Copy on Write

Francisco Javier Thayer Fábrega and Joshua D. Guttman

November 1, 1995

Abstract

The copy-on-write optimization is used in operating systems such as Mach [8] to avoid copying data when large portions of memory are sent between processes in messages. The kernel maintains data structures which allow the system to defer copying any data until either the sender or recipient needs to store a new value. After repeated copies, these structures may grow complex. In this paper, we formalize the copy-on-write optimization using IMPS, an Interactive Mathematical Proof System developed at The MITRE Corporation. We prove formally that the copy-on-write optimization provides the same user-visible behavior as if all the data had been copied immediately in the naïve way.

1 Introduction

One of the prime optimizations of Mach [8, 7, 1] is the so-called copy-on-write optimization. In an operating system like Mach, which emphasizes client-server computing, processes very frequently send other processes, which are acting as servers, large amounts of data. The server process may not even handle the data itself; indeed, frequently the data is simply sent to a third (or fourth ...) process before real actions are taken. Moreover, commonly, little or none of the data will be altered, although the client cannot know when the server will modify particular data. In this situation, traditional operating systems faced a dilemma. One possibility is to copy the data each time that it is sent in a message, but this takes time and is frequently unnecessary. Alternatively the memory to be transferred could be mapped into the virtual address space of the recipient as shared memory. Although this is fast, it is insecure, because the client can see any changes the server may make (and vice versa), and it is unreliable, because the processes must agree on a protocol for accessing and modifying the shared memory.

To resolve this problem, Mach offers the copy-on-write optimization. When memory is sent to another process, Mach does not in fact make a copy; the two processes will in fact read the same data. The microkernel ensures, however, that if either process writes to a portion of the shared memory, then Mach

will silently create a copy of that portion. Thus, each process sees only the modifications it makes itself. This is efficient because a portion of memory is copied only if this is necessary to provide each process with the illusion that memory was not shared.

In Mach, a "portion" of memory means a page as supported by the demand-paged virtual memory system. Copy-on-write is implemented by setting the page protection for shared pages to be read only; an attempt to write to the page causes a page fault. Mach responds to this page fault by copying the page containing the requested address, so that the write may be retried.

1.1 How Copy-on-Write Works

In order for this machinery to work properly, Mach must maintain data structures which indicate where to retrieve not-yet-copied pages and where to write them when they must indeed be copied. In Mach these interrelations are stored in the *memory object* data structures that the kernel maintains to represent the permanent or temporary objects that user processes have allocated in their address spaces.

When a copy is done using the copy-on-write optimization, the kernel creates *two* new memory objects, although no actual data is copied to either. One of the two is returned as the new object, and it will typically be sent to another process. The second is treated specially by the kernel; user processes will not access it. Instead, it serves as a snapshot of the state of the original memory object at the time that the copy was made. For clarity in the discussion below, we will refer to these three memory objects as the original O, the snapshot object S, and the user process's copy object C; we refer to the time the copy operation occurs as t_0.

Two fields in the memory object data structure are relevant; they are illustrated in Figure 1. First, there is a *default* pointer which is set to point from the user's copy C to the snapshot object S, and from S to the original O. Second, there is a *dependent* pointer which is set to point from the original object O to the snapshot object S.

The expected effects of reads and writes are altered in the presence of these pointers. First, any attempt to write to O will cause the page containing the target address to be "pushed" first, by which we mean copied from O to S before being altered. As a consequence, if a is any address at which the original O has data but S does not, then the value of O at a is the same as it was at time t_0. If a is an address for which S does have data, then the value of S at a is the same as the value that O had at time t_0; this is why we call S a snapshot. If a particular page of O has already been pushed, then writes to that page proceed normally, just as if no copy had been made.

Any attempt to write to C will cause the page containing the target address to be "pulled" first. By this we mean that if the data is not already present in C, the operating system copies the data there. If the page is not in physical

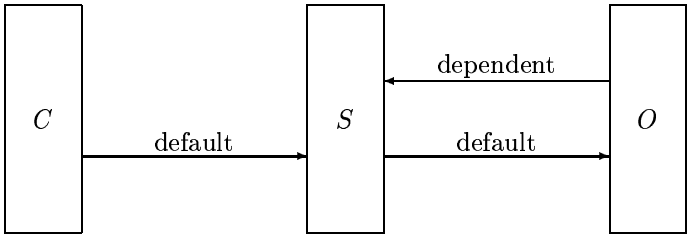

Figure 1: Relations Created by a Copy Operation

memory, the system locates the correct data for that page by following the default pointers. It may be found in S, or alternatively the operating system may follow S's default pointer to retrieve it from O. The page of data, once found, is then copied into C. As a consequence, whenever C does not have data for an address a, no write attempt has occurred, and the value expected for C at that address is the same as the value that O had at time t_0. Once the data has been pulled into C, this write attempt—and subsequent writes—may occur normally.

An attempt to read data from C will cause a search along the default chain. That is, if the data is present in C, then that value is used; otherwise, the default pointer is followed to S. Here again, either the data is present, or else the default pointer is followed to O. The result of the search will be cached. Later fetches from the same page do not require intervention from the operating system unless the virtual memory system has removed or relocated the page of data.

The situation becomes more complex when a second object is copied from the same original. Let us now refer to the initial copy as C_0 and the initial snapshot as S_0. The effect of the second operation depends on whether the original has been altered between t_0 and the time t_1 when the second copy C_1 is made.

When there have been alterations to O before t_1, so that pages have been pushed into S_0, then C_1 must see the new data in O. In this case, C_1 must have its own snapshot S_1. However, Mach ensures that when alterations are made to O, the pages need only be pushed to one snapshot object. This is achieved by changing S_0's default to point to S_1, as shown in Figure 2. If data has been changed between t_0 and t_1, that data is already present in S_0. Hence,

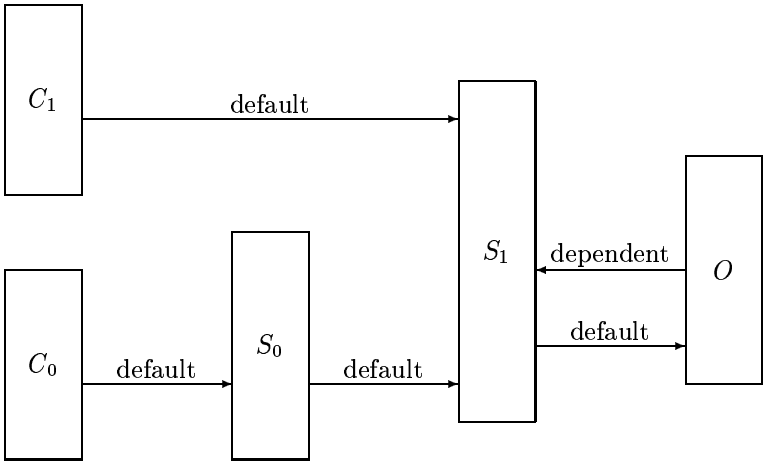

Figure 2: Relations Created by a Second Copy Operation (Original Altered)

the values present in S_0 will be used; later values represented in O or S_1 will never be visible to C_0. On the other hand, data in O altered only after t_1 will be copied to S_1. Since these pages are not represented in S_0, the values in S_1 will show through when Mach follows the default chain. For data that is never changed, the default chain will lead back all the way to O.

By contrast, if no writes have been made against O in the interval, then the initial snapshot S_0, which as yet contains no data, can be reused for both objects (see Figure 3). When, later, alterations are made to O, and the original unchanged pages are moved to S_0, these pages will be visible to both C_0 and C_1.

