DYNAMIC SOFTWARE UPDATING

Michael Hicks

A DISSERTATION
in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2001

Scott M. Nettles
Supervisor of Dissertation

Val Tannen
Graduate Group Chair

Acknowledgements

This dissertation, while an achievement that bears my name, would not have been possible without the help of others, who I would now like to thank. First and foremost, I thank God. He has been a hidden, guiding force throughout my life, and has been a source of strength and consolation during the writing of this document, and indeed throughout my days as a graduate student. As a close second, I thank my family, especially my wife. She has been especially patient and understanding during what was anticipated to be a four year process, though has now ended at six years. My son Timothy, born during this research, became my close companion during the writing of the dissertation. Some days he would be asleep in his stroller while I typed away on a park bench. Many times, he protested as I divided my time between him and my other "baby." In the end, he provided a great joy that made it possible, though challenging, to complete this work. My parents deserve credit, among so many other things, for teaching me that good things come with hard work and perseverance. My brother has served as a constant source of inspiration, showing me that short term sacrifice can lead to everlasting reward, and that with love anything is possible. Without the support of all these people, it would have been difficult indeed for me to complete a Ph.D.

I know how to do research thanks to my advisor. Scott has been a patient and encouraging guide, showing me the way but never demanding I move any faster or slower than I was ready to. He provided structure yet allowed great independence, encouraging exploration and self-improvement. Thanks to him, I have done research in many areas, creating a foundation for exciting things to come. I must also thank my other "advisor," Jonathan Smith. Jonathan was generous enough to support me after Scott left Penn, nearly three years ago. Without him, I would not be getting this degree. Jonathan has been a wonderful friend and support, always looking out for my best interests, despite inconvenience to himself.

I must also thank my committee for their input and feedback. Any competence I have in programming language theory I owe to Benjamin Pierce. His wonderful course on type systems provided insight as well as information, and greatly broadened my understanding in the area. Greg Morrisett has also been a great source of insight, affirming that my intuitions as a "systems" person are completely relevant to the way programming languages should be designed and implemented. Two of Greg's students, Karl Crary and Stephanie Weirich, contributed greatly to the theoretical aspects of this work (specifically Chapters 5 and 6) and in the process taught me a lot about the area. Insup Lee and Mark Segal provided excellent suggestions drawing from their own research experience in dynamic updating.

Working in the Distributed Systems Lab has been a wonderful experience, both personally and professionally. I have made so many friends, each of whom contributed in varying levels to this work and the work that preceded it. Jonathan Moore, a member of the fabled “Mike and Jon show,” has been my research companion on so many projects; I owe much of my success to him. More importantly, Jon, and his wife Jessica, have been two of my best friends. I should also thank Scott Alexander who, as the result of an unsubstantiated claim in his own thesis, inadvertently provided me with my own thesis topic! I also received excellent feedback and encouragement from many others, and did some great work with them, including Luke Hornof, Steve Muir, Jessica Kornblum, Sotiris Ioannidis, and Angelos Keromytis. I will never forget our monumental games of WarCraft II and StarCraft; some day I will find a way to have these experiences add value to my research (it is possible; see Baughman and Levine [BL01]). Thanks so much to everyone; it has been a wonderful six years!

Abstract

DYNAMIC SOFTWARE UPDATING
Michael Hicks
Supervisor: Scott M. Nettles

Many important applications must run continuously and without interruption,yet must be changed to fix bugs or upgrade functionality. To date, no existing, general-purpose methodology for dynamic updating achieves a practical balance between flexibility, robustness, low overhead, and ease of use.

We present a new approach for imperative languages that provides type-safe dynamic updating of native code in an extremely flexible manner (code, data, and types may be updated, at programmer-determined times) and permits the use of automated tools to aid the programmer in the updating process. Our system is based around *dynamic patches*, which contain both the updated code and the code needed to transition from the old version to the new. A novel aspect of our patches is that they consist of *verifiable native code* (or VNC, see [Nec97, MWCG99]), which is native code accompanied by annotations that allow on-line verification of the code's safety. We discuss how patches are generated mostly automatically, how they are applied using our own novel dynamic-linking technology for VNC systems, and how code is compiled to make it updateable.

To concretely illustrate and validate our system, we have implemented a sizeable application: a dynamically updateable web server, called FlashEd. We discuss our experience building and maintaining FlashEd. Performance experiments show that updateable FlashEd runs roughly 2% slower than a static one under various workloads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many computer programs must be "non-stop," that is, provide continuous and uninterrupted service. This is especially true of mission critical applications, such as telephone switches, financial transaction processors, airline reservations and air traffic control systems. In addition, the importance of the Internet and its link with the global economy has broadened needs, making non-stop service important to less sophisticated users participating in e-commerce.

Non-stop systems are not immune to the need for upgrades and bug fixes. In the simplest case, software changes require the system to be shut down, updated, and then brought back on-line. This approach has two consequences. First, any *state* accumulated by the application will be lost when the old application is shut down. Second, any processing in progress at the time of shutdown will be canceled. In some situations, these consequences are acceptable. For example, if the only program state is a cache of some kind, then losing it will only affect performance, not correctness. Similarly, a properly architected transaction system will prevent canceled processing from adversely affecting stable state. However, in the worst case, lost state and canceled processing may translate to lost revenue, compromised safety, and incorrect execution.

Thus, in general, non-stop systems require the ability to update software without service interruption. Our goal is to show that *dynamic software updating* can be achieved in a *practical, general-purpose* manner that is *flexible*, *efficient*, *robust* and is *easy to use.* To demonstrate this thesis, we have built a dynamic updating system which we show has all of the desired characteristics. Furthermore, using this system, we built a non-trivial, dynamically updateable application, the FlashEd webserver. We explain how FlashEd has been updated over time, and in so doing, show that the system is flexible and easy to use. We argue that FlashEd is robust both due to measures that promote program and patch correctness, including automated safety checking and mostly automatic patch generation, and because the compilation and library support for updateability is simple to implement and has only a small trusted component. Finally, we show that the updating system imposes only a negligible overhead on FlashEd's performance.

1.1 Motivation

Enabling dynamic software updating is not a new problem; many solutions exist and are widely deployed. Past approaches have been both general-purpose and application-specific, and generally employ one of two mechanisms to realize dynamic updating: *state transfer* or *dynamic linking*. Many past systems that employ state transfer also use *redundant hardware*. We highlight the elements of these approaches below and point out their weaknesses, motivating our work.

1.1.1 Redundant Hardware

Because systems that require dynamic code updates are non-stop, they frequently employ redundant hardware to support fault tolerance. As a result, many updating approaches assume redundant hardware is present. Generally speaking, to perform an on-line upgrade with these approaches, a secondary machine is loaded with new code, passed the necessary state from the primary, and 'switched over' to become the primary system.

A good example of this approach is employed by ACARS, the digital messaging system used by the airlines, developed and maintained by ARINC, Inc. [ARI]. This system uses a centralized, special-purpose router for relaying messages to and from aircraft. To enable dynamic updates and improve robustness, this router consists of two machines, one a *primary* and the other a *hot standby*. The primary machine software, in addition to performing all message processing, is engineered to communicate its state to the standby machine, either periodically or on-demand. When the system needs to be updated, the new software is loaded on the standby machine, which immediately requests a state transfer from the primary to initialize itself. The two machines then switch roles, so that the updated machine becomes the primary and starts processing messages. The now-standby machine is then loaded with the new software, completing the update. This architecture is similar to the one employed by Lucent for its 5ESS^{TM} circuit switches [5ES]. Some systems avoid transferring state directly between machines by keeping it in stable storage, say by using a database. When a new version of the program starts up, it reads in its startup state from the database.

The primary/secondary architecture is frequently generalized to arbitrarily many machines to increase availability. Incoming transactions are routed to available servers, and servers can be brought up or down as needs demand. For example, Visa makes use of 21 mainframe computers to run its 50 million line transaction processing system. This system is updated as many as 20,000 times per year, but tolerates less than 0.5% downtime [Pes00].[1] Parts of the system are brought down, updated with new software, and then brought back on-line, while the operating mainframes continue to process transactions.

Given its popularity, the technique of combining redundant hardware with software-enabled state transfer is obviously effective. It is especially appealing in situations in which redundant hardware is required anyway to support fault tolerance. However, we would prefer not to formulate an approach to dynamic updating that *requires* redundant hardware, for two reasons:

1. Redundant hardware adds cost and complexity. If a system does not require redundant hardware for other reasons, we would prefer not to require it for software updating. In fact, many systems that require upgradeable, non-stop service do not employ fully-redundant hardware. Examples include communications components (*e.g.*

[1]Twenty-thousand updates per year seems quite high, but I was unable to get VISA to better explain the number. Regardless, we can believe that updates occur reasonably frequently.

routers, firewalls, NAT translators, *etc.*), less sophisticated Internet servers, medical monitoring systems, and others. Furthermore, there are many non-redundant systems that do not necessarily *require* non-stop service but would certainly benefit from it. For example, rather than having to reboot a PC each time its OS is upgraded, we would prefer to realize the updates dynamically.

2. An updating system not requiring redundant hardware will be relevant on systems that happen to use it. On the one hand, divorcing the concerns of fault tolerance (say) from updating may prevent the sharing of implementation mechanisms like state transfer, but on the other hand the resulting design is more general and more modular.

1.1.2 State Transfer

In addition to being used with redundant hardware, state transfer can be performed between processes running on the same machine. In general, making an application updateable via state transfer (whether or not it uses redundant hardware) requires the programmer or system to do three things:

1. Identify the state that will need to be transferred to the new version. We call this the *persistent* state, since it must persist between application versions. All other state is referred to as *ephemeral*.

2. Develop a means to encode and decode the persistent state, and a means to transfer it from the old to the new version.

3. Build the new version to be able to start with the old version's (decoded) state, potentially transforming that state to be usable by the new code.

Application-specific approaches that employ state transfer, like the ones we described in the previous subsection, typically require the programmer to perform all three of these tasks, which can be tedious, and potentially quite difficult or even impossible. For example, relating to point 1, state that is stored in the operating system, like a process's file descriptor associations, cannot readily be transferred between machines or processes, and therefore cannot be made persistent. Similarly, state stored in application libraries may not be available, since it is hidden behind the library interface. An example is the current value of the seed in a random number generator. Regarding point 2, the more complicated the persistent state, the more tedious it is to encode and decode. For example, pointers need to be made platform-independent, so they cannot be captured by simply recording the address in the running process. In addition, depending on how the state is encoded and transferred, restarting the program with that state can be overly time-consuming; for example, when state information is stored incrementally in a log format, the entire log must be replayed to regenerate the state [Seg].

While application-specific state encoding and decoding to transfer state can 'get the job done,' we would prefer to use a more general-purpose approach, ignoring points 1 and 2, focusing on the truly application-specific task of state transformation. That is, we prefer to assume that *all* state in the program is potentially persistent, removing the guesswork and possible error by the programmer. This also frees us from using a database or other special-purpose mechanism, unless other requirements demand it, simplifying program construction. Furthermore, we would rather that the system perform the encoding and decoding of the state automatically, as opposed to requiring the programmer to do so.

A number of general purpose approaches to enable state transfer have been developed. For example, checkpointing [Pla97] and general-purpose persistence [PJW96] are means to generally and automatically capture a program's state for later restart, *e.g.* to support process migration [Smi88]. However, these approaches have a number of problems:

1. Like application-specific state transfer, OS-level datastructures cannot, in general, be captured. This includes file descriptors for open socket connections.

2. Most approaches are process image-dependent, which simplifies the process of capturing and restoring state but makes that state all but unusable to different process images. Since we are interesting in updating a process with new code, its image would obviously change between the capture and restore of the state.

3. Some approaches provide *portable* state capture across different architectures [Hof93, RS97], such that the state is stored in a more abstract form; in principle this should allow it to be used by different process images. However, in practice this is not the case. Though abstract, the captured state still matches the structure of the capturing program (most importantly its stack), making even simple program restructurings problematic. Furthermore, in general, pointers cannot be distinguished from integers without user-assistance; this is the same problem that occurs in conservative garbage collection [BW88]. Finally, using a source-to-source translation to enable portable state capture introduces a potentially significant overhead on running code due to code insertions required to unwind the stack.

Despite these limitations, automatic state transfer has been successfully employed in a number of systems and scenarios to perform or support dynamic updating [GJ93, Hof93, TTA$^+$99, GBHC00]. In particular, when OS-resident data structures need not be captured,[2] or when global state is simple and program structure is not appreciably changed, automatic state transfer can serve as an elegant means to upgrade without halting service.

However, the limitations of state transfer preclude its application to larger, network-oriented systems, like e-commerce servers, since connection data (*i.e.* the socket file descriptor table) is stored in the OS. This rules out a sizeable class of applications if losing connections at update-time is unacceptable. Some problems can be solved using application-specific approaches, but with more effort. In short, to use state transfer as the core dynamic updating technology requires either sacrificing some flexibility when using a general-purpose tool (and potentially performance as well), or adding complexity and cost when using an application-specific approach. Either way, the programmer bears a greater burden to make software updateable.

[2]Some specialty operating systems perform state capture, *e.g.* in EROS [SSF99].

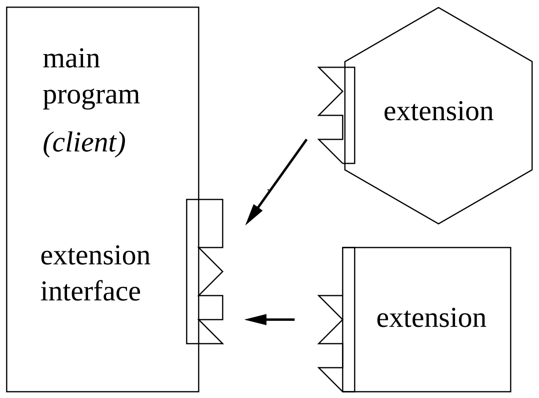


Figure 1.1: Plug-in extensibility: extensions are "plugged-in" to an extension interface in the running program.

1.1.3 Dynamic Linking

Many systems employ *dynamic linking* to realize software adaptability. In some sense, dynamic linking is the converse of state transfer; rather than trying to move a program's state to a different program, we instead move the program to the state. As a result, there is no need for redundant hardware, or even redundant processes, and the programmer no longer needs to identify persistent state or develop a means to capture it and restore it. Instead, the programmer need only transform the state as necessary to work with the new code (a requirement of state transfer as well). In addition, dynamic linking is popular and quite simple to implement, decreasing the system's overall complexity, and we can often verify that loaded code is *safe* (*c.f.* Java [jav96]); together, simplicity and safety improve the system's robustness.

However, dynamic linking has flexibility problems of its own. In its most common use, dynamic linking implements *plug-in extensibility*, an approach in which loaded code is constrained to match a pre-defined *signature* expected by its clients; correctly formed loaded code is called a *plug-in*. Plug-ins are used in many systems, including so-called extensible operating systems (*e.g.* SPIN [BSP$^+$95] and Exokernel [EKO95]), commodity operating systems (*e.g.* Linux), adaptable distributed systems (*e.g.* Cactus [Cac] and [Ens]), web browsers, and others. Plug-in extensibility is insufficient for dynamic updating simply because only parts of the system (the plug-ins) are allowed to change.

Plug-in Extensibility

Essentially, plug-in extensibility is a technique that *abstracts* the shape of loadable code. Loaded code is accessed by the running program, the *client*, through an extension interface. Extensions, while internally consisting of arbitrary functionality, may only be accessed by the client through the extension interface, which does not change with time. This idea is illustrated abstractly in Figure 1.1.

To use plug-in extensibility, the programmer must do two things:

1. Identify those elements in his program that should be subject to change. These will be the plug-ins for the program.

2. Create a common, abstract interface for those components to be used by the rest of the program.

Plug-ins are used in many systems. For example, in the Linux kernel, plug-ins are used, among other things, to implement socket handlers for various protocols. Each handler has an abstract interface for use by the networking code consisting of the socket interface functions. That is, each handler will implement an `open` function, a `connect` function, a `close` function, *etc.* When a user attempts to open a socket of a particular type, say IPX, the handler for that socket type is loaded (if not already present) and its `open` function is invoked. Future uses of that socket will use the IPX handler.

Active, or programmable, network implementations frequently employ plug-in extensibility (*e.g.* [YdS96, HMA$^+$99, WGT98, MBC$^+$99] and others), having the goal of evolving network service on demand. As an example, consider the PLANet [HMA$^+$99] active internetwork. PLANet is based on a two-level architecture that provides lightweight, but limited, programmability in the packets of the network, and more general-purpose extensibility in the routers. Packet headers are replaced by programs written in a special-purpose language PLAN [HKM$^+$98], resulting in much greater flexibility than traditional headers. When packets arrive at a node to be evaluated, their PLAN programs may call node resident *service routines*, which form the second level of the architecture. The service routine space is extensible, allowing new service routines to be installed or removed without stopping the execution of the system. This is implemented by dynamically linking code that implements the new service and registering it in a symbol table used by the PLAN interpreter.

PLANet services are plug-ins. Every time that a PLAN program invokes a service, that service's name is looked up in the symbol table, and the corresponding service routine is returned as a function pointer. Since the lookup code does not know anything about particular services, the type of the function returned must match the extension interface, just as the socket handler used by Linux must match the socket interface. In this case, all services take as arguments a variable-length list of PLAN values and a PLAN packet and return a PLAN value.

Plug-ins are convenient because they abstract the kinds of changes that may be made in the future, and thus give the current code an interface to deal with those changes. In the Linux case, the socket code does not care *what* code it is calling, only that it will perform the proper *kind* of function (like setting up a socket object and returning it). Similarly with PLAN services, the caller (the PLAN interpreter) only cares that the service function performs some action with the given arguments and returns a PLAN value.

While plug-ins can be used to simply and easily implement bounded changes in a program, they cannot easily support *arbitrary* changes dynamically. This is because potentially many parts of the system are *not* plug-ins, and therefore they cannot be changed. For example, while we can add new service routines to PLANet, thereby upgrading the service API for PLAN programs, we cannot alter PLANet's more low-level components, such as the PLAN interpreter itself, the way in which PLAN programs are encoded on the wire, the way packets are queued, *etc.* The code that implements these features is not a plug-in, and therefore not subject to change.

One might argue that a system could be constructed such that *all* (or as many as possible) of its components are plug-ins. As is argued more extensively in [HN00], the task of converting a program by hand so that all of its components are plug-ins is non-trivial. Furthermore, the converted program is much harder to read since it now contains sizeable amounts of 'scaffolding' code to enable various types of plug-in components. Instead, we would prefer that this conversion be automatic, allowing the programmer to write code in the traditional manner. In essence, this is the approach that we, and others, have taken to realizing dynamic software updating.

1.2 Thesis

As we have discussed so far, existing technologies for dynamic software updating have important limitations. Employing redundant hardware adds to system cost and complexity when not otherwise needed, and the enabling technologies of state transfer and dynamic linking are not as flexible as we might like. Many current solutions employ application-specific techniques, which places an extra burden on the programmer, and obviously the formulated approaches are less general.

Many past researchers have recognized these problems, and have formulated general-purpose approaches to dynamic updating that achieve a reasonable level of success. While we defer a more detailed discussion of the strengths and weaknesses of these systems until the next chapter, we can summarize by saying that no prior system emphasizes the system's *practicality* as much as we would like. As a result, we are motivated to explore the space of possibilities for general-purpose, dynamic software updating systems, with an eye toward building a system that is practical.

What makes a practical updating system? A practical system must be flexible; the more limitations it has, the more likely that a change cannot be reflected dynamically, without stopping service. In addition, a practical system must be robust; the more potential there is for error, the more chance that the system will crash or behave incorrectly. Efficiency is also of primary importance because so many non-stop systems must be high-performance. Finally, a system that is hard to use is generally not used at all; therefore, means to ease the burden of crafting dynamic updates are critical to a system's success. Existing general-purpose systems lack in one or more of these areas of flexibility, robustness, efficiency, or ease of use.

In this work, we aim to improve the state of the art in dynamic software updating, drawing on the successes of past efforts, while overcoming many of their limitations. In short, we aim to show that

> *Dynamic software updating*, meaning the arbitrary modification of a program as it runs, can be achieved in a *practical* manner that is *flexible*, *robust*, *efficient*, and is *easy to use*.

To prove this assertion, we have explored the space defined by these criteria, carefully analyzing mechanisms used in prior work, and experimenting with new mechanisms. As a result of this research and analysis, we have built a system that arguably meets our requirements of flexibility, robustness, efficiency, and ease of use. To test our system's practicality, we built a non-trivial application—a webserver—and used our system to update it over time. The experience was eminently useful, not only because we could demonstrate the updating system's practicality, but because the process of updating the webserver shed light on areas we could (and did) improve in the updating system.

We begin in Chapter 2 by more carefully defining the evaluation criteria: flexibility, robustness, efficiency, and ease of use. We then evaluate specific past work in general-purpose dynamic updating in light of these criteria, identifying techniques that are successful, but pointing out areas that need more work. We outline our approach In Chapter 3 and argue that it satisfies the evaluation criteria. The remainder of the dissertation describes our approach in detail.

Chapter 2

Goals

We believe that a practical system for dynamic software updating should satisfy four criteria: flexibility, robustness, efficiency, and ease of use. In this chapter, we define these criteria in detail for our context and argue that they sufficiently measure the practicality of an updating system. We then evaluate past approaches, highlighting what previous systems do and do not do well, setting the stage to present our approach in Chapter 3 and throughout the remainder of the dissertation.

2.1 Criteria of Evaluation

Let us examine the evaluation criteria more closely:

- **Flexibility** We must judge how effectively a system supports dynamic updates. The more flexible a system is, the more likely that we will be able to reflect a needed change at some point in the future. Ideally, a general-purpose updating system should be flexible enough that *any* part of a running program can be updated in any way without requiring downtime. More specifically, after *arbitrarily* altering the source files of a program in creating its next version, we should be able to reflect these changes dynamically, in the running program. The more that an updating system strays from this ideal, the more likely that it will be impossible to reflect a future change dynamically.

  However, the programmer should retain the ability to determine *when*, during program execution, an update is applied. In other words, restrictions on the timing of the update, say to avoid race conditions while manipulating existing state, should be imposed by the programmer, not the system. Again, the more that the system imposes timing restrictions on an update, the more likely it will be that an update cannot be properly applied. The system may provide means to aid the programmer properly time the update based on programmer-provided constraints. For example, it could delay applying an update until certain functions or modules are inactive.

- **Robustness** Many applications that are candidates for dynamic updating are *mission critical*: they must continuously provide correct service. Making a program updateable should not compromise this requirement. In particular, the greater the chance that the system could crash, lose data, perform incorrect operations, or otherwise fail due to an update, the greater the risk to the application that uses it.

While proving that updates are *correct* is undecidable (*cf.* [Gup94]), we can improve the *robustness* of both dynamic updates and the mechanisms for realizing them in a number of tractable ways that, while not providing a full guarantee of correctness, reduce the possibility of error. We have identified five important robustness properties that updating systems should seek to achieve.

Safety Malformed or otherwise incorrect updates should not cause the running system to crash. We can guarantee as much by requiring updates to be *safe*. In particular, a safe update will not perform illegal operations that lead to a crash, such as dereferencing a null pointer, indexing an array outside its bounds, adding an integer to a string, *etc.* Typically, notions of safety encompass *type-safety,* which is a well-understood programming language concept. The Java Virtual Machine (JVM) [LY96] has popularized the use of safety as a security mechanism for loaded code. In our context of non-stop, mission critical systems, safety is an especially powerful property since it rules out crashes, which halt service and could result in lost transactions and/or inconsistent state.

Safety can be verified automatically. In the case of program updates, we prefer to verify for safety *before* the system uses an update, say at load-time. Load-time checking simplifies how safety violations are handled, since an update can be easily discarded once it is known to be unsafe. When safety is ensured by runtime checks, safety violations may go undetected for some time, making it harder to remove the faulty update and return the system to a safe configuration.

Completeness While we cannot rigorously show that a dynamic update is correct, we would at least like to show that it is *complete*, meaning that the update addresses the changes that a new version has made to the old one. In other words, for each change, call it δ, between the old and new source code, there is a corresponding code element of the update that addresses δ.

Well-timedness Just as important as the makeup of a dynamic update is the time that it is applied; choosing an incorrect time may, among other things, result in inconsistent state due to race conditions. We would like to show that the timing of an update will not result in an error.

Simplicity Correct updates are of little use when applied with a buggy implementation. One way to reduce implementation errors is to make it *simple*; the simpler the system, the easier it is to understand, and the greater the likelihood that it is correct. We also prefer a simple updating *methodology.* The more complicated the process of building and updating a system, the greater the chance of error.

In a system that enforces safety, as defined above, we must be concerned about its *trusted computing base* (TCB). In security terminology, the TCB is defined as the system hardware and software that must function correctly in order to enforce a security policy. In our context, the policy being enforced is that loaded code is safe, and therefore the TCB consists of the code that verifies safety, as well as the code upon which this verifier relies. To improve the likelihood that the TCB is correct, we prefer that it be kept small and simple [SS75]. There is also an added motivation in our case: the more elements of the implementation that reside *outside* the TCB, the more of the implementation that is provably safe.

Rollback-enabled While we would prefer that updates be correct before they are applied to the on-line system, some mistakes might slip past testing and verification procedures. Therefore, we desire a means to *roll back* a system to its original form upon discovering that an applied update is buggy. Systems may support rollback for a limited window following an update. For example, rejecting an update during safety checking can viewed as a very small rollback window while rolling back following a failed state transformation, say due to a raised exception, presents a larger window. Rolling back at arbitrary times following an update is obviously the most general.

- **Efficiency** Many systems that require non-stop service are high-performance, *e.g.* web-servers, transaction processors, *etc.* Therefore, enabling a program to be updateable should impact its performance as little as possible.

- **Ease of use** A system's applicability is often determined by its ease of use. Many fine tools and products have been ignored simply because they have not been easy to use. We consider a system easy to use if it reduces the workload on the programmer, and/or reduces the complexity of the tasks to be performed.

  One way to make an updating system easy to use is to clearly separate the process of update development from software development. For example, only after a new version of the software is finished will patches be developed to dynamically update the running system to the new version. This way, developers construct their software and test it without needing to think about updates, effectively making software construction and patch construction two modular components of the development process. Modularity is a well-known technique for reducing complexity.

  Automation can be employed to reduce programmer workload, making a system easier to use. In fact, automatic safety-checking, as described above, is often cited as a means to reduce both programmer workload and software complexity, since it prevents a large class of bugs from cropping up. Other forms of automation may be useful as well, such as means to identify changes between two versions of some software.

We have argued that for a practical system, these criteria are necessary, but we have not argued that, in total, these criteria are *sufficient.* Might there be other criteria important for evaluating updating systems? For example, how *deployable* is the system—can it be readily applied to legacy systems, and/or can it work well with other language families? As another example, how *portable* is the system—can it work on a variety of architectures? Finally, how *elegant* is the system? Does it have a elegant mathematical definition?

These are important criteria, but we believe they are of lesser importance. That is, we need a flexible, robust, usable, and low-overhead system before we can worry about its portability—portability is not central to the goal of building a useful system. However, future work may more readily consider other goals given the foundation we establish here.

