have programmed some quite good numerical algorithms very simply and easily.

5 An Example from Artificial Intelligence

We have argued that functional languages are powerful primarily because they provide two new kinds of glue: higher-order functions and lazy evaluation. In this section we take a larger example from Artificial Intelligence and show how it can be programmed quite simply using these two kinds of glue.

The example we choose is the alpha-beta "heuristic", an algorithm for estimating how good a position a game-player is in. The algorithm works by looking ahead to see how the game might develop, but it avoids pursuing unprofitable lines.

Let game positions be represented by objects of the type *position*. This type will vary from game to game, and we assume nothing about it. There must be some way of knowing what moves can be made from a position: Assume that there is a function,

$$moves :: position \ \rightarrow \ listof\ position$$

that takes a game-position as its argument and returns the list of all positions that can be reached from it in one move. As an example, Fig. 1 shows *moves* for a couple of positions in tic-tac-toe (noughts and crosses). This assumes that it is always possible to tell which player's turn it is from a position. In tic-tac-toe this can be done by counting the ×s and `O`s; in a game like chess one would have to include the information explicitly in the type *position*.

Given the function *moves*, the first step is to build a game tree. This is a tree in which the nodes are labeled by positions, so that the children of a node are labeled with the positions that can be reached in one move from that node. That is, if a node is labeled with position p, then its children are labeled with the positions in (*moves* p). Game trees are not all finite: If it's possible for a game to go on forever with neither side winning, its game tree is infinite. Game trees are exactly like the trees we discussed in Section 2 — each node has a

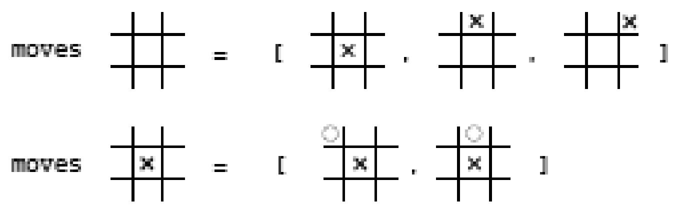


Figure 1: *moves* for two positions in tic-tac-toe.

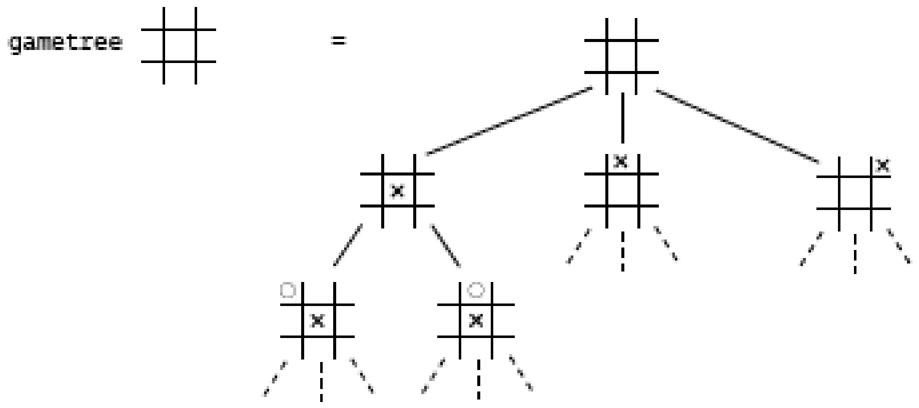


Figure 2: Part of a game tree for tic-tac-toe.

label (the position it represents) and a list of subnodes. We can therefore use the same datatype to represent them.

A game tree is built by repeated applications of *moves*. Starting from the root position, *moves* is used to generate the labels for the subtrees of the root. It is then used again to generate the subtrees of the subtrees and so on. This pattern of recursion can be expressed as a higher-order function,

$$reptree\ f\ a = \ Node\, a\ (map\ (reptree\ f\ )\ (f\ a))$$

Using this function another can be defined which constructs a game tree from a particular position:

$$gametree\ p = \ reptree\ moves\ p$$

As an example, consider Fig. 2. The higher-order function used here (*reptree*) is analogous to the function *repeat* used to construct infinite lists in the preceding section.

The alpha-beta algorithm looks ahead from a given position to see whether the game will develop favorably or unfavorably, but in order to do so it must be able to make a rough estimate of the value of a position without looking ahead. This "static evaluation" must be used at the limit of the look-ahead, and may be used to guide the algorithm earlier. The result of the static evaluation is a measure of the promise of a position from the computer's point of view (assuming that the computer is playing the game against a human opponent). The larger the result, the better the position for the computer. The smaller the result, the worse the position. The simplest such function would return (say) 1 for positions where the computer has already won, −1 for positions where the computer has already lost, and 0 otherwise. In reality, the static evaluation function measures various things that make a position "look good", for example material advantage and control of the center in chess. Assume that we have such a function,

$$static :: position \ \rightarrow \ number$$

Since a game tree is a (*treeof position*), it can be converted into a (*treeof number*) by the function (*maptree static*), which statically evaluates all the positions in the tree (which may be infinitely many). This uses the function *maptree* defined in Section 2.

Given such a tree of static evaluations, what is the true value of the positions in it? In particular, what value should be ascribed to the root position? Not its static value, since this is only a rough guess. The value ascribed to a node must be determined from the true values of its subnodes. This can be done by assuming that each player makes the best moves possible. Remembering that a high value means a good position for the computer, it is clear that when it is the computer's move from any position, it will choose the move leading to the subnode with the maximum true value. Similarly, the opponent will choose the move leading to the subnode with the minimum true value. Assuming that the computer and its opponent alternate turns, the true value of a node is computed by the function *maximize* if it is the computer's turn and *minimize* if it is not:

$$\begin{aligned} &maximize\ (Node\ n\ sub) = \ max\ (map\ minimize\ sub) \\ &minimize\ (Node\ n\ sub) = \ min\ (map\ maximize\ sub) \end{aligned}$$

Here *max* and *min* are functions on lists of numbers that return the maximum and minimum of the list respectively. These definitions are not complete because they recurse forever — there is no base case. We must define the value of a node with no successors, and we take it to be the static evaluation of the node (its label). Therefore the static evaluation is used when either player has already won, or at the limit of look-ahead. The complete definitions of *maximize* and *minimize* are

$$\begin{aligned} &maximize\ (Node\ n\ Nil) = n \\ &maximize\ (Node\ n\ sub) = \ max\ (map\ minimize\ sub) \\ &minimize\ (Node\ n\ Nil) = n \\ &minimize\ (Node\ n\ sub) = \ max\ (map\ maximize\ sub) \end{aligned}$$

One could almost write a function at this stage that would take a position and return its true value. This would be:

$$evaluate \ = \ maximize\ .\ maptree\ static\ .\ gametree$$

There are two problems with this definition. First of all, it doesn't work for infinite trees, because *maximize* keeps on recursing until it finds a node with no subtrees — an end to the tree. If there is no end then *maximize* will return no result. The second problem is related — even finite game trees (like the one for tic-tac-toe) can be very large indeed. It is unrealistic to try to evaluate the whole of the game tree — the search must be limited to the next few moves. This can be done by pruning the tree to a fixed depth,

$$\begin{aligned} \mathit{prune}\ 0\ (\mathit{Node}\ a\ x) &= \mathit{Node}\ a\ \mathit{Nil} \\ \mathit{prune}\ (n+1)\ (\mathit{Node}\ a\ x) &= \mathit{Node}\ a\ (\mathit{map}\ (\mathit{prune}\ n)\ x) \end{aligned}$$

The function (*prune* n) takes a tree and "cuts off" all nodes further than n from the root. If a game tree is pruned it forces *maximize* to use the static evaluation for nodes at depth n, instead of recursing further. The function *evaluate* can therefore be defined by

$$\mathit{evaluate}\ =\ \mathit{maximize}\ .\ \mathit{maptree}\ \mathit{static}\ .\ \mathit{prune}\ 5\ .\ \mathit{gametree}$$

which looks (say) five moves ahead.

Already in this development we have used higher-order functions and lazy evaluation. Higher-order functions *reptree* and *maptree* allow us to construct and manipulate game trees with ease. More importantly, lazy evaluation permits us to modularize *evaluate* in this way. Since *gametree* has a potentially infinite result, this program would never terminate without lazy evaluation. Instead of writing

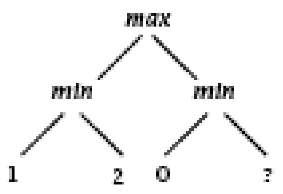
$$\mathit{prune}\ 5\ .\ \mathit{gametree}$$

we would have to fold these two functions together into one that constructed only the first five levels of the tree. Worse, even the first five levels may be too large to be held in memory at one time. In the program we have written, the function

$$\mathit{maptree}\ \mathit{static}\ .\ \mathit{prune}\ 5\ .\ \mathit{gametree}$$

constructs parts of the tree only as *maximize* requires them. Since each part can be thrown away (reclaimed by the garbage collector) as soon as *maximize* has finished with it, the whole tree is never resident in memory. Only a small part of the tree is stored at a time. The lazy program is therefore efficient. This efficiency depends on an interaction between *maximize* (the last function in the chain of compositions) and *gametree* (the first); without lazy evaluation, therefore, it could be achieved only by folding all the functions in the chain together into one big one. This would be a drastic reduction in modularity, but it is what is usually done. We can make improvements to this evaluation algorithm by tinkering with each part; this is relatively easy. A conventional programmer must modify the entire program as a unit, which is much harder.

So far we have described only simple minimaxing. The heart of the alphabeta algorithm is the observation that one can often compute the value returned by *maximize* or *minimize* without looking at the whole tree. Consider the tree:



Strangely enough, it is unnecessary to know the value of the question mark in order to evaluate the tree. The left minimum evaluates to 1, but the right minimum clearly evaluates to something at most 0. Therefore the maximum of the two minima must be 1. This observation can be generalized and built into *maximize* and *minimize*.

The first step is to separate *maximize* into an application of *max* to a list of numbers; that is, we decompose *maximize* as

$$maximize = max\ .\ maximize'$$

(We decompose *minimize* in a similar way. Since *minimize* and *maximize* are entirely symmetrical we shall discuss *maximize* and assume that *minimize* is treated similarly.) Once decomposed in this way, *maximize* can use *minimize′*, rather than *minimize* itself, to discover which numbers *minimize* would take the minimum of. It may then be able to discard some of the numbers without looking at them. Thanks to lazy evaluation, if *maximize* doesn't look at all of the list of numbers, some of them will not be computed, with a potential saving in computer time.

It's easy to "factor out" *max* from the definition of *maximize*, giving

$$\begin{aligned}
maximize'\ (Node\ n\ Nil) &= Cons\ n\ Nil\\
maximize'\ (Node\ n\ l\,) &= map\ minimize\ l\\
&= map\ (min\ .\ minimize')\ l\\
&= map\ min\ (map\ minimize'\ l\,)\\
&= mapmin\ (map\ minimize'\ l\,)\\
&\quad \textbf{where}\ mapmin = map\ min
\end{aligned}$$

Since *minimize′* returns a list of numbers, the minimum of which is the result of *minimize*, (*map minimize′ l*) returns a list of lists of numbers, and *maximize′* should return a list of those lists' minima. Only the maximum of this list matters, however. We shall define a new version of *mapmin* that omits the minima of lists whose minimum doesn't matter.

$$\begin{aligned}
&mapmin\ (Cons\ nums\ rest)\\
&\ \ = Cons\ (min\ nums)\ (omit\ (min\ nums)\ rest)
\end{aligned}$$

The function *omit* is passed a "potential maximum" — the largest minimum seen so far — and omits any minima that are less than this:

$$
\begin{aligned}
&\mathit{omit\ pot\ Nil} = \mathit{Nil}\\
&\mathit{omit\ pot}\ (\mathit{Cons\ nums\ rest})\\
&\quad = \mathit{omit\ pot\ rest}, && \textbf{if}\ \mathit{minleq\ nums\ pot}\\
&\quad = \mathit{Cons}\ (\mathit{min\ nums})\ (\mathit{omit}\ (\mathit{min\ nums})\ \mathit{rest}), && \textbf{otherwise}
\end{aligned}
$$

The function *minleq* takes a list of numbers and a potential maximum, and it returns *True* if the minimum of the list of numbers does not exceed the potential maximum. To do this, it does not need to look at the entire list! If there is any element in the list less than or equal to the potential maximum, then the minimum of the list is sure to be. All elements after this particular one are irrelevant — they are like the question mark in the example above. Therefore *minleq* can be defined by

$$
\begin{aligned}
&\mathit{minleq\ Nil\ pot} = \mathit{False}\\
&\mathit{minleq}\ (\mathit{Cons}\ n\ \mathit{rest})\ \mathit{pot} = \mathit{True}, && \textbf{if}\ n \le \mathit{pot}\\
&\qquad\qquad\qquad\qquad\quad = \mathit{minleq\ rest\ pot}, && \textbf{otherwise}
\end{aligned}
$$

Having defined *maximize′* and *minimize′* in this way it is simple to write a new evaluator:

$$\mathit{evaluate} = \mathit{max}\ .\ \mathit{maximize}'\ .\ \mathit{maptree\ static}\ .\ \mathit{prune}\ 8\ .\ \mathit{gametree}$$

Thanks to lazy evaluation, the fact that *maximize′* looks at less of the tree means that the whole program runs more efficiently, just as the fact that *prune* looks at only part of an infinite tree enables the program to terminate. The optimizations in *maximize′*, although fairly simple, can have a dramatic effect on the speed of evaluation and so can allow the evaluator to look further ahead.

Other optimizations can be made to the evaluator. For example, the alpha-beta algorithm just described works best if the best moves are considered first, since if one has found a very good move then there is no need to consider worse moves, other than to demonstrate that the opponent has at least one good reply to them. One might therefore wish to sort the subtrees at each node, putting those with the highest values first when it is the computer's move and those with the lowest values first when it is not. This can be done with the function

$$
\begin{aligned}
&\mathit{highfirst}\ (\mathit{Node}\ n\ \mathit{sub}) = \mathit{Node}\ n\ (\mathit{sort\ higher}\ (\mathit{map\ lowfirst\ sub}))\\
&\mathit{lowfirst}\ (\mathit{Node}\ n\ \mathit{sub}) = \mathit{Node}\ n\ (\mathit{sort}\ (\mathit{not}\ .\ \mathit{higher})\ (\mathit{map\ highfirst\ sub}))\\
&\mathit{higher}\ (\mathit{Node}\ n1\ \mathit{sub}1)\ (\mathit{Node}\ n2\ \mathit{sub}2) = n1 > n2
\end{aligned}
$$

where *sort* is a general-purpose sorting function. The evaluator would now be defined by

$$
\begin{aligned}
&\mathit{evaluate}\\
&\quad = \mathit{max}\ .\ \mathit{maximize}'\ .\ \mathit{highfirst}\ .\ \mathit{maptree\ static}\ .\ \mathit{prune}\ 8\ .\ \mathit{gametree}
\end{aligned}
$$

One might regard it as sufficient to consider only the three best moves for the computer or the opponent, in order to restrict the search. To program this, it is necessary only to replace *highfirst* with (*taketree* 3 . *highfirst*), where

$$taketree\ n = \ foldtree\ (nodett\ n)\ Cons\ Nil$$
$$nodett\ n\ label\ sub = Node\ label\ (take\ n\ sub)$$

The function *taketree* replaces all the nodes in a tree with nodes that have at most n subnodes, using the function (*take* n), which returns the first n elements of a list (or fewer if the list is shorter than n).