As a consequence of this scheme, reads against C_0 will deliver whatever values O held at time t_0, while reads against C_1 will deliver the values O held at time t_1. Moreover, an alteration to O never requires a page to be pushed to more than one snapshot object.

A succession of copies may be made from the same original. Thus, if the object changes frequently, the default chain may grow long. If copies are made more frequently than the original object changes, the structure may grow bushy. The mechanism in Mach has some additional fine points:

- A copy may start at a specified offset n in the original. In this case, the 0th word in C has the same value as the nth word in O, and generally when there is an ith word in C, it has the same value as the $n+i$th word in O.

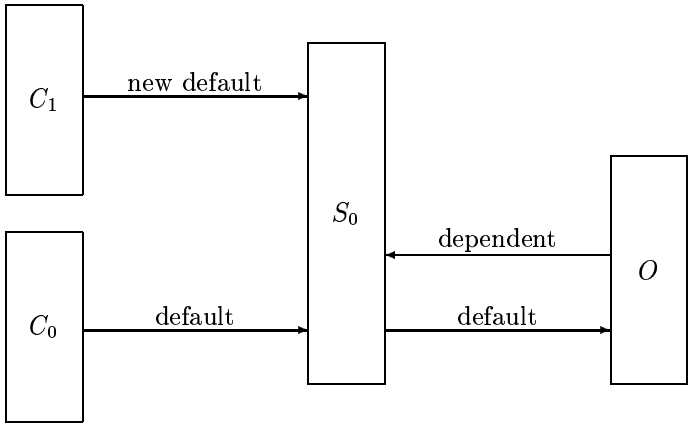

Figure 3: New Relation Created by a Second Copy Operation (Original Unaltered)

- A copy may have a limited length ℓ_C. In this case, the copy contains at most ℓ_C words; more exactly, it has $\min(\ell_C, \ell_O - n)$ words, where ℓ_O is the current length of the original and n is the offset, or 0 if none is specified.

Given that there is a flexible relation between the contents of the original and those of the copy, we found it convenient to specify the semantics of the copy operation in a way that allows any function to interrelate their locations.

1.2 Correctness Condition for Copy-on-Write

Intuitively, the copy-on-write mechanism seems just as good as actually carrying out all the copying; however, questions may remain about just what would occur with complicated sequences of copy operations. It is also not clear which states are legitimate, in the sense of which graph-like structures of *default* and *dependent* pointers would make starting points from which the copy-on-write mechanism behaves as expected.

In this subsection we will describe what correctness means for the copy-on-write mechanism. In the remainder of the paper, we will establish that it really is correct in this sense. As part of carrying out the proof strategy, we will also determine the set of states which provide legitimate starting-points for copy-on-write behavior.

The sense in which copy-on-write is as good as the unoptimized alternative, which we will refer to as *eager* copying, is that user-level processes receive the same services from the operating system no matter which is used, except that

they presumably receive that service more promptly when copy-on-write is used. The intuition is that user processes may request a sequence of actions, and that these actions will issue in the same visible consequences no matter which copy strategy the operating system selects. So if an applications program exhibits some behavior when the operating system offers eager copying, then it will exhibit just the same behavior (except possibly more quickly) if the operating system instead offers copy-on-write.

Thus we view the operating system as providing a collection of services to its clients, including operations to copy memory objects, and to fetch and store values as their contents. We regard these services as providing a state machine interface. In each state, a set of events is offered. Which set of events is available will depend on the state of the machine; this is how the model expresses information made available from the state machine (which, in our case, represents the kernel) to its environment (which, in our case, represents the user-level processes).

The fact that copy-on-write is a faithful refinement of eager copy may then be formalized as the assertion that one machine, which offers services including a copy operation using the copy-on-write strategy, is a faithful refinement of a different machine, which instead offers a copy operation using the unoptimized eager copy. We will refer to the latter as the "abstract" state machine, and to the former as the "concrete," or "implementation," state machine.

One strategy for formalizing this is to introduce an *abstraction function*. The abstraction function yields a state of the abstract machine, when given as argument a state of the implementation machine. The purpose of the abstraction function is to say which abstract state a concrete state corresponds to. An abstraction function may be used when there may be several concrete states, all of which represent the same abstract state. By contrast, in formalizing refinements in which a single concrete state may represent several different abstract states, an abstraction relation is needed, instead of an abstraction function. When a function is usable, as is the case in the current example, it is more convenient to do so. We will refer to the abstraction function as *abstr*.

The main correctness property is the claim that every computation[1] of the abstract state machine corresponds to a computation of the implementation machine and *vice versa*, in a certain sense of "correspond." To state it, we will use $\downarrow$ to mean "is defined," and we will say that $s \simeq t$ (read "s is quasi-equivalent to t") if $(s \downarrow \vee\ t \downarrow) \Rightarrow s = t$. Quasi-equivalence says that the terms have the same denotation or lack thereof, and in our logic [3, 2] it is the condition that justifies substitution of s for t, wherever s is free for t.

If C_i is a computation of the implementation, then an abstract computation C *corresponds* to C_i if there is a non-decreasing function $f : \mathbf{N} \rightharpoonup \mathbf{N}$ onto the domain of C such that $\mathrm{abstr}(C_i(j)) \simeq C(f(j))$, for all j. This notion of

[1] That is, a finite or infinite sequence C of states such that $C(0)$ is an initial state, and for every j where $C(j+1)\downarrow$, there is an operation h such that $C(j+1) = h(C(j))$.

refinement is closely connected with the *storage layout relations* used in VLISP [6, 5].

The refinement theorems are divided into safety theorems and liveness theorems. Suppose first that g is one of the specified operations, with parameters fixed, and h is the implementation version of that operation. Then the safety condition states that $h(\sigma_i) \downarrow \Rightarrow \text{abstr}(h(\sigma_i)) = g(\text{abstr}(\sigma_i))$.

To state the liveness constraints, we will call a function f on implementation states, which is a composition of state machine operations, an *abstract no-op* when $\text{abstr}(f(\sigma_i)) = \text{abstr}(\sigma_i)$. The liveness condition for an abstract operation g and the corresponding concrete operation h states that there is an abstract no-op f such that

$$\text{abstr}(h(f(\sigma_i))) \simeq g(\text{abstr}(\sigma_i))$$

This is a liveness condition: when the abstract machine can undergo operation g from $\text{abstr}(\sigma_i)$, then even if the implementation cannot undergo h from σ_i, it can evolve to an indistinguishable state $f(\sigma_i)$ from which it can undergo h.

In our case, the situation is slightly more complex. There is a contrast among two different ways that memory objects can be used in the Mach-style copy-on-write mechanism. First, there are the objects O and C, which are of interest to user-level processes, and produced at their request. Second, there are the snapshot objects S, which are not of interest to user-level processes, but are rather used by the operating system for its own bookkeeping purposes. These objects are intended not to be user-visible. Operations intended to be abstract no-ops, such as adding a snapshot, are in fact no-ops only if there is a way of masking off the visibility of these system-maintained objects.

One way to do so would be to use an implementation-level state which specified, in one of its components, which objects are user-visible in that state. There were two reasons why we have not done so. First, it appeared to be a distraction from the main line of development, focused on the structure of the default relations. Second, Mach itself does not appear to classify memory objects in this way. Although there is a notion of an internal memory object, in the current implementation (Mach 3.0 [1]), it is used differently. It coincides with "temporary" objects, which are all objects paged against the system's default pager. Many of these objects are user-visible, such as ordinary memory allocated in the address space of a user-level process. Instead, Mach appears to rely on its convention of naming objects by means of ports to ensure that user-level processes do not gain access to snapshot objects. This is a complicated mechanism, which relies on the assumption that no part of the code of the kernel or of the default pager will pass a right to the port to any other user-level process.