System	Flexibility			Robustness					Efficiency	Ease of use
	what	when		Sf	C	W	Si	R		
		D	ok							
Dynamic linking			√	√		√	√		√	
DYMOS	√	√	√	√		√				
Argus	√		√	√		√		√	√	√
Conic				√		√	?		?	√
PODUS	√	√				√			√	
PolyLith			√			√			?	
OSVC						√			√	√
Erlang	√		√			√	√		√	
Dynamic ML				√	√			√	√	√
Dynamic C++			√			√	√		√	
Dynamic Java classes				√						√
DITools			√			√	√		√	
GSU			√			√	√	√	√	
DynInst	√		√			√				

Flexibility timing abbreviations:

D	enforces dynamic constraints
ok	no timing restrictions

Robustness property abbreviations:

Sf	Safety
C	Completeness
W	Well-timedness
Si	Simplicity
R	Rollback-enabled

Table 2.1: Evaluating previous general-purpose updating systems

2.2 Evaluating Past Work

A number of researchers have designed and/or built software-based approaches to dynamic updating, with different emphases. In this section, we examine a number of systems in light of our criteria for practicality, identifying what they do and do not do well. As we shall see, there are number of inherent tradeoffs in the design decisions to be made concerning an updating system.

Table 2.1 summarizes our evaluation of past work on general-purpose dynamic updating. The systems/mechanisms we consider here are the following (presented roughly chronologically): dynamic linking (which exists for various languages), DYMOS [Lee83], Argus [Blo83, BD93], Conic [MKS89, MK85], PODUS [FS91, SF93], PolyLith [Hof93], Online Software Version Change (OSVC) [GJ93, Gup94, GJB96], Erlang [AVWW96, Hau94], Dynamic ML [GKW97, WKG98], Dynamic C++ [HG98], Dynamic Java classes [MPG$^+$00], DITools [SNC00], Guarded Software Updating (GSU) [TTA$^+$99, TTA$^+$00], and DynInst [BH00]. A few other systems have been proposed (*c.f.* [ACR98]) but not fully explored so we do not examine them here. In addition, we consider only past general-purpose approaches, not application-specific methodologies. A more detailed survey of the related work presented here can be found in Appendix B (in §B.3). Other useful surveys are found in Gupta [Gup94] and Segal [SF93].

For the purpose of filling in the table, we make the criteria more quantitative as follows:

- Flexibility is broken down into two parts: *what* changes can be effected dynamically, and *when* those changes can take place during program execution. We consider a system to have fulfilled the *what* part if it allows essentially arbitrary updates; that is, no significant programming language feature is restricted from dynamic updates. We break down when updates can take place into two parts. First, we identify whether the system imposes *no restrictions* on update timing. Second, we note whether the system provides support for enforcing dynamic timing constraints.

- Robustness is broken down into the five sub-criteria we identified: *safety* ('Sf' in the table), *completeness* (C), *well-timedness* (W), *simplicity* (Si), and *rollback-enabled* (R). No system can provide an automatic, provable well-timedness property (demonstrated by Gupta *et. al* [Gup94] to be undecidable), so we consider a system to support well-timedness if it provides enough support that the programmer can ensure well-timedness.

- We consider a system to be efficient if it works with high-performance code, and imposes a negligible runtime cost on programs that use updating relative to programs that do not.

- We consider a system to be easy to use if it provides means to reduce the complexity of the updating problem. For example, the system could provide a clear separation between regular and update development, it could provide automated means of developing patches, *etc.*

The user interested in the particulars of each system is referred to SB.3; we have attempted to make the text below highlight the key aspects of each system without requiring an exhaustive summary of each. Our presentation is structured around the evaluation criteria.

2.2.1 Flexibility

Flexibility is the most important criterion with regard to software updating: the less flexible the system, the more likely that an on-line update will not be possible. On the other hand, higher flexibility often means reduced robustness, in terms of implementation and update complexity, and possibly reduced safety. Some systems have favored robustness over flexibility, and thereby chosen to limit their application domain. For example, Dynamic ML limits flexibility in favor of completeness and ease of use. Nonetheless, for an updating system to be practical, it must support essentially arbitrary updates. We consider the two facets of flexibility below: *what* changes can be effected dynamically by the system, and *when* these changes can be effected.

What can be changed

A few systems satisfy both aspects of the flexibility criterion. The most flexible system is DYMOS, which permits changes to the functions, data, type definitions, and even loop bodies of concurrent programs. Erlang, PODUS, and DynInst are similarly flexible, but do not have special support for loops, and PODUS additionally does not support updating type definitions (but this is of little consequence since loaded code is not type-checked). Argus permits the replacement of groups of distributed, multi-threaded processes called *guardians*.

In contrast, many previous systems sacrifice flexibility in favor of other criteria. Dynamic linking, used by a number of systems [App94, PHL97], nicely supports extensible software, but is less useful for effecting arbitrary change (see §1.1.3). On the other hand, basic implementations are reasonably simple, and because the program bindings are stable, it is easy to see that a dynamic change is correct. Three approaches—OSVC, PolyLith, and GSU—use *state transfer* as their underlying updating technology, and therefore suffer from the limitations of that technology (see §1.1.2). On the other hand, these systems fit well in a distributed context, since state can be transferred to programs on other machines.

Other systems exercise other tradeoffs. Conic, a programming environment for distributed systems, only considers changes at the process-level, allowing the adding, moving, or removing of processes and per-process communication channels. Changes to a process's code are not supported (and therefore state is not preserved), but understanding the effects of an update becomes much simpler. Dynamic ML only permits updating modules that export abstract types, and module signatures may not change in arbitrary ways (*e.g.* functions and data cannot change type). This restriction allows for an elegant use of copying garbage collection to change the implementation of the abstract type. DITools focuses on legacy software customization, and not software evolution, and therefore does not consider a number of useful changes, like dynamically changing type definitions. Dynamic Java classes does not permit user-directed state transformation (the system fills in default values for the new state), nor does it permit arbitrary changes to class signatures. Both restrictions are used to ensure that classes are updated correctly. GSU restricts new code to work with existing state without modification, and requires the new code to have roughly the same external behavior (in terms of the messages it sends) as the old code; these restrictions permit old and new versions of the code to run concurrently, permitting a rollback to the old code if errors are detected (see below).

Nonetheless, while the restrictions made by these systems have added value in other areas, they have made the system less practical in terms of supporting a wide array of dynamic changes.

When changes can take place

While most systems permit changes to take place at any time during program execution, PODUS, OSVC, Dynamic ML, and Dynamic Java classes forbid updates to *active* code (that is, code that has an activation record on the stack). This is typically to ensure that only one version of a module/class may be present in the system at a time. Similarly, Conic requires that updates occur only to modules that are *quiescent*, meaning modules that are not performing any processing (*i.e.* they are waiting to receive transactions). In all of these cases, the intent is to increase update robustness by preventing ill-timed updates. Unfortunately, while some ill-timed updates are prevented, some perfectly legal updates are ruled out as well.

In cases when updates to active code are allowed, most systems transition from old to new code at well-defined points, such as at procedure calls (for Erlang, DYMOS, and DITools), or during object creation (Dynamic C++ classes). DynInst updates take place immediately, since the old code is modified in place to jump to *trampolines* to mitigate entry and exit to new code snippets.

Three systems provide system-enforcement of programmer-determined timing constraints. In Argus, like Conic, modules to be updated must be *quiescent*, but here quiescence is defined by the programmer as part of the module definition. Two systems, DYMOS and PODUS, support delaying updates until certain timing constraints are met. For example, the user could specify that the update should be delayed until some number of modules or procedures are inactive. In these cases, constraints are truly *dynamic*, while in Argus they are statically defined as part of the module. While they provide an obvious increase in flexibility, the use of these mechanisms is largely unproven. Two possible reasons are that determining the appropriate constraints is undecidable in general [GJB96], and there is little experience with realistic applications that use these systems.

2.2.2 Robustness

Without some assurances of robustness an updating system is of little practical use, even if it is very flexible, because the integrity of the non-stop system is in question. This tradeoff between flexibility and robustness exists in other contexts as well. For example, strongly-typed languages like Java lose some of the expressiveness of C (being unable to do pointer arithmetic, manual memory management, *etc.*) to ensure that programs will never crash; however, Java is still very flexible, if not the *most* flexible it could be. In addition, bolstering the other facets of robustness an increase the complexity of the implementation. In the end, a system should strive to provide a high level of robustness while still preserving a reasonably high level of flexibility.

Existing systems favor different sides of the flexibility/robustness tradeoff. Dynamic ML favors robustness in supporting a type-safe language (SML), guaranteeing complete patches, and providing rollback. However, supporting these features has led to a complex implementation. On the other hand, DynInst provides a high-degree of flexibility, but uses an unsafe language, relies on a complicated implementation, and has little support to assure patches are correct. DynInst focuses less on robustness because it is targeted at instrumenting existing programs, rather than modifying them arbitrarily for the long term, simplifying typical modifications. Most other systems fall somewhere in between. We consider each robustness property individually below.

Safety

Providing safety is one of the most effective ways of ensuring a high degree of robustness. In all cases, 'safe' programs will not crash,[1] which is a boon for non-stop systems. Anecdotally,

[1]Safe programs will not crash barring implementation bugs, which is why a small TCB is so important.

statically-typed languages like SML catch many bugs during the type-checking phase. However, as we have said, with safety comes a slight decrease in flexibility, making safe languages inappropriate for all contexts. That said, a number of previous approaches use safe languages like Java and/or SML, while the majority of systems do not, being based on C or C++.

Of the safe languages used, all but Erlang are statically-typed, meaning that a dynamic update's safety can be ascertained at link-time. Erlang is dynamically typed, meaning that type errors may arise at runtime, making it potentially difficult for the system to recover from a faulty update since the error could arise long after the update is applied. A number of soft-type systems have been developed for Erlang, somewhat mitigating this problem [MW97, AA98].

Completeness

A system is complete if a programmer may be assured that an update addresses the changes resulting from a new version's code. Completeness is essentially a syntactic property; if a definition changes between the old and the new version, the update should contain code that deals with the change.

Most past systems have been more concerned with enabling dynamic updating mechanisms than with the form and content of the updates themselves. As such, little attention has been paid to completeness, which is more closely tied to the update and not the mechanism used to realize it. The exception is Dynamic ML, which provides a simple updating interface. When a module is replaced, the implementation of one or more of its abstract types may change. During the update, the existing module code is replaced by the new code, and existing instances of abstract type are translated to the new implementation by some user-provided code. When the user writes the new module version, this new code *must* be provided, ensuring the update to the module is complete. In this case, the narrow scope of what may be updated localizes the notion of completeness. In the general case, completeness is a global property, essentially requiring automated support to prove its presence.

Well-Timedness

Most systems make it possible to ensure update well-timedness, though none can do it without programmer assistance.[2] Only Dynamic ML and Dynamic Java classes provide inadequate support. In these systems, a module update may occur at any time other than when the module is active; while often necessary, module inactivity is not sufficient to guard against race conditions. For example, another module could be manipulating the updated module's state when the update occurs. This problem could be fixed without extensive changes to these systems.

[2]Some work was presented in [Gup94] to prove well-timedness for imperative programs without procedures, under certain circumstances.

Simplicity

A simple implementation is more likely to be correct, and is therefore more likely to not introduce fatal errors. A simple implementation also tends to be more portable. Unfortunately, simplicity is the quality where the flexibility/robustness tradeoff most comes into play. In particular, with more flexibility comes a larger, and potentially more complicated implementation.

Of all of the systems that have simple implementations, only Erlang is both simple ([Hau94] describes a C-based implementation) and sufficiently flexible. Most other systems provide flexibility via a complex group of mechanisms. For example, to enforce system-imposed (or programmer-specified) timing constraints at runtime at least requires support for examining the program stack for return addresses pointing to the relevant module [Lee83, FS91, MPG$^+$00, GKW97], and potentially a means of delaying the update until no such addresses are found [Lee83, FS91]. In the presence of multi-threading, all thread stacks must be traced, and determining activeness may require locking on procedure entry and exit [Lee83]. PODUS uses segmented virtual memory, requiring potentially complex OS support if implemented directly (it can also be simulated in user-space), and Argus leverages language constructs for persistent transactions and recovery, which are difficult to implement.

Systems that enforce type safety require verification software that is part of a potentially large trusted computing base. In particular, a *trusted compiler* is employed by the updating systems of Argus, DYMOS, Conic, Dynamic ML, and the dynamic linkers of OCaml and Haskell. That is, source language safety is checked during compilation, but target language safety is not assured, in effect trusting the compiler not to introduce violations. Argus, OCaml, and Conic go a bit further than this, ensuring that target code is *link-safe*; that is, the interface advertised by the target code (but not confirmed) is verified to be consistent with the context of the running program when it is linked. Java-based systems do verify safety in the target code, and thus the Java-to-JVM compiler (*i.e.* `javac`) can be untrusted. However, since the target code is virtual machine bytecode, we must trust the JVM interpreter, or else trust a just-in-time (JIT) compiler that translates JVM code to machine code following verification.

Rollback-enabled

All of the other robustness properties look to prevent errors before they arise; conversely, rollback can be used to reverse the damage caused by a faulty update. Most past systems have focused on enabling updateable programs, with rollback being considered less important future work.

Two systems provide short-term rollback. Argus provides rollback not in the updating system *per se*, but in the transaction facility leveraged by the updating system. Parts of an update can be made into a transaction, which can be rolled back on failure. Dynamic ML provides a similar kind of rollback. After the new code for an abstract type has been loaded, the existing instances of that type must be converted to match the definition of the new code. If during this translation process an exception is raised, then the entire update is aborted and the system is rolled back to the state just before the update occurred. Dynamic ML leverages copying garbage collection as a mechanism to enable this.

One system, GSU, has proposed a more general form of error detection and rollback. Here, the updated code runs in a separate process concurrently with the existing code, with two changes: the messages sent by the new code are checked for accuracy with programmer-provided *acceptance tests*, while messages that would have been sent by the old version are logged. If an erroneous message is detected, the old version is restored. So that this switch over is semantically consistent, GSU employs checkpointing technology [Pla97] to checkpoint the state of the old version when its is known to be consistent, so that the system can roll back to that state on a failure. Checkpointing is also used to enable the new version to begin with the state of the old version. Unfortunately, enabling GSU error detection and recovery technology seems to limit the flexibility of the underlying updating system (see B.3.13).

2.2.3 Efficiency

Most systems appear to be reasonably efficient (although few demonstrate as much experimentally), adding either no additional runtime overhead, or only a modest one (*e.g.* an added indirection per function call). The few exceptional cases have sacrificed some level of efficiency to gain a more elegant or more flexible updating model.

- DYMOS introduces high per-function call overhead due to extensive locking sequences before and following each call. This support owes to its need to track the active procedures and modules in multiple threads.
- PODUS similarly has a potentially high per-procedure call overhead since it uses system calls into the OS to leverage segmented virtual memory (simulating segmented virtual memory in user-space reduces this cost at the loss of some flexibility). Segmented virtual memory allows multiple versions of code to coexist, allowing for more relaxed transition semantics following update.
- Dynamic Java classes lose significant performance because the majority of the support for dynamic updating is built into a single, bytecode-interpreted JVM to simplify the implementation.
- DynInst loses performance due to its use of trampolines; each change to a function results in a branch to some trampoline code that saves (and restores) machine state before jumping to (and returning from) the new code.

2.2.4 Ease of Use

Few systems have focused on usability, instead favoring flexibility and robustness. As a result, most systems lack a clear separation of update development versus normal development, meaning that code relating to updates becomes intermixed with normal development code, making development and maintenance more difficult and less modular. There has also been little focus on the use of automation to reduce update complexity.

Two systems that nicely separate update code from normal code are OSVC and Dynamic ML. In OSVC, the programmer writes the new version of the program from the old without concern for patches. After development and testing are finished, a *state transformer function* is written to transform the running program's state into a form usable by the new version. In Dynamic ML, similar functions are defined, one per abstract type, for each module that has changed. Dynamic Java classes is a degenerate form of Dynamic ML, whereby the per-class transformation code is generated automatically (and somewhat inflexibly). Conic also separates its *reconfiguration directions* from normal code.

Some systems provide automated support to reduce the complexity of the updating process. Both PODUS and DYMOS provide support for enforcing runtime timing constraints. However, this support is only marginally helpful, as determining the correct constraints can be quite complicated. PODUS does automatically enforce certain syntactic constraints. Argus's use of transactions greatly simplifies keeping runtime state consistent during an update. On the other hand, the syntax and semantics of the Argus language can be unintuitive.

2.2.5 Summary

Maximizing the benefit of an updating system means trading off the various evaluation criteria. The mechanisms and approaches explored in past work are extensive, and each system has focused on certain areas in the design space, in some cases favoring robustness over flexibility or performance, or perhaps the reverse. However, no prior system has effectively balanced the tradeoffs to become truly *practical*. In the next section, we describe how by borrowing mechanisms from these systems, and placing more of an emphasis on simplicity and ease of use, we can arrive a more practical system.

Chapter 3

Approach

Previous systems each have their strengths in the areas of flexibility, robustness, low overhead, and ease of use, but no system is strong in *all* areas. In this chapter we outline our approach and argue that it finds a 'sweet spot' in meeting the evaluation criteria: it provides sufficient flexibility and robustness, imposes a low overhead, and is easy to use. We begin by outlining the strategy we have taken in designing and building our general-purpose updating system, describing the major elements of our system in the process. Following this discussion, we evaluate our system and show that it meets the four evaluation criteria.

3.1 Strategy

In this section we describe the strategy we took in developing our dynamic software updating system. We start by considering the core technology we employ, and then describe how we build on it to ultimately support general-purpose dynamic updating. The presentation of our strategy mirrors the structure of the remainder of the document, so this section serves as a technical overview and outline; where applicable we indicate which chapters develop each point.

1. Build on dynamic linking. There are three obvious mechanisms we could choose as the foundation of our approach: state capture and restore (as used by OSVC and PolyLith), code insertion by trampolining (as in DynInst), or dynamic linking. State capture has known flexibility limitations (see §1.1.2 and §2.2.1). Trampolining has a number of disadvantages, including implementation complexity, execution-time overhead, and platform dependence. Type-safe dynamic linking already meets quite a number of the evaluation criteria, as shown in Table 2.1, but by itself lacks sufficient flexibility (see §1.1.3). Our strategy is to start with dynamic linking and then build the needed flexibility on top of it, while keeping a simple implementation and a small trusted computing base. In essence, as alluded to in the introduction, we develop an automated way that converts every program module into a plug-in.

2. Use verifiable native code. Existing type-safe dynamic linking implementations have one of two drawbacks. In many cases, the trusted computing base includes a trusted compiler, since only source language safety is verified, and not target language safety. Systems like Java do verify target language safety, but at the cost of either using a slow, byte-code interpreter, or by including a large JIT compiler in the TCB. *Verifiable native* *code* (VNC) systems, like Proof-Carrying Code (PCC) [Nec97] and Typed Assembly Language (TAL) [MWCG99], mitigate these problems by permitting *native* code to be verified for safety. This approach avoids both the performance cost of byte-code interpretation and the security cost of having a compiler in the TCB.

For our implementation, we chose to use Typed Assembly Language. TAL defines a framework in which native machine code is coupled with annotations so that it is provably safe. In TAL, 'safe' code is many things: in addition to being type-safe, it is memory-safe (*i.e.* no pointer forging), control-flow safe (*i.e.* no jumping to arbitrary memory locations), and stack-safe (*i.e.* no modifying of non-local stack frames). TAL has been implemented for the Intel IA32 instruction set; this implementation, called TALx86 [MCG+99], includes a TAL verifier and a prototype compiler from a safe, C-like language called Popcorn to TAL. Chapter 4 describes TAL and Popcorn in more detail.

3. Implement dynamic linking for VNC (TAL). Existing VNC systems do not support dynamic linking. As a result, we must first implement dynamic linking for VNC. This implementation should be as simple as possible, have a small TCB, impose a low overhead on linked code, and be flexible enough that we can build the necessary updating mechanisms on top of it.

Roughly, dynamic linking requires three operations: *loading* the new code into the running program, *linking* that code with the existing code, and *managing the symbols* of the code to be used in future dynamic linkages. To maximize flexibility but minimize complexity and the need for trust, we divided our dynamic linking implementation into trusted and untrusted components, such that the trusted part takes care of loading, while the untrusted part takes care of linking and symbol management.

The trusted part, called *TAL/Load*, consists of extensions to the TAL language and runtime system, including a special load primitive that loads a module and verifies it for safety, and runtime *type representations*, needed to build a type-safe symbol table. Our implementation of TAL/Load and its theoretical underpinnings are described in Chapter 5. The untrusted part builds upon TAL/Load to provide compiler and library support for dynamically linking Popcorn modules. We provide an API for Popcorn programs called DLpop, which is similar to DLopen [Lin95], a common dynamic linking API for Unix-based C programs. The implementation of DLpop is described in Chapter 6. Because DLpop consists of TAL code, or code that generates TAL code, it can be verified for safety, improving robustness. The components of our approach are summarized in Figure 3.1.

4. Define a notion of dynamic patch. Given an implementation of dynamic linking for TAL, we need to consider how to build on it to enable dynamic updating. The first step is to define the unit of dynamic update. In our case, an update consists of one or more *dynamic patches*, defined in the first half of Chapter 7 (§7.1). A dynamic patch to an existing module can be described as (1) the new version of the module, and (2) the code and data needed to support updating that module dynamically. An important part of the update code is a state transformer function, like that used in OSVC and Dynamic ML, that computes the new module's starting state from that of the old module.

Because our notion of dynamic patch cleanly separates the new code from the update support code, normal development can take place independently of patch development.

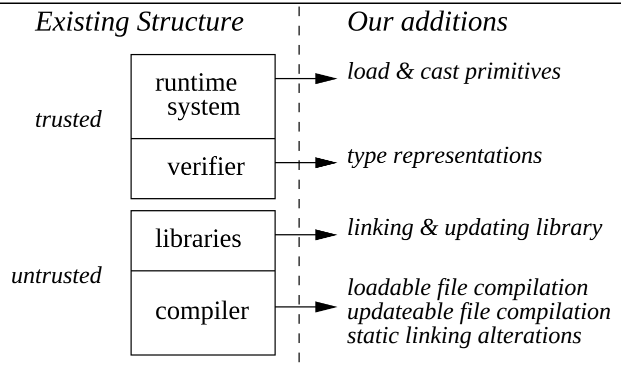


Figure 3.1: The trusted and untrusted components of our implementation in TAL.

Once the appropriate changes have been made to the program, patches can be written by including the new module code and any additional code needed to transform the state. The patch is then compiled into a single TAL module, to be applied by dynamically linking it into the running program.

5. Enable running programs to be dynamically patchable. Once a patch has been dynamically linked, its state transformer function is invoked to transform the existing state, and the running program is 'fixed up' to use the new code. Our approach is that following linking and state transformation, we *relink* the existing program modules to use the new code. If old code is running at the time of update (or is referenced on the stack), it will continue to be used until control exits from it; new calls into the updated module will go into the new code. This is the same approach as taken by Erlang, DYMOS, and others. Care is taken to ensure that code is made *unreachable* as soon as it is no longer needed; this includes removing references to old code from its dynamic patch and from the dynamic symbol table following linking. This way, once the old module is no longer active, it can be safely garbage-collected.

The benefit of this approach is that it reuses functionality already present to support dynamic linking. Doing so keeps the implementation simple, requires no additions to the TCB, and introduces no additional overhead. Furthermore, by allowing running code to be updated, the system places no limits on the time that a patch can be applied. How we enable dynamic patches is described in detail in the second half of Chapter 7 (§7.2).

6. Ensure updates are type-correct and well-formed. The relinking process must be type-correct. In particular, if an updated module changes the type of any of its functions or data, then the system must ensure that doing so does not violate type-safety. The simplest way to do this is to simultaneously update other modules that refer to these functions or data. Alternatively, the programmer can define *stub functions* having the old type to be interposed between old callers and new functions. Stub functions can also be useful for implementing transitional computation, such as incremental state transformation.

We must also properly handle any type definitions that have changed. To do this, we identify changed type definitions during patch creation and transparently rename each type name at compile-time, rather than deal with changes during linking or updating. While the programmer must manually translate old type instances to new ones in the state transformer and/or stub functions, this approach avoids the implementation complexity and reduced flexibility of systems like Dynamic ML and Dynamic Java classes.

Finally, we ensure that an update will not damage the system during state transformation by supporting rollback in the style of Dynamic ML. That is, if any state transformation function raises an exception, the entire state of the system is rolled back to what it was before the update.

7. Generate patches (mostly) automatically. Given a program that is dynamically patchable, we need to consider how we will generate patches for it. Many past approaches have required the programmer to generate patches entirely by hand, notably the state transformer and stub functions. To greatly reduce this burden, we developed a tool to generate patches mostly automatically; it is described in the first half of Chapter 8 (§8.1). The tool identifies all changes to a program from one version to another and generates a patch. Changes are addressed either by generating some patch code, or by inserting a placeholder when generating code is not possible, so that the programmer may address the change. The tool ensures that patches are complete and makes the system easier to use.

8. Ensure patches are well-timed. Given a set of well-formed patches, we need to determine an appropriate time to apply them to the running program. Rather than support mechanisms to enforce user-provided timing constraints at runtime, we instead rely on the programmer to construct the program to perform its own updating, ensuring in advance that updates will be well-timed. As a result, we avoid the implementation complexity imposed by runtime enforcement mechanisms, but lose some flexibility, since the programmer must code the system to anticipate updating. However, in our experience this added burden is minimal, and the benefit of enforcing dynamic constraints is largely unproven. Issues of timing are explained in Chapter 8 (§8.2).

9. Validate the system using a realistic application. A problem of many past systems is that they failed to validate their abstractions on realistic programs. For example, PolyLith, PODUS, Dynamic Java classes, and others only considered toy programs, while Dynamic ML lacks an implementation entirely.

To validate our approach, we decided to build a reasonably complete webserver *incrementally*, starting with a basic implementation and adding sizeable features over time. After deploying the first version of the system publicly, we updated it four times with significant new functionality. This process was critical in informing our design, especially in the development of our patch generator and compiler. We also used the webserver to measure the overhead that updateability imposes on a running program.

3.1.1 Limitations

The approach that we have described has a number of limitations:

1. Our approach is necessarily tied to imperative, C-like languages, since its implementation language, Popcorn, is of this sort. On the other hand, this limitation is also

a feature, since a low-level, imperative languages like C can serve as the target for other language styles. In particular, we have begun to explore how our approach would apply to functional and object-oriented languages; we present our progress on this topic in Chapter 11.

2. We cannot seamlessly support some of the more advanced language features of Popcorn. In particular, we do not support updating exceptions and exception constructors, and have not implemented support for updating function pointers and abstract types. However, we know at least one way to update function pointers and abstract types, and sketch how to do so in Chapter 11. We also do not support updating long-running loops (as DYMOS does).

3. We only have experience with single-threaded programs, owing to the fact that TAL and Popcorn currently do not support multi-threaded programming. However, our approach will work just as well in multi-threaded programs, though ensuring well-timedness becomes more difficult.

4. We do not consider distributed programs. The problem of dynamically updating multiple, distributed processes is necessarily more complex, since it subsumes the problem we consider of updating a single process. However, we feel that our approach fits well into previous approaches to updating distributed systems, like Conic and PolyLith, that require single-process updateability but do not implement it well.

5. We only briefly consider the problem of updating programs for which not all of the source code is available. Being able to support this would be useful for certain *active network* implementations, which allow multiple parties to load code into network routers. As a result, each user may have only a partial view of the entire system, complicating the process of updating. We consider this issue somewhat in Chapter 11.