Another improvement is to refine the pruning. The program above looks ahead a fixed depth even if the position is very dynamic — it may decide to look no further than a position in which the queen is threated in chess, for example. It's usual to define certain "dynamic" positions and not to allow look-ahead to stop in one of these. Assuming a function *dynamic* that recognizes such positions, we need only add one equation to *prune* to do this:

$$\begin{aligned}&prune\ 0\ (Node\ pos\ sub)\\ &\ \ = Node\ pos\ (map\ (prune\ 0)\ sub), \quad \textbf{if}\ dynamic\ pos\end{aligned}$$

Making such changes is easy in a program as modular as this one. As we remarked above, since the program depends crucially for its efficiency on an interaction between *maximize*, the last function in the chain, and *gametree*, the first, in the absence of lazy evaluation it could be written only as a monolithic program. Such a programs are hard to write, hard to modify, and very hard to understand.

6 Conclusion

In this paper, we've argued that modularity is the key to successful programming. Languages that aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough — modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To support modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation. Using these glues one can modularize programs in new and useful ways, and we've shown several examples of this. Smaller and more general modules can be reused more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones. It also provides a target for functional programmers to aim at. If any part of a program is messy or complicated, the programmer should attempt to modularize it and to generalize the parts. He or she should expect to use higher-order functions and lazy evaluation as the tools for doing this.

Of course, we are not the first to point out the power and elegance of higher-order functions and lazy evaluation. For example, Turner shows how both can be used to great advantage in a program for generating chemical structures [3]. Abelson and Sussman stress that streams (lazy lists) are a powerful tool for structuring programs [1]. Henderson has used streams to structure functional operating systems [2]. But perhaps we place more stress on functional programs' modularity than previous authors.

This paper is also relevant to the present controversy over lazy evaluation. Some believe that functional languages should be lazy; others believe they should not. Some compromise and provide only lazy lists, with a special syntax for constructing them (as, for example, in SCHEME [1]). This paper provides further evidence that lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool.

Acknowledgments

This paper owes much to many conversations with Phil Wadler and Richard Bird in the Programming Research Group at Oxford. Magnus Bondesson at Chalmers University, Göteborg, pointed out a serious error in an earlier version of one of the numerical algorithms, and thereby prompted development of many of the others. Ham Richards and David Turner did a superb editorial job, including converting the notation to Miranda. This work was carried out with the support of a Research Fellowship from the U.K. Science and Engineering Research Council.

References

[1] Abelson, H. and Sussman, G. J. *The Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1984.

[2] Henderson, P. "Purely functional operating systems". In *Functional Programming and its Applications.* Cambridge University Press, Cambridge, 1982.

[3] Turner, D. A. "The semantic elegance of applicative languages". In *ACM Symposium on Functional Languages and Computer Architecture* (Wentworth, N.H.). ACM, New York, 1981.

[4] Turner, D. A. "An Overview of Miranda". *SIGPLAN Notices*, December 1986 (this and other papers about Miranda are at: http://miranda.org.uk).

[5] United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, Berlin, 1980.

[6] Wirth, N. *Programming in Modula–II.* Springer-Verlag, Berlin, 1982.

ACM *OOPS Messenger* **4**,4 (Oct. 1993), 2-27.

Equal Rights for Functional Objects[1] or, The More Things Change, The More They Are the Same[2]

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way, Encino, CA 91436 (818) 501-4956 (818) 986-1360 FAX

August, October, 1990, October, 1991, and April, 1992

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

We argue that intensional *object identity* in object-oriented programming languages and databases is best defined operationally by side-effect semantics. A corollary is that "functional" objects have extensional semantics. This model of object identity, which is analogous to the normal forms of relational algebra, provides cleaner semantics for the value-transmission operations and built-in primitive equality predicate of a programming language, and eliminates the confusion surrounding "call-by-value" and "call-by-reference" as well as the confusion of multiple equality predicates.

Implementation issues are discussed, and this model is shown to have significant performance advantages in persistent, parallel, distributed and multilingual processing environments. This model also provides insight into the "type equivalence" problem of Algol-68, Pascal and Ada.

1. INTRODUCTION

Parallel, distributed and persistent programming languages are leaving the laboratories for more wide-spread use. Due to the substantial differences between the semantics and implementations of these languages and traditional serial programming languages, however, some of the most basic notions of programming languages must be refined to allow efficient, *portable* implementations. In this paper, we are concerned with defining *object identity* in parallel, distributed and persistent systems in such a way that the intuitive semantics of serial implementations are transparently preserved. Great hardware effort and expense—e.g., cache coherency protocols for shared memory multiprocessors—are the result of this desire for transparency. Yet much of the synchronization cost of these protocols is wasted on *functional/immutable* objects, which do not have a coherency problem. If programming languages distinguish functional/immutable objects from non-functional/mutable objects, and if programs utilize a "mostly functional" style, then such programs will be efficient even in a non-shared-memory ("message-passing") implementation. Since it is likely that a cache-coherent shared-memory paradigm will not apply to a large fraction of distributed and persistent applications, our treatment of object identity provides increased cleanliness and efficiency even for non-shared-memory applications.

The most intuitive notion of object identity is offered by simple Smalltalk implementations in which "everything is a pointer". In these systems, an "object" is a sequence of locations in memory, and all "values" are homogeneously implemented as addresses (pointers) of such "objects". There are several serious problems with this model. First, "objects" in two different locations may have the same bit pattern both representing the integer "9"; an implementation must either make sure that copies like this cannot happen, or fix the equality comparison to dereference the pointers in this case. Second, the "everything is a pointer" model often entails an "everything is heap-allocated" policy, with its attendant overheads; an efficient implementation might wish to manage small fixed-size "things" like complex floating point numbers directly, rather than through pointers. Third, read access to the bits of an object may become a bottleneck in a multiprocessor environment due to locking and memory contention, even when the object is functional/immutable and could be transparently copied. In light of these problems, we seek a more efficient and less implementation-dependent notion of object identity than that of an address in a random-access computer memory.

A more efficient, but also more confusing, notion of object identity is offered by languages such as Pascal, Ada and C. These languages can be more efficient because they directly manipulate values other than pointers. This efficiency is gained, however, at the cost of an implementation-dependent notion of object identity. To a first

[1]"Functional objects" is triply overloaded, meaning immutable objects, function closures or objects with functional dependencies.

[2]*Plus ça change, plus c'est la même chose*—Alphonse Karr, as translated in [Cohen60,p.214].

approximation, a *value* in these languages is a *fixed-length* configuration of bits which can be manipulated directly, while an *object* is a configuration of bits which is manipulated through a pointer. The restriction on *values* to have only fixed lengths known to the compiler eliminates the possibility of variable-length functional/immutable *values*—e.g., character strings whose length cannot be determined at compile time; such values must be first-class (side-effectable) "objects" manipulated through pointers. The storage allocation and potential dangling reference problems of pointers cause these languages great uneasiness, however. Ada's paranoia about pointers leads to its bizarre and error-prone "**in out**" parameter passing, which it uses in preference to "by-reference" parameter passing, but whose semantics do not preserve object identity. The copying implicit in Ada's "**in out**" parameter passing can be characterized as a clumsy attempt at a software "cache coherency protocol", but it fails to provide the desired transparency.

We call the problem of providing clean, efficient semantics for "object identity" the *object identity crisis*, because the costs of full object identity in persistent, parallel and distributed systems are far greater than in serial, single-process systems. We will argue that the notions of "object identity" and "distinguishable by side-effects/assignment" are equivalent, and that applying this equivalence provides cleaner semantics for argument-passing, result-returning, and the built-in "equality" predicate of a programming language. Our model solves the problem of integrating functional (immutable) objects (e.g., numbers, strings) with non-functional objects by providing all objects with "object identity", but without the usual costs of full object status. This model of object identity also offers an interesting insight into the problem of "type equivalence".

Most of our examples and discussions will center on Common Lisp, because Common Lisp exhibits the wide range of issues we are trying to resolve, Lisp has a well-known, trivial syntax, Lisp has a universal type, and Lisp is intuitively defined in Lisp itself in a "meta-circular" manner. These properties allow the issues to be discussed with a minimum of extraneous detail. Our model is easily transported, however, to any other language in which object identity must be defined.

2. OBJECT IDENTITY

Modern "object-oriented" programming languages and data bases are based upon the notion of "object identity" [Khoshafian86] [King89] [Ohori90]. Every "object" has an identity that is unique even when its attributes are not. For example, object A and object B are distinguishable even when they have the same attributes. In a sense, object identity can be considered to be a rejection of the "relational algebra" view of the world [Ullman80] in which two objects can only be distinguished through differing attributes.[3]

Distinguishing objects can be done in several ways, with the "true" notion of object identity achieved through the finest distinction. Many programming languages provide a primitive equality predicate which can be used to distinguish objects. For example, most "systems programming" languages such as C or Ada compare simple objects like fixed-precision integers and characters for bit-representation equality, and pointers for location equality. As we will show, however, these predicates are often unreliable, in the sense that they sometimes make distinctions that are not otherwise visible, they sometimes fail to distinguish distinct objects, and they sometimes just plain fail. A more reliable, but more expensive, test for identity is "operational equivalence" [Rees86], invented by Morris [Morris68] and popularized by Plotkin [Plotkin75]; the Scheme Report defines operational equivalence as follows:

> Two objects are "operationally equivalent" if and only if there is no way that they can be distinguished, using ... primitives other than [equality primitives]. It is guaranteed that objects maintain their operational identity despite being named by variables or fetched from or stored into data structures. [Rees86,6.2]

This description of "operational identity" given above incorporates some of the most basic notions of "object identity":

- objects retain their identity throughout their lifetime—identity is constant and immutable;
- object identity is preserved by the basic value-transmission operations of the programming language—binding, argument-passing, value-returning, and assignment;[4]
- objects with the same identity are operationally equivalent.

[3]Object identity can be grafted onto a relational system through the use of a "time stamp" or equivalent attribute. This "attribute" is variously called an *oid*, a *surrogate*, an *l-value*, or an *object identifier* [Abiteboul89].

[4]Any value-transmission operation that does not preserve object identity is therefore an implicit *coercion*. Since implicit coercions are some of the more error-prone features of programming languages [DoD78,3B] [Radin78], we propose that coercions in the basic value transmitting operations of programming languages be eliminated.

We would like any predicate `e` for testing object identity to meet the following requirements, which provide bounds on the notion of object identity:

- `e` is an *equivalence relation*: `e`(x,x); `e`(x,y)⇒`e`(y,x); and `e`(x,y)∧`e`(y,z)⇒`e`(x,z)
- `e`(x,y) iff (let z=x in `e`(z,y)) iff (λ(z) `e`(z,y))(x) iff `e`((λ() x)(),y) iff (let z in z:=x; `e`(z,y))
- if (λ(x) f(x)) is a constant (single-valued) *function*, and if `e`(x,y), then `e`(f(x),f(y))
- if `c` is a *constructor* of immutable objects, then `c` is a constant (mathematical) function.[5]

Our requirements for an object identity predicate allow for an equivalence relation that is finer than operational equivalence. Since operational equivalence is undecidable, any decidable test for object identity will have to be strictly finer than operational equivalence. As a result, some functional objects, for example, will be considered distinct even though any application of these objects to identical arguments will produce identical results. However, object identity, as defined by an equality predicate, will be constant, and referential transparency for mathematical functions will be observed.

Lisp is the first major computer language to provide "object identity". A Lisp list can have the same elements in the same order, but it is not necessarily the "same" list. The built-in Lisp predicate `EQ` can be used to make this distinction, but the distinction can be more reliably drawn in an operational fashion using update semantics for determining object identity. Consider the following operational definition of `EQ` for cons cells:[6]

```
(defun eq-cons (x y)
  (let ((oldcar (car x)))
    (rplaca x 'private-symbol-for-eq-cons)
    (prog1 (eq-symbol (car y) 'private-symbol-for-eq-cons)
           (rplaca x oldcar))))
```

The experienced Lisp programmer may not have seen this particular pedagogical definition of `EQ`, but side-effecting functions like `NREVERSE` cannot be properly understood without this intuition.

A considerable amount of effort in beginning Lisp courses is devoted to providing the student a proper model for cons cells [Abelson85,3.3]. The two popular models are the "box-and-pointer diagrams" model introduced in [McCarthy60] and the linear-array-of-pointer-pairs model. Both of these models provide the correct operational semantics for `EQ`, because the `CAR` of a cell can be affected if and only if one applies `RPLACA` to "the" cons cell. Once these models are understood, `EQ` is a well-defined and natural predicate on mutable structured objects.[7]

The Lisp predicate `EQUAL` greatly confuses the situation, however. `EQUAL` tests for structural similarity rather than object identity, by recursing on the `CAR`'s and `CDR`'s of the sub-expressions:

```
(defun equal (x y)
  (or (and (atom x) (atom y) (eq-atom x y))
      (and (equal (car x) (car y))
           (equal (cdr x) (cdr y)))))
```

There are a several problems with `EQUAL`. First, it may diverge in the presence of directed cycles (loops) in one of its arguments, although some (e.g., [Pacini78]) have suggested more sophisticated predicates capable of detecting such cycles. Secondly, it is not *referentially transparent*; two calls to `EQUAL` on the "same" (i.e., `EQ`) arguments can produce different results at different times. Neither of these possibilities is to be desired in a programming language primitive equality test because we would like such a test to always return and we would like object identity to be preserved.

---

[5][Ohori89] states that different invocations of *mutable* constructors are guaranteed to produce distinct results.

[6]Mason's *defined:eq* [Mason86,p.66] uses the same technique. McCarthy's original "recursive functions" paper [McCarthy60] explicitly makes `EQ` *undefined* for cons cells, where the term "undefined" to the recursion theorist typically means "diverges" or "loops". McCarthy's paper also describes a *garbage collection* algorithm which twiddles an object's mark bit, thereby demonstrating the use of side-effects to define object identity.

[7]Garbage collectors have never trusted equality predicates to determine object identity, but have always used side-effect semantics; that's what "mark bits" and "reference counts" are for! A garbage collector can be built which utilizes an equality predicate instead of a mark bit—e.g., by using a hash table instead of a mark bit—but such a garbage collector would likely be slow. We argue later that a garbage collector can sometimes achieve increased performance by not preserving an identity predicate finer than the `egal` predicate defined later.

Yet `EQUAL` is an extremely valuable operation, because the vast majority of Lisp lists are side-effect free—i.e., *"pure"*. Without side-effects, loops cannot be constructed and sublists cannot be modified, and within these restrictions `EQUAL` becomes a well-defined and useful operation.[8]

This tension between the simplicity of `EQ` and the usefulness of `EQUAL` has led to a great deal of confusion. This confusion has now lasted for 30 years, with additional confusion having been recently added by Common Lisp. Since neither `EQ` nor `EQUAL` has the desired semantics for the multiplicity of Common Lisp datatypes, Common Lisp added six more—`EQL`, `EQUALP`, `=`, `CHAR=`, `STRING=`, and `STRING-EQUAL`, yet these also lack the correct semantics.[9] Scheme continues this confusion by offering the three predicates (`eq?`, `eqv?` and `equal?`) which are roughly equivalent to Common Lisp's `EQ`, `EQL` and `EQUALP`.