Thus, we have pursued a different idea. We will add a parameter to the abstraction function. Thus, given an implementation state and a set of "intended" user visible objects, the abstraction function returns an abstract state from which other objects have been hidden. Correctness theorems will naturally assume that certain objects are not user-visible.

In our case, the operations offered by the abstract machine will be:

- *a_copy*, to cause a new memory object C to have its data initialized to equal the data of a specified original memory object O;
- *a_fetch*, to retrieve a word of data from a memory object; and
- *a_store*, to alter the value of a word in a memory object.

The semantics of *a_store* are the eager semantics in which the data of C is initialized right away.

The implementation machine provides some similar operations, and also an additional operation. To avoid confusion, we will label corresponding operations in the implementation machine with a leading *i*:

- *c_copy*, to cause the defaults to be updated for a set of places;
- *c_redirect*, to re-route default pointers from one object to another (always a snapshot), when it can be used instead;
- *c_snapshot*, to cause the defaults to be updated for a set of places, when these are to be used as a snapshot object;
- *c_fill*, to cause a data to be copied from the source determined by the default chain;
- *c_fetch*, to retrieve the word stored in a place by dereferencing through the default chain;
- *c_store*, to alter the value of a word in a memory object, but possible only when the value is immediately present, without using the default chain.

The operation *i-fill* is used to implement both the "push" that copies a page of data from O to S and also the "pull" that copies a page of data to C, from either S or O. The operations *c_fetch* and *c_store* are trivial, and omitted below.

In the pages that follow, then, we will concentrate on the kernel level data structures. However, for our purposes it will be unnecessary to formalize the *dependent* pointers. Our formalization can always consider whether O is in the range of the *default* pointers. By contrast, the implementation must be able to efficiently discover the snapshot S that defaults to O.

We have also chosen here to abstract from the issue of page protection, and the handling of page faults when a process tries to store into a write-protected portion of a copy-on-write object. We have not postponed it because it raises difficult questions. On the contrary, we set it to one side because it seems completely amenable to the method developed in a previous paper [4].

2 The Abstract State Machine

In Mach, *memory objects* are the natural unit. Memory objects are used both to represent permanent storage objects such as files and also temporary storage such as a process's private address space. Conceptually, a memory object is a sequence of locations, indexed by natural numbers. Each location stores a word-sized datum. The association between locations and their contents is modelled by a function γ on $\mathcal{P}$. Thus $\gamma(m, n)$ is the word in memory object m at index n. However, for the correctness of the copy-on-write strategy this product decomposition is totally unnecessary. We thus view memory as a homogeneous set of points called *places* on which is defined a partial function representing stored data; the values that can be stored are called *words*. This abstraction allows the essential mathematical elements to become more apparent. In this model, immediate memory is represented by a partial function γ defined on the set of places. Though simplifying the mathematics, however, the change of view also requires that some commonly understood operations (such as copying a portion of one memory object to another) must be reformulated in a less constrained manner. We will specify a relation between source and target by giving a function f from places to places. When $f(p_1) = p_2$, that will indicate that the value for p_1 should be copied from p_2. The function, thus, maps target to source.

Language 2.1 *language-for-places*
Embedded Language: *h-o-real-arithmetic*
Base Types: $\mathcal{P}$ $\mathcal{W}$

The base types $\mathcal{P}$ and $\mathcal{W}$ denote *places* and *words* respectively.

Theory 2.2 *places*
Language: *language-for-places*
Component Theories: *h-o-real-arithmetic.*

In the view of memory as a product set, "copying" a memory object m to m' in effect extends the domain of the memory contents function γ to a new function γ' obtained by composition with the mapping $(m', i) \mapsto (m, i)$. This motivates the following:

Definition 2.3 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \Phi : \mathcal{P} \rightharpoonup \mathcal{W}$.

$\mathrm{copy}(\Phi, f) = \Phi \circ f$.

Theory: *places*

This general definition allows for copying with an offset, and it allows for copying a limited portion of a memory object.

We want copying to preserve existing memory and not to overwrite memory locations currently in use. On the other hand, when a copy is made, the source

should have contents. Therefore, a restriction must be imposed on copying functions; it is embodied in the following definition. The leading $\Longleftrightarrow$ should be read, "The following are equivalent."

Definition 2.4 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \Phi : \mathcal{P} \rightharpoonup \mathcal{W}$.

$\Longleftrightarrow$

- copy_fn(Φ, f)
- conjunction
  - $\forall x : \mathcal{P} \quad x \in dom\{\Phi\} \supset f(x) = x$
  - $ran\{f\} \subseteq dom\{\Phi\}$.

Theory: *places*

The abstract machine's copy operation may thus be defined:

Definition 2.5 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \Phi : \mathcal{P} \rightharpoonup \mathcal{W}$.

a_copy$(\Phi, f) =$ *conditionally, if* copy_fn(Φ, f)
- *then* copy(Φ, f)
- *else* $\perp[\mathcal{P} \rightharpoonup \mathcal{W}]$.

Theory: *places*

We treat the abstract machine's operation to store with similar generality:

Definition 2.6 *Let* $h, \Phi : \mathcal{P} \rightharpoonup \mathcal{W}$.

a_store$(\Phi, h) = [\, x : \mathcal{P} \mapsto$
conditionally, if $x \in dom\{h\}$
- *then* $h(x)$
- *else* $\Phi(x)\,]$.

Theory: *places*

On the other hand, the abstract machine's fetch operation retrieves a value for a single location. It returns the given state unaltered. However, to determine the word retrieved, it uses a "guessing" style. If the third argument is the correct guess for the value to be retrieved, the state is returned unaltered; otherwise, the result is undefined. This approach is discussed in more detail in [4].

Definition 2.7 *Let* $w : \mathcal{W}, p : \mathcal{P}, \Phi : \mathcal{P} \rightharpoonup \mathcal{W}$.

a_fetch$(\Phi, p, w) =$ *conditionally, if* $w = \Phi(p)$
- *then* Φ
- *else* $\perp[\mathcal{P} \rightharpoonup \mathcal{W}]$.

Theory: *places*

3 Concrete Machine: The State and its Properties

We turn next to the structure of the class of implementation states (denoted by Ω). An implementation state is an ordered pair consisting of a memory contents function γ and a default function δ. We will prove the main theorems in this section, before introducing the concrete machine's operations, and applying them to prove the correctness theorems, in Section 4.

Cartesian Product Sort 3.1 Ω
Components: $[\mathcal{P} \rightharpoonup \mathcal{W}]$ $[\mathcal{P} \rightharpoonup \mathcal{P}]$
Accessors: γ δ
Constructor: make_istate

Much of the mathematical content needed here concerns function iteration and a theory of discrete flows. That material has been developed in a more abstract context; it is presented in the Appendices. Here we will simply import it. The constant first_entry is introduced in Definition A.11, while flow_ext is introduced in Definition A.18.

Translation 3.2 *ind_2-to-place*
Source: *generic-theory-2*
Target: *places*
Fixed Theories: *h-o-real-arithmetic*
Sorts: $\mathbf{I}_1 \to \mathcal{P}$ $\mathbf{I}_2 \to \mathcal{W}$

We will overload the symbols first_entry and flow_ext, and use them for the corresponding notions in this more concrete theory of places and words.