6. Finally, we do not rigorously consider the problem of proving that an update is well-timed. Instead, we provide enough support and flexibility that users can construct applications and updates to be valid; this is no better (and no worse) than previous systems. This limitation is arguably the most important one we have mentioned; providing a means for proving updates are well-timed, or even providing a framework to establish a reasonable well-timedness property, is important future work.

3.2 Evaluating Our Approach

In designing our system, we have focused on retaining a high level of flexibility and performance while bolstering the system's robustness and ease of use.

- Our approach is flexible enough to update all of the major features of Popcorn without any system-imposed timing restrictions.

- The system is made robust by type-safe code (Popcorn and TAL), a simple implementation (dynamic linking and code relinking) with a small trusted computing base (due to TAL and our approach to dynamic linking), has complete patch files, and imposes no unreasonable timing restrictions.

- Our approach is only slightly less efficient than statically linked code; overheads result from load-time verification and an extra indirection imposed by dynamic linking.
- Developing updateable software is simplified by our clean separation of software and patch development, and by our automated patch generator.

Compared to previous systems, our system is nearly as flexible and efficient, but more robust and easy to use. Our relative decrease in flexibility and efficiency is quite minimal overall, and results in significantly larger gains in robustness and ease of use.

3.2.1 Contributions

In proving our thesis, we make the following contributions:

1. Our primary contribution is to show that one can build a *practical*, general-purpose updating system that is flexible, robust, low overhead, and easy to use. No prior approach is as practical. Because we prove this point by construction, a corresponding contribution is an updating system *implementation* that meets the desired criteria.
2. We have developed the first complete framework for safe dynamic linking of (verifiable) native code. The system that we have built is the first to enable dynamic linking of native code in a way that is both safe and flexible enough to support a variety of dynamic linking (and updating) strategies.
3. We have defined and implemented a notion of dynamic patch that cleanly separates the concerns of program and patch development. This simplifies the development process and makes program code more maintainable, since the program does not become 'polluted' with code relating to dynamic patching.
4. We have employed a novel approach to dealing with changes to type definitions by renaming them. This approach works well in practice, and avoids the implementation complexity of true type replacement, as employed in systems like Dynamic ML and Dynamic Java classes.
5. We have developed a tool that mostly automatically generates patches, given two versions of a program. This tool greatly simplifies the process of developing dynamic updates and ensures that updates are complete.
6. We show that VNC technology is flexible enough to support dynamically updateable programs. The use of VNC increases the robustness of both the running program and its dynamic patches.
7. We have built a sizeable updateable application: an updateable webserver. As far as we know, ours is the largest application described in the general-purpose dynamic updating literature to be updated in non-trivial ways over a lengthy course of time.
8. We show by direct measurement that dynamic updateability can impose a low overhead. Ours is one of the few systems to have documented performance data.

3.3 Roadmap

The remainder of this document more fully describes the approach outlined in this chapter, supporting the arguments presented here. Our presentation mirrors the description of the system in §3.1. Following some background in Chapter 4, the next two chapters describe our approach to adding dynamic linking to TAL. Chapter 5 presents the trusted component of our implementation, called TAL/Load, while Chapter 6 describes the implementation of a safe dynamic linking API built on TAL/Load, based on the standard, C-based approach for Unix systems.

The next two chapters describe how we build on dynamic linking to support dynamic updating. In Chapter 7, we describe the how we extend the untrusted dynamic linking mechanisms to support updating. In Chapter 8, we describe the process of building updateable systems using our approach, considering issues of timing and patch generation. As a case study, in Chapter 9, we describe the implementation of an updateable webserver. In Chapter 10, we analyze the performance of dynamic linking and dynamic updating, both component costs and application performance. To measure of application performance impact, we compare the throughput measured for updateable and non-updateable versions of the webserver. Chapter 11 considers future work, and we conclude in Chapter 12.

Chapter 4

Background

In this chapter, we provide some background on Typed Assembly Language and Popcorn, which should help the reader understand the code examples in the rest of the document. The reader not interested in the code examples, but primarily the high-level ideas, can safely skip this chapter, or refer back to it as needed.

4.1 TAL

A concise description of TAL can be found on the TAL home page [TAL99]:

> Typed Assembly Language (TAL) extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules. These typing rules guarantee the memory safety, control flow safety, and type safety of TAL programs. Moreover, the typing constructs are expressive enough to encode most source language programming features including records and structures, arrays, higher-order and polymorphic functions, exceptions, abstract data types, subtyping, and modules. Just as importantly, TAL is flexible enough to admit many low-level compiler optimizations. Consequently, TAL is an ideal target platform for type-directed compilers that want to produce verifiably safe code for use in secure mobile code applications or extensible operating system kernels. We have implemented a variant of TAL for Intel's IA32 architecture called TALx86, and have written a compiler for a safe C-like language called Popcorn to TALx86.

While we could present more detail concerning the features, syntax, and semantics of TAL, doing so would not aid the reader's understanding of this work. Instead, we refer the interested reader to the introductory TAL theory paper [MWCG99]. We will, however, present some relevant information concerning the TALx86 implementation.

The implementation includes a number of tools, including a TAL assembler and linker, called `talc`, and two Popcorn compilers. In addition to performing assembly and linking, `talc` *verifies* TAL files for safety, and verifies that the linking process is safe (the theoretical details of link-checking can be found in [GM99]). The two Popcorn compilers consist of a simple, bootstrap compiler, and more sophisticated, optimizing compiler. The bootstrap compiler is written in Objective Caml (OCaml) [Ler00], a descendant of the functional language ML. This compiler is largely unoptimized, in particular lacking a register allocator, and so it uses the stack heavily. A newer, more sophisticated version of the compiler is written in Popcorn itself. This version implements register allocation as well as a number of traditional optimizations. Our implementation uses the OCaml version of the compiler, essentially because it was easier to modify, although efforts are underway to add the necessary features to the newer compiler as well. An excellent introduction (though somewhat dated) to the TALx86 implementation, particularly in the way Popcorn is compiled to TALx86, can be found in [MCG+99].

4.2 Popcorn

This section describes the essential aspects of the Popcorn language. Sections 4.2.1, 4.2.2, and 4.2.4 are taken from [MCG+99]. Section 4.2.5 is taken largely from the Popcorn language manual; both texts required minor alterations to reflect the current implementation.[1] Section 4.2.6 describes features we added to Popcorn to support the dynamic linking source-to-source translation described in Chapter 6.

The Popcorn language purposely looks like C, and includes some standard enhancements such as more flexible variable declarations and a C++-like namespace mechanism. Unsafe features, such as pointer arithmetic, the generic address operator, and pointer casts, are missing. Compiling these features safely would impose a significant performance penalty on all Popcorn code. Popcorn does have several advanced features not in C such as exceptions and parametric polymorphism.

A program in Popcorn is constructed from one or more files (typically having suffix `.pop`), each containing a number of top-level definitions, including function, data, and type definitions. We consider each Popcorn file as a separate module, where all modules share a global namespace. The Popcorn compiler uses the C preprocessor, so Popcorn programs can use `#ifdef`'s, `#include`, *etc.* Program execution begins in the `pop_main` function, which is analogous to C's `main`.

As a quick example, the following is the 'hello world' program written in Popcorn:

```
#include "core.h"
void pop_main() {
  printf("Hello World\n");
}
```

The included file `core.h` contains frequently used routines, including file I/O. The `printf` 'function' is not actually a function, but special syntax that is expanded by the compiler into a series of calls to functions in `core.h`. Argument processing from the command-line is handled by a separate library, as opposed to using `argc` and `argv`, as in C.

4.2.1 Control Flow

The basic control constructs of Popcorn, such as `if`, `while`, `for`, `do`, `break`, and `continue`, are identical to those in C except that test expressions must have type `bool` (the result type of relational and logical operators is `bool`).

[1]Thanks to the TAL team for providing this text and the permission to use it: Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic.

Popcorn's `switch` construct differs from C in that execution never "falls through" cases. Furthermore, a default case is required unless the other cases are exhaustive. The argument of a switch test expression can be an `int`, `char`, `union`, or `exception`. For example, we could find the first occurrence of the character `'a'` in an array:

```
int i = 0, answer;
while (true)
  switch arr[i] {
   case 'a': answer = i;
             break; // break from while
   default:  i++;
  }
```

Array subscripts are bounds-checked at run time; the above example will throw an exception `ArrayBounds` if `arr` does not contain an `'a'`.

Exceptions may have different types and exception handlers may switch on the name of an exception, as in Java. However, exception names are not hierarchical.

4.2.2 Data

Currently, the simple types of Popcorn are `bool`, `char`, `short`, `int`, `string`, `float` and `unsigned` variants of the numeric types. Unlike C, strings do not require a null-terminator. Arrays (and strings) carry their size to support bounds-checks. A special `size` construct retrieves the size of an array or string.

Popcorn also has tuples which are useful for encoding anonymous structures and multiple return values. The `new` construct creates a new tuple (as well as new `struct` and `union` values). For example, the following code performs component-wise doubling of a pair of ints:

```
*(int,int) x   = new (3, 4);
*(int,int) dbl = new (x.1+x.1, x.2+x.2);
```

Popcorn has two kinds of structure definitions: `struct` and `?struct` (we will refer to these collectively as `struct` definitions from here on). They resemble `struct *` in C. The difference between `struct` and `?struct` is that values of types defined with `struct` cannot be `null`, which is a primitive construct in the language. Values of types defined with `?struct` are checked for null on field access; failure results in a `NullPointer` exception. Note that as in C, field order matters. That is, `struct s { int a; float b; }` is not equivalent to `struct s { float b; int a; }`; these constitute two different types.

Unions in Popcorn are more like ML datatypes than C unions. Each variant consists of a tag and an associated type (possibly void). For example,

```
union tree
{void Leaf; int Numleaf; *(tree,tree)Node};
```

Any value of a `union` type is in a particular variant, as determined by its tag, and may not be treated otherwise. We use `switch` to determine the variant of an expression and bind the corresponding value to a variable. Continuing our example, we can write:

```
int sum(tree e) {
 switch e {
  case Leaf: return 0;
  case Numleaf(x): return x;
  case Node(x): return sum(x.1)+sum(x.2);
 }
}
```

Notice that in declaring a value of `union` type, we do not prepend the `union` keyword (we say `tree e` as opposed to `union tree e`); the same is true for `struct` value declarations. In addition, unlike C, a type declaration may only be made once; to use that declaration in other files requires prepending the declaration with `extern`, as is typically done for data. This feature is important for abstract types, described below.

4.2.3 Functions

Popcorn functions are essentially the same as C functions, with some syntactic differences. First, array and function pointer modifiers for a function's return type follow the argument list; a function `f` that takes no arguments and returns an integer array has type `int f() []` (as opposed to `int[] f()`), and a function `f` that takes no arguments and returns a function from ints to ints has type `int f() (int)`. Second, function pointers have the same syntax as regular function prototypes, *i.e.* the C declaration `int (*f)(int)` is simply `int f(int)` in Popcorn. To avoid ambiguity, function pointers may not be declared as globals.[2] For example, the following code is not allowed:

```
int g(int x) { return x+1; }
int f(int) = g; // error
```

To get around this restriction, global function pointers can be encapsulated in a `struct` or tuple; function pointers as local declarations are not problematic.

4.2.4 Parametric Polymorphism

Popcorn functions, `struct`, and `union` declarations may all be parameterized over types. For example, we can define lists as:

```
?struct <'a>list {'a hd; <'a>list tl;}
```

To declare that a variable `x` holds a list of ints, we instantiate the type parameter: `<int>list x`. Explicit type instantiation on expressions is not necessary; for example, `new list(3,null)` has type `<int>list`. Having polymorphic functions means we can write a length function that works on any type of list. Polymorphism is particularly useful with function pointers. For example, we can write a map function:

[2]Otherwise, it would be unclear to the compiler whether the function `f` in the declaration `extern int f()` were a function or a function pointer, which have different representations

```
<'b>list map<'a,'b>('b f('a), <'a>list l) {
   if (l == null) return null;
   return new list(f(l.hd), map(f, l.tl));
}
```

A call to this function could look like:

```
<int>list x;
...
<string>list y = map(int_to_string, x);
```

4.2.5 Type Abstraction

Popcorn supports type abstraction in two ways: with the qualifier `abstract`, and with a form of existential types [MP88], referred to in Popcorn as `abstype`'s. Prepending a `struct` or `union` declaration with the `abstract` keyword will hide its implementation from external client code. For example, to make an abstract lists implementation, we do:

```
abstract ?struct <'a>list {'a hd; <'a>list tl;}
```

To use this list, clients define a reference to this `list` type without its implementation, as:

```
extern list?<'a>;
```

The `?` and type parameter list `<'a>` are used only if needed (they should be omitted if the type cannot be null and/or is not polymorphic). Abstraction with `abstract` is enforced by the TAL module system, and follows ML-style module systems' notion of *opaque* types (*e.g.*, see [HL94, Ler94]).

The second way of defining abstract types is via a more 'first-class' mechanism called `abstype`'s. Since `abstype`'s are an important part of our dynamic linking implementation, we include a complete tutorial below, taken (with slight formatting changes) from the Popcorn manual that comes with the TALx86 distribution.

Existential Types (`abstype`'s)

Data declared with the `abstype` keyword is a form of first-class abstract data built on existential types [MP88], which are similar in many respects to a very primitive form of object type. `abstype`'s are particularly useful when one wants to manipulate heterogeneous data structures. Like a `struct` or `union`, an `abstype` can be polymorphic. Unlike `struct`s or `union`s, an `abstype` can abstract or hide certain types as well. The typical use of an `abstype` is when we want to export some but not all information about a type. For instance, when representing objects, we may want to hide the types of the instance variables but expose the types of the methods. Furthermore, the methods should take the instance variables as extra arguments. As another example, a closure may be represented by an abstract environment and a function which, when given the environment and an argument, produces a result.

Like `struct`s and `union`s, `abstype` values are created by using `new`. To manipulate an `abstype` value, we must use the `with` construct to open up the abstracted type in some scope.

As a simple example, suppose we have two different representations for two-dimensional points, polar and cartesian, with appropriate operations defined on them:

```
extern struct polar {int mag; int angle;}
extern polar add_polar(polar x, polar y);
extern polar sub_polar(polar x, polar y);
extern polar mul_polar(polar x, polar y);

extern struct cartesian {int xcoord; int ycoord;}
extern cartesian add_cart(cartesian x, cartesian y);
extern cartesian sub_cart(cartesian x, cartesian y);
extern cartesian mul_cart(cartesian x, cartesian y);
```

Unfortunately, Popcorn does not allow one to mix polar and cartesian values directly, for example by mixing them in a list. `abstype`'s allow us to abstract the particular representation of a type (in this case, whether a point is polar or cartesian) and package up the operations on values of those types. For example, we might define a generic point object as follows:[3]

```
struct <a>point_rep { a data;
                      a add_point(a,a);
                      a sub_point(a,a);
                      a mul_point(a,a); };

abstype point[p] = <p>point_rep;
```

Informally, the `abstype` definition defines a new type `point` that hides or abstracts the representation of the field data (`p`) allowing us to mix different point representations. For example, we might define:

```
point polar_point(int mag, int angle) {
  <polar>point_rep pol =
    new point_rep(new polar(mag, angle),
                  add_polar, sub_polar, mul_polar);
  return new point(pol);
}

point cartesian_point(int x, int y) {
  <cartesian>point_rep car =
```

[3]Using conventional lambda-calculus-style notation, the Popcorn code be expressed with the following type definitions:

$$\begin{array}{lll} \mathit{point\_rep} = & \Lambda a.\{ & \mathit{data} : a, \\ & & \mathit{add\_point} : a \times a \to a, \\ & & \mathit{sub\_point} : a \times a \to a, \\ & & \mathit{mul\_point} : a \times a \to a \ \} \\ \mathit{point} = & \exists p.(\mathit{point\_rep}\ [p]) & \end{array}$$

That is, *point_rep* is a type operator that takes a type argument a, while *point* is an existential type that applies the type operator *point_rep* to the type variable p.

```
    new point_rep(new cartesian(x, y),
                  add_cart, sub_cart, mul_cart);
  return new point(car);
}
```

Notice that both function definitions return point values and that the return type makes no mention of whether the representation of the point is polar or cartesian. Indeed, at the point where we create a point (`new point(pol)` or `new point(car)`), we have abstracted the representation. This is a lot like casting an object of a particular class in Java to one of the interface types that the class implements—you lose the specific information about what kind of object it is and must manipulate it through its abstract interface.

With these two definitions for creating points that are either polar or cartesian, we can define, for instance, a list that mixes both kinds of points:

```
<point>list points = new list(polar_point(10,15),
                       new list(cartesian_point(0,0),
                          new list(polar_point(3,3),null)));
```

We can then write a function to manipulate the list through the exposed interface. A simple example follows:

```
<a>point_rep double_point_rep<a>(<a>point_rep pr) {
   a new_data = pr.add_point(pr.data,pr.data);
   return new point_rep(new_data,pr.add_point,
                         pr.sub_point,pr.mul_point);
}

point double_point(point x) {
   with pr[p] = x {
     <p>point_rep new_pr = double_point_rep(pr);
     return new point(new_pr);
   }
}
```

In this example code, we first define a polymorphic function which, when given a `point_rep` where the data has type `a`, returns a new point representation where the data has the same type. We do so by simply adding the old point representation data to itself and packaging this up with the operations. But this function manipulates `<a>point_rep` values, not points. The `double_point` function does what we need to take a point with any representation and apply the `double_point_rep` function to that representation.

The body of `double_point` uses a `with` statement to 'open up' the abstract point `x`. Within the scope of the `with` statement, the variable `pr` is bound to the point representation and the type variable `p` is bound to the type of the point representation's data. That is, `pr` has type `<p>point_rep` (for some type named by `p`) and can only be used within the scope of the `with` body. Within the body, we call the `double_point_rep` function on `pr.` Given any type `p`, `double_point_rep` will take a `<p>point_rep` and produce a `<p>point_rep`, thus we get back a `<p>point_rep` for a result. We then abstract `p` again by placing the new `<p>point_rep` value `new_pr` in a point. Finally, we return the new point as the result of the function.

We could then use the function `double_point` on each element of our list `points` to double the value of each point:

```
<point>list dpoints = List::map(double_point, points);
```

Here we use the well-known *map* function, denoted `List::map`, which applies the function provided as its first argument (in this case `double_point`) to each element of the list provided as its second argument (in this case `points`), returning the results in a new list, in this case bound to the variable `dpoints`.

The `with` statement allows us to unpack or open up an abstract data type within some scope. All this really means is that it gives us a way to name the type for a limited amount of code and to get to the underlying value. Notice that if we open up two different points, then the type-checker forces us to use different names for the two points and thus, their respective point representation types cannot be treated as the same:

```
point point1,point2;

with pr1[p1] = point1 {
  with pr2[p2] = point2 {
    pr2.add_point(p1.data,p2.data);  // fails to type-check!
  }
}
```

In the above example, the attempt to add `point1`'s data and `point2`'s data fails to type-check. The reason is that `point1`'s representation might be incompatible with `point2`'s representation. One might be tempted to rewrite the code so that we replace `p2` with `p1`:

```
with pr1[p1] = point1 {
  with pr2[p1] = point2 {     // fails to type-check!
    pr2.add_point(p1.data,p2.data);
  }
}
```

but the Popcorn type-checker will reject this. In general, Popcorn requires that semantically distinct type variables be syntactically distinct—it does not implicitly α-vary them.

4.2.6 Added Features

To support converting files to be loadable and updateable via a source-to-source translation, described in detail in Chapters 6 and 7, we needed to add some features to Popcorn. We briefly describe those features here.

The & Operator

Supporting a generic & operator in a safe manner is problematic, so the original Popcorn design left it out. A central problem is that taking the address of a stack-allocated value can result in a dangling pointer if the result escapes outside the scope of the value's declaration (*e.g.* if a local variable's address is returned to the caller).

On the other hand, when data is not tied to a local scope, acquiring its address is safe. In particular, taking the address of global values is always safe. In addition, taking the address of heap-allocated data is also safe, as long as the value that results is well-formed. A simple case is taking the address of a particular field in a `struct` or tuple. For example, given our `struct` definition of the `<'a>list`, above, we could do the following:

```
<int>list l = new list(1,new list(2,null));
*(<int>list) l_elem = &(l.tl);
l_elem.1 = new list(3,null);
```

In the first line, we create a two-element list. In the second line, we acquire the address of the second field of the first element. The result has the same type as the field, but with an added level of indirection; this is encoded in a tuple type. In the third line, we dereference the address to assign a new list element. This effectively changes the `tl` pointer in the list `l` to point to the new element. The overall effect is that the list `l` now has values `1` and `3`, rather than `1` and `2`.

As another example, we can take the address of a tuple member:

```
*(int,int,int) three = new (1,2,3);
*(int) field2 = &(three.2);
field2.1 = 3;
```

This code has the effect of taking the address of the second field of the declared tuple, and then assigning to it the value `3`. These approaches extend to nested structures and tuples as well.

In total, we have implemented & to work on global variables and the fields of all `struct` and tuple values. While arrays and strings, which are always heap-allocated in Popcorn, could be handled as well, they are less straightforward because they each have an associated header that stores the length, needed for bounds checks. Taking the address in the middle of a string would require constructing a new header to alias the substring and store its length. We could add this feature if needed.

First-class Exception Constructors

Popcorn exceptions are first-class, but initially, Popcorn *exception constructors* (*a.k.a* exception names) were not. An exception constructor can be thought of as the *tag* that distinguishes a particular exception from another. For example, while there can be many `NullPointer` exceptions, there is only one `NullPointer` exception constructor, shared by all exceptions that bear its name. A particular exception constructor is declared as

```
exception Foo(int);
exn e = new Foo(1);
```

Here we declare some new exception constructor `Foo`, and then declare a `Foo` exception, storing it in variable `e`. To support deferring the identity of a exception constructor until dynamic link time, we allow exception constructors to be first-class. Continuing our example:

```
<int>exncon econ = Foo;
exn e2 = new econ(1);
```

Here, we declare a variable `econ` of type `<int>exncon`, meaning that it can hold any exception constructor that carries an `int` argument. We assign to `econ` the construct `Foo`. We can then use `econ` to build an exception, as we do for `e2`. We similarly allow fields in `switch case` statements to refer to variables containing exception constructors, rather than just exception constructor 'constants.' For example:

```
bool catch_exn(<void>exncon x, exn e) {
  try
    raise e;
  handle z {
    switch z {
    case x:
      { print_string("caught passed exception");
        return true; }
    ...
}
```

Here we define a function that takes as arguments an exception constructor `x` and an exception `e`. The function raises the exception `e`, catches it, and checks its constructor in the `switch`. The `x` in the `case` refers to the variable `x` passed in to the function.

Identifiers in Global Initializers

To simplify the construction of the indirection tables used for dynamic linking (see §6.2.1), we allow global initialization expressions to contain variables, as long as the variable's use is 'constant.' For example:

```
int a = 1;
int b = a; // illegal
```

This expression is illegal since `b`'s value depends on the value of `a`. However,

```
int a = 1;
int *(b) = &a; // legal

int x(int y) { return y+1; }
struct fnptr { int f(int); }
fnptr = new fnptr(x); // legal
```

This code is legal because the expressions `&a` and `x` (appearing in the constant expression `new fnptr(x)`) are constant, and can be calculated at compile-time.

Type Representations

TAL and Popcorn both employ a *type-erasure* semantics. In particular, while types are used in a first-class manner (particularly in polymorphic functions and data), they are entirely parametric, meaning that the particular identity of the type is not of concern, and therefore does not contribute to the runtime computation. As a result, first-class types can be safely eliminated from the final executable code. However, we may wish to use types *intensionally* [HM95], meaning that we wish to examine their identity to determine program behavior. In this way, so-called intensional polymorphism allows for computations on types. In this setting, types *cannot* be erased from the resulting executable, since they take part in the computation.

In this work, we use types intensionally in our approach to dynamic linking. To keep TAL's type-erasure semantics but still be able to compute with types at runtime, we introduce *type representations* into the normal term language. We defer discussion on the syntax and use of these representations until §5.4.1, where we explain how they are used by our dynamic linker.

Chapter 5

Dynamic Linking in TAL

Dynamic linking is the foundation of our approach to dynamic updating. For purposes of performance and robustness, we have opted to use verifiable native code as our implementation platform, for which no well-designed methodology for dynamic linking previously existed. For example, in the PCC Touchstone system, dynamic linking was ad-hoc, crafted to support loading extensions into a non-PCC OS kernel [NL96], while Special J [CLN$^+$00], a PCC system for Java, lacks dynamic linking support altogether as of this writing. Therefore, to implement dynamic updating in VNC based on dynamic linking, we need to first design dynamic linking facilities for verifiable native code and implement them for TAL.

Rather than do something 'quick and dirty' just to support updating, we wanted our implementation to stand on its own. Our approach was designed to meet three criteria:

1. **Flexibility**. While we are primarily concerned with dynamic linking as the foundation of dynamic updating, we prefer to design a dynamic linking approach that is general. We should be able to support typical source language linking entities, *e.g.*, Java classes, ML modules, or C object files; and their loading and linking operations. Furthermore, adding updateability should add little or no complexity to the basic approach.

2. **Security**. Type-safe dynamic linking has been proposed as a means to run untrusted code, since type-safe code is certain not to access information surreptitiously, such as by creating a pointer from an integer and dereferencing it. We must therefore take care in designing and implementing our approach so that it is secure. In particular, the type system we use must be sound, and the trusted computing base (TCB) of our implementation should be small.

   The fact that code is type-safe is only of value if the type system used to check the code is *sound*; that is, programs that are type-safe cannot be 'ill-behaved,' such as being able to forge a pointer. Therefore, the definition of the type system and the proof that well-typed programs always behave in a secure manner is of critical importance.

   The term 'trusted computing base' comes from security terminology, and in our case refers to the infrastructure that ensures loaded code is type-safe. A bug in the trusted computing base could lead to a security violation, since some code that is apparently safe is actually not so, and so may be able to exploit this weakness to, say, forge a pointer. Early implementations of Java were found to be insecure due to this sort of failure [Dea97]. To reduce the possibility of bugs, we prefer a small (and simple) trusted computing base. The result is improved confidence in the system's security and consequently its robustness.

By themselves, VNC systems like TAL employ sound type systems and have small TCB's. Since we are adding dynamic linking to VNC, we want to do so in a way that preserves the type system's soundness does not overly expand the TCB.

3. **Efficiency**. Dynamic linking should impose little or no overhead above statically-compiled programs, in terms of both space and time.

The goals for dynamic linking mirror our goals for dynamic updating, and thus in meeting them we set a firm foundation for our updating approach.

In this chapter, we present our dynamic linking framework for TAL, called TAL/Load; this is joint work was done with Stephanie Weirich and Karl Crary.[1] Our framework consists of several small additions to TAL that enable us to *program* dynamic linking facilities in a type-safe manner, rather than including them as a monolithic addition to the TCB. Our additions are simple enough that a formal proof of soundness is straightforward. Furthermore, our approach is not specific to TAL; it should be possible to implement it in other verifiable native code systems. Ours is the first complete framework for dynamic linking in verifiable native code.