Why have so many different notions of equality been defined? We believe that this confusion has been caused by the tension between clean semantics and efficient execution. While every Lisp object can be provided with "object identity" by allocating a separate address for each number and character, such a Lisp is extremely slow because every arithmetic operation is accompanied by a memory allocation operation that is required to hold the new number.[10] However, the overhead of providing every number with "object identity" seems silly when the typical number is itself smaller than an object pointer. This space overhead, combined with the time overhead for allocation and dereferencing, has led to modern Lisps wherein the bits for a number's representation are stored in pointer-like form, with no allocation or dereferencing necessary—e.g., Interlisp's *inums* [Teitelman74]. When represented in this "immediate" format, numbers can be compared with `EQ`, which typically performs a simple pointer comparison, instead of `EQUAL`, which would require dereferencing.

Unfortunately, not all numbers can be represented in such a compact format; e.g., "bignums"—extended precision integers—require the allocation of additional storage, and hence require the use of a true pointer in order to avoid manipulating variable-length objects. Rather than requiring a programmer to constantly test whether an integer is large or small before making the appropriate comparison, Common Lisp defines the `EQL` predicate, which dereferences numbers when necessary.[11] Dereferencing when comparing numbers is consistent, however, because numbers are *immutable* objects whose components cannot be side-effected.[12] Thus, two instances of a number can sometimes be distinguished using `EQ` when they are supposed to be the same number; `EQ` draws too fine a distinction in this case.[13]

Strings provide more examples of `EQ`/`EQUAL` confusion. We would like to provide a primitive mechanism for comparing strings which treats each string as a whole, without requiring us to iterate through the individual string elements. In addition to the general usefulness of such a predicate, character strings and bit strings can often be compared more efficiently by considering the characters in groups rather than individually; this desire for efficiency usually results in this predicate becoming a primitive [Baker90b]. Common Lisp strings are instances of Common Lisp arrays, however, and hence their elements can be changed through assignment. By our operational definition of `EQ` given above, therefore, different strings are different, even when their spellings are currently the same.

Most strings, however, are *constant*—i.e., their spellings are never modified. The Common Lisp semantics which provide first-class object identity for strings means that storage allocation and dereferencing is always required, even when the strings are small and constant. In a distributed programming environment in which strings are incorporated

---

[8]McCarthy's `EQUAL` function [McCarthy60] applies only to immutable cons cells, and within that context, his `EQUAL` meets our requirements. He purposely crippled `EQ` (hence its name) to apply only to non-cons cells. `EQ` does not meet our requirements for object identity for pure cons cells because in most Lisp systems `(EQ (CONS x y) (CONS x y))` yields false—i.e., `CONS` is not a function.

[9]Steele reports that the ANSI Common Lisp committee concluded that "object equality is not a concept for which there is a uniquely determined correct algorithm" [Steele90,p.109]. This paper suggests otherwise.

[10]*MacLisp* [Moon74] numbers were allocated in this fashion, but small integers were "uniquized" with a table.

[11]Dereferencing can be avoided, allowing `EQ` to be universally used for comparing numbers, if all numbers are *uniquized*. Uniquization is the same process whereby symbols which are spelled the same are required to point to the same address, and hence be the same object. However, uniquization for arbitrary numbers has traditionally been found to be more expensive than dereferencing.

[12]Interlisp's `setn` primitive allows for unspeakable violence to numbers; its use is reserved for those who mutilate Fortran's call-by-reference constants by the light of the full moon.

[13]This superfine distinction can be embarrassing, as Common Lisp is allowed to copy numbers whenever it feels the urge [Steele90,p.104]. The noise often heard when this kind of bug is discovered is "eek!".

into messages, the overhead of providing `EQ`-style object identity for strings can be extremely expensive. For this reason, Cedar [Swinehart86] offers the notion of "ropes", which are functional (immutable) strings, and hence can be copied at will.[14] Unfortunately, neither Common Lisp nor Scheme has yet seen fit to provide for functional strings, although *AutoLISP* [Autodesk88] has *only* functional strings. Thus, strings are another datatype in which `EQ` is often too fine, and `EQUAL` is often too coarse.

The representation of numbers and strings are instances of a more generic functional representation problem. Whether functional objects are to be represented directly or as pointers is an implementation issue that should be hidden from the programmer. Therefore, such notions as "shallow equality" and "deep equality" [Atkinson89] are wrong-headed, because they allow implementation decisions to "leak through" to the programmer. For example, an object with a functional attribute might represent the attribute directly if it is small and if it belongs to a type which can be represented in a number of bits which can be fixed at compile time; alternatively, it could represent the attribute through a pointer to another functional object. Using shallow and deep equality predicates, however, the programmer could distinguish these representations, even though this distinction can only confuse him since the objects are functional.

Common Lisp *hash tables* and *property lists* present another important identity problem; an important use of hash tables is to give properties to non-symbols. The choice of the proper predicate to use (`EQ`, `EQL`, `EQUAL`, etc.) for the hash table/property list depends upon the kinds of objects to be used as keys: numbers should use `EQL`, while strings should use `EQUAL`. Hashing a number in an `EQ`-hash table will probably result in losing the property, because numbers which are `EQL` are not necessarily `EQ`; whether two numbers remain `EQ` over time is not guaranteed. Hashing a string into an `EQUAL`-hash table does not actually hash the string itself, but the functional *contents* of the string. If the implementation erroneously hashed the string itself, then a later modification of the string would cause the hash table code to crash or otherwise fail. Unfortunately, Common Lisp does not offer a single kind of hash table which will work uniformly and reliably on every object; such a hash table would be required for a generic "memoizing" function [Bird80]. We later define a predicate `EGAL` for the universal determination of object identity which can be used for such generic memoizing.

In the next section, we argue that neither `EQ` nor `EQUAL` is wrong; `EQ` is correct for mutable cons cells and `EQUAL` is correct for immutable cons cells. The major mistake of Lisp is in not distinguishing the two kinds of cons cells based on their mutability. Goto [Goto74] [Goto76] introduced this distinction, but for a different, although related, purpose.

3. OUR MODEL OF OBJECT IDENTITY

A. Mutability Definition of Object Identity

Our model for object identity is similar to Scheme's concept of "operational identity" [Rees86], in which objects which *behave* the same should *be* the same. However, since "behave the same" is undecidable for functions and function-closures, we back down from "operational identity" to "operational identity of data structure representations". Operational identity for data structures is much easier than operational identity for function-closures, because there are only a few well-defined operations on data structures, but function-closures can do anything. We define a single, computable, primitive equality predicate called `EGAL` which we show is consistent with the notion of "operational identity of data structures". *Egal* is the obsolete Norman term for *equal,* and *Égalité* is the French word for social equality. "During the seventeenth century *two parallel vertical lines* were frequently used [to denote equality], especially in France, instead of =" [Young11]; we will later find that || is a remarkably satisfying infix symbol for *egal*.

Our model for object identity distinguishes mutable objects from immutable objects, and mutable components of aggregate objects from immutable components. We consider an immutable component of an object to be an integral part of the object's identity, since it cannot be separated from the object. Unlike a normalized (factored) relational database, which attempts to *minimize* the size of a "key" which holds the essence of an entity [Ullman80,s.5.4], we *maximize* the size of the object "key" to include all of its static components. Because these components are static, we cannot create any "update anomalies" with this policy. In particular, this object identity can be used as a key to a Common Lisp hash table [Steele90,p.435], and no hash entries will become inaccessible as a result of a key element being modified.

[14]The term "rope" is curious; presumably functional ropes are "lighter-weight" than mutable strings. "Thread" would have been a better choice, but that term was already taken. "Filament" might have been the best choice. Common Lisp *keywords* are rough analogues to Cedar ropes.

We will build up our definition of `EGAL` incrementally, starting from a relatively simple base. An object is *immutable* if all of its (top-level) components are immutable, otherwise it is *mutable*. Briefly, if an object is mutable, then we compare it using an "address-like" comparison, while if an object is immutable, then we recursively compare its components. This recursion is only used to *define* the semantics of `EGAL`; a given implementation may not need to recurse. For example, Goto's "hash consing" [Goto74] [Goto76] allows functional lists to be compared without recursion. Below is a first approximation to `EGAL`:

```
(defun egal (x y)
  (and (egal (type-of x) (type-of y))
       (cond ((symbolp x)      (eq x y))
             ((numberp x)      (egal-number x y))
             ((consp x)        (eq x y))
             ((vectorp x)      (egal-vector x y))
             ((functionp x)    (egal-function x y))
             ((hash-table-p x) (egal-hashtable x y))
             ((streamp x)      (egal-stream x y))
             ((mutable-structure-p x)
                               (eq x y))
             (t                (every #'(lambda (component)
                                          (egal (funcall component x)
                                                (funcall component y)))
                                      (components (type-of x)))))))
```

B. Equality of Symbols and Numbers

The first clause of `EGAL` concerning symbols is not strictly necessary, since symbols are mutable (i.e., their values and properties can be changed), and hence they are included in the set of mutable structures. The second clause concerning numbers is also not strictly necessary, because numbers are immutable and are handled by the last clause. Since there are many types of Common Lisp numbers, however, we explicitly show how to compare them, as follows:

```
(defun egal-number (x y)
  (and (egal (type-of x) (type-of y))
       (cond ((complexp x)
              (and (egal-number (realpart x) (realpart y))
                   (egal-number (imagpart x) (imagpart y))))
             ((rationalp x)
              (and (egal-number (numerator x) (numerator y))
                   (egal-number (denominator x) (denominator y))))
             ((floatp x)
              (and (= (float-sign x) (float-sign y)) ; for IEEE-754[15]
                   (= x y)))
             ((and (fixnump x) (fixnump y)) (eq x y))
             ((and (bignump x) (bignump y))
              (every #'eq (digits x) (digits y)))
             (t nil))))
```

The use of strict equality "=" to compare two floating-point numbers is essential for object identity. In particular, the unfortunate tendency of some programming languages (e.g., APL) to include some sort of "fuzz" in the comparison of floating-point numbers is not acceptable. This is because such a fuzzy comparison is an *analytic* notion of closeness, and not an *algebraic* equivalence relation. In particular, fuzzy comparisons are not transitive. Since one of our goals is to preserve the semantics of table lookup, a fuzzy comparison would make such a notion ill-defined, because there may be several table keys which are sufficiently close to a test key. If a programmer or language designer wishes to have a fuzzy comparison, it should be a predicate distinct from the object identity predicate.

[15]Either IEEE `=` is not an identity predicate or IEEE `atan` is not a function, because `0.0=-0.0`, but `atan(0.0,-0.0)≠atan(-0.0,-0.0)`; due to this and other reasons, IEEE `-0.0` is an *algebraic abomination*.

Common Lisp does not support user-defined objects analogous to bignums and complex numbers, which would be EQL but not necessarily EQ; EQUALP descends into structures regardless of mutability. Since none of these equality predicates are CLOS generic functions [Steele90,ch.28], their behavior cannot be overloaded by user-supplied methods. The above definition for EGAL, however, allows the user to trivially define functional rational and complex number types himself using defstruct as follows, because EGAL will automatically recurse into the components of an immutable object.

```
(defstruct (complex
             (:constructor complex (realpart &optional (imagpart 0))))
  "Immutable (functional) complex numbers."
  (realpart 0 :read-only t :type real)
  (imagpart 0 :read-only t :type real))

(defstruct (ratio
             (:constructor / (numerator denominator)))
  "Immutable (functional) ratio numbers."
  (numerator   0 :read-only t :type integer)
  (denominator 1 :read-only t :type integer))
```

C. Equality of Vectors and Strings

If Common Lisp defined immutable vectors (and hence immutable strings), EGAL would utilize the following code:

```
(defun egal-vector (x y)
  (cond ((and (mutable-vector-p x) (mutable-vector-p y))
         (eq x y))
        ((and (immutable-vector-p x) (immutable-vector-p y))
         (and (= (length x) (length y))
              (dotimes (i (length x) t)
                (unless (egal (aref x i) (aref y i))
                  (return nil)))))
        (t nil)))
```

D. Equality of Functions and Function-Closures

The most problematic datatypes for equality are the function objects—*simple* functions ("compiled-functions") and function *closures* ("FUNARGS" [Moses70]). We would like to compare closures properly because they can be used for object-oriented programming based on *delegation* instead of *inheritance* [Snyder86] [Lieberman86] [Ungar87]. *Simple* functions access only their arguments and global variables, and therefore require no environment pointer; e.g., the C language [ANSI-C] provides simple function objects, but not function closures. Simple function objects in a functional language which do not reference global names are called *combinators*. Function closures are data structures incorporating both a simple function and an "environment" mechanism which provides the values of the "free variables" in the simple function; e.g., Algol-68, Pascal and Lisp offer lexical closures; Ada can approximate function closures using *tasks* [Lamb83].

True operational equivalence for function objects—even for simple functions—is impossible to compute, because the equivalence of programs is undecidable. Rather than "throw out the baby with the bath water" by refusing to compare functions, however, we desire to compare function closures in the same manner as the data structures that they may emulate. Consider the following ("object-oriented") simulation of Lisp's cons cells:

```
(defun cons (x y)
  #'(lambda (m &optional z)
      (caseq m (car x)
               (cdr y)
               (consp t)
               (rplaca (setq x z) nil)
               (rplacd (setq y z) nil))))

(defun car (c) (funcall c 'car))

(defun cdr (c) (funcall c 'cdr))

(defun rplaca (c x) (funcall c 'rplaca x))

(defun rplacd (c y) (funcall c 'rplacd y))
```

Since these simulated cons cells are mutable, we would like each of them to have a separate object identity, since they can be distinguised through mutations. Since "`#'`" ("`FUNCTION`") constructs a new function-closure object, Common Lisp's `EQ` has the correct semantics for this case. Consider, however, the simulation of *immutable* list cells:

```
(defun functional-cons (x y)
  #'(lambda (m)
      (caseq m (car x)
               (cdr y)
               (consp t))))

(defun car (c) (funcall c 'car))

(defun cdr (c) (funcall c 'cdr))
```

These cons cells are "read-only", since there are no `setq`'s to the free variables in the closure. We desire the `EGAL` predicate to perform correctly in this case, as well as in the mutable case above. `EGAL` can only get the correct answer if closures themselves are "functional" objects which allow `EGAL` to recursively examine their structure. In the case of our functional cons, the closure consists of 3 elements: the code for the closure and the values of `x` and `y`. If the components `x` and `y` are immutable, then the closure itself is immutable, because the code pointer cannot be changed. We therefore define `EGAL` on functions and function-closures as follows:

```
(defun egal-function (x y)
  (and (egal (type-of x) (type-of y))
       ((simple-function-p x) (eq x y))
       ((closure-p x)
        (and (egal-function (code x) (code y))
             (egal-environment (env x) (env y))))))
```

This definition for `EGAL` will work properly on both versions of `cons` because while the closure itself is in both cases immutable, the environment in the mutable case will itself be mutable, while the environment in the immutable case will be immutable. We have only succeeded in transforming the equality problem of closures into the equality problem of environments, however.