Transported Symbols 3.1 *Transport:*
first_entry $\to$ first_entry flow_ext $\to$ flow_ext iterate $\to$ iterate
Translation: *ind_2-to-place*

In the concrete machine, the values stored immediately in memory are used in combination with the default pointers. If a place p has a value stored immediately, in the sense that $\gamma(\sigma)(p)$ is well-defined, then it represents that value. If it does not, then we consult the default function $\delta(\sigma)$. We apply $\delta(\sigma)$ as many times as needed to reach the immediately available memory. This is formalized using flow_ext.

Definition 3.3 *Let* $\mathrm{pl} : \mathcal{P}, \sigma : \Omega$.

$$\mathrm{promote}(\sigma, \mathrm{pl}) = \mathrm{flow\_ext}(\delta(\sigma), \gamma(\sigma), \mathrm{pl}).$$

Theory: *places*

The *bound memory* of a state is the set of places for which promote is well defined. It is written $\mathcal{P}_{\mathbf{B}}$.

Definition 3.4 *Let* $\sigma : \Omega$.

$$\mathcal{P}_{\mathbf{B}}(\sigma) = dom\{[\, p : \mathcal{P} \mapsto \text{promote}(\sigma, p)\,]\}.$$

Theory: *places*

Thus we will refer to the immediate memory of a state, meaning the set of p such that $\gamma(\sigma)(p)$ is well-defined, as well as the bound places of the state, which are all the places from which we can eventually reach immediate memory, by following default pointers.

Theorem 3.5 *bound%place-characterization*

$$\forall \sigma : \Omega \quad \mathcal{P}_{\mathbf{B}}(\sigma) = \{p : \mathcal{P} \mid \text{first\_entry}(\delta(\sigma), dom\{\gamma(\sigma)\}, p)\downarrow\}.$$

Theory: *places*

We introduce now an auxiliary abstraction function. This is not the abstraction function mentioned in the introduction; that has an additional parameter to specify the set of places intended to be user-visible. This auxiliary function corresponds to the case in which all places are intended to be visible.

Definition 3.6 *Let* $\sigma : \Omega$.

$$\text{abstr}(\sigma) = [\, m : \mathcal{P} \mapsto \text{promote}(\sigma, m)\,].$$

Theory: *places*

Observe that $\mathcal{P}_{\mathbf{B}}(\sigma) = \text{dom}(\text{abstr}(\sigma))$.

One ingredient of the copy-on-write strategy is the factorization of the copy function, copy $= \delta' \circ \delta'$, where δ' is an extension of the default function for the given state. One application of δ' brings us from the copy object to a snapshot; another is needed to bring us to the original. As the state evolves, the number of times that the default must be applied to bring us back to immediate memory may increase.

Thus, we are interested in the places x in the domain of the copy function, such there is some n such that $\delta^n(x) \in dom\{\gamma\}$.

The notation of the following definition requires some explanation: Given an $x : \mathcal{P}$, we are interested in the first entry of the iterates $f^n(x)$ in the set $\mathcal{P}_{\mathbf{B}}(\sigma)$. The first such iterate (if it exists) is of the form $f^{n_0}(x)$ for some integer n_0 depending on x. We will denote this $f^{[\sigma]}$.

Definition 3.7 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

$$f^{[\sigma]} = [\, x : \mathcal{P} \mapsto \text{first\_entry}(f, \mathcal{P}_{\mathbf{B}}(\sigma), x)\,].$$

Theory: *places*

This function symbol is referred to as *raise* in running prose (or theorem names).

The next result explains how the copy-on-write strategy works. In a way, the result is stated backwards, because we regard the user-supplied copy function as being of the form $f^{[\sigma]}$ for some f. This function f must agree with the existing default function $\delta(\sigma)(x)$ where it counts, namely on the set $\mathcal{P}_{\mathbf{B}}(\sigma)$.

Theorem 3.8 *default-modification*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$ implication
- $\forall x : \mathcal{P}$ *s. t.* $x \in \mathcal{P}_{\mathbf{B}}(\sigma)$, $\delta(\sigma)(x) \simeq f(x)$
- conjunction
  - $\mathrm{abstr}(\mathrm{make\_istate}(\gamma(\sigma), f)) = \mathrm{copy}(\mathrm{abstr}(\sigma), f^{[\sigma]})$
  - $\mathrm{copy\_fn}(\mathrm{abstr}(\sigma), f^{[\sigma]})$.

Theory: *places*

The other crucial lemma justifying the copy-on-write strategy allows the default pointers to be redirected. The lemma states that we can alter the default $\delta(\sigma)$ to f, wherever f is defined, under a condition. The alteration will not change the abstract, user-visible state. We isolate the condition in the predicate c_redirect_fn. A redirection function f for a state σ has its range disjoint from its domain. Also, its range is disjoint from the immediate memory of γ. Finally, f does not affect $\delta(\sigma)$ in the sense that $\delta(\sigma)(x) = \delta(\sigma)(f(x))$, where f is defined at all.

Definition 3.9 *Let* σ : istate, f : place $\rightharpoonup$ place.

$\Longleftrightarrow$
- $\mathrm{c\_redirect\_fn}(f, \sigma)$
- conjunction
  - $ran\{f\} \cap dom\{\gamma(\sigma)\} = \emptyset$
  - $dom\{f\} \cap ran\{f\} = \emptyset$
  - $\forall x$: place *s. t.* $x \in dom\{f\}$, $\delta(\sigma)(x) = \delta(\sigma)(f(x))$.

Theory: *places*

Theorem 3.10 *default-multiplicity-reduction*

$\forall f$: place $\rightharpoonup$ place, σ : istate implication
- $\mathrm{c\_redirect\_fn}(f, \sigma)$
- $\mathrm{abstr}(\mathrm{make\_istate}(\gamma(\sigma), [\, x : \mathrm{place} \mapsto$
  conditionally, if $x \in dom\{f\}$
  - *then* $f(x)$
  - *else* $\delta(\sigma)(x)\,]))$

  $= \mathrm{abstr}(\sigma)$.

Theory: *places*

Definition 4.4 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

c_copy$(\sigma, f) =$ *conditionally, if* c_copy_fn(σ, f)
- *then* make_istate$(\gamma(\sigma), f)$
- *else* $\perp\Omega$.

Theory: *places*

Figure 4: The Concrete Copy Operation

4 Concrete Machine: The Operations and their Correctness

The definition of a primitive copy operation for the implementation state machine is given in Figure 4. The safety and liveness theorems justifying it are Theorems 4.5–4.6. Each of these theorems is a straightforward consequence of the Default Modification Theorem (3.8). The liveness theorem also requires the auxiliary notion *lower*, and a lemma relating it to the *raise* operation $f^{[\sigma]}$.

Definition 4.1 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

$\Longleftrightarrow$
- c_copy_fn(σ, f)
- $\forall p : \mathcal{P} \quad s.\ t. \quad p \in \mathcal{P}_{\mathbf{B}}(\sigma)$, $f(p) \simeq \delta(\sigma)(p)$.

Theory: *places*

Definition 4.2 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

lower$(\sigma, f) = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
- *then* $\delta(\sigma)(p)$
- *else* $f(p)\,]$.

Theory: *places*

Theorem 4.3 *raise-lower-composition*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega \quad s.\ t. \quad$ a_copy$(\text{abstr}(\sigma), f) \downarrow$, $f = (\text{lower}(\sigma, f))^{[\sigma]}$.

Theory: *places*

Theorem 4.5 *copy-safety*

Definition 4.8 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

c_redirect$(\sigma, f) =$ *conditionally, if* c_redirect_fn(f, σ)
- *then* make_istate$(\gamma(\sigma), [\, x : \mathcal{P} \mapsto$ *if* $x \in dom\{f\}$ *then* $f(x)$
  else $\delta(\sigma)(x)\,]$)
- *else* $\perp\Omega$.