The remainder of this chapter is organized as follows. We begin with informal definitions of both linking and dynamic linking, to make our discussion more concrete. Next, we motivate and outline TAL/Load, our framework for dynamic linking in TAL. Next, we formalize this framework and prove it sound. Finally, we describe our implementation in TAL. In the next chapter, we discuss how TAL/Load has been used at the core of our Popcorn implementation of DLopen [Lin95], a UNIX library that provides dynamic linking services to C programs. We also informally present how to program other linking approaches using TAL/Load. Performance data is presented for both dynamic linking and dynamic updating in Chapter 10.

5.1 Background

To understand how to build a dynamic linker for TAL, a low-level language, we need to understand what is typical of dynamic linkers in high-level languages. Knowing this, we can design lower-level primitives in TAL on which to map higher-level dynamic linking abstractions. We begin by describing how programs are linked statically, and then examine how dynamic linking changes the landscape. We ultimately break down how a typical dynamic linker is implemented using code for loading, linking, and symbol management.

5.1.1 Static Linking

Most non-trivial programs are constructed from one or more program *modules.* While the definition of a module differs with programming language, most often a module is a collection of *definitions* which map symbol names (or simply, *symbols*) to code fragments;

[1]As of this writing, Karl Crary is at Carnegie-Mellon University, and Stephanie Weirich is at Cornell University. A breakdown of each person's contributions is presented at the end of the next chapter.

the delineation of a module is typically a single source file. For example, consider the following C module, stored in the file `f.c`:

```
int d = 5;
int g(int x) {
  return x+d;
}
int f(int x) {
  return g(x)+1;
}
```

This module contains three definitions: symbol `d` is mapped to the constant 5, and symbols `g` and `f` are mapped to functions. All of the symbols mentioned in the module code refer to definitions in the module itself.

Programs can consist of more than one module, in which case some symbols will refer to definitions in other modules. For example, we could break up `f.c` into two files, `f.c` and `g.c`, shown in Figure 5.1.

`g.c`:

```
int d = 5;
int g(int x) {
  return x+d;
}
```

`f.c`:

```
extern int g(int);
int f(int x) {
  return g(x)+1;
}
```

Figure 5.1: Two C modules to be linked together.

Now the function `g` and the integer `d` are defined in the file `g.c`, while the function `f` is defined in `f.c`. As a result, the symbol `g` mentioned in `f` is *external* to `f.c`; we also say that module `f.c` *imports* the symbol `g`.

The modules `f.c` and `g.c` can be combined together into a single program by a process called *linking*. Given two or more modules as arguments, a program called a *linker* combines the modules' code and *resolves* any references to externally-defined symbols by matching those references with the appropriate definitions in other modules. In the case of `f.c` and `g.c`, the linker would resolve the reference to `g` in `f.c` with the definition of `g` in the file `g.c`. The linker will only resolve a module's imported symbols with symbols *exported* by other modules; that is, a module may have definitions that are not available to other modules during linking. For example, in C (and Popcorn), such definitions are prepended with the keyword `static`.

5.1.2 Dynamic Linking

Originally, the process of linking only occurred *statically* to construct programs from separately-compiled modules. That is, linking always occurred before program execution, with the requirement that all references be resolved before the program could run.

However, many programming environments now support *dynamic linking*, in which the program may invoke a dynamic linker at runtime to extend itself with new modules. When new modules are added, their imports are resolved with the exports of the modules in the running program.

A dynamic linker must perform three tasks: it must *load* the module into the memory of the running program; it must *link* the loaded module with the running program; and it must *manage* the program's symbols for use in future dynamic linkages. While all dynamic linkers implement these tasks, the the details differ. Consider each task more closely:

Loading Loading entails reading the module from disk or from the network, making some well-formedness checks, and mapping it into the program's address space. Well-formedness checking varies with the programming environment. For example, Java modules, called *classfiles*, are verified to respect certain safety properties, including type safety, before they are added to a program. By contrast, C object file loaders are only concerned only with the well-formedness of the object file metadata—no checks are made to assure that the code contained therein is type safe (or even well-formed).

Linking Linking entails resolving the imports of the loaded module with the exports of the running program. The linking process also varies with programming environment. In particular, there is the question of *when* imports are resolved, and the question of *how* dynamic linking is implemented.

Linking is typically performed at one of two times: load-time or on-demand. In the former case, all of the module's imports are resolved immediately after it is loaded; if a reference cannot be resolved then the linkage fails. In the latter case, imports are resolved just before the program references them. This amortizes the linking process over the program's execution and avoids linking symbols not needed by a particular run. The C dynamic linking interface on Unix systems, called DLopen [Lin95], allows both linking styles, while Java links classes on-demand.

Dynamic linking is typically implemented either via *indirection* or *code rewriting*. In the former case, external references are compiled to be indirected though a module-local table. Linking then consists of filling in the table. When using native code, some extra code fragments can be used to allow references to be resolved on-demand (*cf.* ELF [TISC95]), while a virtual machine architecture like the JVM can incorporate on-demand linking into its instruction semantics. Code rewriting links modules by rewriting the code so that references point directly to the appropriate external definitions. The ELF dynamic linking [TISC95] standard, often implemented in DLopen, uses the indirection approach (the indirection table is called a *global offset table*, or GOT), while the Linux kernel's loadable module facility (called *modutils*), uses the rewriting approach.

Symbol Management To link a loaded module into the running program requires that the addresses of the program's symbols be available to the dynamic linker. *How* these symbols are maintained (*i.e.* what kind of datastructure) and *which* symbols to use are questions of symbol management. Considering the latter question, there are many circumstances in which certain symbols should not be available during linking. As mentioned above, C definitions declared to be `static` should not be available when linking a loaded module since these symbols are considered local to the module in which they are defined. In addition, symbols may be precluded from linking for security concerns. In Objective Caml (OCaml) [Ler00], program modules define a 'safe' interface that is a subset of the actual module interface; only the safe interface is made available during dynamic linking. This approach is taken one step further in ALIEN [AHI+00, Ale98], in which each symbol's availability depends on the privilege of the loading code. There is even the possibility that symbols may be resolved with different values. For example, untrusted code could be linked against a version of `open` that only works for files in the `/tmp` directory, while trusted code is resolved with the normal `open` function.

5.2 TAL/Load

Because TAL serves as the target for high-level languages, dynamic linking in TAL must be general enough to accommodate the wide variety of dynamic linking approaches described above. We begin by considering a straightforward but flawed means of adding dynamic linking in TAL, to motivate our actual approach.

Consider defining a primitive, load_0, that dynamically loads and links a TAL module into the running program. Informally, load_0 might have the type:

$$\mathsf{load}_0 : \quad \forall\alpha : \mathsf{sig}.\, \mathtt{bytearray} \rightarrow \alpha\ \mathtt{option}$$

That is, load_0 takes two arguments: the expected *signature* of the loaded module, stored in the variable α, and the binary representation of the module, stored as a `bytearray`. The signature is a description of the module's contents, including the names and types of its functions, as well as the names and definitions of its user-defined types. load_0 parses the bytearray, checks it for well-formedness, and links any unresolved references in the module to their exported definitions in the running program. It compares the module's actual signature with the expected one α; if the signatures match, it returns the module to the caller. If any part of this process fails, load_0 returns *null* to signal an error.[2]

As an example, say we have some TAL module that corresponds to the file `f.c` that we presented in Figure 5.1 on page 40. This module compiled to TAL is stored in the file `f.tal`. If we have some program that wishes to dynamically link in `f.tal` and invoke its function `f`, the program could contain the following (Popcorn-like) code:

```
m = load0 ([sig f : int → int end], read_file("f.tal"));
if (m != null)
  return m.f(12);
else
  ... handle error ...
```

Here we call load_0 with the signature as its first argument, which indicates that the loaded module should contain a single function `f` that maps integers to integers. The second argument is a bytearray containing the contents of `f.tal`, obtained by some function

[2]The `option` type in SML usually defines a constructor `NONE` to indicate undefined values; we use the more Popcorn-like *null* in our exposition instead.

`read_file`. The module is loaded and linked, and then stored in the variable `m`; its reference to `g` is resolved with a value defined in the running program. If dynamic linking is successful, then `m` is not null, so we can invoke its function `f`. Otherwise, an error occurred and we have to take some action.

While reasonably simple and intuitive, there are a number of obstacles to implementing this approach.

1. We require a way to manipulate modules as data, so that they can be stored in variables when returned from load_0, giving modules 'first-class' status. In the context of a rich type system, first-class modules require a complicated formalization (*e.g.*, Lillibridge [Lil97]) with restrictions on expressiveness; as a result, most ML-variants (and TAL as well) do not permit modules to be manipulable as data, relegating them to 'second-class' status [HMM90, Ler94, MTHM97].

2. The signature argument to load_0 is essentially a type being used as data. Implementing types as data is typically done using a *type-passing* semantics, which requires that types have a runtime representation, but one that is not under the explicit control of the programmer. TAL prefers a *type-erasure* semantics whereby all of the typing annotations can be stripped away without affecting the program's computation; these two semantics are incompatible.

3. All of the code for loading, linking, and symbol management occurs as part of load_0. As a result, the system's flexibility is diminished, since the policy decisions concerning linking and symbol management are fixed. For example, load_0 performs all of its linking at load-time, precluding a Java-like semantics where linking is done on-demand. In addition, it provides no means to incorporate source-language or security policies concerning symbol management, such as precluding module-local or protected symbols from linking. Furthermore, the entire implementation of load_0 is trusted (since it is outside the TAL type system), reducing system security.

   To improve flexibility, we could parameterize load_0 to accommodate different styles of linking, or to break it into trusted component parts that closely map to common source-language operators—this approach is taken in TMAL [Dug00]. However, all dynamic linking functionality would still be within the TCB.

To allow dynamic linking operations to be more flexible and to reduce the additions to the TCB, we reduce the prominence of the load_0 primitive and make it part of a dynamic linking *framework* with which we can *program* source-level dynamic linking approaches. That is, rather than expect load to perform the tasks of loading, linking, and symbol management, we reduce the role of load to just loading, and allow linking and symbol management to be programmed in TAL itself, using features already present in TAL, along with a few carefully selected new features. The result is improved flexibility, since policy decisions concerning linking and symbol management can now be programmed using TAL. We also improve system security, since we only expand the TAL TCB with code to support loading.

We call our framework TAL/Load. We now describe this framework informally by explaining how it addresses the three problems mentioned above:

1. We avoid the use of first-class modules. First-class modules are theoretically problematic because modules may contain type definitions (referred to as *type components*) as well as function and data definitions (collectively referred to as value definitions). The difficulty arises because the meaning of a type component depends on the module that the type is defined in. That is, if M and N are arbitrary expressions of module type having a type component t, it is difficult at compile-time to determine if the type $M.t$ is equal to (is the same name as) $N.t$. The problem arises because we do not know the identities of types $M.t$ and $N.t$, and therefore must use their names (including the paths) to compare them. Type components are an instance of *named types* (*a.k.a.* branded types). Named types are also used to implement generative types (such as structs in C or datatypes in ML).

   TAL avoids the type component problem by making named types globally scoped. As a result, no two modules in a program may define a type having the same name, and therefore the identity of a type can always be known at compile-time, based on just its name. There is no need for load_0 to return a value of module type (which consists of both type and value definitions), so our new loading primitive, called load, returns a tuple containing the module's exported value definitions instead. Any exported type definitions are added to a global *program type interface* maintained by the loader for the running program. This interface is a list of currently defined types and their definitions used during type-checking to ensure that imported type definitions of modules loaded later are consistent with ones already defined. In essence, the loader is responsible for ensuring that no loaded code defines duplicate type names.

2. Rather than require a type-passing semantics for the type argument to load, we use an explicit representation of types as data, in the style of Crary *et al.* [CWM98]. We create special values, called *type representations*, that correspond one-to-one with the types they represent, and the relationship between the two is known by the type-checker. This allows type representations to participate in the proof of type-safety but still be under the explicit control of the programmer.

3. Since we expect linking and symbol management to be programmed in TAL, rather than fixed as part of load, we restrict load to load only those TAL modules that do not have any imported values; *i.e.* modules are required to be *closed* with respect to values. This ensures that load will not be responsible for linking TAL modules and managing the program's value symbols. On the other hand, TAL modules can import externally-defined type definitions, which are maintained in the program type interface, as described above.

   To support linking, we compile source-language modules that import values to be closed TAL modules; the idea is to encode the mechanisms needed to implement linking as TAL code. For example, we could compile source-level external references into local data 'cells' to ultimately store the external values. After the module is loaded, these cells are filled in appropriately by the dynamic linker, also written in TAL. In essence, this is the indirection approach to implementing linking, as described in the previous section. For example, consider once again the module `f.c` in Figure 5.1 on page 40. We can close this module as shown below:

`f.c`:

```
extern int g(int);
int f(int x) {
  return g(x)+1;
}
```

`f.c` compiled to be closed:

```
static int (*g)(int) = null;
int f(int x) {
  return g(x)+1;
}
```

Here we have translated the `extern` for `g` into a function pointer. Initially this value is set to null, but following dynamic loading, the linker will set it with the value of `g` as defined in the running program. In a type-safe language like Popcorn, null-checks will be inserted to ensure that `g` is not dereferenced by the function `f` if it has not been filled in.

For the dynamic linker to track the running program's symbols, we can program a type-safe symbol table in TAL. To do so, we use type representations, existential types [MP88] and a special checked_cast operator to implement type dynamics [ACPP91]. We go into greater detail about compiling for dynamic linking and how we implemented the symbol table in the next chapter.

For the remainder of this chapter we more carefully describe TAL/Load. In the next section, we present a formalization of the load primitive in a variant of polymorphic lambda-calculus that captures the relevant elements of TAL/Load. We prove that this calculus, which we call the load-calculus, is type-safe. In Section 5.4, we describe the implementation of TAL/Load in the TALx86 [MCG$^+$99] implementation of TAL. We show that our implementation adds little to the TALx86 trusted computing base. In particular, the majority of the functionality of load—unmarshalling and type-checking TAL object files—is *already* a part of the TALx86 TCB. Finally, we close with some discussion on how we could increase the flexibility of load with a simple extension.

5.3 The load-calculus

We designed the load-calculus to balance two tensions. We wanted it to be simple enough that a proof of soundness would not be overly tedious, but enough like TAL that a correspondence between the two, in terms of the intended result of type soundness, is believable. To balance these tensions, the load-calculus is essentially a variant of the well-studied polymorphic lambda calculus [Gir71, Rey74], for which proofs of soundness are well-known, but is formulated using an 'allocation-style' semantics. In this formulation (*cf.* [MH97]), a program's heap is explicitly considered, and thus programs more closely correspond to actual machine-language programs. Because TAL programs are by nature imperative, programs can alter values stored in the heap, essentially treating heap locations as reference cells, *e.g.* [Har94]. Programs also keep a program type environment, described informally in the previous section, for the purpose of modeling named types.

In this section, we present the load-calculus. We begin by describing an untyped version of the calculus, giving a flavor for the evaluation of programs, and showing how ill-formed programs can go 'wrong.' Next, we add types, with the intention that a well-typed cannot go 'wrong;' this is the calculus's key property of *type soundness*. Finally, we add named types to the formulation, to model type components found in TAL modules. The entire

$$
\boxed{\begin{array}{l} i \in \mathcal{Z} \\ L \in \mathsf{Labels} \\ x \in \mathsf{Vars} \end{array}}
\qquad
\begin{array}{llcl}
\textit{expressions} & e & ::= & i \mid x \mid \lambda x.e \mid e_1\ e_2 \\
 & & \mid & L \mid \texttt{ref}\ e \mid \texttt{assign}\ e_1\ e_2 \mid !e \\
 & & \mid & \mathsf{load}\ e_0\ e_1\ e_2 \\
\textit{values} & v & ::= & i \mid \lambda x.e \mid L \\
\textit{heaps} & H & ::= & \{L_1 = v_1, \ldots, L_n = v_n\} \\
\textit{programs} & P & ::= & (H, e)
\end{array}
$$

Figure 5.2: Untyped load-calculus Syntax

calculus and its proof of soundness is presented in Appendix A.[3] This work was originally done jointly with Stephanie Weirich from Cornell University [HW00], but the presentation here is completely new.

5.3.1 The Untyped load-calculus

We first present the syntax of the untyped load-calculus, and then describe how its programs evaluate, using an operational semantics.

Syntax

The syntax of the untyped load-calculus is shown in Figure 5.2. A program P consists of *heap* H and an *expression* to evaluate e. The heap models a program's memory, including its code and data, while the expression models its execution. Comparing load-calculus programs to typical, UNIX-like processes, the heap is equivalent to a process's code segment, static data segment, and runtime heap; while the expression represents the program counter and the stack.

Expressions can be divided into three classes. The first contains the standard, lambda-calculus expressions: variables (x), integers (i), abstractions ($\lambda x.e$), and applications ($e_1\ e_2$). The second class contains expressions relating to the program's heap (using the standard interface for reference cells, *e.g.* [Har94]): labels (L), allocation ($\texttt{ref}\ e$), assignment ($\texttt{assign}\ e_1\ e_2$), and dereference ($!e$). The third class contains the load expression used to perform dynamic linking.

A heap is represented as a finite map from labels L to values v. Values are a subset of expressions e consisting of integers (i), labels (L), and abstractions ($\lambda x.e$). Intuitively, a label L is an address in memory, pointing to either a function or some data. Labels are created either statically, as part of the initial program, or dynamically, though allocation. For example, in modeling the module `g.c` shown on page 40 in Figure 5.1, the initial program heap would be:

$$\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}$$

That is, the label `d` maps to the integer 5, and the label `g` maps to a function that adds its argument to the value stored at label `d` (the ! operator indicates that the label should

[3]The formalization in the appendix includes the use of a *type heap mask*, described in Section 5.5.

$\boxed{(H, e) \mapsto (H', e')}$

$$
\begin{array}{lcll}
(H, (\lambda x{:}\tau.e)\ v) & \mapsto & (H, e[v/x]) & \textit{(beta)} \\
\\
(H, \texttt{ref}\ v) & \mapsto & (H \uplus \{L = v\}, L) & \textit{(ref)} \\
\text{where } L \notin \texttt{dom}(H) & & & \\
\\
(H, !L) & \mapsto & (H, v) & \textit{(deref)} \\
\text{where } H(L) = v & & & \\
\\
(H, \texttt{assign}\ L\ v) & \mapsto & (H[L = v], v) & \textit{(assign)}
\end{array}
$$

$$
\frac{(H, e) \mapsto (H', e')}{\left\{\begin{array}{c}
(H, e\ e_2) \mapsto (H', e'\ e_2) \\
(H, v\ e) \mapsto (H', v\ e') \\
(H, \texttt{ref}\ e) \mapsto (H', \texttt{ref}\ e') \\
(H, !e) \mapsto (H', !e') \\
(H, \texttt{assign}\ e\ e_2) \mapsto (H', \texttt{assign}\ e'\ e_2) \\
(H, \texttt{assign}\ v\ e) \mapsto (H', \texttt{assign}\ v\ e')
\end{array}\right\}} \quad \textit{(congruence)}
$$

Figure 5.3: Operational rules for the untyped calculus, excluding load

be dereferenced to obtain the value stored there). The heap is also used to store values allocated by the program at runtime using the expression `ref` e; intuitively, this expression evaluates e and then stores the resulting value at a newly-allocated memory location. The contents of a memory location can be changed with `assign` .

Following convention, we consider expressions to be equivalent up to α-conversion of lambda-bound variables. Heap labels may be α-converted as well, following the intuition that addresses in a program may be relocated without affecting the program's correctness. To make the code examples more meaningful, we include additional operators, like addition on integers. The standard lambda-calculus is powerful enough to encode our shorthand changes (addition on integers can be encoded using Church numerals), so our result is not compromised.

Semantics

We define the operational semantics for the load-calculus using a deterministic, one-step reduction operator $\mapsto$, following a call-by-value discipline. The rules, not including load, are shown in Figure 5.3. The first four rules are *computation* rules, which define how the program evaluates by rewriting, while the remaining are *congruence* rules, which define the order of evaluation to be left-to-right, call-by-value.

The *beta* rule performs function application via substitution; we define $e[e'/x]$ as the capture-avoiding substitution of the term e' for each occurrence of the variable x in the term e. The next three rules operate on the heap. Notationally, we write $H(L)$ to denote v in the heap $H = \{\ldots, L = v, \ldots\}$. For the heap $H = \{L_1 = v_1, \ldots, L_n = v_n\}$, $\texttt{dom}(H)$ refers to the set $\{L_1, \ldots, L_n\}$ and $\texttt{rng}(H) = \{v_1, \ldots, v_n\}$. If $H = \{\ldots, L = v, \ldots\}$, then let $H[L = v']$ be the heap $\{\ldots, L = v', \ldots\}$; this operation is undefined if $L \notin H$. The *ref* rule allocates a unique label L in the heap and stores the value v there. The *deref* rule extracts the value v mapped to by label L in the heap. The *assign* rule overwrites the existing mapping for label L in the heap with one from L to v.

As a simple example, consider the evaluation of the following program, based on our translation of the file `g.c` (see Figure 5.1) above. This program invokes the function `g` with the argument 4, ultimately returning 9:

$$\begin{array}{lll}
(\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}, (!\texttt{g}\ 4)) & \mapsto & (\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}, ((\lambda x.(x + !\texttt{d}))\ 4)) \\
 & \mapsto & (\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}, (4 + !\texttt{d})) \\
 & \mapsto & (\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}, (4 + 5)) \\
 & \mapsto & (\{\texttt{d} = 5; \texttt{g} = \lambda x.(x + !\texttt{d})\}, 9)
\end{array}$$

As another example, the following program starts with an empty heap, allocates a label to store the integer 4, and then increments it by 1:

$$\begin{array}{lll}
(\{\},\ (\lambda x.\,\texttt{assign}\ \ x(!x + 1))\ (\texttt{ref}\ 4)) & \mapsto & (\{L_1 = 4\},\ (\lambda x.\,\texttt{assign}\ x\ (!x + 1))\ L_1) \\
 & \mapsto & (\{L_1 = 4\},\ \texttt{assign}\, L_1\ (!L_1 + 1)) \\
 & \mapsto & (\{L_1 = 4\},\ \texttt{assign}\, L_1\ (4 + 1)) \\
 & \mapsto & (\{L_1 = 4\},\ \texttt{assign}\, L_1\ 5) \\
 & \mapsto & (\{L_1 = 5\},\ 5)
\end{array}$$

The running program can load new code into itself using the load primitive. Loaded code has the form (H, e), which is the same as a normal program, except that e should be thought of as the loaded code's *initialization expression*. Following the semantics for load in TAL/Load, described in §5.2, we would expect the initialization expression to return the values defined in the program. Informally, loading code with load consists of 1) merging the loaded code's heap with that of the running program, 2) executing the loaded code's initialization expression, and 3) continuing computation in the running program using the result of initialization. If the merge operation fails, then the initialization expression is not invoked, and the running program continues on an alternate path.

The crux of the load operation is the merging of the program heap and the loaded code's heap. Heap merging is disjoint union, defined formally below.

Definition 5.3.1 (Heap Merge)

$$\frac{H_1 \mid H_2}{H_1\, \texttt{merge}\, H_2 \Rightarrow H_3}\ (H_3 = H_1 \uplus H_2)$$

where we define $H_1 \mid H_2$ $\texttt{dom}(H_1)$ *and* $\texttt{dom}(H_2)$ *are disjoint*
$H_1 \uplus H_2$ *Union of disjoint maps, defined if* $H_1 \mid H_2$

$\boxed{(H, e) \mapsto (H', e')}$

$$\frac{H \ \texttt{merge}\ H_i \Rightarrow H'}{(H, \mathsf{load}\ i\ e_s\ e_f) \mapsto (H', e_1\ e_i)} \left(\hat{i} = (H_i, e_i)\right) \qquad (\mathit{load} - \mathit{success})$$

$$(H, \mathsf{load}\ i\ e_s\ e_f) \mapsto (H, e_f) \qquad (\mathit{load} - \mathit{failure})$$
otherwise

$$\frac{(H, e) \mapsto (H', e')}{(H, \mathsf{load}\ e\ e_s\ e_f) \mapsto (H', \mathsf{load}\ e'\ e_s\ e_f)} \qquad (\mathit{congruence})$$

Figure 5.4: Rules for load in the untyped calculus

Because we permit α-conversion of heap labels, if the condition $H_1 \mid H_2$ is not met, we can α-convert one of the heaps. This is because loaded programs are expected to be *closed*, meaning that all labels mentioned therein (whether in the heap or the expression part), refer to labels defined in the heap, and thus the program is completely *relocatable.* This expectation is in contrast to typical formulations of heap linking (*e.g.* MTAL [GM99], program fragments [Car97], TMAL [Dug00], *etc.*); these systems assume that free labels refer to (and will be resolved with) definitions in programs to be linked against, and thus the names of the labels have meaning. In our case, we assume any sort of linking will take place in the term language, as we sketched in §5.2 (and show in more detail in §5.3.3).

The formal rules for load are shown in Figure 5.4. The first argument i to load is the binary representation of a program, represented as an integer. Loading begins by converting the binary representation i into some program (H_i, e_i); we use $\hat{\cdot}$ as some function that maps integer arguments to programs, modeling a filesystem. The second two arguments to load are the success and failure expressions. If $H \ \texttt{merge}\ H_i \Rightarrow H'$, then the heap of the running program H can be merged with the heap of the loaded code H_i, then we use the $\mathit{load} - \mathit{success}$ rule, in which the success expression e_s is applied to the loaded program's initialization expression e_i and execution continues with the merged heap H'; otherwise the expression e_f is used (*i.e.*, when using $\mathit{load} - \mathit{failure}$) with the original program heap H. In the untyped calculus, a failure expression is not really necessary since load can never really fail; however, when we add types, a number of potential failure conditions arise.

As an example, consider the following. The running program loads some code that defines a single function g that adds 1 to its argument; the initialization expression returns g. After loading the program, the running program invokes the function g on the value d, defined in its own heap.

$$\begin{array}{lll}
(\{\mathtt{d} = 5\},\ \mathsf{load}\ i\ (\lambda f.f\ !\mathtt{d})\ 0) & & \text{where } \hat{i} \ = \ (\{\mathtt{g} = \lambda x.x+1\}, !\mathtt{g}) \\
& \mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x+1\},\ (\lambda f.f\ !\mathtt{d})\ !\mathtt{g}) \\
& \mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x+1\},\ (\lambda f.f\ !\mathtt{d})\ \lambda x.x+1) \\
& \mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x+1\},\ (\lambda x.x+1)\ !\mathtt{d})
\end{array}$$

$$\boxed{\alpha \in \mathsf{TypeVars}}$$

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \texttt{int} \mid \tau \to \tau \mid \tau\ \texttt{ref} \mid \alpha \mid \forall\alpha.\tau \\
\textit{heap types} & \Phi & ::= & \{L_1 : \tau_1, \ldots, L_n : \tau_n\} \\
\\
\textit{expressions} & e & ::= & \ldots\ \lambda x : \tau.e \mid \Lambda\alpha.e \mid e[\tau] \\
 & & \mid & \mathsf{load}[\tau]\ e_0\ e_1\ e_2 \\
\textit{values} & v & ::= & \ldots\ \lambda x : \tau.e \mid \Lambda\alpha.e \\
\\
\textit{type contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha \\
\textit{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}$$

Figure 5.5: Typed load-calculus syntax, minus named types (changes from Figure 5.2)

$$\begin{array}{ll}
\mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x + 1\},\ (\lambda x.x + 1)\ 5) \\
\mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x + 1\},\ 5 + 1) \\
\mapsto & (\{\mathtt{d} = 5, \mathtt{g} = \lambda x.x + 1\},\ 6)
\end{array}$$

In the calculus, well-defined programs either evaluate to *answers*, in which the expression part is a value (thus having the form (H, v)), or they diverge (*i.e.* never terminate). Ill-defined programs are ones in which the expression part of the program is not a value, but there is no possible evaluation rule to apply. Sometimes such programs are termed *stuck programs* (*cf.* [MH97]). Though we will not attempt to identify all of the syntactic forms of stuck programs, the intuition is that the program has been incorrectly constructed. For example, the following program is stuck:

$$(\{\}, !(\lambda x.x))$$

No rule from Figure 5.3 can be used to further evaluate this program, and $!(\lambda x.x)$ is not a value. The dereference operator expects its argument to be a label from the heap, but here its argument is an abstraction.