In order to understand how to compare environments, we must first look at their structure. An environment defines the values of the variables which are "free" in a "code" object. This definition can be in the form of an "association list", which is a list of pairs <variable-name,value>, which is searched to find the value, or the environment can be in the form of a vector which is indexed by an index number associated with each variable. Unfortunately, some language interpreters include in a closure environment values for variables which are not needed by the closure code. While these inclusions may cause excessive storage usage [Baker92a], they should not affect the course of the computation. We must therefore make sure that these extraneous values in closure environments do not affect the result of our equality comparison.

Extraneous values can be ignored in two ways: by making sure that the closures are not constructed with extraneous values, or by modifying environment comparison to ignore these extraneous values. Many Common Lisp compilers implement closures without extraneous values, but many Common Lisp interpreters implement closures with extraneous values.[16] Since we do not allow compiled closures to compare equal to interpreted closures, even when the compiled code resulted from compiling the interpreted code, we do not have to worry about comparing environments with incompatible structures. Furthermore, interpreted closures can compare equal only if they have *identical* (i.e., `EQ`) code—no matter how it looks when printed.[17] Therefore, the comparison of interpreted environments has to be performed in the presence of a list of "essential" (i.e., non-extraneous) values, as follows:

```
(defun egal-environment (vars x y)
  (every #'(lambda (v) (egal (lookup v x) (lookup v y)))
         vars))
```

[16]Some compilers perform the "optimization" of sharing environment structures among closures [Kranz86], even though this optimization may cause a form of "storage leak", wherein the extraneous objects avoid garbage collection [Baker92a].

[17]We later argue that Lisp source code should be functional; in this case, source code which prints the same will compare the same, i.e., be `EGAL`.

But wait! It would seem that this definition of environment comparison would allow EGAL to recurse on the comparison of our mutable cons cells as defined above. However, this is not correct. To achieve the correct semantics for mutable local variables which may be captured by closures, all Lisp systems perform a program transformation equivalent to that we call "cell introduction" [Sandewall74] [Kranz86]. Cell introduction provides for the binding of the assignable free variables to an anonymous "cell" which is then read and written instead of the variable itself. Below is the code for our mutable cell after cell introduction:

```
(defun cons (x y)
  (let ((x (make-cell x))
        (y (make-cell y)))
    #'(lambda (m &optional z)
        (caseq m (car (cell-value x))
                 (cdr (cell-value y))
                 (consp t)
                 (rplaca (setf (cell-value x) z) nil)
                 (rplacd (setf (cell-value y) z) nil)))))
```

As a result of cell introduction, the free variables x,y of our closure are now bound to cells instead of directly to their initial values. Thus, while EGAL on these environments will recurse down to the cell level, it will stop there because cells are mutable. Thus, even though the top-levels of closures created at different times will compare equal, the cells created by make-cell at these different times will not compare equal, so the semantics of EGAL are correct.

There remains a nagging problem with closure comparisons, however. The notion of a "free variable" of a lambda-expression is not well-defined, because a smart compiler may be able to eliminate all references to certain free variables. Such an optimization may change the behavior of EGAL in a way visible to the programmer if the closure becomes functional as a result of the optimization. One approach would be to include a variable in the closure if it appears free in the unoptimized source code, even if all of its occurences are later optimized away. Another approach would be to have the language provide a means for the programmer to declare which free variables he would like included in the closure, and thus participate in the EGAL comparison. An interpreter/compiler can easily check that this list is a superset of the variables which are free in the (optimized) code. Such a declaration was a necessity when Lisp variables were dynamically scoped, and is a welcome documentation aid for any lambda-expression in a lexically scoped Lisp. Below is shown one possible format for such a free variable declaration in Common Lisp:

```
#'(lambda (x)
    (declare (free a b c)) ; If a "free" declaration is given,
    (list x a b))          ; every free var must be in the list.
```

The final problem for the equivalence of functions involves *mutually recursive* functions. If the functions are known to the compiler to be simple, then the recursion may be closed by the assembler or linking loader, in which case EGAL will not be aware of the implicit recursive cycles. On the other hand, if a nest of mutually recursive functions are created by Common Lisp's LABELS (Scheme's letrec), then typical implementations will create cyclic environment structures to handle the recursion. Of course, cyclic environments are not required to implement recursive functions if we use the "Y combinator" trick from the lambda calculus [Gabriel88], as the factorial example below demonstrates.

```
(defun fact (f n) (if (= n 0) 1 (* n (funcall f (- n 1)))))
(defun factorial (n) (fact #'fact n))
```

Since our environment structures must be functional and finite, mutual recursion must therefore be implemented by means of the Y combinator shown for factorial, or by means of cyles created with side-effectable cells. Let us call the Y combinator method "pure" and the cyclic method "impure". Although the functions created using the two methods are "applicationally equivalent", their different natures can be distinguished by means of EGAL. Since the majority of language implementations utilize side-effects and cycles, and since it is difficult to hide the existence of the side-effectable cells used in these cyclic environments—e.g. Scheme's letrec [Bawden88], it is easier to use the impure method as the *definition* of mutual recursion. We therefore define Common Lisp's recursive LABELS in terms of non-recursive FLET, as follows:

```
(labels ((foo (x) << body of foo >>)
         (bar (y z) << body of bar >>))
  << body of labels using foo and bar. >>) ; is defined by:
```

```
(let ((foo-cell nil)
      (bar-cell nil))
  (flet ((foo (x) (funcall foo-cell x))
         (bar (y z) (funcall bar-cell y z)))
    (setq foo-cell #'(lambda (x) << body of foo >>)
          bar-cell #'(lambda (y z) << body of bar >>))
    << body of labels using foo and bar. >>))
```

A persistent, parallel and/or distributed object-oriented programming language might want to offer both "pure" and "impure" mechanisms for implementing mutual recursion, since their equivalence semantics would differ. The "impure" mechanism would offer traditional semantics, while the "pure" mechanism would exhibit functional behavior; this functional behavior being especially important in the case where the programmer was lazy and the functions did not actually call one another in a mutually recursive manner.

In summary, our model of equivalence for function closures is not based on "behavioral equivalence"—because that is undecidable—but on "representational equivalence". While "representational equivalence" provides a finer distinction than "operational equivalence", and may not provide all the equivalence we might like—e.g., equivalence under lexical variable "alpha-renaming"—we believe that "representational equivalence" provides an extremely useful extention of data structure equivalence into the realm of closures. Finally, "representational equivalence" does not prove embarrassing, since identity is constant and identical arguments to mathematical functions give identical results.

E. Equality of Hash Tables

Hash tables are a concrete implementation of an abstract function represented by the set of its <domain-element,range-element> association pairs. The hash tables of Common Lisp [Steele90,p.435] are mutable in two different ways. First, the domain of the function can be changed by adding or deleting association pairs. Second, the range-element associated with a particular domain-element can be changed. As a result of this modifiability, standard Common Lisp hash tables should compare `EGAL` if and only if they are identically the same hash table.

Even if a hash table restricted association pairs themselves to be immutable, the hash table itself would still be mutable because associations could be added or deleted. In this case, `EGAL` hash tables must still be identical.

The only case that would allow for consistent recursion into the structure of a hash table (as Common Lisp does with `EQUALP` [Steele90,p.108-109]) would be a constant (immutable) hash table, which represents a constant function with constant domain. In this case, the tables should be compared as extensional sets of association pairs, where the ordering of the association pairs is immaterial (as in `EQUALP`). A nonconstant function with constant domain could be simulated by such a constant hash table by binding each domain element to a different assignable cell.

The concept of a "lazy" hash table whose keys are not fully evaluated is discussed in the section on lazy values.

F. Equality of I/O Streams

There are mutable objects whose state is not changed by assignment. The most significant of these are *input/output* objects—usually input/output *streams*. One can simply define object identity through some sort of address comparison, as is done with assignable mutable objects. Such an *ad hoc* definition is not required, however. Streams are usually complex objects, incorporating a number of mutable elements such as buffers and buffer pointers, and in these cases the buffers and buffer pointers will "carry" the object's identity.

One can also conceive of unbuffered streams which are implemented as readable or writable memory locations in the manner of the "memory-mapped I/O" found in many microprocessors. For unbuffered output streams, the mutability is obvious due to the use of assignment to modify the contents of the "port". For unbuffered input streams, the "port" is read-only, but it is also *volatile* (i.e., incapable of being cached), meaning that it can change from reference to reference without ever being written. In both of these situations, the port should be considered a "mutable" cell, even if the mutating is being done by, or being noticed by, an external agent.

G. The Assignable Cell (Reference) Model

In the exposition of `EGAL`, above, we allowed for structures which have both immutable and mutable components. There is another model introduced by Algol-68 [vanWijngaarden77], however, which is cleaner because there is only one kind of mutable object—the *cell* (Algol-68 called it a "reference")—and all other objects are immutable. In the cell model, all data structures—with the exception of a cell—are immutable, and structure components which are intended to be updated are bound to cells which hold the initial value. The process of converting from assignable components to cells is called "cell introduction". We have already seen one example of cell introduction in the section on function closures, but here we uniformly introduce cells for the mutable components of all data structures.

It might seem that cell introduction would change the semantics of our EGAL equality predicate. This is not the case, however. While EGAL may have to recurse an additional level through a component, EGAL always stops at a cell. While the immutable structure which holds the cells together may be replicated at some point during the computation (there is no way to tell), EGAL will always recurse through the structure to get to the cells. The creation of an object may involve the creation of several new cells, but because the object's backbone is immutable, these cells are bound together into the structure for life. Therefore any copies of the object's immutable backbone will still hold the "same" cells. These mutable cells of an object are thus the "carriers" of a mutable object's identity, because differences in objects of the same data type can only be determined by EGAL recursing down to the level of cells.

If cells are the carriers of a mutable object's identity in our model, then the only way to generate nonforgeable "object ID's"—i.e., "social security numbers"—for such mutable objects is to provide a dummy cell as part of an object's definition. Since independent mutability is the essence of identity in our model, this cell can be as small as a single bit, because identity can be distinguished by "dithering" this single bit. Trivial mutable objects of this kind can be used for unforgeable keys, because the only way one could have gotten such a key is by communicating with some object which already had it, since the generator of such objects is guaranteed to produce a new, never-before-seen (modulo garbage collection) object. Of course, if the object identity predicate does not diddle this bit, and if we do not attempt to garbage collect these keys, then the storage for the bit itself can be optimized away.

In the cell model, then, all objects have the form of *finite trees* in which all of the interior nodes are immutable and all of the leaves—which can be shared—are either cells or atomic constants. All objects are finite trees, because it is impossible to create infinite trees—i.e., directed cycles—without side-effects. Furthermore, none of the interior nodes share, or more accurately, some of the internal nodes may share, but this sharing cannot be detected through side-effects or through equality testing—i.e., the tree is not re-entrant. Thus in the cell model, EGAL always performs a finite tree comparison. This restriction to *finite*, *acyclic* objects has significant implications for type equivalence, discussed later.

H. Object Input/Output

In Lisp, the input/output operations READ and PRINT make weak attempts to respect object identity. READ reads a single "object", while PRINT produces a READ-able representation of a single object. Originally, READ was to be a mathematical inverse of PRINT, so that the printed representation of an object was to be a faithful rendering of the object. When READ and PRINT are restricted to immutable cons cells, numbers and symbols, this goal is achieved—READ'ing such a PRINT'ed representation of an object A will produce an object B that is EQUAL to object A. Lisp I/O has never recovered, however, from the introduction of mutable objects, and READ/PRINT fail to respect object identity in the presence of mutable objects, even though some attempts are made to handle sharing and cycles.[18] Proposals for abstract I/O in other languages—e.g., [Herlihy82]—also fail to preserve object identity, because they go too far in preserving sharing for immutable objects, and do not go far enough to preserve sharing for mutable objects.

There are occasions—e.g., for Interlisp "dump" files ("poor man's persistence"), and when debugging—where the preservation of true object identity would be welcome.[19] Some sort of "social security number" is then required of every mutable object, and this number must be universal across all installations. Relational database systems which attempt to preserve object identity also require a similar "timestamp", or equivalent universal naming device. It should be clear that if every mutable object is uniquely tagged when performing an identity-preserving PRINT, the cost of dealing with mutable objects becomes substantially higher than the cost of dealing with immutable objects.

I. Copying

Many languages offer the ability to copy objects, including Common Lisp [Steele90,s.19.5]. Given our discussions of object identity, however, the status of such copying operations is very much in doubt. If an object is mutable, it has object identity, and copying produces a non-identical object. In other words, such a copying operation is an attempt to produce a "clone", and the depth of copying required to produce the correct semantics is highly

[18]Modern Lisps offer a more compact, but humanly-unreadable I/O format called "FASL" format (don't ask), which also attempts to handle sharing and cycles. FASL format also fails to respect object identity, however.

[19]The text of Autolisp interpreted user functions [Autodesk88] is often swapped to and from disk using PRINT and READ; the efficacy of this scheme depends upon the immutability of Autolisp cons cells and character strings.

application- and implementation-dependent. Therefore, no standardized copying operation (such as that for Common Lisp structures [Steele90,p.477]) will be useful.

If an object is immutable, on the other hand, every copy is identical to the original, and there is no way to distinguish among them within the programming language. Copying is therefore well-defined, because it is the identity function, but useless as a standardized function because it does nothing. Therefore, the concept of a standardized copying operation is always useless at best, and highly misleading at worst. Perhaps this ambiguity is one reason why the Common Lisp Object System (CLOS) [Steele90,s.28] has dispensed with default object copiers for each class.

An interesting situation occurs within a copying garbage collector. By definition, a copying garbage collector must preserve object identity, so the curious programmer can often find valuable insight into a programming language's notion of object identity by examining an implementation's garbage collector code. On the other hand, the collector's very copying seems to indicate a contradiction. This contradiction is resolved by the introduction of the notion of "forwarding pointers" [Baker78], which redirect object references from their original location to their new location so that the "same" object is referenced. Traditional copying garbage collectors leave forwarding pointers even for functional objects, but we have shown in this paper that forwarding pointers for functional objects are not necessary. In fact, we conjecture that 1) most recyclable garbage is functional in nature, and 2) substantial performance improvements in garbage collection algorithms can be obtained through clever optimizations for functional objects. For example, the "transitive size" of a functional object can be calculated incrementally as it is constructed, which allows any potential copier to dynamically and efficiently evaluate the trade-offs between copying and sharing. Furthermore, the reduction of the number of forwarding pointers can increase the "locality" (and hence efficiency) of a virtual memory copying garbage collector.

J. Lazy Values

Some modern languages—e.g., Scheme—offer a programmer-annotated method for achieving *lazy evaluation*. Lazy evaluation is a strategy for evaluating an expression which defers the evaluation of subexpressions as long as possible, in the hope that the value of the subexpression will never be required. For example, the evaluation of the expression denoting an argument for some call to a function may be deferred until the function itself requires the value, at which point the argument expression evaluation will be forced. Of course, the function may simply return the argument value, or store it into some data structure, in which case the evaluation may continue to be deferred. One could take the point of view that the only time that an evaluation is truly forced is when the answer is required to be printed out, but if such a printout is very large (perhaps infinite), then a lazy language may defer much of the evaluation even then. To put lazy evaluation on a more solid foundation, a lazy language usually defines a number of primitive functions as *strict*, meaning that they coerce some portion of their arguments into "ground terms".