Theory: *places*

Figure 5: The Redirect Operation

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega \quad s.\ t. \quad$ c_copy$(\sigma, f)\downarrow$,
abstr(c_copy(σ, f)) = a_copy(abstr(σ), $f^{[\sigma]}$).

Theory: *places*

Theorem 4.6 *copy-liveness*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega \quad s.\ t. \quad$ a_copy(abstr$(\sigma), f)\downarrow$,
abstr(c_copy$(\sigma,$ lower$(\sigma, f))) =$ a_copy(abstr$(\sigma), f)$.

Theory: *places*

Theorem 4.7 *copy-liveness-corollary*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega \quad s.\ t. \quad$ a_copy(abstr$(\sigma), f)\downarrow$,
c_copy$(\sigma,$ lower$(\sigma, f))\downarrow$.

Theory: *places*

We now introduce—in Figure 4—an operation *c_redirect* which simply redirects default pointers. Its correctness is an immediate consequence of the Default Multiplicity Reduction Theorem 3.10.

Theorem 4.9 *c%redirect-no-op*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega \quad s.\ t. \quad$ c_redirect$(\sigma, f)\downarrow$,
abstr(c_redirect(σ, f)) = abstr(σ).

Theory: *places*

Definition 4.10 *Let* $u :$ sets$[\mathcal{P}], g, f : \mathcal{P} \rightharpoonup \mathcal{W}$.

$\Longleftrightarrow$
- $f \cong_u g$
- $f\restriction u = g\restriction u$.

Definition 4.12 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

c_snapshot(σ, f) = *conditionally, if* a_copy(abstr$(\sigma), f) \downarrow$
- *then* c_copy$(\sigma,$ lower$(\sigma, f))$
- *else* $\perp\Omega$.

Theory: *places*

Figure 6: The Snapshot Operation

Theory: *places*

When a snapshot is added to the state, the snapshot set is the set of locations that will newly come into service. These are the places in the domain of f which are not already bound places. By the condition for *c_snapshot* to be defined, f must be the identity for bound places.

Definition 4.11 *Let* $f : \mathcal{P} \rightharpoonup \mathcal{P}, \sigma : \Omega$.

snapshot_set$(\sigma, f) = dom\{f\} \cap (\mathbf{C}(\mathcal{P}_{\mathbf{B}}(\sigma)))$.

Theory: *places*

The snapshot operation causes no change in the user-visible portion of the state, so long as the snapshot set is not regarded as user-visible. The symbol *ó* here should be read "is disjoint from."

Theorem 4.13 *c%snapshot-no-op*

$\forall f : \mathcal{P} \rightharpoonup \mathcal{P}, v :$ sets$[\mathcal{P}], \sigma : \Omega$ implication
- conjunction
  - v *ó* snapshot_set(σ, f)
  - c_snapshot$(\sigma, f) \downarrow$
- user_eq(abstr(σ), abstr(c_snapshot$(\sigma, f)), v)$.

Theory: *places*

Using the following definition, we can introduce (in Figure 4) the last interesting state machine operation, *c_fill*.

Definition 4.14 *Let* $a :$ sets[place], $\sigma :$ istate.

ifill(σ, a) = make_istate([$p :$ place $\mapsto$
conditionally, if $p \in a$
- *then* promote(σ, p)
- *else* $\perp$word], default$(\sigma))$.

Definition 4.16 *Let* $a : \text{sets}[\mathcal{P}], \sigma : \Omega$.

$\text{c\_fill}(\sigma, a) =$ *conditionally, if* $dom\{\gamma(\sigma)\} \subseteq a$
- *then* $\text{ifill}(\sigma, a)$
- *else* $\perp\Omega$.

Theory: *places*

Figure 7: The Fill Operation

Theory: *places*

There is a restriction on a in the theorem establishing that *ifill* is a no-op. The restriction has the effect of ensuring that the *ifill* does not throw away memory that already has immediate contents in σ.

Theorem 4.15 *ifill-abstraction*

$\forall \sigma : \text{istate}, a : \text{sets[place]} \quad s.\ t. \quad dom\{\gamma(\sigma)\} \subseteq a,$
$\text{abstr}(\text{ifill}(\sigma, a)) = \text{abstr}(\sigma).$

Theory: *places*

Theorem 4.17 *c%fill-no-op*

$\forall \sigma : \Omega, a : \text{sets}[\mathcal{P}] \quad s.\ t. \quad \text{c\_fill}(\sigma, a) \downarrow,$
$\text{abstr}(\sigma) = \text{abstr}(\text{c\_fill}(\sigma, a)).$

Theory: *places*

5 Mach: Three Cases for Copying

With these tools in hand, we can now turn to the details of Mach's treatment of the copy-on-write optimization. In order to avoid unnecessary snapshot objects and redundant copying, the actual Mach system distinguishes three cases. Suppose that in state σ a user-level copy is requested, and let the function h carry the places of the desired copy object C to the corresponding places of the original O. Let h^+ be

$$[\, p : \text{place} \mapsto \text{ if } p \in \text{bound\_place}(\sigma) \text{ then } p \text{ else } h(p) \,]$$

and let h' be $\text{lower}(\sigma, h)$. Observe that $h' = \text{lower}(\sigma, h^+)$.

Theorem 5.1 *lower-h%plus*

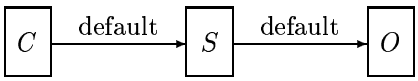

Figure 8: Relations Created by an Initial Copy Operation

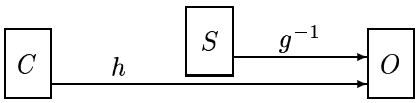

Figure 9: Relations before Redirection (Case 1)

$\forall \sigma : \Omega, h, h^+ : \mathcal{P} \rightharpoonup \mathcal{P}$ implication
- $h^+ = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
  - *then* p
  - *else* $h(p)\,]$
- $\text{lower}(\sigma, h^+) = \text{lower}(\sigma, h)$.

Theory: *places*

In the first case, the original has—as yet—no copies or snapshot objects. Formally, this is represented by the case where $\text{ran}(h)$ and $\text{ran}(\delta(\sigma))$ are disjoint. In this case, to produce the relations pictured in Figure 8, three steps will be required. First, the system can apply $c\_copy(\sigma, h')$ to simulate the intended effect of the *a_copy*, obtaining a new state σ_1. Second, a snapshot is selected. Formally, this consists of choosing a bijection on places g such that $\text{ran}(g) = \text{ran}(h)$ and $\text{dom}(g)$ is unused. The snapshot object is the set $\text{dom}(g)$. If there is no such g, then the system has run out of storage to use for its internal snapshot objects, and the operation must fail. Otherwise, let $\sigma_2 = c\_snapshot(\sigma, g)$. Let $\sigma_2 = c\_copy(\sigma, h)$. Finally, in the third step, we reduce the multiplicity (i.e., reduce the number of default pointers into $\text{ran}(h)$), by letting $\sigma_3 = c\_redirect(\sigma_2, (g^{-1} \circ h))$.

Definition 5.2 *Let* $\sigma : \Omega, h : \mathcal{P} \rightharpoonup \mathcal{P}$.

$\Longleftrightarrow$
- $\text{h\_ok}(h, \sigma)$
- conjunction
  - $ran\{h\} \subseteq \mathcal{P}_{\mathbf{B}}(\sigma)$
  - $dom\{h\}$ ó $\mathcal{P}_{\mathbf{B}}(\sigma)$.