In the next subsection, we add types to the calculus. The type system is designed so that well-typed programs never become stuck, and thus (in the real world) never crash.

5.3.2 Adding Types

Now we add types to the calculus presented thus far. Because TAL/Load supports type components in its modules, we will further extend the calculus presented here to support *named types*, but we defer doing so until the next subsection.

The syntax of the calculus modified to include types is shown in Figure 5.5. If not shown in the figure, the syntax is the same as in Figure 5.2. Types τ include the integer type `int`, arrow types $\tau \to \tau$, reference cell types (*i.e.* heap label types) τ `ref`, type variables α, and polymorphic types $\forall\alpha.\tau$. Heap types Φ map heap labels to types τ. Both the abstraction and load expressions have been decorated with types, and we have added expression forms to support parametric polymorphism: type abstraction ($\Lambda\alpha.e$) and type

$\boxed{\Delta \vdash \tau}$

$$\Delta \vdash \texttt{int} \qquad \frac{\alpha \in \Delta}{\Delta \vdash \alpha}$$

$$\frac{\Delta \vdash \tau' \qquad \Delta \vdash \tau}{\Delta \vdash \tau' \to \tau} \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \texttt{ref}\ \tau} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha . \tau}\ (\alpha \notin \Delta)$$

$\boxed{\vdash \Phi}$

$$\frac{\cdot \vdash \tau \quad \text{(for each } \tau \in \texttt{rng}(\Phi))}{\vdash \Phi}$$

Figure 5.6: Well-formedness for types and heap types

application ($e[\tau]$). Values now include type abstractions. To support type-checking, we define type contexts Δ as lists of type variables, and contexts Γ as lists of mappings from variables x to types τ.

Static Semantics

The typed calculus defines judgments to assert that a program is well-formed. Well-formedness is defined inductively. That is, a program is well-formed if its heap H and its expression e are well-formed. Heaps are well-formed if they may be given some well-formed heap type Φ, and similarly, expressions are well-formed if they may be given some well-formed type τ.

The judgments for type and heap type well-formedness are shown in Figure 5.6. Types are checked for well-formedness in relation to a type variable context Δ. This context is used to make sure a type variable α is properly quantified. A heap type Φ is well-formed if all of the types mentioned in its range are well-formed.

The judgments for expressions, heaps, and programs are shown in Figure 5.7. Most of expression typing rules are standard; noteworthy is the rule for load. As mentioned earlier, the first term argument, which is mapped at runtime to a program, must have type `int`. The type argument τ' indicates the expected type of the loaded program's initialization expression. The second term argument is the 'success-expression' which is applied to the loaded code's initialization expression, so it must take an argument of type τ', returning a result of type τ. The final term argument is the 'failure-expression' which is executed if loading fails; its type must match the return type τ of the success condition so that the overall type of the load expression will be τ. A heap H is well-formed if the values in its range have the type indicated by the given heap type, and a program (H, e) is well-formed if its heap and expression are well-formed.

Dynamic Semantics

The operational semantics of the typed calculus is the same as that of the untyped calculus (see Figures 5.3 and 5.4) with two exceptions. First, there is an additional evaluation rule

$\boxed{\Phi; \Delta; \Gamma \vdash e : \tau}$

$$\frac{\begin{array}{c}\Phi; \Delta; \Gamma \vdash e_1 : \texttt{int} \\ \Phi; \Delta; \Gamma \vdash e_2 : \tau' \to \tau \\ \Phi; \Delta; \Gamma \vdash e_3 : \tau\end{array}}{\Phi; \Delta; \Gamma \vdash \mathsf{load}[\tau']\ e_1\ e_2\ e_3 : \tau}$$

$$\Phi; \Delta; \Gamma \vdash i : \texttt{int} \qquad \Phi; \Delta; \Gamma \vdash x : \Gamma(x) \qquad \Phi; \Delta; \Gamma \vdash L : \Phi(L)\ \texttt{ref}$$

$$\frac{\Phi; \Delta; \Gamma, x{:}\tau' \vdash e : \tau \qquad \Delta \vdash \tau'}{\Phi; \Delta; \Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau} \qquad \frac{\Phi; \Delta; \Gamma \vdash e_1 : \tau' \to \tau \qquad \Phi; \Delta; \Gamma \vdash e_2 : \tau'}{\Phi; \Delta; \Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Phi; \Delta, \alpha; \Gamma \vdash e : \tau}{\Phi; \Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\Phi; \Delta; \Gamma \vdash e : \forall\alpha.\tau \qquad \Delta \vdash \tau'}{\Phi; \Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

$$\frac{\Phi; \Delta; \Gamma \vdash e : \tau}{\Phi; \Delta; \Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}} \qquad \frac{\Phi; \Delta; \Gamma \vdash e : \tau\ \texttt{ref}}{\Phi; \Delta; \Gamma \vdash !e : \tau} \qquad \frac{\begin{array}{c}\Phi; \Delta; \Gamma \vdash e_1 : \tau\ \texttt{ref} \\ \Phi; \Delta; \Gamma \vdash e_2 : \tau\end{array}}{\Phi; \Delta; \Gamma \vdash \texttt{assign}\ e_1\ e_2 : \tau}$$

$\boxed{\vdash H : \Phi}$

$$\frac{\Phi; \cdot; \cdot \vdash H(L) : \Phi(L) \quad (\text{for each } L \in \texttt{dom}(H))}{\vdash H : \Phi}$$

$\boxed{\vdash (H, e) : \tau}$

$$\frac{\vdash \Phi \qquad \vdash H : \Phi \qquad \Phi; \cdot; \cdot \vdash e : \tau}{\vdash (H, e) : \tau}$$

Figure 5.7: Well-formedness for expressions, heaps, and programs

for type application (which is standard):

$$(H, (\Lambda\alpha.e)[\tau]) \mapsto (H, e[\tau/\alpha]) \quad (\textit{tapp})$$

That is, a type application substitutes the type τ for every occurrence of type variable α in the body e of the abstraction. Second, the *load-success* rule changes to verify that the loaded program is well-formed:

$$\frac{\begin{array}{c}\vdash (H_i, e_i) : \tau \\ H\ \texttt{merge}\ H_i \Rightarrow H'\end{array}}{(H, \mathsf{load}[\tau]\ i\ e_1\ e_2) \mapsto (H', e_1\ e)} \left(\hat{i} = (H_i, e_i)\right) \quad \textit{(load-success)}$$

As before, heap merging $H\,\texttt{merge}\,H_i \Rightarrow H'$ must be well-defined, but in addition the loaded program must be well-formed, $\vdash (H_i, e_i) : \tau$, whose type τ matches the type argument passed to load. As a result of this change, we require a type-passing semantics because the type argument passed to load is used at runtime.[4]

5.3.3 Adding Named Types

So far, the calculus that we have considered does not define programs (*i.e.* modules) as having type components. In this subsection, we complete the formulation of the calculus by adding a *type environment* to our notion of program, which allows for the definition of named types. We begin by motivating our approach to named types, and then present the additions to the calculus to support them.

Motivation

The load primitive forbids the loading of programs with free variables in the heap; one interpretation of linking would allow such programs, and would resolve these undefined references with definitions in the program's heap. Instead, we expect that the source-level modules with external references will be compiled to closed TAL modules, and the process of linking at the source module level will be reflected in the compiled TAL code.

As we briefly outlined in the previous section, closing a module by compilation is fairly simple. For example, consider the following SML module, perhaps forming part of an I/O library, that supports the opening and reading of text files:

```
structure TextIO =
struct
  type instream = int
  val openIn : string -> instream = ...
  val inputLine : instream -> string = ...
  ...
end
```

This module consists of the value definitions `openIn`, `inputLine`, and maybe others, as well as the type component `instream`, implemented as an integer. A client of this module might be something like:

```
fun doit () =
  let val h = TextIO.openIn "myfile.txt" in
    TextIO.inputLine h
  end
```

Say we wanted to dynamically link this code into a program that uses the `TextIO` module. We need to compile it so that it no longer makes reference to the externally defined `TextIO` module. One way to do this is to convert externally referenced values into locally defined references to values:

[4]As mentioned in the last section, we are able to use type-erasure semantics in TAL/Load by introducing term representations for types, in the style of Crary *et al.* [CWM98].

```
val TextIO_openIn : (string -> int) option ref = ref NONE
val TextIO_inputLine : (int -> string) option ref = ref NONE
fun doit () =
  let val h = getOpt (!TextIO_openIn)
              "myfile.txt" in
    getOpt (!TextIO_inputLine) h
  end
```

We initially fill the reference with NONE, indicating it has no value, and when the module is dynamically loaded, the reference is filled in with the proper value.

However, we run into difficulty when we have externally defined values of *named type*. Named types are noteworthy because they are only considered equal if their names match, regardless of whether their implementations do. In particular, consider the following (SML-like) code sequence:

```
named type t1 = int
named type t2 = int
fun bad_eq x:t1 y:t2 =
  x = y
```

The code for the function bad_eq would fail to type-check because the values x and y have different named type, even though they both are implemented as integers.

To see the problem with closing a module that has values of named type, consider if TextIO wished to hold the type instream abstract. As a result, we cannot replace the type with its definition int, as we did above. Instead, our attempt to close the client code as before would result in:

```
val TextIO_openIn : (string -> TextIO.instream) option ref = ref NONE
val TextIO_inputLine :
  (TextIO.instream -> string) option ref = ref NONE
```

While external references to values have been eliminated, we still have the external references to the type TextIO.instream. We cannot easily create a 'hole' for these type references like we did for values because type equality needs to be checked at load-time when load checks the loaded code for well-formedness. Because named types must be known by the type-checker, our solution to this problem is to extend our notion of program to include a *type interface* which notes all of the named types, and their definitions, used in the program. This is possible because, unlike the SML module system, the TAL module system uses a global type namespace, meaning that a given type name can only be defined once in the whole program. In SML, types with the same name are differentiated by the module they are defined in.

For the remainder of this subsection, we present the syntax and semantics of the load-calculus having support for named types.

Syntax

The extensions to the typed calculus syntax to support named types are shown in Figure 5.8. Programs are extended to include a type interface Θ of the form (X_I, X_E), which

	types	τ	$::=$	$...\ n$
	type environments	X	$::=$	$\{n_1 = \chi_1, \ldots, n_n = \chi_n\}$
	type environment values	χ	$::=$	$\top \mid \tau$
	type interfaces	Θ	$::=$	(X_I, X_H)
	expressions	e	$::=$	$...\ \texttt{reveal}\ e \mid \texttt{hide}_n\ e$
	values	v	$::=$	$...\ \texttt{hide}_n\ v$
$\boxed{n \in \mathsf{TypeNames}}$	*programs*	P	$::=$	(Θ, H, e)

Figure 5.8: load-calculus syntax including support for named types

is a pair of *type environments* that map type names n to their implementations; types τ are extended to include type names. X_I mentions the named types imported from other modules, and X_E mentions named types defined by (exported from) this one. For example, the type interface of the SML-like program fragment above that defines types `t1` and `t2`, would be:

$$(\{\}, \{\texttt{t1} = \texttt{int}, \texttt{t2} = \texttt{int}\}$$

In this case, the type names `t1` and `t2` map to types τ (`int`). However, type names can also map to $\top$ to implement type abstraction. For example, to make the named type `instream` abstract, the type interface of the client code above (defining the function `doit`) would be:

$$(\{\texttt{instream} = \top\}, \{\})$$

and the interface for `TextIO` would be the reverse:

$$(\{\}, \{\texttt{instream} = \top\})$$

Values of named type are considered isomorphic to the values of the named type's definition; we use the coercions `hide` and `reveal` to witness the isomorphism. In the example above, to convert the value 1 to have type `t1`, we would do $\texttt{hide}_{\texttt{t1}}\ 1$. Converting it back to an integer would simply require a `reveal`; *i.e.* $\texttt{reveal}\ (\texttt{hide}_{\texttt{t1}}\ 1)$. A `reveal` is not permitted if the named type is abstract, disallowing the code from 'looking' at the value; we formalize this case in the static semantics, below.

Static Semantics

The well-formedness judgments must be adjusted to account for the type environment; the judgments from Figures 5.6 and 5.7 are shown in Figure 5.9. There is one additional judgment, $\vdash X$, for type environment well-formedness. For all of the old judgments, well-formedness now additionally requires a type environment in its context.

For types, the judgment changes from $\Delta \vdash \tau$ to $X; \Delta \vdash \tau$. All of the type well-formedness rules from Figure 5.6 are the same, except that we add a type environment to the context; there is also the additional rule for named types which states that a named

$\boxed{X; \Delta \vdash \tau}$

$$\frac{n \in \mathtt{dom}(X)}{X; \Delta \vdash n}$$

$\boxed{\vdash X}$

$$\frac{X; \cdot \vdash \tau \quad (\text{for each } \tau \in \mathtt{rng}(X))}{\vdash X}$$

$\boxed{X \vdash \Phi}$

$$\frac{X; \cdot \vdash \tau \quad (\text{for each } \tau \in \mathtt{rng}(\Phi))}{X \vdash \Phi}$$

$\boxed{X \vdash H : \Phi}$

$$\frac{X; \Phi; \cdot; \cdot \vdash H(L) : \Phi(L) \quad (\text{for each } L \in \mathtt{dom}(H))}{X \vdash H : \Phi}$$

$\boxed{X; \Phi; \Delta; \Gamma \vdash e : \tau}$

$$\frac{X; \Phi; \Delta; \Gamma \vdash e : n}{X; \Phi; \Delta; \Gamma \vdash \mathtt{reveal}\ e : \tau}\ (X(n) = \tau) \qquad \frac{X; \Phi; \Delta; \Gamma \vdash e : \tau}{X; \Phi; \Delta; \Gamma \vdash \mathtt{hide}_n\ e : n}\ (X(n) = \tau)$$

$\boxed{X_P \vdash (\Theta, H, e) : \tau}$

$$\frac{\begin{array}{cc} \vdash X_I \uplus X_H & X_I \uplus X_H \vdash \Phi \\ X_I \uplus X_H \vdash H : \Phi & X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau \end{array}}{X_P \vdash ((X_I, X_H), H, e) : \tau}\ (X_H \mid X_P)$$

Figure 5.9: Additional and/or modified rules defining well-formedness for types, heap types, expressions, heaps, and programs

type n is well-formed if it is mentioned in the type environment. The change to this judgment is reflected into the judgments for heap and heap type well-formedness. The new judgment for type environment well-formedness states that a type environment is well-formed if all of the types mentioned in its range are well-formed. Named types may be mutually recursive, but a well-formed type environment must be closed; all of the type names appearing in its range must appear in its domain.

For expressions, all of the judgment rules from Figure 5.7 are the same, except that a X is added to the context. There are two new rules for named types, one for $\mathtt{hide}_n$ and the other for `reveal`. We use `reveal` to coerce an expression e having some named type n. The result has type τ, where n maps to τ in the type environment X. We use $\mathtt{hide}_n$ to coerce an expression e to named type n; if e has type τ then the type environment X must map n to τ. The semantics allows for named types to be opaque (abstract). In particular, the expression $\mathtt{reveal}\ e : \tau$ is only well-typed if $X(n) = \tau$. To make n abstract, we set $X(n)$ to $\top$, forbidding the coercion to the implementation type. In practice, label n is made abstract to loaded code by mapping it to $\top$ in the type heap mask X during loading.

Dynamic Semantics

The operational semantics changes only slightly from what we have presented so far. First, all programs are extended to include a type interface. Except for load, each of the existing operational rules only in that Θ is added to the definition of the program. For example, the *beta* rule becomes

$$(\Theta, H, (\lambda x{:}\tau.e)\ v) \mapsto (\Theta, H, e[v/x]) \quad (\textit{beta})$$

Second, we add evaluation rules for named types, which also make use of the type interface:

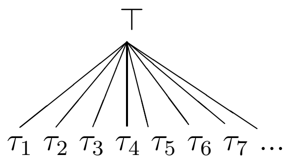
$$(\Theta, H, \texttt{reveal}(\texttt{hide}_n\ v)) \quad \mapsto \quad (\Theta, H, v) \qquad\qquad (\textit{reveal})$$

$$\frac{(\Theta, H, e) \mapsto (\Theta', H', e')}{\left\{ \begin{array}{c} (\Theta, H, \texttt{hide}_n\ e) \mapsto (\Theta', H', \texttt{hide}_n\ e') \\ (\Theta, H, \texttt{reveal}\ e) \mapsto (\Theta', H', \texttt{reveal}\ e') \end{array} \right\}} \quad (\textit{congruence})$$

In essence, *unroll* guarantees that a value that has been coerced to a named type cannot be examined until it has been revealed. Though we do not prove as much here, for well-typed programs, the `hide` and `reveal` keywords can be erased without affecting the program's behavior; we do this in our implementation.

The last change is that the semantics of load must accommodate type interfaces. We must additionally merge the type interfaces of the loaded code and the running program, on the condition that doing so is well-typed. The type interface merging operator `link` must make sure that the named types defined in loaded code mesh appropriately with named types already provided by the program. The type interface merging operator `link` is defined in terms of some operators and relations on type environments and type environment values, shown in Figure 5.10. We write $X(n)$ to denote χ in the heap $X = \{\ldots, n = \chi, \ldots\}$. For the heap $X = \{n_1 = \chi_1, \ldots, n_n = \chi_n\}$, $\texttt{dom}(X)$ refers to the set $\{n_1, \ldots, n_n\}$ and $\texttt{rng}(X) = \{\chi_1, \ldots, \chi_n\}$.

The type environment value operations are derived from the following poset:



$\top$
τ_1 τ_2 τ_3 τ_4 τ_5 τ_6 τ_7 ...

That is, the $\sqcap$ and $\leq$ operators have their standard definitions given this relationship between type environment values, where the list of types $\tau_1, \tau_2, \ldots$ represents the enumeration of all possible types τ. The type environment operator $\oplus$ extends the $\sqcap$ operator to type environments, operating on members with the same type name; $\precsim$ is the extension of $\leq$, and $\diamond$ is the extension of $\diamond$.

Type interface linking is defined formally in Definition 5.3.2. Stated informally, linking two type interfaces (X_I^1, X_H^1) and (X_I^2, X_H^2) results in combining the two export environments $(X_H^1 \uplus X_H^2)$, and resolving any type names in the imports of both interfaces with the

Type Environment Values χ

Operators		
meet	$\chi_1 \sqcap \chi_2$	$\top \sqcap \chi = \chi \quad \chi \sqcap \top = \chi \quad \chi \sqcap \chi = \chi$
approximates	$\chi_1 \leq \chi_2$	$\chi \leq \top \quad \chi \leq \chi$
Relations		
compatible	$\chi_1 \diamond \chi_2$	$\chi_1 \leq \chi_2$ or $\chi_2 \leq \chi_1$

Type Environments X

Operators		
restriction	$X_1 - X_2$	X_1 restricted to labels not in $\texttt{dom}(X_2)$
disjoint union	$X_1 \uplus X_2$	Union of disjoint maps, defined if $X_1 \mid X_2$
merge	$X_1 \oplus X_2$	Union of compatible maps (defined if $X_1 \diamond X_2$), maps $n \in \texttt{dom}(X_1) \cap \texttt{dom}(X_2)$ to $X_1(n) \sqcap X_2(n)$
Relations		
disjoint	$X_1 \mid X_2$	$\texttt{dom}(X_1)$ and $\texttt{dom}(X_2)$ are disjoint
link compatible	$X_1 \precsim X_2$	For n in $\texttt{dom}(X_1) \cap \texttt{dom}(X_2), X_1(n) \leq X_2(n)$
compatible	$X_1 \diamond X_2$	For n in $\texttt{dom}(X_1) \cap \texttt{dom}(X_2), X_1(n) \diamond X_2(n)$

Figure 5.10: Type environments and type environment values: operators and relations

same names defined in the exports. We can do this by merging the imports ($X_I^1 \oplus X_I^2$), and then subtracting any labels mentioned in the combined exports: $(X_I^1 \oplus X_I^2) - (X_H^1 \uplus X_H^2)$. These operations are only well-formed if (1) the two export environments are disjoint (*i.e* $X_H^1 \mid X_H^2$), and (2) if the two imports are compatible (*i.e.* $X_I^1 \diamond X_I^2$). We also add the restrictions $X_H^1 \precsim X_I^2$ and $X_H^2 \precsim X_I^1$. This prevents the import environment from one interface from replacing $\top$ mapped to by some name in the export environment of the other interface. Doing so would break the abstraction enforced by `hide` and `reveal`.

Definition 5.3.2 (Type Interface Linking)

$$\frac{X_I^1 \diamond X_I^2 \qquad X_H^1 \precsim X_I^2 \qquad X_H^2 \precsim X_I^1 \qquad X_H^1 \mid X_H^2}{(X_I^1, X_H^1)\,\texttt{link}\,(X_I^2, X_H^2) \Rightarrow (X_I^3, X_H^3)} \left(\begin{array}{c} X_H^3 = X_H^1 \uplus X_H^2 \\ X_I^3 = ((X_I^1 \oplus X_I^2) - X_H^3) \end{array} \right)$$

We do not require that all of a module's type imports be resolved during linking; that is, the running program's import type environment is not required to be empty.[5] Not requiring resolved imports facilitates loading modules one at a time that have mutually-recursive type definitions. In particular, each loaded module indicates in its imports the definitions of types it expects from other modules to be loaded (akin to an `extern` declaration for types). When the next module is loaded, its exports are confirmed to match the previously loaded module's imports.

[5]To implement this relaxation requires a uniform representation of named types; in our case, all named types are pointer-types.

Unlike heap labels, α-conversion is not permitted on type names; this is because type names participate in linking, so that each particular name has meaning. Furthermore, we assume that type environments X_H export all of their values. With a slight modification we could follow the approach of MTAL [GM99] and designate a subset of the domain of X_H as exports, where the rest of the heap consists of *local type names.* These local names would be allowed to α-vary during linking, since they are limited to the scope to the defining module and not the whole program. We take this approach in our implementation.

The operational rule for *load-success* now becomes:

$$\frac{\begin{array}{c} X_H \vdash \hat{i} : \tau \\ H\,\texttt{merge}\,H_i \Rightarrow H' \\ (X_I, X_H)\,\texttt{link}\,(X_I^i, X_H^i) \Rightarrow (X_I', X_H') \end{array}}{\begin{array}{l} ((X_I, X_H), H, \mathsf{load}[\tau]\ i\ e_2\ e_3) \mapsto \\ ((X_I', X_H'), H', e_2\ e_i) \end{array}} \left(\hat{i} = ((X_I^i, X_H^i), H_i, e_i)\right) \quad \textit{(load-success)}$$

The only difference from the rule in the previous subsection is the condition on type interfaces. That is, we must be able to link the type interface of the running program with that of the loaded code: $(X_I, X_H)\,\texttt{link}\,(X_I^i, X_H^i) \Rightarrow (X_I', X_H')$.

5.3.4 Properties of the load-calculus

The important formal property of this system is that it is *type-safe* (this property is also called *type-soundness*). In particular, if a program is well-typed, it will execute in a well-defined fashion indefinitely, or until it completes with a particular value; a well-typed program therefore cannot get 'stuck.' Formally stated:

Theorem 5.3.3 (Type Safety) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto^* (\Theta', H', e')$ *then* (Θ', H', e') *then either* e' *is a value or can be further reduced by some rule of the operational semantics.*

Note that $\mapsto^*$ is the multi-step reduction relation, indicating zero or more applications of the single-step relation $\mapsto$. Type safety is proven using the standard technique of showing *subject reduction* and *progress*:

Lemma 5.3.4 (Subject Reduction) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto (\Theta', H', e')$ *then* $\vdash (\Theta', H', e') : \tau$

Lemma 5.3.5 (Progress) *If* $\vdash (\Theta, H, e) : \tau$ *and* e *is not a value, then there exists a* (Θ', H', e') *such that* $(\Theta, H, e) \mapsto (\Theta', H', e')$.

Stated informally, subject reduction indicates that if a given program has a type τ, and it can take (at least) one reduction step, then the resulting program, after applying the reduction rule, still has type τ. Progress indicates that if a well-typed program cannot take an evaluation step, then it must be a value having type τ. All of the relevant proofs are presented in Appendix A.

5.4 Implementation

We implemented TAL/Load in TALx86 as follows. First, we added a trusted library to the base TALx86 implementation that implements the load primitive. Second, to preserve TAL's type-erasure semantics and to allow computing with types, we added term representations for types to TAL. Finally, we added a checked_cast operator to permit the construction of a type-safe dynamic symbol table in TAL; how we use checked_cast is described in the next chapter. For the remainder of this section, we present more detail on these three additions to TALx86.

5.4.1 Passing Types at Runtime

Recall that load requires that a type argument τ' be examined at runtime to make sure it matches the type of the the loaded code. However, TAL maintains a *type-erasure* (as opposed to *type-passing*) semantics, meaning that all types are erased at runtime, since they are assured to not have a computational effect. The addition of load to TAL as defined previously would violate the premise that types do not have computational effect, and therefore we would not be able to erase the types.

To preserve TAL's type-erasure semantics, we have added term representations for types, following the approach of Crary *et al.'s* λ_R [CWM98]. This way, rather than pass a type argument to load at runtime, we can pass a *term argument representing that type*, allowing the types to be safely erased. Informally, λ_R defines term representations for types, called R-terms, and types to classify these terms, called R-types. For example, the term to represent the type `int` would be $R_{\texttt{int}}$, and the type of this term would be $R(\texttt{int})$. The type $R(\tau)$ is a singleton type; for each τ there is only one value that inhabits it—the representation of τ. Therefore the type-checker guarantees the correspondence between a type variable checked statically and the representation of that type used at runtime.

So that this addition to the TAL trusted computing base can be kept small, we do two things. First, we represent R-terms using the binary format for types already used by the TAL disassembler.[6] Second, we do not provide any way within TAL to dynamically introduce or deconstruct R-terms, such as via appropriate syntax and `typecase` [CWM98]. Doing so would require that we reflect the entire binary format of types into the type system of TAL. Instead, we only allow the introduction of R-terms in the static data segment by a built-in directive. Consequently, only closed types may be represented. This restriction has proven problematic in some cases, which we elaborate on further in the next chapter.

5.4.2 load

In our implementation, load essentially has type:

$$\mathsf{load} : \forall\alpha.\,(R(\alpha) \times \texttt{bytearray}) \rightarrow \alpha\ \texttt{option}$$

In addition to the `bytearray` containing the module data, load takes a term representation of its type argument, as described above. The actions of load are illustrated in Figure 5.11.