Obviously, the uncertainty in the time of evaluation of a Scheme delay can lead to non-determinism in a language offering side-effects, as Scheme does. However, an expression in a functional language, once evaluated, will always yield the same value on additional evaluations. Therefore, most lazy implementations cache this value into a "value cell", thus avoiding the redundant evaluation of this particular expression. This use of value caches is sometimes called "call by need", whereas the uncached strategy is called "call by name".

Scheme `delay` creates a lazy value; `(delay` <exp>`)` creates a lazy value which can be forced into a strict value by means of the `force` expression, which is the left inverse of `delay`. In other words, `(delay (force` <exp>`))`≡`(delay` <exp>`)`, but `(force (delay` <exp>`))`≡<exp>. Lazy evaluation, e.g., using `delay`, can be used to create infinite functional data structures, which contradicts one of our major desiderata—that `egal` always terminate promptly.

Prior to this section, `egal` has been defined as a *strict* operation, because it recursively forces its arguments until it has explored enough to determine whether they are indeed the same. The only delays left unforced will occur in arguments that are not identical, and arguments which evaluate to the same expression will be transitively forced.

If we were to implement the Scheme `delay` and `force` ourselves, by generating an assignable cache cell in the manner suggested in the Scheme document [Rees86], then we would not achieve the correct semantics unless `egal` strictly forced its arguments. Otherwise, `(egal (delay 3) 3)` would not yield true, because the function closure for the `delay` would not compare `egal` to the ground number `3`. One could modify `egal` to be slightly less strict if it recursively forced evaluation until it compared two `delays`. Then, if the `delays` themselves were identical, which could be determined by comparing their cache cells, then `egal` could return true without further forcing the expression; otherwise, the `delays` would have to be further forced. This modification was suggested by Hughes [Hughes85] for use with "lazy memo-functions", which allow certain computations on infinite data structures to terminate.

Since our intention is the provision of a robust object identity predicate for widespread use, rather than for research programming languages, we feel that the efficiency and simplicity of a strict `egal` is more important than the transparent provision of lazy infinite data structures. Therefore, we suggest that `egal` remain strict, and those intent on implementing infinite laziness are welcome to do so using explicit assignable cells within an abstract data type. In this way, the lazy memo-functions of [Hughes85] can be readily and efficiently emulated.

4. OBJECT IDENTITY IN PERSISTENT, PARALLEL AND DISTRIBUTED SYSTEMS

Object identity becomes a much more important issue for *parallel* systems than it is in a single-process system. In a single-process system, object identity can be the same notion as that of an address, because dereferencing such an object is not expensive. With multiple processes, however, accessing an object involves some sort of locking to protect it from multiple unsynchronized accesses which could compromise its atomicity. For example, the implementation of a push-down stack in Common Lisp can utilize a vector with a "fill pointer" which keeps track of the current length of the vector. Pushing or popping such an object involves two operations—the access to the "top" of the stack and the adjustment of the fill pointer; these two operations must be grouped into an atomic transaction, or else the object will not provide correct stack semantics.

Persistent systems [Sandewall75] [Atkinson87] [Atkinson88] [Alagic89] require the logging or journaling of updates to objects, so that a consistent world can be recovered in the case of a crash. *Distributed* systems require more complexity, because in addition to having to be synchronized, an object must also be localized, and any references to this object which come from places far away from its location will be quite expensive. Although schemes have been described which enable mutable objects to be replicated in many places, these schemes are complicated and expensive; therefore, we will consider only models in which each mutable object has but one location.

We have argued that immutable functional objects have "object identity", but they do not require synchronization because they cannot be modified, and they do not require localization, because they can be replicated at will. As a result, functional objects can be much more efficient than non-functional objects in a distributed system; this is the reason for Cedar's functional strings (called "ropes") [Swinehart86]. For example, if we were to create a list of objects in a distributed Lisp system, traditional Lisp semantics would demand that each list cell have "object identity", because Lisp list cells are mutable. As a result, a function call from one location to another would pass a pointer to this list, and the called function would then have to interrogate each list cell in turn (perhaps using a locking protocol) to acquire references to the listed objects. If, however, our Lisp had functional list cells, we could pass this list by copying it at the time of the function call (in the manner of a "remote procedure call"), and the called function would then have its own private copy of this list after only one interaction with the calling function instead of many. Keeping consistent semantics in this situation is possible only if argument lists are functional.

The efficiency of functional arguments in parallel and distributed systems does not come from either a consistent reference-passing or a consistent copying policy, but from the freedom to do *either* at any time. The consistent reference policy allows for great efficiency for infrequently referenced or very large objects, while the consistent copying policy allows for great efficiency for frequently referenced and/or small objects. By basing object identity upon an abstract concept like mutability instead of an implementation-dependent concept, we can achieve efficiency in a more portable fashion.

Ada's non-deterministic, non-transparent semantics (by-reference v. copying) for its "**in out**" parameter-passing mode has the same efficiency goals as our object-identity proposal; Ada, however, must rely on the good intentions of the programmer, because the difference between the two mechanisms is all too evident. Since "the road to hell is paved with good intentions", our scheme does not rely upon them for correct behavior, unlike Ada's "**in out**" parameter-passing.

5. OBJECT IDENTITY IN MULTILINGUAL ENVIRONMENTS

The ability of a program in a language to call subprograms in other languages has become an feature essential to a programming language's ability to survive the recent "standardization craze". The mechanisms required for interlingual communication are quite similar to those required for heterogeneous distributed systems—even when the various languages run within the same address space.

Communication *per se* in a multilingual single-address-space environment is not as big of a problem as protecting one language environment from the other. The differing structure formats of different languages can usually be handled by relatively straight-forward format coercion routines. Much more problematical is the fight between the language run-time systems over the control of storage allocation and control of object identity.

Functional argument structures can reduce interlingual communication costs, just as they do in a distributed environment. Copies of functional objects can be freely made using the recipient language's own storage management system, if necessary, so that no storage management problems arise; functional return values are handled similarly. Of course, if the functional objects are small enough, then no storage need be allocated.

In the case of mutable objects, it is very likely that the different languages have different formats and thus the receiver of an object cannot access it directly. In this case, the sending language can pass the object after it has been encased in a translation container which interprets actions requested by the receiver. This translation container is essentially a function closure or an Algol-60-like "thunk". Thus, even when a mutable object must be passed, the existence of cheap functional objects—argument lists and closures—can still reduce the overall costs.

Unfortunately, the publication of object references creates a substantial "garbage collection" problem in distributed and heterogeneous environments. Since the owner of the object is never sure when the last object reference has been forgotten, he is obliged to keep the referenced object alive indefinitely. We have no elegant solution for this problem, but the aggressive replacement of mutable objects by functional objects can reduce the amount of "persistent garbage" that is created by interlingual communication.

6. MOSTLY FUNCTIONAL PROGRAMMING

Due to the growing importance of persistent, parallel and distributed programming systems, and due to the costs of mutable objects in such systems, "functional" and "mostly functional" programming styles have been advocated [Backus78] [Knight86]. There are numerous advantages to eliminating gratuitous side-effects. Not only do we gain in efficiency in persistent, parallel and distributed systems, but we also make the job of compilers and verifiers much easier [Baker90d]. Less obviously, the freedom to copy functional objects or not can improve the underlying operations of storage management [Baker78] [Baker92a]; for example, *forwarding pointers* are not required when a functional object is moved because the sharing of functional objects is not detectable by `EGAL`.

Providing for mostly functional programming in Common Lisp requires the addition of functional list cells and functional arrays. Such provisions simplify many complications of Common Lisp. There are many rules regarding the sharing/non-sharing/modifiability of constants in source code;[20] these rules disappear with the single policy that traditional parenthesized lists and quoted strings `READ` as functional objects. Making `CONS` and `LIST` produce functional list cells, and including a new constructor (`WCONS`?) for mutable list cells,[21] offers the efficiency of functional lists to the 95+% of instances where lists are used functionally. Of course, `EVAL` should accept source code only in the form of functional lists; modifiable source code has been considered an anathema for the past 30 years. The problem of whether `&REST` arguments in argument "lists" are copied or not disappears when `&REST` arguments become functional.

Functional strings can have interesting implementations. A representation of a string as a tree whose leaves are individual characters offers O(1) complexity concatenation; while element indexing is more expensive, it is usually less common than concatenation.

Functional arrays are known to have performance problems, because a succession of single-element updates may convert a problem of linear complexity into one of quadratic complexity [Bloss89]. Many solutions to these problems have been proposed [Schwartz75ab] [Hudak85] [Hudak86] [Hudak89] [Bloss89] [Baker90a] [Baker91a]. On the other hand, functional array semantics can allow the experimentation with completely new array implementations, such as quad-trees [Wise87]. The incorporation of functional arrays into existing programming languages would allow for the extensive testing of these ideas, while preserving the option of traditional mutable arrays.

The status of primitive similarity predicates that are coarser than `EGAL` is marginal in a mostly functional language. For example, suppose we wish to compare a mutable character buffer with a constant string. Since these two objects have different types, there is no obvious mechanism to compare the two, and they might have different representations. The language could offer a primitive coercion from a mutable character vector into a functional string, which could then be compared for equality. Any other similarity primitive would be an attempt to optimize this two-step process.

7. CALL-BY-REFERENCE VERSUS CALL-BY-VALUE

Despite the thirty years that the problem of "call-by-reference" versus "call-by-value" in argument-passing semantics of Algol-like programming languages has been discussed, it has never been clearly resolved. While many undergraduate texts [Aho86,s.7.5] and language manuals give a definition of each policy, there has always been a nagging feeling that these definitions for Algol-60, Fortran and Ada are too *ad hoc* and too implementation-oriented.

[20]This is called the "coalescing of constants" problem [Steele90,p.691-4].

[21]Given the availability of `defstruct`, perhaps primitive mutable list cells are obsolete.

With the proliferation of persistent, parallel and distributed systems (with, e.g., "remote procedure calls"), the problem of resolving the precise definition of argument-passing semantics has become urgent.

There are two orthogonal issues in argument-passing semantics: evaluation ordering (e.g., "eager" versus "lazy"), and "object identity". Unfortunately, these two issues interact in very complex ways in non-functional languages, and this interaction has produced a great deal of confusion. Evaluation ordering is well-defined and relatively well-understood, because it can be studied in the context of the functional lambda calculus which has well-defined and precise semantics. However, the "object identity" issues have not been well-understood, precisely because they are intimately tied up with the side-effects introduced in non-functional languages.

We claim that the only argument-passing model that is consistent in non-functional languages is *call-by-object-reference,* i.e., passing object identities. Exactly *when* this object identity is determined relative to the computation inside the called function is the issue of evaluation order, but *what* this object identity means should always be well-defined. Our model offers a seamless integration of functional and imperative programming, because if an argument is functional, it will be passed by value, and if it is mutable, it will be passed by "reference". Any other argument-passing model involves an implicit coercion of mutable objects or mutable components into their values, and the timing of this coercion may be difficult to determine or control. For example, the coercion semantics of `EQUAL` applied to standard, mutable list cells is best described as "lazy coercion to value", where only the portion of the argument examined is actually "coerced". The complexity of such gradual coercions is one of the main reasons for our rejection of the traditional argument-passing approach.

Lisp is sometimes characterized as "call-by-value", but this is incorrect. Lisp approximates call-by-object-reference semantics extremely well, and the single lacuna can be easily repaired. The reason Lisp appears to be call-by-value is that arguments which are functional data structures appear to have been passed in toto. However, it is the nature of functional data structures that whether they are passed by value or by reference is not determinable, hence we consider these data structures to have been passed by "object reference".

"Call-by-value" coerces arguments into their "values", which causes mutable portions of a structure to be replaced by their current values; Lisp does not usually do this. The single exception is caused by variables, which are made variable by `SETQ`. If we make variable binding permanent, then we can eliminate this exception. `SETQ` of global variables is trivially replaced by `SET` of the quoted variable, while `SETQ` of local variables can be eliminated through "cell introduction" for mutable local variables, discussed in several sections above. After these changes, the binding of all lambda variables is constant (although they are sometimes bound to mutable cells), and argument-passing becomes completely uniform. Of course, when cells are introduced for mutable lambda variables, any variable instances—including those in argument lists—are replaced by expressions which access the cell's value, thus making any coercion explicit.

8. TYPE IDENTITY, TYPE EQUIVALENCE AND ABSTRACT DATA TYPES

A. Comparing Type Objects

There has been as much confusion over type identity as there has been over object identity, although the type identity problem is usually referred to as the type *equivalence* problem [Aho86,s.6.3] [Wegbreit74] [Welsh77]. The type identity problem is to determine when two types are equal, so that type checking can be done in a programming language.[22] Algol-68 takes the point of view of "structural" equivalence, in which nonrecursive types that are built up from primitive types using the same type constructors in the same order should compare equal, while Ada takes the point of view of "name" equivalence, in which types are equivalent if and only if they have the same name. We will ignore the software engineering issues of which kind of type equivalence makes for better-engineered programs, and focus on the basic issue of type equivalence itself. We note that if a type system offers the type `TYPE`—i.e., it offers first-class representations of types as accessible objects during execution, then the type identity problem is subsumed by the object identity problem [Goldberg83] [Queinnec88].

Type systems are usually constructed starting from a small set of primitive types—boolean, character, IEEE-single-float—and building up utilizing *products* (record structures and arrays), *sums* (*unions*) and *functions*. Any *finite* type expression built in this way can be "easily" compared for type equivalence using structural equivalence, although comparing unordered union types with less than quadratic complexity requires some sophistication. Equivalence problems arise in two areas: recursive types and abstract data types. Recursive types cause problems because their proper description requires either infinite type expressions, or type expressions with directed cycles [Aho86,s.6.3]. In either case, the obvious simple recursive structural equivalence algorithm will fail. Abstract data types cause

[22]Whether type checking is performed statically during compilation or dynamically during execution is irrelevant for our purposes.

problems because they are supposed to be *opaque*; i.e., equivalence should depend only upon the identity of the abstract data type itself, and not upon whatever representation is currently being used to implement it [Guttag77].

Regardless of their theoretical problems, recursive data types are handled in the following way by Pascal, C, Ada, etc. The programmer first declares that the recursive type is a type (we have purposely fuzzed the distinction between "type" and "type name"), but gives no details; then he uses the type to construct another type. Finally, he gives the full definition of the recursive type.[23] Inside of the Pascal, C, Ada, etc., compiler, it is obvious what is happening. A new type node is defined, but its contents are not filled in. This type node is referenced by one or more other type nodes. Finally, the type node is *updated* with the full definition of the type. We notice two things about this implementation of recursive types: a recursive type involves a true directed cycle in the type definition data structure, and building a recursive type involves a side-effect (assignment) on the type definition data structure. Directed cycles cannot be built without side-effects, and conversely, side-effects are only needed to build these cycles, because non-cyclic types can be built in a functional manner.