Theory: *places*

Definition 5.3 *Let* $\sigma : \Omega, h, g : \mathcal{P} \rightharpoonup \mathcal{P}$.

$\Longleftrightarrow$

- g_h_ok(g, h, σ)
- conjunction
  - bijective_on$\{g, dom\{g\}, ran\{h\}\}$
  - $dom\{g\}$ *ó* $\mathcal{P}_{\mathbf{B}}(\sigma)$
  - $dom\{g\}$ *ó* $dom\{\delta(\sigma)\}$
  - $dom\{g\}$ *ó* $dom\{h\}$.

Theory: *places*

Definition 5.4 *Let* v : sets$[\mathcal{P}], g : \mathcal{P} \rightharpoonup \mathcal{P}$.

$\Longleftrightarrow$

- snapshot_not_visible(g, v)
- $dom\{g\}$ *ó* v.

Theory: *places*

Theorem 5.5 *case-1-mach-copy-thm*

$\forall \sigma : \Omega, g, h, f, h^+, h' : \mathcal{P} \rightharpoonup \mathcal{P}, v$: sets$[\mathcal{P}]$ implication

- conjunction
  - h_ok(h, σ)
  - $ran\{h\}$ *ó* $ran\{\delta(\sigma)\}$
  - g_h_ok(g, h, σ)
  - snapshot_not_visible(g, v)
  - $h^+ = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
    - *then* p
    - *else* $h(p)\,]$
  - $h' =$ lower(σ, h^+)
  - copy_fn(abstr$(\sigma), h^+)$
  - $f = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(c_copy(\sigma, h'))$
    - *then* p
    - *else* $g(p)\,]$
- a_copy(abstr$(\sigma), h^+)$
  $\cong_v$ abstr(c_redirect(c_snapshot(c_copy$(\sigma, h'), f), g^{-1} \circ h)$).

Theory: *places*

The second case arises when a copy has already been made of all or part of the original O, and moreover some data has been altered, causing data to be copied into the snapshot. Here a new snapshot object S_1 is created, and the old snapshot is redirected to it (Figure 10). The second case really subsumes

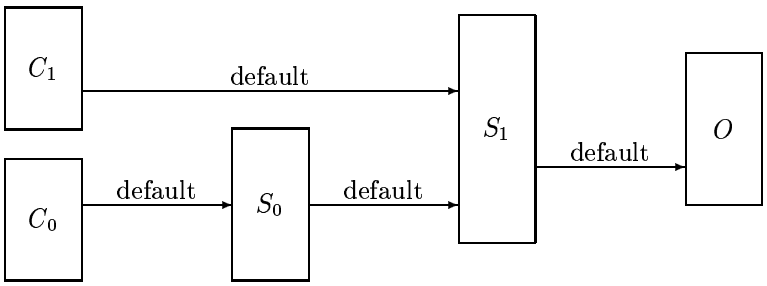

Figure 10: Relations Created by a Second Copy Operation (Original Altered)

the first, taking C_0 and S_0 to be empty, and we represent it, like the first case, as a composition of three state machine operations. Formally, this composite can be applied correctly even in case 3 below; it simply leads to a less efficient situation in which storage is wasted and longer chains of default pointers need to be traversed.

In this case, as in case 1, let g be a bijection on places such that $\mathrm{ran}(g) = \mathrm{ran}(h)$ and $\mathrm{dom}(g)$ consists of unused places. If no such g exists, we have run out of storage. Otherwise, let $sigma_1 = c\_copy(\sigma_1, h')$ to simulate the intended effect of the *a_copy*. Second, we add the new snapshot using $c\_snapshot(\sigma_1, f)$, where f is the identity on the bound places of σ_1 and agrees with g elsewhere. Let σ_2 be the result. Finally, in the third step, we reduce the multiplicity (i.e., reduce the number of default pointers into $\mathrm{ran}(h)$) by redirecting the defaults of the new copy object and the old snapshot object at once. The result is $\sigma_3 = c\_redirect(\sigma_2, (g^{-1} \circ h'))$.

Here, as in the other cases, it is quite easy to prove that the two states $\mathrm{a\_copy}(\mathrm{abstr}(\sigma), h^+)$ and

$$\mathrm{c\_redirect}(\mathrm{c\_snapshot}(\mathrm{c\_copy}(\sigma, h'), f), g^{-1} \circ h')$$

are indistinguishable to the user if the latter is well-defined. The work is showing that the conditions are met to execute the second and third steps. We show the theorem that establishes this.

Theorem 5.6 *c%redirect-c%snapshot-c%copy-definedness-case2*

$\forall \sigma : \Omega, g, h, f, h^+, h' : \mathcal{P} \rightharpoonup \mathcal{P}, v : \mathrm{sets}[\mathcal{P}]$ implication

- conjunction
  - $\mathrm{h\_ok}(h, \sigma)$
  - $\mathrm{g\_h\_ok}(g, h, \sigma)$
  - $\mathrm{snapshot\_not\_visible}(g, v)$
  - $h^+ = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
    - *then* p
    - *else* $h(p)\,]$
  - $h' = \mathrm{lower}(\sigma, h^+)$
  - $\mathrm{copy\_fn}(\mathrm{abstr}(\sigma), h^+)$
  - $f = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\mathrm{c\_copy}(\sigma, h'))$
    - *then* p
    - *else* $g(p)\,]$
- $\mathrm{c\_redirect}(\mathrm{c\_snapshot}(\mathrm{c\_copy}(\sigma, h'), f), g^{-1} \circ h') \downarrow$.

Theory: *places*

Theorem 5.7 *case-2-mach-copy-thm*

$\forall \sigma : \Omega, g, h, f, h^+, h' : \mathcal{P} \rightharpoonup \mathcal{P}, v : \mathrm{sets}[\mathcal{P}]$ implication

- conjunction
  - $\mathrm{h\_ok}(h, \sigma)$
  - $\mathrm{g\_h\_ok}(g, h, \sigma)$
  - $\mathrm{snapshot\_not\_visible}(g, v)$
  - $h^+ = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
    - *then* p
    - *else* $h(p)\,]$
  - $\mathrm{copy\_fn}(\mathrm{abstr}(\sigma), h^+)$
  - $h' = \mathrm{lower}(\sigma, h^+)$
  - $f = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\mathrm{c\_copy}(\sigma, h'))$
    - *then* p
    - *else* $g(p)\,]$
- $\mathrm{a\_copy}(\mathrm{abstr}(\sigma), h^+)$
  $\cong_v \mathrm{abstr}(\mathrm{c\_redirect}(\mathrm{c\_snapshot}(\mathrm{c\_copy}(\sigma, h'), f), g^{-1} \circ h'))$.

Theory: *places*

The third case arises when a copy object already exists, but no immediate data has been filled into the snapshot (Figure 11). Formally, this is the case where $\mathrm{ran}(h) \subset \mathrm{ran}(\delta(\sigma))$, but $\mathrm{dom}(\gamma(\sigma))$ is disjoint from the inverse image of $\mathrm{ran}(h)$ under $\delta(\sigma)$. We also require that the portion of $\delta(\sigma)$ yielding values in $\mathrm{ran}(h)$ should be a bijection. Let g be this subfunction of $\delta(\sigma)$, and let s be $\mathrm{dom}(g)$.

Definition 5.8 *Let* $\sigma : \Omega, s : \mathrm{sets}[\mathcal{P}], h, g : \mathcal{P} \rightharpoonup \mathcal{P}$.