[6]The binary representation of a named type is a string containing the name; this prevents problems with 'revealing' the implementation type of abstract types that occurs with type dynamics [ACPP91, ACPR95].

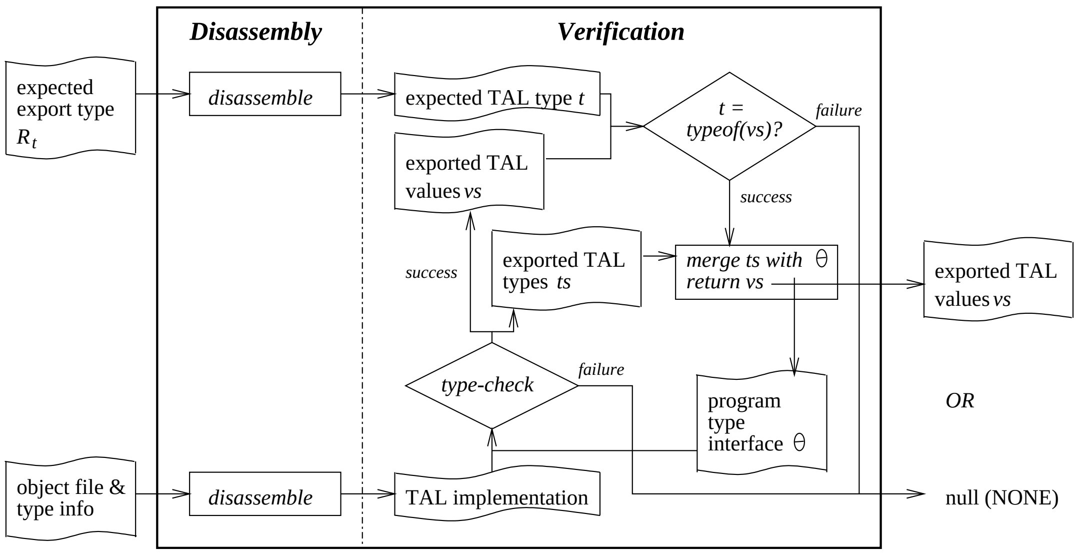


Figure 5.11: The implementation of load

In the figure, the square boxes indicate unconditional actions, and the diamond boxes indicate actions that may succeed or fail. Each square and diamond box has data inputs and outputs, indicated as wavy boxes; the arrows illustrate both data- and control-flow. Using components of the TALx86 system, load performs two functions:

1. **Disassembly** The first argument R_t indicates the expected type t of the exports, and must be disassembled into the internal representation of TAL types. To succeed, type t should be of tuple type, where each element type represents the type of one of the object file's exported values. The second argument to load is a byte array representing the object file and its typing annotations. This information is parsed by the TAL disassembler to produce the the appropriate internal representation: a *TAL implementation*.

2. **Verification** The TAL implementation is then type-checked in the context of the program's current type interface Θ, essentially following the procedure described for the load-calculus. This is done by calling the static link checking code [GM99], providing Θ as an argument. If type-checking succeeds, the result is a list of exported values and exported types. The values are gathered into a tuple, the type of which is compared to the expected type. If the types match, the tuple is returned (within an option type) to the caller, and the exported named types are combined with Θ to form the new program type interface. On failure, *null* (*i.e.*, `NONE`) is returned.

The majority of the functionality described above results in no addition to the TAL trusted computing base. In particular, the TAL type-checker and disassembler are already an integral part of the the TCB. To implement our trusted library, we combined the OCaml code that implements these features in TALx86 with code specific to load. This new code performs two tasks: it loads the object file, and maintains the program type interface Θ.

The loading code is written in C, and serves as the library's entry-point, and the maintenance code is written in OCaml, so as to interface existing OCaml code for processing TAL modules.

Loading

The TALx86 assembler compiles TAL modules to two separate files: an *object file*, containing the code and data, and a *types file*, containing the type annotations; for simplicity, the contents of these two files have been denoted as a single `bytearray` argument in Figure 5.11. TALx86 is implemented to work on both Windows and Linux, and the object file format on each system differs; Windows uses COFF [COF] object files, and Linux uses ELF [TISC95] object files. The TALx86 assembler produces the format correct to the OS on which it was compiled and run.

When load is called, the C code implementing the loader is invoked, which in turn calls the OCaml code to perform the majority of the actions depicted in Figure 5.11. Once the OCaml code returns with success, the object file is loaded into memory. This loading code is based on that used by the Linux kernel to dynamically load modules (modutils). We describe the code for ELF object files; COFF files are similar.

First, the file is parsed, performing well-formedness checks and extracting the ELF file's section headers, which describe the file's format. The file must be a relocatable object file, as is normally produced by a compiler for separate compilation, *e.g.* by `cc -c`. The sections of interest are the code and data sections, the relocations section, and the symbol tables. Second, the code and data are logically arranged in the order and alignment specified by the file and the ELF standard, and the total required size is computed. Third, any externally-defined symbols are resolved—more on this below. Finally, an appropriately-sized buffer is allocated and the code and data are copied to that buffer (TAL uses garbage collection, so the buffer is allocated using the GC allocator).[7] This code is then relocated to work relative to the allocated buffer's address. Finally, the address of the buffer is returned to the caller (which is the result of load).

There are two undesirable features of this implementation. First, the process of parsing the object file duplicates work done by the TALx86 disassembler (albeit for a different purpose: the disassembler must combine the object file with the types file to produce a TAL implementation). This is a result of the type-checking code being written in OCaml and the loading code being written in C. One way to fix this problem would be to integrate the two implementation components, say in OCaml. However, this would be a time-consuming task, and not of great research benefit.

The more troublesome feature is that we resolve (*i.e.* link) external symbols during the loading process. Part of the motivation of our approach was to perform linking outside the TCB, avoiding the additional complexity. In fact, most symbols *are* linked by mechanisms outside the TCB, as we show in the next chapter. However, there are some *trusted* symbols that cannot easily be linked in this way. These symbols are part of *macro* instructions that

[7]This allocation is necessary; we cannot reuse the buffer containing the object file data to avoid the copy. The reason is that load effectively changes the type of the buffer argument from `bytearray` to some type α. Placing the object file contents in a fresh buffer prevents surreptitiously modifying the given buffer via an alias still having `bytearray` type. We could avoid this copy by proving that no aliases exist, *e.g.* by using alias types [WM00].

TAL instruction	Machine Code expansion
	malloc
`malloc ptr, 8`	`pushl $0x8`
	`movl GC_malloc, %eax`
	`call *%eax`
	`addl $0x4, %esp`
	`testl %eax, %eax`
	`je out_of_memory`

Figure 5.12: Code sequence for TAL "macro" instructions

are a part of TAL. TAL macro instructions are those instructions that don't map directly to a machine instruction, but instead to a machine instruction sequence; this sequence may include references to external symbols. The expansion of the `malloc` instruction is shown in Table 5.12. We can see that this expansion makes two external calls, one to `GC_malloc`, and the other to `out_of_memory`. The file cannot be closed with respect to these calls, as shown in §5.3.3, because they are primitive.

Therefore, when a file containing a `malloc` instruction is dynamically loaded, the external calls must be resolved by the loader. We do this by rewriting the code directly, using the relocations provided in the object file. Patching symbols in the object file directly, in this manner, has two ramifications. First, the code of the loaded file cannot be shared between (OS-level) processes because the patched symbols, like `GC_malloc`, may be at different addresses in each process. Second, none of the patched symbols can be dynamically updated because a patched symbol cannot be re-bound. This implies that we cannot dynamically update the garbage collector, for example.

In practice, there are very few symbols that form part of macro instructions that need to be patched—for TALx86 (*v2.0*) there are 10 symbols. Furthermore, since these are all trusted language elements, it is reasonable that they may not be updated; only code that can be verified as safe may be dynamically linked in our approach. As it turns out, we also allow certain symbols introduced by the compiler to be linked implicitly in this manner, for convenience; right now, we close a file for loading via a source-to-source translation preceding compilation. Our initial implementation integrated this translation with compilation, and so it was able to properly deal with symbols added by the compiler. It may be that a hybrid approach is possible, but we have not explored this.

Given that we must link some symbols implicitly—that the module does not truly have to be 'closed'—it is reasonable to ask "why not link all symbols in this way?" The answer is that it would greatly reduce our flexibility and our security. As motivated in §5.2, by moving symbol management outside of the TCB, we can better control how symbols are stored (*i.e.* what datastructure), how they are apportioned among users of various privilege levels, how they are interfaced, *etc.*, without changing the trusted computing base; instead we can rely on the system to verify that this code is safe. Furthermore, we can *update the linking and management code at runtime*, as needs change. For example, we could imagine wanting to increase the strength of the cryptosystem used to authenticate the source of some loaded code used to make linking decisions. If we were to totally rely on the implicit linking technology described here, we could not do any of these things.

While implicit linking seems to be necessary for TAL macro instructions, it may be that our approach could be improved. In particular, if the symbols referred to by macro sequences (*e.g.* `GC_malloc`) were always loaded at the same address, then we could share the code between processes. Given that most modern operating systems support separate, per-process address spaces, and that both ELF and COFF files allow the loaded address for a program component to be specified, this should be possible. It would furthermore allow the relocation process to take place outside of the TCB, preceding the call to load. The disassembler would then check for the particular, fixed address when checking the well-formedness of macro instruction sequences, rather than looking for an external symbol reference.

Maintaining the Program Type Interface

Our implemented program type interface follows the form of the one described in §5.3.3. For simplicity, we wrote this code in OCaml, reusing existing, trusted datastructures. Representations of type interfaces (X_I, X_E) already exist as part of the types files; they are used to verify static link consistency. The initial Θ is initialized in a small bit of C code generated by the TAL static linker after it has determined the program's type interface. Computing the new type interface at runtime is done using this same trusted code for static link verification. In short, maintaining the type interface at runtime largely reuses existing, trusted code, and therefore does not noticeably expand the TCB.

5.4.3 checked_cast

Aside from providing type information to load, R-types are also useful for implementing dynamic types [ACPP91]. Dynamic types may be used to implement type-safe symbol management, as we describe in the next section. Therefore we allow limited examination of R-terms with a simple primitive called checked_cast:

$$\mathsf{checked\_cast} : \forall\alpha.\forall\beta.\,(R(\alpha) \times R(\beta) \times \beta) \to \alpha\ \mathtt{option}$$

Informally, checked_cast takes a value of type β and casts it to type α if the types α and β are equal. This operation is trivial to add as comparing types is part of the TAL type-checker. Therefore it does not add to the TCB. With a full implementation of λ_R including `typecase`, checked_cast does not need to be primitive [Wei00].

5.5 Discussion

Part of the benefit of having symbol management and linking outside of the TCB is that it allows greater flexibility: we can use different mechanisms, as the situation calls for, for implementing linking policy. For example, we could assign users a cryptographic key which is checked before each user loads code in the system. When linking takes place, information related to the user's privilege is used to determine what symbols are available. Different policies can be used in different situations, without affecting the TCB.

Specializing the linking of *types* to consider user-privilege requires extra *trusted* machinery because types are managed and linked *inside* the TCB by the type-checker. To increase flexibility and keep the TCB small, we observe that while type-linking must be trusted, type management need not be. In particular, we can allow the caller of load provide a *type environment mask* to restrict the program type interface. The idea is that the TCB keeps track of the 'standard' program interface, while the linking library (that is, the library that mitigates access to the load primitive, like the DLpop library described in the next chapter) maintains a number of more restricted interfaces, each one mapped from a certain level of privilege. Before load is called, the library will check the privilege of the calling user, and then apply the appropriate type environment mask to load.

In this section, we first present some alterations to the load-calculus, in particular to parts pertaining to load, to support type environment masks; the complete formulation and proof of soundness of the load-calculus presented in Appendix A includes this mask. We then touch on how we might implement such masks in TAL/Load.

5.5.1 The load-calculus with Type Environment Masks

A type environment mask is simply a type environment that is more restrictive than the running program's export type environment. It is used by the caller to limit the definitions that may be seen by the loaded code. To support masks, we augment load expression to take an additional argument:

$$\textit{expressions} \quad e \quad ::= \ldots\ \mathsf{load}[\tau]\ e_0\ e_1\ e_2\ e_3$$

Now, the first (term) argument is the binary representation of the type environment mask, while the remaining arguments are as before: the representation of the loaded program, the success expression and the failure expression. The typing rule for load changes to become:

$$\frac{\begin{array}{c} X;\Phi;\Delta;\Gamma \vdash e_0 : \texttt{int} \\ X;\Phi;\Delta;\Gamma \vdash e_1 : \texttt{int} \\ X;\Phi;\Delta;\Gamma \vdash e_2 : \tau' \rightarrow \tau \\ X;\Phi;\Delta;\Gamma \vdash e_3 : \tau \end{array}}{X;\Phi;\Delta;\Gamma \vdash \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3 : \tau}$$

The type environment mask is an 'integer;' as with the loaded code argument, the operational rule for load uses the function $\hat{\cdot}$ to map this integer to an actual type environment at runtime, simulating a filesystem.

The operational rules for load are shown in Figure 5.13. The *load-failure* and *congruence* rules are essentially the same as before, altered to support the extra mask argument. The *load-success* rule changes so that when linking the running program's type interface (X_I, X_H) with the loaded program's interface (X_I^i, X_H^i), we replace X_H with the mask X^h. This way, any type names defined by the program but not mentioned in the mask will be unavailable to loaded code. The result of the link operation is the type interface (X_I', X_H''). This export type environment X_H'' is merged with the program's export type environment X_H to restore any type names that were removed by the mask during linking.

$\boxed{(\Theta, H, \mathsf{load}\ e_0\ e_1\ e_2\ e_3) \mapsto (\Theta', H', e')}$

$$\frac{\begin{array}{c} X_H \vdash \hat{i} : \tau \\ H\ \mathtt{merge}\ H_i \Rightarrow H' \\ X_H \leq X^h \qquad X_I^i \mid (X_H - X^h) \\ (X_I, X^h)\ \mathtt{link}\ (X_I^i, X_H^i) \Rightarrow (X_I', X_H'') \end{array}}{\begin{array}{l} ((X_I, X_H), H, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \mapsto \\ ((X_I', X_H'), H', e_2\ e_i) \end{array}} \left(\begin{array}{c} \hat{h} = X^h \\ \hat{i} = ((X_I^i, X_H^i), H_i, e_i) \\ X_H' = X_H'' \oplus X_H \end{array} \right) \quad \textit{(load-success)}$$

$$\begin{array}{l} (\Theta, H, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \mapsto (\Theta, H, e_3) \\ \text{otherwise} \end{array} \quad \textit{(load} - \textit{failure)}$$

$$\frac{(\Theta, H, e) \mapsto (\Theta', H', e')}{\left\{ \begin{array}{c} (\Theta, H, \mathsf{load}[\tau]\ e\ e_1\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ e'\ e_1\ e_2\ e_3) \\ (\Theta, H, \mathsf{load}[\tau]\ v\ e\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ v\ e'\ e_2\ e_3) \end{array} \right\}} \quad \textit{(congruence)}$$

Figure 5.13: Operational rules for load using a type environment mask

The mask is intended to be more restrictive than the program's heap, so it must meet two conditions:

- $X_H \leq X^h$
  This condition guarantees that the mask defines strictly fewer (but compatible) types than the program's export type environment. The operator $\leq$ is defined as follows:

  $$X_1 \leq X_2 \textit{ iff } X_1 \precsim X_2 \text{ and } \mathtt{dom}\,(X_2) \subseteq \mathtt{dom}(X_1)$$

- $X_I^i \mid (X_H - X^h)$
  This condition guarantees that the loaded code's import type environment X_I^i does not import any type names defined by the program's export type environment but not defined by the mask. This condition is necessary to keep the final export type environment X_H' disjoint with the final import environment X_I', so that the final program is well-formed (see Figure 5.9).

The complete load-calculus and proof of soundness in Appendix A includes the type heap mask as we have defined it here.

5.5.2 Implementing Type Environment Masks

Implementing type environment masks would require only a small effort, with the majority of code occurring outside the TCB. The only operation that affects the TCB is our addition of a type environment mask parameter to load. Like the object file argument, the type mask can just be a bytearray, whose contents is understood by the implementation of load.

There is no need to extend the type system to qualify type environment masks because a mask is not reflected in any direct way into the running program. This is in contrast to the type argument to load, which must match the type of the value returned. Type environment masks already have a binary format for storage in TAL types files, and so we can just adopt that format and use the existing assembly/disassembly code, thereby not changing the TAL TCB.

There are a number of design decisions to be made in terms of how type environment masks could be used in practice, *e.g.* how they could be created and managed. We discuss this in future work (§11.5.2).

Chapter 6

DLpop: Dynamic Linking with TAL/Load

TAL/Load forms the trusted component of our dynamic linking approach. In this chapter we describe the untrusted component, which builds on TAL/Load to provide full-featured dynamic linking facilities. More concretely, we present the implementation of a type-safe version of DLopen [Lin95], a standard dynamic-linking methodology for C on Unix systems. Our version, called DLpop, provides the same functionality for Popcorn.

Describing DLpop in detail serves two purposes. First, it concretely demonstrates the flexibility afforded by TAL/Load. Second, and more importantly, it forms the foundation of our approach to dynamic updating, which we call DLpop/update. DLpop and DLpop/update share the same interface, and much of the same implementation, but have different semantics: in DLpop, symbol bindings are fixed at load-time and cannot be changed, but in DLpop/update, bindings may be redirected to new, updated code and data. How we modified DLpop to create DLpop/update is described in Chapter 7.

A reasonable question is why we chose to implement a version of DLopen, rather than some other dynamic linking interface, like COM. The answer is twofold. First, because our source language is Popcorn, which is C-like, we wanted to implement a library familiar to C programmers. Second, DLopen provides as much or more functionality than other approaches, and is therefore the most general, validating our claim of TAL/Load's flexibility. This claim is further bolstered in Section 6.4.2 where we explain how we might have programmed other approaches, including Java classloaders [jav96], Windows DLL's and COM [com01], Objective Caml's Dynlink [Ler00, Rou96], Flatt and Felleisen's Units [FF98], and SPIN's domains [SFPB96], among others.

We begin by describing DLpop and the ways in which it differs from DLopen, and then follow with a description of our implementation written in TAL/Load.

6.1 DLpop: A Type-safe DLopen

Most Unix systems provide some compiler support and a library of utilities (interfaced in the C header file `dlfcn.h`) for dynamically linking object files. We call this methodology *DLopen*, after the principal function it provides; our version is called DLpop. DLpop's library interface, depicted in Figure 6.1, is identical to DLopen except that it is type-safe. We describe this interface in detail below, noting differences with DLopen; a thorough

```
extern handle_t;
extern handle_t dlopen(string fname);
extern a dlsym<a>(handle_t h, string sym, <a>rep typ);
extern void dlclose(handle_t h);

extern exception WrongType(string);
extern exception FailsTypeCheck;
extern exception SymbolNotFound(string);
```

Figure 6.1: DLpop library interface

description of DLopen may be found in the Unix documentation [Lin95].

DLpop and DLopen both provide three core functions:

- `handle_t dlopen(string fname)`
  Given the name of a TAL object file, `dlopen` dynamically loads the file and returns a `handle_t` to it for future operations. Imports in the file (*i.e.,* symbols declared `extern` therein) are resolved with the exports (*i.e.,* symbols not declared `static`) of the running program and any previously loaded object files. Before it returns, `dlopen` will call the function `_init` if that function is defined in the loaded file. In DLpop (but not DLopen), `dlopen` type-checks the object file, throwing the exception `FailsTypeCheck` on failure. In addition, the exception `SymbolNotFound` will be raised if the loaded file imports a symbol not present in the running program, or `WrongType` if a symbol in the running program does not match the type expected by the import in the loaded file. In DLopen, such error conditions are reported by returning a *null* `handle_t`; in DLpop, `handle_t`'s are guaranteed never to be *null.*

- `a dlsym<a>(handle_t h, string sym, <a>rep typ)`
  Given the handle for a loaded object file, a string naming the symbol, and the representation of the symbol's type, `dlsym` returns a pointer to the symbol's value. In DLopen, `dlsym` does not take a type argument, and the function returns an untyped pointer (*null* on failure), of C type `void *`, which requires the programmer to perform an unchecked cast to the expected type. In contrast, our version takes a type representation argument `typ` to indicate the expected type; this type is checked against the actual type at runtime. In practice, `typ` will always be of pointer-type since the value returned is a reference to the requested symbol. As in TAL, we have extended Popcorn with representation types `<a>rep`, implementing them with TAL *R*-types. The term representing type `t` in Popcorn is denoted `repterm@<t>`. Because we cannot create the representation of a type with free type variables in TAL, the type argument `a` to `dlsym` must also be a closed type. If the requested symbol is not present in the object file, the exception `SymbolNotFound` is thrown; if the passed type does not match the type of the symbol, the exception `WrongType` is thrown.

- `void dlclose(handle_t h)`
  In DLopen, `dlclose` *unloads* the file associated with the given handle. In particular, the file's symbols are made unavailable to future linkages, and the memory for the file is freed; the programmer must make sure (hope?) there are no dangling pointers to symbols in the file. In DLpop, `dlclose` nulls out the references the library has to the loaded file, so it will be unavailable for future linkages. If the user program does not reference the loaded file, then nulling these references makes the file unreachable, and so it can be garbage collected. There is thus no possibility for dangling pointers.

While the basics are the same, DLopen has some advanced features that are not currently supported by DLpop. They are:

1. DLopen automatically loads object files upon which a dynamically loaded file depends, thus conveniently supporting recursive references between files.

2. DLopen supports the ability to optionally resolve function references *on-demand*, rather than all at load-time, as long as the underlying object file format supports it. ELF object files do support on-demand linking of functions using a procedure linkage table [TISC95].

3. DLopen provides a kind of finalization by calling the user-defined function `_fini` when unloading object files.

We foresee no technical difficulties in adding these features should the need arise. In particular, we have implemented a variant of `dlopen` that allows the caller to specify a list of object files to load, and these files may have mutually-recursive references. This has the prototype:

```
extern handle_t dlopens(string filenames[]) [];
```

Extending `dlopen` to implicitly load needed files on demand would mean adding a bit more information to the file during compilation. On-demand function symbol resolution is feasible: a possible compilation strategy to support it is described below, and another approach is described in Section 6.4.2. Lastly, finalization is implemented in most garbage collectors, in particular the Boehm-Demers-Weiser collector [BW88] used in the current TAL implementation. It would be simple to modify the loader described in §5.4.2 to associate the user's finalizer with the memory allocated for the loaded file.

Figure 6.2 depicts a simple use of DLpop. The user statically links the file `main.pop`, which, during execution, dynamically loads the object file `loadable.o` (the result of compiling `loadable.pop`), looks up the function `g`,[1] and then executes it. The dynamically linked file also makes an external reference to the function `f`, which is resolved at load time from the exports of `main.pop`.

Our implementation of DLpop is similar to implementations of DLopen that follow the ELF standard [TISC95] for dynamic linking, which requires both library and compiler support. In ELF, dynamically loadable files are compiled so that all references to data

[1]Note that the type argument to `dlsym` is inferred by the Popcorn compiler. The explicit syntax is `int g(int) = dlsym(h,"g", repterm@<int(int)>)@<int(int)>;`

Dynamically linked code: `loadable.pop`

```
extern struct t {                    /* imported definition of type t   */
  int a; int b;
}
extern int f(t);                     /* imported function; uses t       */
static int cnt;                      /* static variable (not exported)  */
static void _init() {                /* load-time initializer           */
  cnt = 5;
}
int g(int i) {                       /* exported function               */
  t T = new t(cnt++,i);                 /* reference to imported type t */
  return f(T);                          /* reference to imported func f */
}
```

Static code: `main.pop`

```
static int num = 0;                  /* static variable (not exported)  */
struct t {                           /* exported type definition t      */
  int a; int b;
}
int f (t T) {                        /* exported function f; uses t     */
  num++;
  return T.a + T.b;
}
void pop_main () {
  handle_t h = dlopen("loadable");   /* load the file                   */
  int g(int) =                       /* look up function g              */
    dlsym(h, "g", repterm@<int (int)>);
  g(3);                              /* call g                          */
  dlclose(h);                        /* close the file                  */
}
```

Figure 6.2: DLpop dynamic loading example

are indirected through a *global offset table* (GOT) present in each object file. Each slot in the table is logically labeled with the name of the symbol to be resolved. When the file is loaded dynamically, the dynamic linker fills each slot with the address of the actual exported function or value in the running program; these exported symbols are collected in a *dynamic symbol table*, used by the dynamic linker. This table consists of a list of hashtables, one per object file, each constructed at compile-time and stored as a special section in the object file. As files are loaded and unloaded, the hashtables are linked and unlinked from the list, respectively.

We describe our DLpop implementation below, pointing out differences with the ELF approach. While similar in spirit, DLpop is inherently more secure than DLopen: because it is written in TAL/Load, all operations are verifiably type-safe. A mistake in our implementation may result in incorrect behavior but not a loss of safety, which could result in a crash. We first describe the changes we made to the Popcorn compiler, and then describe how we implemented the DLpop library.

6.2 Compilation

As in the ELF approach, files involved in dynamic linking must be specially transformed. In our initial implementation, this transformation occurred as part of code generation within the compiler. Later, we moved to a source-to-source translation, preceding standard compilation. The chief benefit is that debugging is easier, since we can pretty-print the results of the transformation as more-readable Popcorn, rather than less-readable TAL code. It also provided for a more modular design, since all issues with dynamic linking are separated from those of code generation. Finally, it makes the dynamic linking transformation Popcorn compiler-independent. More specifically, once some additional features are added to the Popcorn optimizing compiler (see §4.2.6), we can gain the benefit of its superior code.

6.2.1 Dynamically Linked Files

The dynamic transformation is performed in three stages. As an example, Figure 6.3 shows the entire translation for the dynamic code in Figure 6.2. All of the code that is added or altered as a result of the transformation is shown in boxes. In the translated code, many of the automatically-generated identifiers have suffixes, like `__1`, to avoid name-clashes. During the discussion below, these suffixes are omitted, where possible, for brevity.

Global Offset Table (GOT) First, we define a GOT for the file, and translate references to externally defined functions and data to refer to slots in the GOT. Here, the call `f(T)` in the function `g` is changed to be `GOT.f(T)`. In ELF, the GOT is part of the object file, while in DLpop the GOT is implemented in the verifiable language (TAL, as compiled from Popcorn). As a consequence, the table is verifiably well-typed. In Figure 6.3, the GOT has type `GOT_t`, which is a structure-type containing a single element of function type; this element will be filled in at load-time with the value of `f`, from `main.pop`. The actual GOT is declared in the variable `GOT`. To avoid null-checks, each GOT slot is initialized to a dummy value of the correct type, where possible. In this case, a function of

```
static int cnt = 0;
extern struct t {
  int a; int b;
}
static int g (int i) {
  t T = new t(cnt++, i);
  return (GOT.f(T));
}
static void _init () {
  cnt = 5;
}
static exception exncon__2(string);
static int fn__3 (t a) {
  raise (new exncon__2("f"));
}
static struct GOT_t {
  int f (t);
}
static GOT_t GOT = new GOT_t{f=fn__3};
static bool is_updated_flag = false;
static bool done_init_flag = false;
void
dyninit_loadable<b,c> (a lookup <a>(b,string,<a>rep),
                       b lookup_closure,
                       void update <a>(c,string,<a>rep,a),
                       c update_closure,
                       bool no_init) {
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "g", repterm@<int (int)>, g);
  }
  GOT.f = lookup(lookup_closure, "f", repterm@<int (t)>);
  if (!no_init)
    if (!done_init_flag) {
      done_init_flag = true; _init();
    };
}
```

Figure 6.3: Compilation of dynamically loadable code. Additions or alterations as part of the compilation are boxed.

the correct type, `fn__3`, has been created, which simply raises an exception. To use `fn__3` in the initialization expression, we needed to allow top-level initialization expressions refer to identifiers, so long as the resulting expression was still constant (see §4.2.6). For slots having abstract type, we cannot create this dummy value, so we initialize the slot to null and insert null checks for each table access in order to satisfy the type-checker. The TAL verifier guarantees that these null checks cannot be left out by mistake.