Abstract data types are handled in a similar way by Eiffel, Ada, etc.[24] The programmer first declares the *specification* of the data type, which involves giving it a name.[25] Later in the compilation, linking or execution, the datatype is associated with an *implementation*, which provides a representation for the datatype. Since a user of the abstract data type is not allowed to know the implementation of the datatype, because it is opaque, it cannot know its representation, and therefore it cannot perform structural equivalence. Inside the Eiffel, Ada, etc., compiler, it is obvious what is happening. A new type node is defined, but its contents are not filled in. This type node is referenced by one or more other type nodes. Finally, the type node is *updated* with the full definition of the type when the implementation is given. We notice that an abstract data type seems to involve a side-effect on the type system when the implementation is finally given, and this side-effect seems to be inherent in the notion of abstract data type, since abstract data types can be given new implementations.[26]

We make the claim that recursive data types for persistent languages with the type `TYPE` are most naturally defined as abstract data types whose *implementation* depends upon the abstract data type itself. In other words, since we wish to provide abstract data types anyway, and since the abstract data type provides us with the side-effect we need to achieve recursive type circularity, we might as well enjoy the conceptual simplicity this scheme provides. There is then no need for complex structural equivalence algorithms for recursive types, as is required for Algol-68, because the cycles in all recursive types are now broken by an abstract data type which must compare identically. Furthermore, the intermediate states involved in building a recursive data type are usually visible from a parallel process in a parallel/distributed/persistent system having explicit type objects, in which case the type will display its mutability, and therefore may as well be an abstract data type.[27]

In summary, we have described a type system which provides a small set of primitive immutable types (*boolean*, *bit*, etc.), a small set of primitive type constructors (*product*, *sum*, *function*, etc.), an operation to create a new, mutable opaque type (not to be confused with a Milner-style type *variable* [Milner78]), and an operation to assign a mutable type a type-expression value. Given this type system, the type identity/type equivalence problem is solved by the straight-forward application of our `EGAL` equality predicate. Type values are finite trees which are constructed from leaves which are either immutable atomic types (*boolean*, *bit*, etc.) or mutable opaque types. The backbones of these trees are immutable structures (*product*, *sum*, etc.). `EGAL` performs structural equivalence down to the leaves, which compare for identity.

The difference between languages with structural type equivalence and languages with name type equivalence is now clear. Languages with structural type equivalence (e.g., Algol-68) use a permanent binding of names to type values, so that the type comparison recurses through the type name. Languages with name type equivalence (e.g., Ada) use

---

[23]In simple cases these steps may be performed within a single type definition, but these rarely occur [Baker80].

[24]We purposely ignore the Common Lisp Object System [Steele90,ch.28], because the types it introduces are not completely opaque, and hence they are not true abstract data types. For example, the slot structure of a type can be ascertained by examining its "class precedence list".

[25]The new data type itself is the key issue for us; we ignore any operation signatures also given that refer to the new datatype.

[26]This updating involves a bit of checking to make sure that the operation signatures match, but these details do not change the side-effect nature of providing an implementation for an abstract data type.

[27]In an analogous situation, the internal side-effects involved in the implementation of Scheme's `letrec` can be exposed using continuations, allowing a functional subset of Scheme to emulate side-effectable cells! [Bawden88].

an assignable binding of names to type values, so that the type comparison stops at the type name. By forcing recursive types to be abstract data types, Ada could dispense with a whole set of rules and restrictions for "incomplete types", which are used in Ada to construct recursive types.

Our assertion that type identity is intimately bound up with the mutability of an abstract data type is essentially a *modal* concept. Modal logic [Hughes68] deals with the notions of "possibility" and "necessity", and Kripke models of these logics utilize the notions of "multiple worlds", where "possibility" is intimately related to "accessibility" of other worlds. A data type is thus abstract if we can "conceive of" multiple implementations of it; i.e., if there exist worlds accessible from the current world in which the implementation differs. In modal "dynamic logic" of programs [Pratt79], accessibility is achieved through one or more assignment statements, so it should be possible to unite these two concepts. This modal notion of abstract data types is closely related to the "existential" notion of abstract data type espoused by [Cardelli85] and [Mitchell88].

B. Comparing Objects of Abstract Data Types

Abstract data types [Guttag77] are so-called because they present the specification of a set of objects which is abstracted from the representation of those objects. It is therefore important to distinguish between the external (specificational) view of an abstract object and the internal (representational) view of the object. The simplest treatment of object identity for abstract objects is to ignore the abstraction and use the object identity of the representation; this choice is the only one available to programmers in languages which do not support abstract data types. This simple treatment has two drawbacks—1) it allows the equivalencing of two objects which *accidentally* have the same representation—e.g., the rational number 2/3 and the Gaussian integer 2+3*i*; and 2) it requires that the abstract data type representations always be kept in *canonical form*—e.g., Ada has this problem because its "=" operator is not easily overloaded with any other interpretation. For example, the comparison of (functional) rational numbers would be performed by comparing their numerators and denominators; this comparison would only work correctly for pairs with no common divisors in which the sign appeared uniformly in either the numerator or the denominator. When a canonic form exists, it can greatly improve the efficiency and reliability of the overall program, because the equality predicate is under the complete control of the implementation and hence optimized and trusted. Unfortunately, some abstractions may not have a canonical form, or else continual canonicalization may be inefficient. In these cases, the programmer may wish to offer an *abstract object identity* for objects of the abstract data type which is different from the representational object identity; he will then need to overload (genericize) the EGAL predicate.

EGAL avoids the confusion between abstract and concrete objects by its first clause which requires that the datatypes themselves be EGAL. Since an abstract datatype itself includes a mutable cell which carries the type's identity, it is this cell which is compared when abstract datatypes are compared. An object of the abstract type can be "opened up" to reveal its *representation*, which is another object of another—usually concrete—type. By requiring the conjunction of abstract type equivalence and concrete representation equivalence, EGAL will distinguish between abstract objects which may be abstractly equivalent, but have different representations—perhaps as the result of a new implementation of the abstract type having been instantiated. Although this equivalence may be finer than we might desire, at least it will not lead to contradictions.

```
(defun egal-abstract-type (x y)
  (and (egal (type-of x) (type-of y))
       (egal (representation x) (representation y))))
```

We would now like to be able to overload EGAL for abstract data types in a way that provides a more abstract equality, but does not compromise EGAL's basic principles. If the proper overloading of EGAL can be automatically derived from the other information given in defining the abstract data type, then EGAL's nature will be preserved. We can derive the requirements for abstract EGAL through a number of observations.

Rule I. Two abstract objects can be EGAL only if they have the same (abstract) data type.

Rule II. EGAL operating on abstract objects (abstract EGAL) cannot be finer than EGAL operating on their representations (representational EGAL). This rule also guarantees that abstract EGAL is reflexive.

Rule III. Abstract EGAL must be symmetric. A smart compiler could check this condition at compile time by comparing the given user code to the user code with its parameters reversed; if the two are equivalent, then the user code is symmetric.

Rule IV. Abstract EGAL must be transitive. A smart compiler could check this condition at compile time by noticing when the abstraction calls another predicate which is already known to be transitive—e.g., if it calls EGAL on component values.

Some of the above rules can be easily enforced by inserting any user definition for `EGAL` into the following prototype:

```
(defun abstract-egal (x y)
  (and (egal (type-of x) (type-of y))             ; Rule I.
   (or (egal (representation x)
             (representation y))                  ; Rule II.
    (let ((user-egal (get-egal (type-of x))))
      (or (funcall user-egal x y)
          (funcall user-egal y x))[28]            ; Rule III.
       ...))))                                    ; other Rules.
```

Unfortunately, the requirement for transitivity cannot be so easily guaranteed. For example, the transitivity of the grade-school cross-product rule for comparing rational numbers depends upon deep properties of integer multiplication—associativity, commutivity and cancellation. If one is preparing "mission-critical" or "standards-quality" abstract data types, it might be reasonable to ask for a mathematical proof of transitivity before allowing the compilation of a user `EGAL` definition. Under normal conditions, however, this would be impractical. Short of a mathematical proof, we can achieve some assurance from the prototype above; we believe this approach preferable to the approach of many "object-oriented" languages which allow any behavior of an identity predicate whatsoever.[29]

An even more difficult case is that of an abstract object whose specification claims the object to be immutable, but whose representation utilizes mutable objects and *benevolent* side-effects. The most obvious example is that of the use of a reference to RAM storage to represent a functional object (e.g., a Lisp *bignum*); traditional abstract datatype systems handle this situation by protecting the object from outside access either intentionally, through the published operations, or unintentionally, by clobbering storage through wild pointers. A more sophisticated example is the use of benevolent side-effects to improve the performance of functional arrays through *shallow binding* [Baker91b] (also called *trailers* [Bloss89]). If the implementation is flawed in some way that allows the internal state to become visible, then havoc will result. Proofs of immutable behavior are even more difficult than in the rational number case, above, although work is progressing on type systems that can automatically check simple cases [Gifford86] [Lucassen87].

As a result of these problems, we recommend that `EGAL` remain a simple, reliable, decidable predicate which compares types and representations, as in the following code. While `EGAL` will sometimes provide a finer equivalence relation than the abstract one desired, it cannot cause embarrassment due to violation of the basic rules of object identity, because it tests for identity as preserved by the basic value transmission operations of the programming language.. Furthermore, this `EGAL` is consistent with the earlier definition of `EGAL` on closures, which are often used to implement abstract data types (this consistency can be seen by equating the notion of an abstract type object with the notion of the code for a closure). Systems defining abstract datatypes might consider providing a new "generic" predicate that defaults to `EGAL` for primitive datatypes and can be overloaded for user-defined abstract datatypes; this generic predicate would then be used for table lookup routines such as `assoc`, `member`, etc., instead of passing an equality predicate as an argument.

```
(defun egal (x y)
  (and (egal (type-of x) (type-of y))
    (cond ((abstract-type-p (type-of x))
           (egal (representation x) (representation y)))
          << the other clauses for egal, as before. >>
           )))
```

9. COMPARISON WITH PREVIOUS WORK

Philosophers and logicians—including Aristotle—have long discoursed on the *properties* and *attributes* of *objects*. Von Wright, according to [Hughes68,p.184], distinguishes between *formal* properties, "whose belonging to an object is always either necessary or impossible", and *material* properties "whose belonging to an object is always contingent". If we interpret "necessary" and "possible" according to the modal systems of "dynamic logic", which

[28]A smart compiler can often optimize out this second call to `user-egal` which guarantees symmetry.

[29]We note that this `EGAL` prototype does not conform to the usual object-oriented dispatching mechanisms (e.g., Smalltalk and Common Lisp Object System (CLOS) [Steele90,ch.28]), because we do not call the specific user function first. In CLOS, `EGAL` would be a non-generic function which eventually called the specific user function.

models imperative sequencing and assignment [Pratt79], then formal properties are equivalent to functional properties (components) of an object while material properties are equivalent to mutable properties (components) of an object. Obviously, material properties—being contingent—cannot contribute to an object's identity, while formal properties—being necessary—are an inherent part of an object's identity.

McCarthy's *Lisp* language[McCarthy60] clearly demonstrated his understanding of side-effect semantics for object identity through his use of "mark bits" in a mark-sweep garbage collector.[30] His `EQ` function was restricted to work only on *atomic* (non-structured objects), and his recursive `EQUAL` function worked correctly for all objects because his cons cells were immutable. His followers forgot to re-examine the situation, however, once `RPLACA` and `RPLACD` had been introduced into Lisp, perhaps for reasons of compatibility.

[Steele78,39-43] came tantalizingly close to drawing our conclusion: "The concept of side effect is inseparable from the notion of equality/identity/sameness", and later: "the only way one can determine that two objects are the same is to perform a side effect on one and look for an appropriate change in the behavior of the other". This quote makes inexplicable the later conclusion of the ANSI Common Lisp committee, of which Steele was a member, that "object equality is not a concept for which there is a uniquely determined correct algorithm" [Steele90,p.109].[31] The radical consequences of the obvious conclusion were apparently too much for either Steele or the ANSI committee to accept—that there were *two* different kinds of list cells, mutable and immutable, with coercions between them, and recognizing this might have sacrificed upwards compatibility.

Common Lisp [Steele84] [Steele90] defines the predicates `EQ`, `EQL`, `EQUAL`, `EQUALP`, which are listed in fine-to-coarse order. In addition, Common Lisp offers `"="`, which is `EQUALP` restricted to numbers. `EQ` essentially compares pointers, while `EQL` relaxes `EQ` and allows functional objects like characters and numbers to be correctly compared.[32] `EQUAL` structurally compares conses, strings and bit-vectors, while `EQUALP` structurally compares all vectors and fuzzes upper/lowercase character distinctions. Unfortunately, Common Lisp arrays cannot be defined as "read-only", so functional arrays (including functional strings and bit-vectors) cannot be defined. Common Lisp does allow for "read-only" components of structures, but `EQUAL` does not decompose structures and `EQUALP` decomposes all structures.

Scheme [Rees86] defines the predicates `eq?`, `eqv?` and `equal?`. `eq?` is essentially pointer comparison (Common Lisp's `EQ`), while `eqv?` is a crude attempt at achieving "operational equivalence", which is not at all well-defined; `eqv?` is approximately the same as Common Lisp's `EQL`, except on functional closures. Finally, `equal?` is Scheme's approximation to Common Lisp's `EQUALP`, since it approximates structural equivalence. While Scheme allows for `eqv?` to implement our version of `EGAL`, it does not require it, and we are not aware of any Scheme which follows our semantics.[33]

Goto [Goto74] [Goto76] researched the notion of "hash consing" (invented by Ershov for common subexpression detection [Ershov58]) which provided an efficient implementation of `EQUAL` for *functional* cons cells. However, we are more interested here in the fact that he provides a separate datatype for functional cons cells than in the fact that they can be efficiently compared. He provides the same semantics for `EQ` that we do for `EGAL`, but accomplishes this by forcing the "uniquizing" of functional cons cells so that the existing `EQ` works correctly. We, on the other hand, do not constrain the implementation of `EGAL`, but allow `EGAL` to handle non-uniquized functional cons cells.

MacLisp [Moon74] was the first Lisp to provide stack-allocated numbers which became "first-class", when they escaped the boundaries of dynamic extents [Steele77]. As a result, `EQ` for numbers became problematical, since numbers could be moved without notice, and these techniques resulted in Common Lisp's `EQL` predicate. This paper can be considered a generalization of this `EQL` technique to all functional objects. The generalization of the stack-allocation techniques of Maclisp to arbitrary objects is considered in [Baker92a]. MacLisp also had a function

[30]The idea of mark-sweep garbage collection is at least 4,000 years old; the Jewish holiday *Passover* celebrates the effectiveness of the first distributed marking algorithm against a particularly deadly sweeping algorithm [*Exodus*12:23-27].

[31]The recent incorporation of CLOS (Common Lisp Object System) into Common Lisp requires the precise definition of object identity, because CLOS generic functions can "dispatch" on particular objects (using `EQL`).

[32][Moon89] suggests that "the Common Lisp function `EQL` compares two object references and returns *true* if they refer to the same object". While Moon's statement is usually correct, most people mistakenly read the "if" as though it said "if and only if".

[33]Curiously, [Rees86] defines operational equivalence for *mutable* structured objects, but not for *immutable* (functional) structured objects, and hence operational equivalence is ill-defined.

`PURCOPY`, which coerced its argument into an immutable form for sharing among processes in a time-sharing system. MacLisp had no predicates to distinguish pure from impure objects, however.