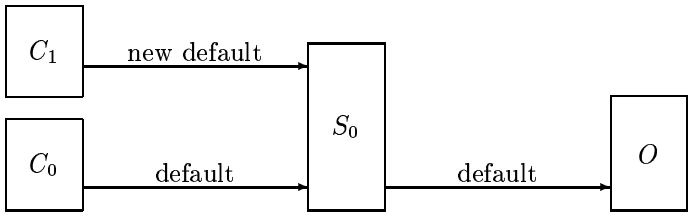

Figure 11: New Relation Created by a Second Copy Operation (Original Unaltered)

$\Longleftrightarrow$

- $\text{g\_h\_ok}_3(g, h, s, \sigma)$
- conjunction
  - $\text{bijective\_on}\{g, s, ran\{h\}\}$
  - $dom\{g\}$ *ó* $dom\{\gamma(\sigma)\}$
  - $dom\{g\}$ *ó* $dom\{h\}$
  - $\forall p : \mathcal{P} \quad s.\ t. \quad g(p)\downarrow$,

$g(p) = \delta(\sigma)(p)$.

Theory: *places*

In this case, we can achieve the desired result in two steps. First, we apply $c\_copy(\sigma, h')$ to simulate the intended effect of the *a_copy*. Next, we apply *c_redirect* to the resulting state, using $g^{-1} \circ h$ as the redirection function.

Theorem 5.9 *case-3-mach-copy-thm*

$\forall \sigma : \Omega, g, h, h^+, h' : \mathcal{P} \rightharpoonup \mathcal{P}, s, v : \text{sets}[\mathcal{P}]$ implication

- conjunction
  - $\text{h\_ok}(h, \sigma)$
  - $\text{g\_h\_ok}_3(g, h, s, \sigma)$
  - $\text{snapshot\_not\_visible}(g, v)$
  - $h^+ = [\, p : \mathcal{P} \mapsto$ *conditionally, if* $p \in \mathcal{P}_{\mathbf{B}}(\sigma)$
    - *then* p
    - *else* $h(p)\,]$
  - $h' = \text{lower}(\sigma, h^+)$
  - $\text{copy\_fn}(\text{abstr}(\sigma), h^+)$
- $\text{a\_copy}(\text{abstr}(\sigma), h^+)$
  $\cong_v \text{abstr}(\text{c\_redirect}(\text{c\_copy}(\sigma, h'), g^{-1} \circ h))$.

Theory: *places*

A Iteration of Functions in IMPS

A.1 Basic Properties of Iteration

In order to deal with the Copy-on-Write strategies, we need to define iteration and state a few general facts about it. This is done in the context of the IMPS theory generic-theory-1.

Definition (Recursive) A.1 *Let* $n : \mathbf{Z}, x : \mathbf{I}_1, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1$.

iterate$(f, x)(n) =$ *conditionally, if* $n = 0$
- *then* x
- *else* $f(\text{iterate}(f, x)(n-1))$.

Theory: *generic-theory-1*

Note that the expression iterate$(f, x)(n)$ is usually written $f^n(x)$.

Theorem A.2 *iterate-front-back-lemma*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, z : \mathbf{Z} \quad f(\text{iterate}(f, x)(z)) \simeq \text{iterate}(f, f(x))(z)$.

Theory: *generic-theory-1*

The next result says that once an iterate becomes undefined it remains undefined.

Theorem A.3 *iterate-definedness-refinement*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, z, j : \mathbf{Z}$ implication
- conjunction
  - $0 \leq j$
  - $j \leq z$
  - iterate$(f, x)(z)\downarrow$
- iterate$(f, x)(j)\downarrow$.

Theory: *generic-theory-1*

Theorem A.4 *iterate-additivity*

$\forall m, n : \mathbf{Z}, x : \mathbf{I}_1, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1 \quad s.\ t. \quad 0 \leq n \wedge 0 \leq m,$
$\text{iterate}(f, \text{iterate}(f, x)(n))(m) \simeq \text{iterate}(f, x)(n+m)$.

Theory: *generic-theory-1*

Quasi-constructor A.5 *Let* $f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \text{sets}[\mathbf{I}_1]$.

invariant$\{s, f\}$ *expands to* $\forall m : \mathbf{I}_1 \quad s.\ t. \quad m \in s \wedge f(m)\downarrow,$
$f(m) \in s$.

Language: *pure-generic-theory-1*

Theorem A.6 *iterate-invariance*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, z : \mathbf{Z}, a : \mathrm{sets}[\mathbf{I}_1]$ implication
- conjunction
  - invariant$\{a, f\}$
  - $x \in a$
  - $0 \leq z$
  - iterate$(f, x)(z)\downarrow$
- iterate$(f, x)(z) \in a$.

Theory: *generic-theory-1*

Theorem A.7 *iterate-locality*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, z : \mathbf{Z}, a : \mathrm{sets}[\mathbf{I}_1]$ *s. t.* invariant$\{a, f\} \wedge x \in a$,
iterate$(f, x)(z) \simeq$ iterate$(f\restriction a, x)(z)$.

Theory: *generic-theory-1*

The entry_index of f relative to a set s is the first index n for which the iterate $f^n(x)$ lands in s. Later we define first_entry which is the value $f^n(x)$.

Definition A.8 *Let* $x : \mathbf{I}_1, s : \mathrm{sets}[\mathbf{I}_1], f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1$.

entry_index$(f, s, x) =$ set_min$(\{m : \mathbf{Z} \mid \mathrm{iterate}(f, x)(m) \in s\})$.

Theory: *generic-theory-1*

Theorem A.9 *entry%index-characterization*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \mathrm{sets}[\mathbf{I}_1], x : \mathbf{I}_1, n : \mathbf{Z} \quad \Longleftrightarrow$
- entry_index$(f, s, x) = n$
- conjunction
  - iterate$(f, x)(n) \in s$
  - $\forall m : \mathbf{Z} \quad m < n \supset \neg(\mathrm{iterate}(f, x)(m) \in s)$.

Theory: *generic-theory-1*

Theorem A.10 *entry%index-definedness*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \mathrm{sets}[\mathbf{I}_1], x : \mathbf{I}_1 \quad \Longleftrightarrow$
- entry_index$(f, s, x)\downarrow$
- $\exists n : \mathbf{Z} \quad \mathrm{iterate}(f, x)(n) \in s$.

Theory: *generic-theory-1*

Definition A.11 *Let* $x : \mathbf{I}_1, s : \mathrm{sets}[\mathbf{I}_1], f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1$.

first_entry$(f, s, x) =$ iterate$(f, x)($entry_index$(f, s, x))$.

Theory: *generic-theory-1*

Theorem A.12 *first%entry-locality*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, a, s : \mathrm{sets}[\mathbf{I}_1]$ *s. t.* invariant$\{a, f\} \wedge x \in a$,
first_entry$(f, s, x) \simeq$ first_entry$(f\restriction a, s, x)$.

Theory: *generic-theory-1*

Theorem A.13 *first%entry-restriction-lemma*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, a, s : \text{sets}[\mathbf{I}_1] \quad s.\ t. \quad \text{invariant}\{a, f\} \wedge x \in a,$
$\text{first\_entry}(f, s, x) \simeq \text{first\_entry}(f, s \cap a, x).$

Theory: *generic-theory-1*

Theorem A.14 *first%entry-characterization*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \text{sets}[\mathbf{I}_1], x, y : \mathbf{I}_1 \quad \Longleftrightarrow$
- $\text{first\_entry}(f, s, x) = y$
- conjunction
  - $y \in s$
  - $\exists n : \mathbf{Z} \quad \text{iterate}(f, x)(n) = y \wedge (\forall m : \mathbf{Z} \quad m < n \supset \neg(\text{iterate}(f, x)(m) \in s))$.