`dyninit` function Second, we generate a special `dyninit` function that will be called at load-time to fill in the slots in the GOT with the proper symbols. This approach differs from ELF, in which the GOT is filled by a dynamic linker, implemented as part of the DLopen library. From the loading program's point of view, the `dyninit` function abstracts the process of linking a file.

The `dyninit` function takes as arguments two other functions, `lookup` and `update`, that provide access to the dynamic symbol table; two *closures* for use by those functions, `lookup_closure` and `update_closure`, respectively; and a flag `no_init` that indicates whether to call the user-defined `_init` function. For each symbol address to be stored in the GOT, `dyninit` will look up that address by name and type using the `lookup` function, and fill in the appropriate GOT slot with the result. Similarly, `dyninit` will call `update` with the name, type, and address of each symbol that it wishes to export. In both cases, the respective closure argument is passed to the function as its first argument. This argument is used by the caller of `dyninit` to provide information that customizes the `lookup` and `update` functions; because closures are parametrically polymorphic, the implementation of closures may be used differently by the caller in different circumstances without affecting `dyninit`. Finally, if an `_init` function is defined in the file then it is called when the `no_init` flag is set to false. Because the `dyninit` function consists only of TAL code, all linking operations are verifiably type-safe. This verification prevents, for example, `lookup` from requesting a symbol by name, then receiving a symbol of an unexpected type. In an untype-checked setting, as in DLopen, this operation could result in a crash.

When just one module is being loaded, by calling `dlopen`, the `dyninit` function is called just once. If any linking error occurs during `dyninit`'s execution, *e.g.* if a symbol requested by `lookup` is not found, then this error is simply propagated to the caller of `dlopen`. However, `dlopens` is constructed to make two linking passes; this supports linking modules that have mutually-recursive references. During the first pass, all of the modules being loaded will register their exported (*i.e.* non-`static`) symbols by calling `update`. In the example, only the function `g` is registered, as the variables `_init` and `cnt` have been declared `static`. If any of the `lookup` calls fails during the first pass, the error is ignored since the symbol may not have been registered yet by another module being loaded. During the second pass, no updates are performed because the variable `is_updated_flag` was set during the first pass. The `lookup`s are then performed again; this time, a failed symbol resolution truly signifies an error, and is propagated to the caller. The `no_init` flag is set to true during the first pass, so that the `_init` function is only called once, after all linking has been completed. Furthermore, the file records when its `_init` function has been called by setting the `done_init` flag, so it will not be called on any future invocations.

Making exports `static` Finally, because the exports of dynamically linked files are designated by `dyninit`, the object file should only export `dyninit` itself, changing all other symbols to be `static`. This way, the call to `load` within `dlopen`(s) will always expect to get the `dyninit` function as a result.

6.2.2 Statically Linked Files

Statically linked files are only changed by adding a `dyninit` to export symbols to dynamically linked files. At startup, the program calls the DLpop function `dlinit` for each `dyninit` function of its statically linked files; `dlinit` will call `dyninit` with properly constructed `lookup` and `update` functions. Figure 6.4 shows the translated static code of Figure 6.2. Notice that only `f` and `pop_main` are dynamically exported via a call to the `update` function, because `num` is declared `static`.

The code that calls `dlinit` for each statically linked file is currently inserted by the TAL static linker. These calls are stored in the same bit of C code generated to register the program type interface (see §5.4.2). While expedient, having the TAL linker generate code to call `dlinit` for each `dyninit` function is an unsatisfying mixing of abstractions. The TAL linker should only be concerned with TAL/Load, and not the way that it is used. Instead, a better approach would be to have a phase preceding static linking that, given the object files and libraries to link, generates Popcorn code to perform the `dlinit` calls. Having this phase would completely decouple DLpop from TAL/Load, and would reduce the size of the TCB, since the generated code can be properly type-checked.

6.3 The DLpop Library

The DLpop interface in Figure 6.1 is implemented as a Popcorn library. The central element of the library is a type-safe implementation of the dynamic symbol table for managing the symbols exported by the running program. We first describe this symbol table, and then describe how the DLpop functions are used in conjunction with it.

DLpop encodes the dynamic symbol table much as in ELF, as a list of hashtables mapping symbol names to their addresses, one hashtable per linked object file. Each time a new object file is loaded, a new hashtable is added. The dynamic symbol table is constructed at start-up time by calling the `dyninit` functions for all of the statically linked object files, as mentioned in the previous section.

Each entry of the hashtable contains the name, value, and type representation of a symbol in the running program, with the name as the key. So that entries have uniform type, we use existential types [MP88] to hide the actual type of the value:[2]

`objfile_ht : <string,` $\exists\alpha.\ (R(\alpha) \times \alpha)$`> hashtable`

To update the table with a new symbol (the result of calling `update` from `dyninit`), we pack the value and type representation together in an existential package, hiding the value's type, and insert that package into the table under the symbol's key.

When looking up a symbol having some type α, we query the hashtable for the symbol with the given name. If present, the hashtable returns an entry containing a value of some

[2]The type `<`τ_1, τ_2`> hashtable` contains mappings from τ_1 to τ_2.

```
static int num = 0;
struct t {
  int a; int b;
}
int f (t T) {
  num++;
  return T.a + T.b;
}
void pop_main () {
  handle_t h = dlopen("loadable");
  int g(int) = dlsym(h, "g", repterm@<int (int)>);
  g(3);
  dlclose(h);
}
```

```
static bool is_updated_flag = false;
void
dyninit_main<b,c> (a lookup <a>(b,string,<a>rep),
                   b lookup_closure,
                   void update <a>(c,string,<a>rep,a),
                   c update_closure,
                   bool no_init) {
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "f", repterm@<int (t)>, f);
    update(update_closure, "pop_main", repterm@<void ()>, pop_main);
  }
}
```

Figure 6.4: Compilation of statically linked code. Additions or alterations as part of the compilation are boxed.

```
static abstype entry[a] = *(<a>rep,a);
static a find <a> (<<string,entry>hashtable>list start,
                   <<string,entry>hashtable>list stop,
                   string name, <a>rep typ)
{
  while (start != stop) {
    try {
      entry h = hash_lookup(start.hd, name);
      with er[b] = h do { // found it
        return pop_checked_cast (er.2, er.1, typ);
      }
    } handle e {
      switch e {
      case Not_found: {   // symbol not found
        start = start.tl;
      }
      case Failure(s): {  // cast failed
        raise WrongType(name);
      }}
    }
  }
  raise SymbolNotFound(name);
}
```

Figure 6.5: Popcorn code for dynamic symbol table lookup

abstract type and a representation of that type: $\exists\beta.(R(\beta) \times \beta)$. This value is unpacked, binding a type variable β, and two term variables, `table_rep` and `table_value`. These two term variables are the type representation and the value being looked up, having type $R(\beta)$ and β, respectively. With the call

checked_cast $[\alpha][\beta]$`(r, table_rep, table_value)`

the type representations `r` and `table_rep` are compared, converting `table_value` from type β to type α if they match.

In our implementation, the function `find` implements symbol lookup as described above. Given a portion of the dynamic symbol table, a symbol name, and its type representation, `find` returns the value for the symbol, having the type specified. The code for `find` is shown in Figure 6.5.

The type $\exists\alpha.(R(\alpha) \times \alpha)$ is declared as the type `entry`; the Popcorn syntax for the type is in the first line of the figure.[3] For each hashtable between the given start and stop entries in the dynamic symbol table, `find` looks up the symbol with the call to `hash_lookup`. If found, it unpacks the entry with the syntax `with er[b] = h do`; this

[3]See §4.2.5 for a tutorial on existential types in Popcorn.

binds the existential type variable in the entry to b, and binds the tuple *(<b>rep,b) to the variable er. It then performs a Popcorn version of checked_cast; on success, the cast value is returned and on failure the exception Failure is raised (which is immediately caught the exception WrongType is raised instead). If the call to hash_lookup fails, then it will throw an exception Not_found; this is caught and the loop continues with the next hashtable. If the end of the list of hashtables is reached and the symbol has still not been found, then the exception SymbolNotFound is raised.

The DLpop library essentially consists of wrapper functions for load and the dynamic symbol table manipulation routines:

- dlopen
  Recall that dlopen takes as its argument the name of an object file to load. First it opens and reads this object file into a bytearray. Because of the compilation strategy we have chosen, all loadable files should export a single symbol, the dyninit function. Therefore, we call load with the dyninit function's type and the bytearray, and should receive back the dyninit function itself as a result. If load returns NONE, indicating an error, dlopen raises the exception FailsTypeCheck. Otherwise, a new hashtable is created, and a custom update function is crafted that adds symbols to it. The returned dyninit function is called with this custom update function, as well as with a lookup function that works on the entire dynamic symbol table. After dyninit completes, the new hashtable is added to the dynamic symbol table, and then returned to the caller with abstract type handle_t.

- dlsym
  This function receives a type argument (call it α) and three term arguments: a handle_t, h; a string representing the symbol name, s; and the representation of the type α, r. Because the handle_t object returned by dlopen is in actuality the hashtable for the object file, dlsym simply attempts to look up the given symbol in that hashtable, as described above, raising the exception SymbolNotFound if the symbol is not present. If the value is found, but the checked_cast operation fails, the expected type does not match the actual type of the value in the table, and the exception WrongType is raised.

- dlclose
  The dlclose operation simply removes the hashtable associated with the handle_t from the dynamic symbol table. Future attempts to look up symbols using this handle will be unsuccessful. Once the rest of the program no longer references the handle's object file, it will be safely garbage-collected.

As a closing remark, we emphasize the value of the way in which we implemented DLpop. By using TAL/Load, much of DLpop was implemented within the verifiable language, and was therefore provably safe. Only load and λ_R constitute trusted elements in its implementation, and these elements are themselves small. If some flaw exists in DLpop, the result will not constitute a violation of safety.

6.4 Discussion

We now look more closely at some of the aspects of DLpop. In particular, we present a more detailed justification for our use, and the benefit, of the `dyninit` function. We then discuss how we might implement other dynamic linking strategies using TAL/Load, showcasing its flexibility.

6.4.1 Examining `dyninit`

Rather than add the `dyninit` function to fill in the GOT's of loaded files and note their exported symbols, we could have easily followed the ELF approach of writing a separate dynamic linker, called at startup and from `dlopen`. Rather than construct a `dyninit` function per file, we would generate a symbol table per file, stored in the static data segment; this is essentially what ELF does, but it stores the table in a special section of the object file. When calling `load`, `dlopen` would expect to be returned both this table and the global offset table; all other symbols would be made static. The file symbol table would then be added to the global, dynamic symbol table, and then the entries in the GOT would be filled in. This would save time during linking (since the export table is already constructed), and might save space in the file.

However, we have found that abstracting the process of linking to calling a function in the loaded file has a number of benefits. First, it allows the means by which an object file resolves its imported symbols to change without affecting the DLpop library. For example, in order to save space, we could allow GOT entries to be null by changing them to `option` type, or we could eliminate the GOT altogether by using runtime code generation, as described in Section 6.4.2. If we knew that many symbols could remain unused by the loading program (as is likely with a shared library), we could resolve them on-demand by making the dummy functions perform the symbol resolution, rather than doing so in the `dyninit` function; this approach is shown in Figure 6.6. One tricky technical point is that we must use *existential* types (declared `abstype` in Popcorn) to store the `lookup` function with its closure. That is, the `lookup` function is *packed* along with its closure argument as an existential `dynlookup`; this package is unpacked just before the need to call `lookup`.

Second, using `dyninit` allows the loaded file to customize operations performed at link-time. For example, we could imagine that in a mobile code setting certain symbols may not be available to untrusted applications. Because the mobile code is performing its own linking via `dyninit`, it can react to linkage failures and take appropriate action. For example, if it finds that a certain expected service is not available for linking, it could choose to request an alternate service instead.

Finally, `dyninit` simplifies the implementation of policy decisions made by the loading code with regard to symbol management. For example, the loading code may wish to restrict access to some of its symbols based on security criteria [SFPB96]; in this case, it could customize the `lookup` function provided to `dyninit` to throw an exception if a restricted symbol is requested. Customization can be done using the closure argument to `lookup`; for example, we could implement it as the list of restricted symbols. As another possibility, `lookup` could choose to provide an alternative, safe version of a symbol, rather than prevent access altogether. For example, if the loaded code imports `open`, `lookup` could provide a version that only allows opening files in the `/tmp` directory.

Using `dyninit` has drawbacks as well. In addition to the added start-time overhead, and additional space in the loaded file, it is likely that the type of the `dyninit` function will change with the source language or linking strategy used, meaning that modules using different `dyninit`'s cannot be intermixed. For example, as we explain in the next chapter, there are a number of changes we could make to `dyninit` to support dynamic updating (although the type presented here is sufficient). There is also the security threat of code within `dyninit` consuming excessive resources during linking; but this risk exists anyway in the user-defined `_init` function, and from untrusted code in general. More experience will be needed to determine whether `dyninit`'s benefits outweigh its costs, but our current impression is very positive.

6.4.2 Programming Other Linking Strategies

Using our framework TAL/Load, we can implement safe, flexible, and efficient dynamic linking for native code, which we have illustrated by programming a safe DLopen library for Popcorn. Many other dynamic linking approaches have been proposed, for both high and low level languages. In this subsection we do two things. First, we describe the dynamic linking interfaces of some high level languages, describe their typical implementations, and finally explain how to program them in TAL/Load, resulting in better security due to type safety and/or reduced TCB size. Second, we look at some low-level mechanisms used to implement dynamic linking, and explain how we can program them in our framework. Overall, we demonstrate that TAL/Load is flexible enough to encode typical dynamic linking interfaces and mechanisms, but with a higher level of safety and security.

Java

In Java, user-defined classloaders [jav96] may be invoked to retrieve and instantiate the bytes for a class, ultimately returning a `Class` object to the caller. A classloader may use any means to locate the bytes of a class, but then relies on the trusted functions `Classloader.defineClass` and `Classloader.resolveClass` to instantiate and verify the class, respectively. When invoked directly, a classloader is analogous to `dlopen`. Returned classes may be accessed directly, as with `dlsym`, if they can be cast to some entity that is known statically, such as an interface or superclass—this is analogous to the type argument passed to `dlsym`. In the standard JVM implementation, linking occurs on-demand as the program executes such that when an unresolved class variable is accessed, the classloader is called to obtain and instantiate the referenced class. All linking operations occur within the TCB: checks for unresolved class variables occur as part of JVM execution, and symbol management occurs within `resolveClass`.

We can implement classloaders in TAL/Load by following our approach for DLpop: we compile classes to have a GOT and a `dyninit` function to resolve and register symbols. A classloader may locate the class bytes exactly as in Java (*i.e.*, by any means programmable in TAL), and `defineClass` simply becomes a wrapper for a function similar to `dlopen`, which calls load and then invokes the `dyninit` function of the class with the dynamic symbol table.

```
static int cnt = 0;
extern struct t { int a; int b; }
static int g (int i) {
  t T = new t(cnt++, i);
  return (GOT.f(T));
}
static void _init () {
  cnt = 5;
}
static exception exncon__2(string);
extern ?struct <a>Core::Opt { a v; }
static abstype fn[b] = *(a f<a>(b,string,<a>rep), b);
static <fn>Core::Opt dynlookup_closure = null;
static a dynlookup<a>(string name, <a>rep typ) {
  if (dynlookup_closure == null) raise (new exncon__2("dynlookup"));
  with f[b] = dynlookup_closure.v do
    return f.1(f.2,name,typ);
}
static int fn__3 (t a) {
  GOT.f = dynlookup("f", repterm@<int (t)>);
  return GOT.f(a);
}
static struct GOT_t { int f (t); }
static GOT_t GOT = new GOT_t{f=fn__3};
static bool is_updated_flag = false;
static bool done_init_flag = false;
void dyninit_loadable (a lookup<a>(string,<a>rep),
                      b lookup_closure,
                      void update<a>(string,<a>rep,a),
                      c update_closure,
                      bool no_init) {
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure,"g", repterm@<int (int)>, g);
  }
  dynlookup_closure = ^Core::Opt(^fn(^(lookup,lookup_closure)));
  if (!no_init)
    if (!done_init_flag) {
      done_init_flag = true; _init();
    };
}
```

Figure 6.6: Compilation of dynamically loadable code to resolve functions on-demand. Differences from Figure 6.3 are boxed.

To support incremental linking, we can alter the compilation of Java to TAL (hypothetically speaking) in two ways. We first compile the GOT, which holds references to externally defined classes, to allow null values (in contrast to DLpop where we had default values). Each time a class is referenced through the GOT, a null check is performed; if the reference is null then we call the classloader to load the class, filling in the result in the GOT. Otherwise, we simply follow the pointer that is present. As in the strategy depicted in Figure 6.6, the `dyninit` function no longer fills in the GOT at load-time; it simply registers its symbols in the dynamic symbol table. As in all cases with TAL/Load, this approach moves both symbol management and the check for unresolved references into the verifiable language, reducing the size of the TCB.

Windows DLL's and COM

Windows allows applications to load Dynamically Linked Libraries (DLL's) into running applications, following an interface and implementation quite similar to DLopen and ELF, respectively, with some minor differences (see Levine [JRL00], pps. 217–222). Like DLopen and ELF, DLL's are not type-safe and would therefore benefit in this regard from an implementation in TAL/Load.

DLL's are often used as a vehicle to load and manipulate Common Object Model (COM) [com01] objects. COM objects are treated abstractly by their clients, providing access through one or more interfaces, each consisting of one or more function pointers. All COM objects must implement the interface **IUnknown**, which provides the function `QueryInterface`, to be called at runtime to determine if the object implements a particular interface. `QueryInterface` is called with the globally unique identifier (GUID) that names the desired interface. GUIDs are not incorporated into the type-system (at least not for source languages like C and C++), and thus, as with `dlsym`, the user must cast the object's returned interface to the type expected, with a mistake potentially resulting in a crash.

Implementing COM in TAL/Load would be straightforward, with the added benefit of proven type-safety for interfaces. `QueryInterface` could be changed to take type parameter $R(t)$ in addition to the GUID of the expected interface, ensuring the proper type of the returned interface.

Objective Caml Modules

OCaml provides dynamic linking for its bytecode-based runtime system with a special `Dynlink` module; these facilities have been used to implement an OCaml applet system, MMM [Rou96]. `Dynlink` essentially implements `dlopen`, but not `dlsym` and `dlclose`, and would thus be easy to encode in TAL/Load. In contrast to the JVM, OCaml does not verify that its extensions are well-formed, and instead relies on a trusted compiler. OCaml dynamic linking is similar to that of other type-safe, functional languages, *e.g.* Haskell [PHL97].

To be of use, applets' initialization procedures are expected to modify data structures in the running program to point into their code. For example, the running program might define a global variable `current_applet`, which points to a function implementing the current applet. When the new applet is loaded in, its initialization code alters the `current_applet` variable to point to its entry function. This approach has the benefit that no runtime type analysis (such as by using `checked_cast`) is required, but it is less flexible. OCaml modules may never be unloaded, which also simplifies the implementation but is inflexible.

A TAL/Load implementation of the OCaml interface would improve on its current implementation [Ler00] in two ways. First, all linking operations would occur outside of the TCB. Second, extension well-formedness would be verified rather than assumed.

Units

Units [FF98] are software construction components, quite similar to modules. A unit may be dynamically linked into a static program with the **invoke** primitive, which takes as arguments the unit itself (perhaps in some binary format) and a list of symbols needed to resolve its imports. Linking consists of resolving the imports and executing the unit's initialization function. **Invoke** is similar to `dlopen`, but the symbols to link are provided explicitly, rather than maintained in a global table.

Units could be implemented following DLpop, but without a dynamic symbol table. Rather than compiling the `dyninit` function to take two functions, `lookup` and `update`, it would take as arguments the list of symbols needed to fill the imports. The function would then fill in the GOT entries with these symbols, and then call the user-defined `_init` function for the unit. The implementation for **invoke** would call `load`, and then call the `dyninit` function with the arguments supplied to **invoke**. Part of our initial inspiration for the `dyninit` function came from the Units' **invoke** function.

The current units implementation [FF98] is similar to the one we have described above, but is written in Scheme, a dynamically typed language. Therefore, while linking errors within `dyninit` may be handled gracefully in our system (since they will result in thrown exceptions), in Scheme they will result in runtime type errors, halting system service.

SPIN

The extensible operating systems community has explored a number of approaches to dynamic linking. For example, the SPIN [BSP$^+$95] kernel may load untrusted extensions written in the type-safe language Modula-3. In SPIN, dynamic linking operates on objects called domains [SFPB96], which are collections of code, data, and exported symbols. Domains are quite similar to Units, with the functionality of **invoke** spread among separate functions for creation, linking, and initialization, along with other useful operations, including unlinking and combining. All of these operations are provided by the trusted `Domain` module. Furthermore, all operations are subject to security checks based on runtime criteria. For example, when one domain is linked against the interface of another, the interface seen may depend on the caller's privilege.

We can implement domains using techniques described above, with the addition of filters to take security information into account. TAL/Load would improve on the security of the current SPIN implementation in the same ways as OCaml: less of the domain implementation must be trusted to ensure safety, and integrity of extensions can be verified, rather than relegated to a trusted compiler.

TMAL

The TAL module system implemented for TALx86, MTAL (Modular Typed Assembly Language [GM99]), provides a typed version of standard static linking facilities. Typed Module Assembly Language (TMAL) [Dug00] is an alternative module system for TAL that provides a different model of linking, including dynamic linking. Our work in TAL/Load is an extension to TAL to allow dynamically linking MTAL modules. Therefore, TMAL and TAL/Load can be seen as two ways to solve similar problems. TMAL has not been implemented.

TMAL adds a simple notion of first-class modules to TAL; by using explicit coercions accompanied by runtime checks, the type system remains decidable. The operations provided for TMAL module values are much like those for SPIN domains, described above. Two modules can be linked together to form a third module, and the circumstances of linking can be customized. In particular, coercions are provided to remove exported names from a module, and to rename its types and/or values. In addition, modules can be linked with symbols from the program (rather than other modules).

TMAL also provides primitives for reflection. In particular, TMAL's `dlsym_v` is essentially the same as DLpop's `dlsym`. MTAL, and thus TAL/Load, makes the simplification that all named types are global. As a module is loaded, its type components are added into the global namespace. However, in TMAL, first-class modules can contain type components, which introduces a level of hierarchy. As a result, TMAL provides a `dlsym_t` operation for looking up a type component of a module, to be used prior to retrieving a value that has that type.

Finally, TMAL provides primitives for creating and loading dynamically-linked libraries, respectively; the latter operation is similar to `load`, and the former is something that we do at compile-time.

The major difference between TAL/Load and TMAL is that TAL/Load is intended for *programming* the sorts of operations that TMAL provides as primitive; the result is a smaller TCB. On the other hand, the goal of TMAL is to preserve and statically verify the constraints expressed by the source module language at the assembly language level. We could easily implement the majority of TMAL using TAL/Load, where the notion of `handle` as implemented in DLpop is analogous to a first-class module TMAL. Breaking the linking functionality out of DLpop's `dlopen` into the various TMAL linking primitives would be straightforward for values, but tricky for types, though still possible; *e.g.* we can use a type environment mask (see §5.5.1) to hide global types from loaded modules, and we could use existential types to implement something like `dlsym_t`. However, in such an implementation, some properties that could be statically verified by TMAL, would have to be dynamically checked by `load`.

On the other hand, programming provides flexibility. In the case of values, we could even program additional module coercions, since they essentially control a module's symbol table. For example, we could add security information to the table to be used during linking, as is done in SPIN.

Low-level Dynamic Linking Mechanisms

A useful reference of low-level, dynamic linking mechanisms may be found in Franz [Fra97]. One technique that he presents, which has been used to implement some versions of DLopen (as opposed to the ELF methodology [TISC95]), is called load-time rewriting. Rather than pay the indirection penalty of using a GOT, the dynamic linker rewrites each of the call-sites for an external reference with the correct address. This is essentially how we link the trusted symbols in TAL macro sequences, as described in §5.4.2.

This technique is a simple form of runtime code generation. Popcorn and TAL have been extended to support type-safe runtime code generation; the extension is called Cyclone [HJ99]. We could use Cyclone to implement load-time rewriting outside of the TCB. Rather than compile functions to indirect external references through a GOT, we instead create template functions that abstract their external references. When `dyninit` is called, each template function is invoked with the appropriate symbols (found by calling `lookup`), returning a custom version of the original function, closed with respect to the provided symbols. This function is then registered with the dynamic symbol table using `update`. The advantage of this approach is that the process of rewriting can be proven completely safe.

For example, the static code in Figure 6.2 could be compiled as shown in Figure 6.7.[4] This strategy actually eliminates *all* the overhead from linking, as the indirection for the function invocation in bar is also eliminated (its value, rather than location, is emitted by `fill`).

The major disadvantage of code rewriting, in general, is that the code may not be shared between processes. There are also two notable disadvantages of our particular implementation. First, mutually recursive functions are problematic because their template functions must be called in a particular order. This requires all the relevant `lookup`s to be made before constructing each function to `update`; therefore we cannot take the two-pass approach described in §6.2 in which we perform all of the `update`s before we do all of the `lookup`s. One possible solution is to use one level of indirection for recursive calls, backpatching the correct values. How to insert indirections judiciously at compile-time is likely to be tricky, however. Another disadvantage is that template functions make copies of the functions they abstract, rather than filling in the holes in place; making copies is more general, but not necessary in our context. However, the overall cost of doing this should be low (especially relative to verification).