AutoLISP [Autodesk88] is the only Lisp we know of which offers only functional strings and functional cons cells. Unfortunately, it is not pure, because it offers `SETQ`. String and cons cell functionality is not the result of theoretical considerations, but the result of swapping objects to disk using `PRINT` and `READ`. This is because any side-effects (e.g., sharing) to non-atoms are forgotten when the object is swapped in.

Parallelizing Lisp has been the subject of much research. [Baker77] introduces the concepts of *eager* evaluation and *futures*; *MultiLisp* [Halstead84] [Halstead85] implements and studies these concepts in great detail. *ParaTran* [Katz86] [Tinker88] utilizes the ideas of "time warp" to synchronize sequential Scheme on a parallel processor. *QLisp* [Gabriel84] introduces futures into Common Lisp. [Larus89] investigates aliasing in Lisp data structures, while [Harrison88] suggests new—more functional—list structures for Lisp in a parallel processing environment. Parallel Lisp constructs for SIMD architectures are considered in [Hillis85] and [Steele86].

Smalltalk [Goldberg83] [Digitalk88] has two equality predicates—"`==`", for "object identity", and "`=`", for "equality". Smalltalk's "`==`" is roughly Lisp's `EQ`, while Smalltalk's "`=`" (on built-in classes) is roughly Lisp's `EQUALP`, which descends into objects regardless of their mutability. Smalltalk intends "`=`" to be an equivalence relation coarser than "`==`", but a programmer can define "`=`" to mean anything at all, including "`not ==`". Smalltalk has no notion of functional objects which can be compared extensionally.

Prolog [Warren77] implicitly assumes the existence of object identity and an object identity predicate for the operation of its unification algorithm. Since most Prologs are "pure"—no side-effects to data structures—the proper definition of object identity has never been an issue. However, recent attempts to integrate Prolog-like mechanisms into (impure) Lisp [Robinson82] and Scheme [Ruf89] require a more precise notion of object identity; e.g., [Ruf89] incorrectly uses `eq?` rather than `eqv?`.

Algol-68 [vanWijngaarden77] is apparently the first standard language to rigorously separate the concepts of "value" and "name" (" mode ref" ≈ "assignable cell"), which model has been followed most closely by ML. Algol-68 provides the object identity predicate ":=:" only for two name/refs, which compares the names themselves for identity, not the currently-assigned values [vanWijngaarden77,5.2.2]; any attempt to compare a name to a value yields a compile-time error, since a name/ref does not have the same mode/type as a value. The "=" operator is initially overloaded only for the arithmetic, boolean and character types, but the programmer must overload "=" himself for any other type. Algol-68 does not have abstract data types, but does allow for recursively defined types which must be extensionally compared, thereby producing the complexity traditionally associated with Algol-68 type equivalence.

CLU [Liskov77] has a concept of object identity which is similar to ours, in that it distinguishes mutable from immutable objects. CLU defines mutability relative to the operations supplied by the programmer of an abstract data type, however, making immutability substantially more difficult to prove [Bloom76]. CLU also allows for the definition of recursively defined types "without explicit reference types" [Liskov77]; however, the example given in that paper contradicts this assertion through its use of assignment. Herlihy [Herlihy82] describes a mechanism for changing the representation of an abstract "value" during transmission in CLU, and uses the notion of "value equality" to determine the fidelity of this transmission. Unfortunately, the definition of value equality is left up to the particular abstract datatype, and this predicate does not have to be an equivalence relation—e.g., his example of floating-point numbers and rectangular/polar complex numbers uses *closeness* instead of *equality*. While Herlihy uses the term "call-by-value", he actually describes "call-by-object-reference" (although he uses the term "name" instead of "object reference"), because true call-by-value wouldn't care about sharing and cycles. Finally, his attempts to preserve sharing and cycles are not well-defined, because he does not acknowledge the mutability of "names", even though his implementation works exactly like a copying garbage collector, including the updating of forwarding pointers stored in the name-to-object "maps". Since the object identity of these "names" is not preserved outside a single run of a particular program, he underestimates the real costs of full object identity in a persistent database.

ML [Harper86] has a notion of equality very similar to ours. "On references [assignable cells], equality means identity; on objects of other types ..., it is defined recursively in the natural way" [Harper86,s.7.2]. ML makes no attempt to compare function closures, however, even when they simulate data structures. Ohori [Ohori90] makes the case that ML references (cells) are the most appropriate means for introducing object identity into a pure functional language.

C [ANSI-C] has only a single predicate "`==`" which acts like Common Lisp `EQ`. C "`==`" requires the objects being compared to be either arithmetic values (including character values) or pointers. C accesses arrays only through pointers, so C "`==`" does not descend into the contents of arrays. Since C does not define "`==`" for structures, it finesses the issue of how best to compare them. C approximates the notion of a functional object through the "`const noalias`" declaration, which can be applied to structure components; C "`==`" does not cope with this declaration, however, because pointer comparison compares addresses. *C++* [Stroustrup86] follows C, but allows "`==`" to be overloaded. There are no requirements on user-supplied definitions for "`==`"; in fact, "`!=`" could be defined as equality and vice versa.

Ada [Ada83] has a single predicate "`=`" which acts like Common Lisp `EQUALP` because it recurses on the components of structures and the elements of arrays. Ada "`=`" does not descend through pointers, however, so it cannot run into the possible circularity of `EQUALP`. Object identity for accessed objects is determined by address, which is consistent, because Ada does not allow the declaration of constant (non-assignable) aggregate components. Since "`=`" compares the contents of aggregate variable components, "`=`" is not referentially transparent. "`=`" in Ada is not easily overloaded, and since the functionality of an aggregate structure is not encoded in its type and cannot be determined by the program at run-time, it is difficult to implement our `EGAL` semantics.

While one cannot declare a component of an Ada object immutable ("constant"), one can declare an "entire" object constant. Unfortunately, if one compares two pointers to such constant objects, the pointers are not dereferenced in the comparison—i.e., they are compared intensionally instead of extensionally. In a noble attempt at cleanliness, Ada restricts formal mode "**in**" parameters of subprograms to be "read-only". This restriction is an attempt to convey the notion that arguments are coerced into immutable values before being passed as arguments. Unfortunately, this restriction is independent of the arguments, and does not extend to pointer-dereferencing. Ada's read-only parameters thus cause more confusion than they resolve.

Ada retains the notion of "mode" from Algol-68, but separates it from the notion of "type". Ada modes are orthogonal to Ada types, and apply only to parameter-passing. Although one can utilize Ada's "**in out**" mode to partially preserve object identity for *variables* (Ada's mutable objects) [Baker91b], object identity in Ada is generally preserved only for "access" types—i.e., Algol-68/ML *references*, i.e., *pointers*. Much of Ada's complexity in semantics and implementation is a result of its unwise decision to separate modes from types, and from its attempt to emulate "cache coherency" itself in software using mode "**in out**".

Hewitt's *Actor* systems [Atkinson77] [Hewitt78] [Agha86] incorporate the notion of "serialized" and "unserialized" actors. Actors with state are serialized, while functional actors need not be serialized. Serialized actors and unserialized actors correspond roughly to objects with "object identity" and functional objects, respectively. The discrete linear time ordering required of any actor with internal state [Hewitt78] is essence of "serializability", required for (cache) coherence.

MacLennan [MacLennan82] [MacLennan85] independently provides arguments quite similar to ours for a notion of object identity essentially the same as ours. He does not, however, consider the equality problem for function closures.

Gifford [Gifford86] has been examining the inclusion of side-effect information into type systems. He has not examined the issue of object identity, but his type systems should be capable of providing the information necessary to implement our `EGAL` predicate.

Linda [Carriero89] utilizes a pattern-matched database of structured values ("tuples") for communication among independent processes in an MIMD parallel processing environment. A Linda *tuple* is a functional data structure which can have additional structures as components. Tuples are not recursive and are compared recursively—i.e., tuples are finite functional objects, and Linda has no mutable objects other than the global database itself.

Relational data base theory has the concepts of "key", "primary key", "functional dependency" and "normalization". Functional dependencies can be operationally exposed through "update anomalies" which occur when functional dependencies are violated. Update anomalies can be minimized through normalization, which factors the relations according to their functional dependencies. Normalization attempts to whittle an object's identity down to its "primary key", while we grow the object identity to include all of its functional dependencies. Immutable relations cannot be updated, however, and therefore cannot be used to detect update anomalies and functional dependencies. Since these functional dependencies can all be viewed as immutable attributes of the object (whether direct attributes or transitive attributes dependent upon an immutable direct attribute), the inclusion of all of these attributes as part of the object cannot cause an inconsistency. The essence of our argument, when rephrased in relational data base terminology, is that whether immutable relations are normalized can't be determined through update anomalies, and hence can't matter.

Vianu's conception of "object identity" [Vianu88] seems identical to ours, although it is couched in the form of relational algebra rather than in the form of object-oriented programming. Earlier work [Lyngbaek87], however, separated the world into "literal" (i.e., functional/immutable) objects and "non-literal" (i.e., objects with "true" object identity).

10. CONCLUSIONS

Although efficient object-oriented applications can be designed and built in any language, clean semantics for object identity are required in order to eliminate the need for heroic measures [Baker91c]. We have shown that object identity is best defined by the transitive closure of immutable attributes of an object. Object identity is thus defined by side-effect semantics in a way analogous to the way that normal forms in relational algebra are defined by update anomalies.

Our discussions have revealed the `EQ`/`EQUAL` problem to be a straight-forward typing error. Applying `EQ` to a functional list or applying `EQUAL` to a mutable list are type errors of the same sort as applying a floating-point equality predicate to two integers. That mutability has not previously been considered an important issue in programming language typing must be the result of an artificially low price on the assignment operation in von Neumann architectures.

Our primitive `EGAL` predicate is the coarsest predicate consistent with object identity, and therefore `EGAL` should be the finest equality predicate primitive in a language. In this way, many inconsistencies and anomalies disappear, including some especially troubling ones involving hash tables and property lists. Our notion of object identity provides a firm foundation for the introduction of immutable list cells, immutable arrays and strings, and immutable structures into modern programming languages, including Common Lisp. These immutable objects immediately solve many problems which have troubled language standards committees; for example, many of the problems of "typing for declaration" and "typing for discrimination" [Steele90,p.53] stem from imprecise object identity. Immutable objects can also lead to improved performance in persistent, parallel and distributed execution environments.

Our notion of object identity provides for a clean semantics of argument-passing and value-returning, eliminating many *ad hoc* and confusing rules with a single notion—*call-by-object-reference*. Call-by-object-reference is precise, and is robust when used in persistent, parallel and distributed systems.

By providing a better understanding of object identity and its costs, we hope to advance the cause of "mostly functional programming" [Knight86], which tries to reduce the number of "gratuitous" side-effects. Mostly functional programming results in higher efficiencies in persistent, parallel and distributed processing environments because logging, communication and synchronization costs are reduced.

ACKNOWLEDGEMENTS

Many thanks to Matthias Felleisen, Dan Friedman, Robert Kessler, André van Meulebrouck, and Carolyn Talcott for their comments on early drafts of this paper. We thank the referees for bringing the details of object identity in Algol-68 and CLU to our attention, and for their other helpful suggestions.

REFERENCES

Abelson, H., & Sussman, G.J.. *Structure and Interpretation of Computer Programs*. MIT Press, Camb., MA, 1985.

Abiteboul, Serge, and Kanellakis, Paris C. "Object Identity as a Query Language Primitive". *Proc. 1989 ACM SIGMOD Conf., Sigmod Record 18*,2 (June 1989),159-173.

AdaLRM: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Adams, Norman, and Rees, Jonathan. "Object-Oriented Programming in Scheme". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.,* Snowbird, UT, 1988,277-288.

Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Camb., MA, 1986.

Aho, A.V., *et al*. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Alagic, Suad. *Object-Oriented Database Programming*. Springer-Verlag, New York, 1989.

ANSI-C. *Draft Proposed American National Standard Programming Language C.* ANSI, New York, 1988.

Atkinson, Malcolm P., and Buneman, O. Peter. "Types and Persistence in Database Programming Languages". ACM *Computing Surveys 19*,2 (June 1987),105-190.

Atkinson, M.P., Buneman, P., and Morrison, R. (eds.) *Data Types and Persistence*. Springer-Verlag, Berlin, 1988.

Atkinson, M.P., Bancilhon, F., DeWitt, D., Dittrick, K., Maier, D., and Zdonik, S. "The Object-Oriented Database System Manifesto". *Proc. First Deductive and Object-Oriented Database Conf.,* Kyoto, Japan, Dec. 1989.

Autodesk. *AutoLISP® Release 10 Programmer's Reference*. TD111-05.2, Autodesk, Inc., Sausalito, CA, 1988.

Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". *CACM 21*,8 (Aug. 1978),613-641.

Baker, Henry, and Hewitt, Carl. "The Incremental Garbage Collection of Processes". *Proc. ACM Symp. on AI and Prog. Langs., Sigplan Notices 12*,8 (Aug. 1977),55-59.

Baker, Henry. "List Processing in Real Time on a Serial Computer". *CACM 21*,4 (April 1978),280-294.

Baker, Henry. "A Source of Redundant Identifiers in Pascal Programs". *ACM Sigplan Not. 15*,2 (Feb. 1980),14-16.

Baker90a: Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Funct. Prog.*, June 1990.

Baker90b: Baker, Henry. "Efficient Implementation of Bit-vector Operations in Common Lisp". ACM *LISP Pointers 3*,2-3-4 (April-June 1990),8-22.

Baker90d: Baker, Henry. "The Nimble Type Inferencer for Common Lisp-84". Tech. Rept., Nimble Comp., 1990.

Baker91a: Baker, Henry. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *ACM Ada Letters XI*,5 (July/Aug. 1991),79-90.

Baker91b: Baker, Henry. "Shallow Binding Makes Functional Arrays Fast". *ACM Sigplan Not. 26*,8 (Aug. 1991),145-147.

Baker91c: Baker, Henry. "Object-Oriented Programming in Ada83—Genericity Rehabilitated". *ACM Ada Letters XI*,9 (Nov./Dec. 1991),116-127

Baker92a: Baker, Henry. "CONS Should not CONS its Arguments, or A Lazy Alloc is a Smart Alloc". ACM *Sigplan Notices 27*,3 (March 1992),24-34.

Baker92b: Baker, Henry. "The Buried and Dead Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements". ACM *Lisp Pointers V*,2 (Apr-June 1992), 11-19.

Bawden, A. Pure Scheme emulation of cells from network mail circa 1988. Obtained from M. Felleisen.

Bird, R.S. "Tabulation Techniques for Recursive Programs". *ACM Comp. Surveys 12*,4 (Dec. 1980),403-417.

Bloom, Toby. "Immutable Groupings". CLU Design Note 61, MIT LCS, Aug. 16, 1976.

Bloss, A. "Update Analysis and the Efficient Implementation of Functional Aggregates". *Proc. 4th ACM/IFIP Conf. Funct. Progr. & Comp. Arch.*, London, Sept. 1989,26-38.

Burton, F. Warren. "A Note on Higher-Order Functions versus Logical Variables". *Info. Proc. Let. 31* (1989),91-95.

Cardelli, Luca, and Wegner, Peter. "On Understanding Types, Data Abstraction, and Polymorphism". ACM *Computing Surveys 17*,4 (Dec. 1985),471-522.