Theory: *generic-theory-1*

Theorem A.15 *first%entry-minimality*

$\forall g, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \text{sets}[\mathbf{I}_1] \quad \text{implication}$
- $\forall x : \mathbf{I}_1 \quad g(x) \simeq$ *conditionally, if* $x \in s$
  - *then* x
  - *else* $g(f(x))$
- $\forall x : \mathbf{I}_1 \quad s.\ t. \quad \text{first\_entry}(f, s, x)\downarrow,$

$\text{first\_entry}(f, s, x) = g(x).$

Theory: *generic-theory-1*

Theorem A.16 *first%entry-definedness*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, s : \text{sets}[\mathbf{I}_1], x, y : \mathbf{I}_1 \quad \Longleftrightarrow$
- $\text{first\_entry}(f, s, x)\downarrow$
- $\exists n : \mathbf{Z} \quad \text{iterate}(f, x)(n) \in s.$

Theory: *generic-theory-1*

Theorem A.17 *first%entry-equation*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, s : \text{sets}[\mathbf{I}_1] \quad \text{first\_entry}(f, s, x) \simeq$ *conditionally, if* $x \in s$
- *then* x
- *else* $\text{first\_entry}(f, s, f(x))$.

Theory: *generic-theory-1*

A.2 Flow Extension and its Properties

In the following definition, we view f as generating a discrete flow. The function g is extended along the flow as follows:

Definition A.18 *Let* $x : \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1$.

$\text{flow\_ext}(f, g, x) = g(\text{first\_entry}(f, \mathit{dom}\{g\}, x))$.

Theory: *generic-theory-2*

Theorem A.19 *flow%ext-definedness*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1 \quad \Longleftrightarrow$
- $\text{flow\_ext}(f, g, x) \downarrow$
- $\exists n : \mathbf{Z} \quad g(\text{iterate}(f, x)(n)) \downarrow$.

Theory: *generic-theory-2*

Theorem A.20 *flow%ext-minimality-lemma*

$\forall g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, h : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2$ implication
- $\forall x : \mathbf{I}_1 \quad g(x) \simeq$ *conditionally, if* $h(x) \downarrow$
  - *then* $h(x)$
  - *else* $g(f(x))$
- $\forall x : \mathbf{I}_1 \quad s.\ t. \quad \text{flow\_ext}(f, h, x) \downarrow$,

$\text{flow\_ext}(f, h, x) = g(x)$.

Theory: *generic-theory-2*

Theorem A.21 *flow%ext-minimality*

$\forall g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, h : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2$ implication
- $\forall x : \mathbf{I}_1 \quad g(x) \simeq$ *conditionally, if* $h(x) \downarrow$
  - *then* $h(x)$
  - *else* $g(f(x))$
- $\forall x : \mathbf{I}_1 \quad s.\ t. \quad \text{flow\_ext}(f, h, x) \downarrow$,

$\text{flow\_ext}(f, h, x) \simeq g(x)$.

Theory: *generic-theory-2*

Theorem A.22 *first%entry-iteration*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, x : \mathbf{I}_1, a, s : \text{sets}[\mathbf{I}_1] \quad \text{first\_entry}(f, \{x : \mathbf{I}_1 \mid \text{first\_entry}(f, s, x) \downarrow\}, x) \simeq$
conditionally, if $\text{first\_entry}(f, s, x) \downarrow$
- *then* x
- *else* $\perp \mathbf{I}_1$.

Theory: *generic-theory-1*

Theorem A.23 *domain-of-flow%ext-lemma*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1 \quad dom\{[\, x : \mathbf{I}_1 \mapsto \text{flow\_ext}(f, g, x)\,]\} = \{x : \mathbf{I}_1 \mid \text{first\_entry}(f, dom\{g\}, x) \downarrow\}$.

Theory: *generic-theory-2*

Theorem A.24 *flow%ext-idempotency*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1 \quad \text{flow\_ext}(f, [\, x : \mathbf{I}_1 \mapsto \text{flow\_ext}(f, g, x)\,], x) \simeq \text{flow\_ext}(f, g, x)$.

Theory: *generic-theory-2*

Theorem A.25 *locality-of-flow%ext*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1, a, s : \text{sets}[\mathbf{I}_1] \quad s.\ t. \quad \text{invariant}\{a, f\} \wedge x \in a,$
$\text{flow\_ext}(f, g, x) \simeq \text{flow\_ext}(f \restriction a, g \restriction a, x)$.

Theory: *generic-theory-2*

Theorem A.26 *locality-of-flow%ext-corollary*

$\forall f, f_1 : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g, g_1 : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1, a, s : \text{sets}[\mathbf{I}_1]$ implication

- conjunction
  - invariant$\{a, f\}$
  - $\forall m : \mathbf{I}_1 \quad m \in a \supset (f(m) \simeq f_1(m) \wedge g(m) \simeq g_1(m))$
  - $x \in a$
- flow_ext$(f, g, x) \simeq$ flow_ext(f_1, g_1, x).

Theory: *generic-theory-2*

Theorem A.27 *flow%ext-recursive-equation*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1$ flow_ext$(f, g, x) \simeq$ *conditionally, if* $g(x) \downarrow$

- *then* $g(x)$
- *else* flow_ext$(f, g, f(x))$.

Theory: *generic-theory-2*

Theorem A.28 *flow%ext-restriction-lemma*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1$ flow_ext$(f, g, x) \simeq$ flow_ext$(f \restriction \mathbf{C}dom\{g\}, g, x)$.

Theory: *generic-theory-2*

Theorem A.29 *flow%ext-restricted-invariance*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2$ invariant$\{dom\{[\, x : \mathbf{I}_1 \mapsto$ flow_ext$(f, g, x)\,]\}, f \restriction \mathbf{C}dom\{g\}\}$.

Theory: *generic-theory-2*

Theorem A.30 *flow%ext-complement-invariance*

$\forall f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2$ invariant$\{\mathbf{C}dom\{[\, x : \mathbf{I}_1 \mapsto$ flow_ext$(f, g, x)\,]\}, f\}$.

Theory: *generic-theory-2*

Theorem A.31 *flow%ext-domain-monotonicity*

$\forall g : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, f : \mathbf{I}_1 \rightharpoonup \mathbf{I}_1, h : \mathbf{I}_1 \rightharpoonup \mathbf{I}_2, x : \mathbf{I}_1$ implication

- conjunction
  - $dom\{g\} \subseteq dom\{h\}$
  - flow_ext$(f, g, x) \downarrow$
- flow_ext$(f, h, x) \downarrow$.

Theory: *generic-theory-2*

References

[1] CMU Mach Project. Mach 3.0 source code. Available at `<URL:http://www.cs.cmu.edu/afs/cs/project/mach/public/www/sources/sour%ces_top.html>`, 1993.

[2] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M93B-138, The MITRE Corporation, Bedford, MA, November 1993.

[3] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[4] Joshua D. Guttman and Dale M. Johnson. Three applications of Formal Methods at MITRE. In M. Naftalin, T. Denvir, and Miguel Bertran, editors, *FME '94: Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 55–65. Springer Verlag, 1994.

[5] Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, 1995.

[6] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[7] Keith Loepere. Mach 3 kernel interfaces. Technical report, Open Software Foundation, Cambridge, MA, July 1992. Jointly copyright by Open Software Foundation and Carnegie-Mellon University.

[8] Keith Loepere. OSF Mach final draft kernel principles. Technical report, Open Software Foundation, Cambridge, MA, May 1993. Jointly copyright by Open Software Foundation and Carnegie-Mellon University.

Contents