Carriero, N., and Gelernter, D. "Linda in Context". *CACM 32*,4 (1989),444-459.

Cohen, J.M., and Cohen, M.J. *The Penguin Dictionary of Quotations*. Penguin Books, Middlesex, England, 1960.

Cointe, Pierre. "Metaclasses are First Class: the ObjVlisp Model". *Proc. OOPSLA '87, Sigplan Notices 22*,12 (Dec. 1987),156-167.

Digitalk, Inc. *Smalltalk/V 286 Tutorial and Programming Handbook*. Digitalk, Inc., Los Angeles, CA, 1988.

DoD. *"STEELMAN": Department of Defense Requirements for High Order Computer Programming Languages*, June 1978.

Ershov, A.P. "On Programming of Arithmetic Operations". *Doklady, AN USSR 118*,3 (1958),427-430, transl. Friedman, M.D., *CACM 1*,8 (Aug. 1958),3-6.

Gabriel, R.P., and McCarthy, J. "Queue-Based Multi-Processing Lisp". *Proc. 1984 ACM Symp. on Lisp and Funct. Prog.*, (Aug. 1984),25-44.

Gabriel, R.P. "The Why of Y". *ACM Lisp Pointers 2*,2 (Oct.-Dec. 1988),15-25.

Gifford, David K., and Lucassen, John M. "Integrating Functional and Imperative Programming". *Proc. 1986 ACM Conf. on Lisp and Funct. Progr.*, Aug. 1986,28-38.

Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. McGraw-Hill, New York, 1983.

Goto, E. "Monocopy and Associative Algorithms in an Extended Lisp". Univ. of Tokyo, May 1974.

Goto, E., and Kanada, Y. "Recursive Hashed Data Structures with Applications to Polynomial Manipulations". *SYMSAC 76*.

Graube, Nicolas. "Reflexive Architecture: From ObjVLisp to CLOS". *Proc. ECOOP '88*, Springer-Verlag, Berlin, 1988,110-127.

Gunn, H.I.E., and Morrison, R. "On the Implementation of Constants". *Info. Proc. Let. 9*,1 (1979),1-4.

Guttag, John. "Abstract Data Types and the Development of Data Structures". *CACM 20*,6 (June 1977),396-404.

Halstead, R. "Implementation of MultiLisp: Lisp on a multiprocessor". *Proc. 1984 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1984),25-43.

Halstead, R. "MultiLisp: A language for concurrent symbolic processing". ACM *TOPLAS 7*,4 (Oct. 1985),501-538.

Harper, R., *et al*. "Standard ML". Tech. Rept. ECS-LFCS-86-2, Comp. Sci. Dept., Edinburgh, UK, March, 1986.

Harrison, Luddy, and Padua, David A. "PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp". *Proc. 1988 Conf. on Supercomputing*, St. Malo, France, 1988,527-538.

Herlihy, M., and Liskov, B. "A Value Transmission Method for Abstract Data Types". *ACM TOPLAS 4*,4 (Oct. 1982),527-551.

Hewitt, Carl, and Atkinson, Russell. "Synchronization in Actor Systems". *Proc. POPL 4* (Jan. 1977),267-280.

Hewitt, Carl E., and Baker, Henry. "Actors and Continuous Functionals". *Proc. IFIP Working Conf. on Formal Descr. of Progr. Concepts*, Aug. 1977, in Neuhold, Erich ed., *Formal Description of Programming Concepts*. North-Holland, Amsterdam, 1978,367-390.

Hilden, J. "Elimination of Recursive Calls using a Small Table of 'Randomly' Selected Function Values". *BIT 16* (1978),60-73.

Hillis, W. Daniel. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

Hudak, P., and Bloss, A. "The aggregate update problem in functional programming systems". *Proc. 12'th ACM POPL*, Jan. 1985.

Hudak, P. "A Semantic Model of Reference Counting and its Abstraction". *Proc. 1986 ACM Lisp and Funct. Progr. Conf.*, Camb. MA,351-363.

Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages". ACM *Computing Surveys 21*,3 (Sept. 1989),359-411.

Hughes, G.E., and Cresswell, M.J. *An Introduction to Modal Logic*. Methuen and Co., London, 1968.

Hughes, John. "Lazy Memo-functions". *Proc. Funct. Progr. & Computer Arch.,* Nancy, France, 1985,129-146.

Kahan, W. "Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit". Proc. Jt. IMA/SIAM Conf. on The State of the Art in Numerical Analysis, U. Birmingham, April 1986, Iserles, A., and Powell, M.J.D., Eds., Clarendon Press, Oxford, 1987.

Kale, L.V. "The Chare Kernel Parallel Programming System". *Int'l. Conf. on Parallel Programming,* Aug. 1990.

Katz, Morris J. *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme.* M.S. Thesis, MIT, Camb., MA, June 1986.

Keller, R.M., and Lindstrom, G. "Toward Function-Based Distributed Database Systems". Tech. Rep. 82-100, Dept. of Computer Sci., U. Utah, Jan. 1982, 37p.

Kent, William. "A rigorous model of object reference, identity, and existence". *J. O.-O.Progr.*, (June 1991),28-36.

Khoshafian, Setrag N., and Copeland, George P. "Object Identity". *Proc. OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),406-416.

Kim, Won, and Lochovsky, Frederick H., *eds*. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading, MA, 1989.

King, Roger. "My Cat is Object-Oriented". in [Kim89],23-30.

Klimov, Andrei V. "Dynamic Specialization in Extended Functional Language with Monotone Objects". *Proc. ACM PEPM'91*, New Haven, CT, June 1991, 199-210.

Knight, Tom. "An Architecture for Mostly Functional Languages". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),105-112.

Kranz, David, *et al*. "Orbit: An Optimizing Compiler for Scheme". *Proc. Sigplan '86 Symp. on Compiler Constr.* (June 1986),219-233.

Krasner, Glenn, *ed*. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

Lamb, D.A., and Hilfinger, P.N. "Simulation of Procedure Variables using Ada Tasks". *IEEE Trans. Soft. Eng. SE-9*,1 (Jan. 1983),13-15.

Lang, Kevin J., and Pearlmutter, Barak A. "Oaklisp: An Object-Oriented Scheme with First Class Types". *OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),30-37.

Larus, James Richard. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph.D. Thesis, UC Berkeley, also published as Rep. No. UCB/CSD/89/502, May, 1989.

Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*,6 (June 1983),419-429.

Lieberman, Henry. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". *Proc. ACM/Sigplan OOPSLA'86*,214-223.

Liskov, *et al*. "Abstraction Mechanisms in CLU". *CACM 20*,8 (Aug. 1977),564-576.

Lomet, David B. "Objects and Values: The Basis of a Storage Model for Procedural Languages". *IBM J. Res. & Dev. 20*,2 (March 1976),157-167.

Lucassen, John M. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. Ph.D. Thesis, also MIT/LCS/TR-408, MIT, 1987,153p.

Lyngbaek, Peter, and Vianu, Victor. "Mapping a Semantic Database Model to the Relational Model". *Proc. ACM SIGMOD 1987 Conf., Sigmod Record 16*,3 (Dec. 1987),132-142.

MacLennan, Bruce J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.

MacLennan, Bruce J. "A Simple Software Environment based on Objects and Relations". *Proc. ACM Sigplan Symp. on Lang. Issues in Progr. Envs., Sigplan Not. 20*,7 (July 1985),199-207.

Maier, David, *et al*. "Development of an Object-Oriented DBMS". *OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),472-482.

Mason, Ian A. *The Semantics of Destructive Lisp*. Ctr. for the Study of Language and Info., Stanford, CA, 1986.
McCarthy, J. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *CACM 3*,4 (1960),184-195.
McCarthy, J., *et al*. *LISP 1.5 Programmer's Manual*. MIT Press, Camb., MA, 1965.
Milner, Robin. "A Theory of Type Polymorphism in Programming". *JCSS 17* (1978),348-375.
Mitchell, J.C. and Plotkin, G.D. "Abstract Types Have Existential Type". ACM *TOPLAS 10*,3 (July 1988),470-502.
Moon, David A. *MacLisp Reference Manual, Rev. 0*. Proj. MAC, MIT, April 1974.
Moon, D.A. "The Common Lisp Object-Oriented Programming Language Standard". in Kim, W., and Lochovsky, F.H., *eds. Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989,49-78.
Morris, J.H. *Lambda-Calculus Models of Programming Languages*. Ph.D. Thesis, MIT, 1968.
Moses, Joel. "The Function of FUNCTION in Lisp". Memo 199, MIT AI Lab., Camb., MA, June 1970.
Mostow, J., and Cohen, D. "Automating Program Speedup by Deciding What to Cache". *Proc. IJCAI-85*, L.A., CA, Aug. 1985, 165-172.
Novak, G.S., Jr. "Data Abstraction in GLISP". *Proc SIGPLAN'83, Sigplan Notices 18*,6 (June 1983),170-177.
Ohori, A., Buneman, P., and Breazu-Tannen, V. "Database Programming in Machiavelli—a Polymorphic Language with Static Type Inference". *Proc. 1989 Sigmod Conf.*, Portland, also *Sigmod Record 18*,2 (June 1989(,46-57.
Ohori, Atsushi. "Representing Object Identity in a Pure Functional Language". *Proc. 3rd Int'l. Conf. on Database Theory*, Paris, Dec. 1990.
Pacini, G., and Simi, M. "Testing Equality in Lisp-like Environments". *BIT 18* (1978),334-341.
Padget, Julian, and Nuyens, Greg. The EuLisp Definition, Version 0.6. Univ. of Bath, Bath, Eng., July, 1989.
Plotkin, G.D. "Call-by-name, call-by-value, and the lambda-calculus". *Theor. Comput. Sci. 1* (1975),125-159.
Pratt, V.R. "Process Logic". *ACM POPL 6*, (1979),93-100.
Pugh, William. "An Improved Replacement Strategy for Function Caching". *Proc. ACM Conf. on Lisp & Funct. Progr.*, Snowbird, UT, July, 1988,269-276.
Queinnec, Christian, and Cointe, Pierre. "An Open Ended Data Representation Model for Eu_Lisp". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.*, Snowbird, UT, 1988,298-308.
Radin, George. "The Early History and Characteristics of PL/I". *ACM Sigplan History of Prog. Langs. Conf., Sigplan Not. 13*,8 (Aug. 1978),227-241.
Rees, J. and Clinger, W., *et al*. "Revised Report on the Algorithmic Language Scheme". *Sigplan Notices 21*,12 (Dec. 1986),37-79.
Robinson, J.A., and Sibert, E.E. "LOGLISP: Motivation, Design, and Implementation". In Clark, K.L., and Tärnlund (eds), *Logic Programming*, Academic Press, 1982,299-314.
Ruf, Erik, and Weise, Daniel. "Nondeterminism and Unification in LogScheme: Integrating Logic and Functional Programming". *Proc. 4'th ACM Funct. Prog. Langs. and Computer Arch.*, Sept. 1989,327-339.
Sandewall, Erik. "A Proposed Solution to the FUNARG Problem". 6.894 course notes, MIT AI Lab., 1974. This solution was used in the MIT Lisp Machine [Greenblatt].
Sandewall, Erik. "Ideas about Management of Lisp Data Bases". AI Memo 332, MIT AI Lab., May 1975; also *Proc. IJCAI 4* (1975),585-592.
Sandewall, Erik. "Programming in an Interactive Environment: the Lisp Experience". ACM *Computing Surveys 10*,1 (March 1978),35-71.
Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1* (1975),161-194.
Schwartz, J.T. "Optimization of very high level languages—II. Deducing relationships of inclusion and membership". *J. Computer Lang. 1*,3 (1975),197-218.
Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages". *Proc. ACM/Sigplan OOPSLA'86*,38-45.
Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, AI Lab., MIT, May 1978.
Steele, G.L., and Hillis, W.D. "Connection Machine Lisp: Fine-grained Parallel Symbolic Processing". *1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),279-297.
Steele, G.L. "Fast Arithmetic in Maclisp". *Proc. 1977 Macsyma User's Conf.*, NASA Sci. and Tech. Info. Off. (Wash., DC, July 1977),215-224. Also AI Memo 421, MIT AI Lab., Camb., MA.
Steele, G.L., and Sussman, G.J. "The Art of the Interpreter, or The Modularity Complex (Parts Zero, One, and Two)". MIT AI Memo 453, May 1978, 73p.
Steele, G.L. *Common Lisp: the Language*. Digital Press, Burlington, MA, 1984.
Steele, G.L. *Common Lisp: the Language—Second Edition*. Digital Press, Bedford, MA, 1990.
Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

Swanson, Mark R., *et al*. "An Implementation of Portable Standard Lisp on the BBN Butterfly". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.*, Snowbird, UT, July 1988,132-141.

Swinehart, D., *et al*. "A Structural View of the Cedar Programming Environment". ACM *TOPLAS 8*,4 (Oct. 1986),419-490.

Teitelman, Warren. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, 1974.

Teitelman, W., and Masinter, L. "The Interlisp programming environment". *Computer 14*,4 (Apr. 1981),25-34.

Tinker, Pete, and Katz, Morry. "Parallel Execution of Sequential Scheme with ParaTran". *Proc. 1988 ACM Conf. on Lisp and Funct. Prog.*, (July 1988),28-39.

Turner, D.A. "A new implementation technique for applicative languages". *SW—Pract.&Exper. 9* (1979),31-49.

Ullman, Jeffrey D. *Principles of Database Systems*. Computer Science Press, Potomac, MD, 1980.

Ungar, David, and Smith, Randall B. "Self: The Power of Simplicity". *Proc. '87 OOPSLA, Sigplan Notices 22*,12 (Dec. 1987),227-242.

van Wijngaarden, A., *et al*. "Revised Report on the Algorithmic Language Algol 68". *ACM Sigplan Not. 12*,5 (May 1977),1-70.

Verity, J.W., and Schwartz, E.I. "Software Made Simple—Will Object-Oriented Programming Transform the Computer Industry?". *Business Week* cover story, Sept. 30, 1991,92-100.

Vianu, Victor. "A Dynamic Framework for Object Projection Views". ACM *TODS 13*,1 (March 1988),1-22.

Warren, D.H.D., Pereira, L.M., and Pereira, F. "Prolog—the language and its implementation compared with Lisp". *Proc. Symp. on A.I. and Prog. Langs., Sigplan Not. 12*,8 (Aug. 1977),109-115.

Wegbreit, Ben. "The Treatment of Data Types in EL1". *CACM 17*,5 (May 1974),251-264.

Welsh, J., *et al*. "Ambiguities and insecurities in Pascal". *SW—Prac. & Exper. 7*,6 (1977),685-696.

Wiebe, Douglas. "A Distributed Repository for Immutable Persistent Objects". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),453-465.

Wise, David. "Matrix algebra and applicative programming". In Kahn, G., *ed*., *Funct. Progr. Langs. and Computer Arch., Lect. Notes in Comp. Sci 274,* Springer-Verlag, 1987,134-153.

Wulf, W. A., *et al*. "BLISS: A Language for Systems Programming". *CACM 14*,12 (Dec. 1971),780-.

Young, J.W.A., *ed*. *Monographs on Topics of Modern Mathematics Relevant to the Elementary Field.* Longmans, Green & Co., 1911. Reprinted by Dover Publications, 1955.