Acknowledgements

As mentioned in the introduction of the previous chapter, the work presented in the last two chapters was done jointly with Karl Crary and Stephanie Weirich. Karl had the basic idea of the `load` primitive to perform basic loading of closed terms (analogous to a primitive available in the Standard ML of New Jersey implementation). It was my extension to say

[4]The syntax for Cyclone may be found in [HJ99]. Intuitively, the syntax `codegen(...)` copies the code between the parentheses, returning a pointer to a fresh function. At each occurrence of `fill(`x`)` within the `codegen`, the value x is inserted directly within the copied code. In this way, the generated code is *specialized* to make constants of values not known until runtime.

```
static int cnt = 0;
extern struct t {
  int a; int b;
}
static void _init () {
  cnt = 5;
}
static int make_g (int f(t)) (int) {
  return (codegen(
    int g(int i) {
      t T = new t(cnt++, 1);
      return fill(f)(T); })
  );
}
static bool done_init_flag = false;
void
dyninit_loadable<b,c> (a lookup<a>(b,string,<a>rep),
                       b lookup_closure,
                       void update<a>(c,string,<a>rep,a),
                       c update_closure,
                       bool no_init) {
  int f(t) = lookup(lookup_closure,"f",repterm@<int (t)>);
  int g(int) = make_g(f);
  update(update_closure,"g", repterm@<int (int)>, g);
  if (!no_init)
    if (!done_init_flag) {
      done_init_flag = true;
      _init();
    };
}
```

Figure 6.7: Compilation of dynamically loadable code to use runtime code generation. Differences from Figure 6.3 are boxed.

that we could build source-level linking strategies on top of this; Karl suggested the use of existentials coupled with term representations for the symbol table. Stephanie and I split the initial implementation in the compiler, and she wrote the initial DLpop library in TAL. Stephanie also implemented type representations for TAL and designed the important checked_cast primitive (which she later did a separate paper on [Wei00]). I implemented load and the final DLpop infrastructure, including the source-to-source translation (plus the features added to Popcorn to enable this), and the Popcorn version of DLpop to support mutually-recursive references. I also did all the measurements and most of the writing. Stephanie split the theoretical work with me.

Chapter 7

Dynamic Updating

We now turn our attention to the core of this dissertation, dynamic software updating. Our dynamic updating approach consists of four basic steps. For each module that has changed from the previous version:

1. Construct a *dynamic patch* that reflects the differences.
2. Compile the dynamic patch to a loadable TAL file.
3. Dynamically link the file into the running program.
4. Transition the running program to use the patch.

As demonstrated in the next three chapters, our dynamic updating design and implementation aims to be practical, cleanly meeting all of the evaluation criteria defined in Chapter 2. First, we inherit from TAL/Load and DLpop a simple yet flexible and safe framework for loading files. Next, we define a notion of patch that separates a changed module's code from the additional code and data needed to support updates. As a result, we cleanly separate the processes of software and patch development, making the system easier to use and maintain. Finally, we extend DLpop to support updating, reapplying its core linking technology with only slight modifications to the DLpop library and compilation approach. As a result, we preserve our simple implementation and small trusted computing base, and impose no additional overhead.

This chapter describes the design and implementation of our approach, arguing that it is flexible, robust, and easy to use. We begin by defining our notion of dynamic patch, and then explore mechanisms we have considered to enable dynamic patching, the mechanisms we chose, and why. We then present DLpop/update, an extension of DLpop that implements dynamic updating. In the next chapter, we complete our argument by describing how to build dynamic patches mostly automatically and how to ensure that their application is well-timed.

7.1 Dynamic Patches

A *dynamic patch* describes the dynamic changes between two versions of a program module. How we define a dynamic patch influences both the system's flexibility and its ease of use: it should ideally be able to express arbitrary changes to a file, and it should cleanly separate constructs required for patching from the new code, allowing the software development

```
static int num = 0;
struct t {
  int a; int b;
}
int f (t T) {
  num++;
  return T.a + T.b;
}
```

Figure 7.1: Example file `main.pop` (without the `pop_main` function)

process to be cleanly separated from patch development. It also affects the system's robustness, as implementing the patch semantics could be quite difficult, resulting in a large and/or complex implementation. We present our notion of dynamic patch incrementally, arriving at a definition that is suitably flexible, all the while keeping the new code separate from code germane to patching.

Dynamic patches differ from static patches, such as those created and applied using the Unix programs `diff` and `patch`, because they must deal with the state of the running program. We can abstractly define a dynamic patch of some file f as the pair (f', S), where f' is the new version of the file and S is an optional *state transformer function*, used to convert the existing state accumulated by f to a form usable by the new code f'. The transformer is defined such that the old and new code have their own state, and thus the old state is copied to the new code and then properly transformed. This notion of patch is similar to Gupta [Gup94], except he defines essentially a single patch for the entire program, instead of one patch per changed file.

In our system, patches may be used to reflect nearly arbitrary changes to a file dynamically, particularly to its function and data definitions, and its type definitions. We look at each of these cases in turn, and then describe some of the limitations of our patch definition.

7.1.1 Changes to Code and Data

As an example, consider once again the file `main.pop` from Figure 6.2 (page 71), a subset of which is shown in Figure 7.1. The function `f` increments `num` to track the number of times it was called and returns the sum of the two fields of its argument, which has type `t`. Suppose we modify `f` to return the product of its arguments, creating a new version `main.pop`*(2)*. The dynamic patch that converts `main.pop` to `main.pop`*(2)* is shown in Figure 7.2[1]. The state transformer function `S` is trivial: it copies the existing value of `num` in the old version f to the `num` variable in the new version f'. In general, arbitrary transformations are possible.

[1]This figure illustrates an abstract notion of a patch; the actual syntax for our implementation is presented in §7.3.1.

new version `main.pop`*(2)*:

```
static int num = 0;
struct t {
  int a; int b;
}
int f (t T) {
  num++;
  return T.a * T.b;
}
```

state transformer S

```
void S () {
  main.pop(2)::num =
    main.pop::num;
}
```

Figure 7.2: Dynamic patch for `main.pop`: $(\mathtt{main.pop}(\mathit{2}), S)$

Stub Functions

Because patches are applied to individual files, rather than whole programs, there is a problem in applying a single patch if exported code or data *changes type*: existing referers of changed items will access them at the old (now incorrect) type. For example, consider a new version of `f` that adds a new argument to `f` (call the new file version `main.pop`*(3)*); existing callers in different modules will still call `f` with a single argument, constituting a type error. In general, this problem can be 'corrected' by simultaneously applying patches to correct the callers. In most situations, it would not make sense to do otherwise; in transitioning from one version of a program to another, it only makes sense to patch all of the files that changed.

On the other hand, in some situations a changed file's callers cannot be changed. For example, in some proposed *active networks*, multiple parties may download code into routers to customize packet processing. In this case, one party may wish to update his code, but cannot update the callers of that code belonging to other parties. As another example, we may wish to break down a large update into several smaller updates, so that the process of updating the system is less disruptive; Lee [Lee83] considers a way of methodically breaking down larger updates into smaller ones. For these situations, we allow patches to include *stub functions*. A stub function has the same type as the old version, and is interposed between old callers and new definitions to get the types right.

The patch for `main.pop` to `main.pop`*(3)* is $(\mathtt{main.pop}(\mathit{3}), S, \{\mathtt{f} \rightarrow \mathtt{stub\_f}\})$, shown in Figure 7.3. The third part of the patch is a mapping between functions in `main.pop` and the stub functions that should replace them. In this case, the stub function for `f`, called `stub_f`, simply inserts a default value for the new argument to `f`. Existing callers of `f` will now call `stub_f`, while code loaded later will link against the new `f`, at the new type.

Even when all patches are applied at once and/or the types of updated functions do not change, stub functions can be used to perform incremental, transitional computation. For example, suppose we have an event-based system. Every time the program is prepared to process an event, it calls the function `get_next_event`, which returns a value of type `event`. Now, say we wish to update the way that events are gathered; that is, we wish to change the implementation of `get_next_event` to perform some additional, or different new version main.pop*(3)*:

```
static int num = 0;
struct t {
  int a; int b;
}
int f (t T, int x) {
  num++;
  return T.a * T.b * x;
}
```

state transformer S

```
void S () {
  main.pop(3)::num =
    main.pop::num;
}
int stub_f(t T) {
  return f(T,0);
}
```

Figure 7.3: Dynamic patch for `main.pop`: $(\texttt{main.pop}(3), S, \{\texttt{f} \rightarrow \texttt{stub\_f}\})$

operations. Assume that `get_next_event` keeps a queue of events waiting to be processed.

We could write a patch for this change in one of two ways. We could write a state transformer function S to copy the contents of the old queue of events for use by the new `get_next_event`. Alternatively, we could use a stub function to retrieve the queued events using the old function, and then switch to using the new one when no old events remain, roughly:

```
static bool got_old_events = false;
event stub_get_next_event() {
  if (!got_old_events)
    event e = Old::get_next_event(); /* old version */
    if (e != null) return e;
    else got_old_events = true;
  }
  return get_next_event();           /* new version */
}
```

Here we differentiate between the old version of `get_next_event` and the new one by prepending the old version with `Old::`. Using a stub in this way deals with the old state incrementally, as opposed to performing all transitional computation at patch-time. The obvious benefit is that the pause at load-time due to state transformation is less; this may be critical to reduce service outage when transforming large amounts of state.

On the other hand, existing code will always first call the stub function, even after all of the old state has been processed, imposing extra overhead. To avoid this cost, we could define special syntax to allow a stub function to update its clients to point the actual function, rather than the stub, once the state transformation is complete. The above code would change to something like:

new version `main.pop`(4_1):

```
static int num = 0;
struct t {
  int a; int b; int c;
}
int f (t T) {
  num++;
  return T.a * T.b * T.c;
}
```

state transformer S

```
void S () {
  main.pop(4)::num =
    main.pop::num;
}
int stub_f(Old::t T) {
  t T2 = new t(T.a,T.b,0);
  return f(T2);
}
```

Figure 7.4: Dynamic patch for `main.pop`: $(\texttt{main.pop}(4_1), S, \{\texttt{f} \rightarrow \texttt{stub\_f}\})$

```
event stub_get_next_event() {
  event e = Old::get_next_event(); /* old version */
  if (e != null) return e;
  else {
    RELINK("get_next_event",get_next_event);
    return get_next_event();       /* new version */
  }
}
```

The call to `RELINK` would cause existing clients to call the actual function on subsequent calls. How this construct would be implemented would depend on the mechanisms used to realize dynamic updating in general, which we discuss later in this chapter.

There is no obvious construct for data analogous to stubs. Thus, if a patch changes the type of some global variable, then all the functions in the running program that refer to that variable must also be changed. In the case that global data is declared `static`, no additional files are involved since only the functions in the local file itself are affected. When data is exported, however, all other files that refer to that data must be updated.

7.1.2 Changes to Type Definitions

Finally, changes may also occur to type definitions, which declare the named types of the program. In the example above, rather than change the function `f` to have two arguments, we could have changed the definition of `t` to include a third field. In this case, how we choose to express this change in the patch depends on our implementation strategy. We now present our technique of choice; we consider another approach in the next subsection.

A simple way to express a changed type is to syntactically differentiate between the type's old definition and its new one, so that we can manipulate data conforming to both types. A patch in this style is shown in Figure 7.4. Here the stub function `stub_f` takes an argument having the old type `t`, syntactically shown as having type `Old::t`. It then creates a value of type `t`, copying the existing fields from the argument, and assigning a 0 for the third field. Finally, the new `f` function is called with the newly created value.

This style of patch implies that the program may have values of both the old and new version of `t` during its execution. The benefit here is that dealing with changes to data due to changed type is completely in the programmer's control, and the implementation is not made more complex.

7.1.3 Limitations

Our notion of patch is designed to be flexible, but it has some limitations to ensure a simple implementation. In particular, the patch definition does not provide a means to deal with data on the stack, and its requirement that programmers manually handle existing data may sometimes prove burdensome; we consider each point in turn.

Transforming the Stack

The state transformation function S considers global state, including the heap and static data segment, but not the stack. As a result, there is no direct means for the programmer to transform data on (or pointed at from) the stack. In contrast, a system like Dynamic ML can automatically transform all data (having the changed abstract type) wherever it may be.

Preventing direct manipulation of the stack has two consequences. First, old code and data may, for a time, be active along with new code. This is advantageous in that there is no need to translate the return address of the running code to return instead to the new version of its caller. If the caller changed significantly, it would be difficult to decide where to return instead. On the other hand, having two versions of a function or data active in the program may lead to incorrect and/or unpredictable behavior. One way to handle multiple versions is to use stub functions. That is, when old code calls new functions, it will do so through the stub functions. These functions could attempt to ensure a consistent semantics, but admittedly determining reasonable behavior could quickly become quite complicated in even simple situations.

The second consequence is that the implementation burden is reduced. In particular, there is no need to support a general way of traversing the stack. Type-safe, runtime stack traversals would would require extra semantic information be available at runtime, such as that needed by a debugger. Adding such a mechanism in a system like TAL would also increase its trusted computing base.

In the end, we opted for the simpler implementation at the cost of some reduced flexibility. As we describe in §9.1.1, to avoid dealing with the stack at patch time, we constructed our application to only permit updates when the stack was essentially empty. This required slightly restructuring the application. There is reason to believe that systems like TAL may ultimately support stack-tracing, to allow type-safe implementation of debuggers, garbage collectors, thread schedulers, security checkers, *etc.* It would be interesting to revisit this issue at that point and experiment with possible approaches.

Transforming Existing Data

In supporting the updating of type definitions, we require the programmer to explicitly deal with data of the old type (either by ignoring it or converting it to the new type). While new version main.pop(4_2):

```
static int num = 0;
struct t {
  int a; int b; int c;
}
int f (t T) {
  num++;
  return T.a * T.b * T.c;
}
```

state transformer S

```
void S () {
  main.pop(4)::num =
    main.pop::num;
}
Old::t convert_t(t T) {
  t T2 = new t(T.a,T.b,0);
  return T2;
}
```

Figure 7.5: Alternative notion of dynamic patch for main.pop: $(\texttt{main.pop}(4_2), S, \{\}, \{\texttt{t} \rightarrow \texttt{convert\_t}\})$

simple and intuitive, this choice places an added burden on the programmer. An alternate semantics might allow only one version of a type's data to be present at any given time, as enforced by the *system*. In this case, we may instead include *type conversion functions* in the patch to indicate how data of the old type should be transformed to the new, and we leave it to the system to invoke these functions when needed. Using this semantics, our patch file might be like that shown in Figure 7.5.

In this case, there is no need to explicitly define a stub function (as we did in Figure 7.4. The idea is that the system will properly convert the data using `convert_t` as needed; to be general we would include an additional mapping in the patch file description to indicate which conversion function should be applied for each changed type.

This alternative approach is roughly the one employed by Dynamic ML [GKW97]. Its disadvantage is its greater burden on the implementation, since now data must be tagged with its type, and some means must exist for finding and converting the data, and dealing with erroneous conditions. Furthermore, a system-directed, automatic approach to converting old data may be too inflexible for certain applications. In particular, we may want to translate data differently depending on where the data lives (the stack, the heap, *etc.*) or in what part of the program it is used.

We favor programmer-directed data conversion because it results in a simpler implementation, with a smaller trusted computing base. Furthermore, our experience with FlashEd (see §9) has been that manual conversion is not difficult. However, if experience were to show that the benefits of automated conversion outweigh its implementation complexity, we could easily add it to our approach. That is, the techniques we present in this dissertation complement those used to implement system-directed conversion.

In summary, our preferred definition of dynamic patch is $(f, S, \textit{stub set})$, where f is the new code, S is the state transformer function (acting on the heap and static data), and *stub set* is a set of mappings from functions to their corresponding stubs. This notion is both flexible and easy to use, as it can reflect nearly arbitrary changes between two files, and the code germane to dynamic patching is cleanly separated (as the state transformer and stubs) from the new code. The fact that we can generate patches mostly automatically, as described in the next chapter, further bolsters our claims of ease of use.

7.2 Enabling Dynamic Patches

Before going into the specific details of our implementation of dynamic patches, we first present the motivation for, and an overview of, our approach. We begin by presenting our general implementation strategy. Next, we consider possible mechanisms that we could use in the context of our strategy. Finally, we evaluate these mechanisms and settle on our overall approach. The details of our implementation are presented in the next section.

Our general strategy is to realize dynamic updating by dynamic linking. That is, we dynamically link one or more dynamic patches into the running program, transform the state in the program, and finally transition to using the new code. To do this, we must solve two problems:

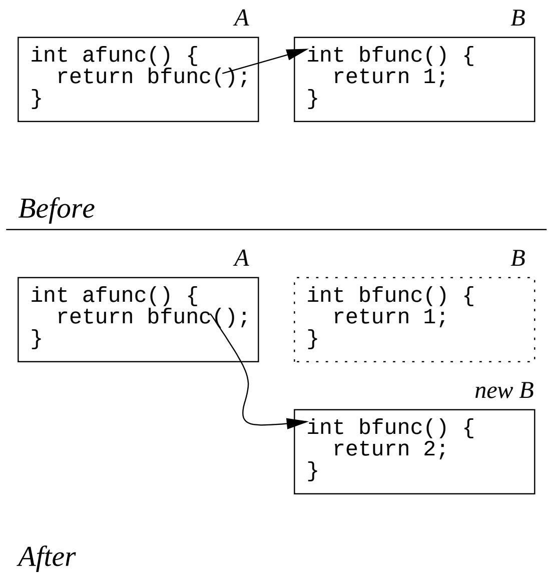
1. Given the concrete syntax for a set of dynamic patches, we must compile these patches to loadable TAL files in order to dynamically link them into the program.

2. We must enable the program to be dynamically patched. That is, not only must the program be able to dynamically link the patches, but the existing code must be transitioned to use the new code in those patches. For example, callers of existing functions must be redirected to use the new versions of those functions present in the patches. We term the process of enabling a files references to b redirected as enabling a file to be *updateable*.

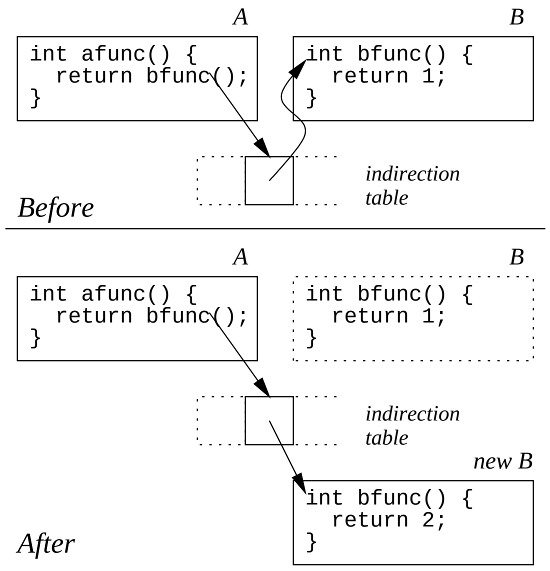
We solve these problems as follows; we consider the second problem first. As with dynamic linking, we enable updateability via special compilation in combination with some library routines. In particular, an updateable program must have each of its modules compiled to be updateable, and these modules must be linked with the updating library.

In determining how to solve the first problem, we observe that patches must also be updateable. That is, once the program has been patched, the code in the patch is now part of the program, and could itself be itself updated at a later time. Therefore, whatever mechanisms we use to realize updating must be applied not only to the statically linked modules of the program, but also to the dynamic patches themselves. In other words, dynamic patches must be both loadable *and* updateable, while statically linked modules are updateable only. Rather than compile patch files to TAL files directly, we simplify and modularize our implementation by first compiling patch files to Popcorn files, and then compiling these Popcorn files to be loadable and updateable. This way, we reuse the same compiler code needed for normal Popcorn files.

We defer discussing how we compile patch files to Popcorn files until the next section. For the remainder of this section, we evaluate mechanisms for enabling a Popcorn file to be updateable, considering tradeoffs between flexibility, efficiency, and ease of use. To understand the mechanisms needed, we consider what parts of the program could change as a result of a patch: the code, the data, and the type definitions. We start with mechanisms to enable updating code and data, and then mechanisms for updating type definitions.



(a) Code relinking



(b) Reference indirection

Figure 7.6: Two ways to update code and data

7.2.1 Code and Data Updates

Once a patch has been dynamically linked and the state has been transformed, existing function calls and data must be redirected to the stubs and new definitions in the patch. There are basically two ways to do this: either by *code relinking* or by *reference indirection*.

With code relinking, the rest of the program is relinked after loading a patch; as a result, all references to the old definitions will be redirected to refer to the new ones. This idea is illustrated in Figure 7.6(a), where function `afunc` is relinked to call the new version of function `bfunc`.

Reference indirection requires modules to be compiled so that references to other modules are indirected through a global indirection table. An update then consists of loading a patch and altering appropriate entries in the table to point to the patch. The analogue of the example in Figure 7.6(a) is shown in Figure 7.6(b).

Each approach has advantages and disadvantages. With relinking, the process of updating is *active*: the dynamic linker must go through the entirety of the program and 'fix up' any existing code to point to the new code. With reference indirection, updating is *passive*: the existing code is compiled to *notice* changes. As a result, the linker does not need to keep track of the existing code and simply makes changes to the table, but at the cost of an extra indirection to access definitions through the table. In both cases it is the responsibility of the state transformer function to convert data that contains pointers to changed function and data definitions. For example, if the program defines a table of function pointers, and some of the functions pointed to have changed, then the state transformer function must go through this table and fix the pointers to point to the new code. However, the reference indirection approach could be modified to allow function and data pointers to be updated automatically; we consider this possibility more below.

We have chosen to use code relinking because it has two main benefits: it avoids the extra indirection, which reduces overhead, and it is simple to implement, enhancing robustness. In particular, we implement code relinking by reusing the code that links a loaded module (with some modification), applying it to the existing program modules following the loading and linking of the patch. The only apparent burden is the need to keep track of the code to be able to relink it, but we must do this already, since we use the old code to resolve external references in loaded patches.

Ultimately, we could take a hybrid approach in which some elements are compiled to notice updates, and others must be relinked. One possibility that we have explored is to compile function pointers to have an extra indirection, while still requiring code references to be relinked. This would ease the requirement that the state transformer translate pointer data. We describe this idea in detail in §7.4.1.

7.2.2 Updating Type Definitions

We must also consider changes made to the definitions of named types in patched files. Again, there are basically two approaches we could take: *replacement* or *renaming.* With replacement, applying the patch *replaces* the existing notion of the type definition with a new one. For example, consider some module that defines type `t` as

```
struct t {
  int a;
}
```

A patch changes this definition to be

```
struct t {
  int a;
  int b;
}
```

To update this definition dynamically, in a sound manner, we must do two things:

1. *Update the type-checking context.* That is, we must type-check loaded files and patches using the new `t` rather than the old one. Therefore, the mapping between types and their definitions used by type-checker, called the *type-checking context*, must reflect the new definition.

2. *Update existing instances of* `t`. All data in the program that is ostensibly of type `t` must be compatible with the new `t`. Otherwise, we might pass a value having the old type `t` to a function expecting the new type and get incorrect behavior. Therefore, in the absence of subtype relationship between the old and new versions, we must convert any existing data of the old type (whether in the heap, stack, or static data area) to the new one. Furthermore, any code that makes use of the old type elsewhere in the program must itself be replaced. One exception is in the case that the type is abstract; then only the abstract data type code (ADT) must be replaced.

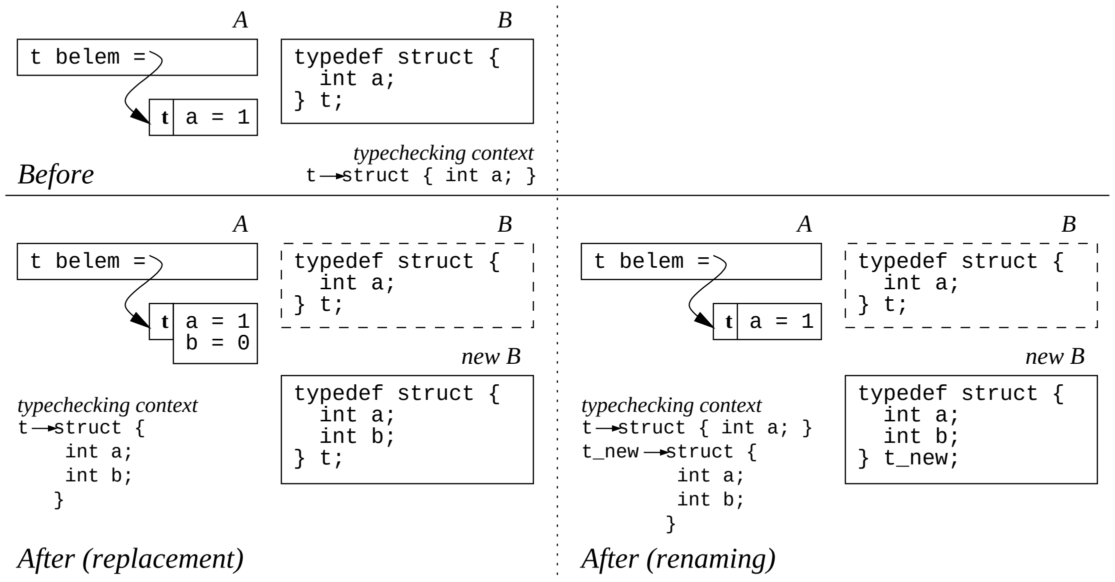


Figure 7.7: Two methods of updating type definitions: *replacement* and *renaming*

This approach is illustrated on the left side of Figure 7.7. Here, the result of updating the definition of `t` in module B is that the existing instance of `t belem` is converted to the new type, and the type-checking context is properly altered. If module B additionally defined code that made use of `belem`, then this code would have to be changed as well, to properly process `belem` at the new type.

The alternative approach is type renaming. Instead of allowing type definitions to be replaced, we maintain a fixed notion of a type definition, and rely on the compiler to define a new type that *logically* replaces the old one by syntactically *renaming* occurrences of the old name with the new one. Renaming is similar to the idea of α-conversion in scoped programming languages, in which a type definition can override a definition of the same name in a surrounding scope; the overriding type is renamed to avoid the clash.

As a consequence of renaming at compile-time, existing instances of the old type are left as they are when the patch is applied; the state transformer function and/or the stub functions in the patch can be used to convert old instances at update-time or later. The type-checking context retains its definition of the old type and adds a new one for the new type. This approach is shown in the right side of Figure 7.7. Now, the new version of B defines a different type `t_new` as the logical replacement of `t`. The existing instance of `t`, `belem`, is left as it is, and the type-checking context retains its definition of `t` and adds the new one for `t_new`. In effect, the programmer has been given the responsibility of converting `belem` to have type `t_new`, if necessary.

There are advantages to both approaches. Type replacement, in general, is quite flexible and easy to use: it maintains the identity of a type within the program but lets its definition change. The system updates the values of the changed type (perhaps using user-provided code), so long as the programmer has updated all of the modules that use that type. However, because the program has no notion of the old and new versions of the type, the system must ensure that it can (logically) convert all of the old types invisibly. This restriction prevents an update to a type while code in the program is using values of that type [GKW97]. In contrast, type renaming only allows the loading of new types to logically replace existing ones, placing more burden on the programmer to convert values from the old to new type. However, renaming provides more freedom in timing updates, since the program is 'aware' of both versions.

Implementing type renaming is quite simple, requiring no runtime support beyond dynamic linking. To be practical it does require a standard method for renaming type definitions at compile-time so that different developers do not choose clashing or inconsistent names, which would result in program errors. This problem can be solved by taking a cryptographic hash (*e.g.* using MD5 [OG97]) of the type's definition (including its name) to arrive at a consistent name. In contrast, type replacement requires a way to find and classify existing instances as having a given type, and a way to change them from the old version to the new. Furthermore, to ensure that type updates do not occur when code that uses them is active requires heavyweight mechanisms to track when modules are in use [FS91, Lee83, GJ93].

We favor the simpler type renaming approach over the more complex, though easier to use, type replacement approach. Type renaming is more likely to be correctly implemented because it is simple, and is more portable, relying on facilities available in type-safe dynamic linkers. In particular, we do not have to make any changes to TAL/Load to support renaming, while we would have to make extensive changes to both the TAL runtime and TAL/Load to meet all of the requirements of type replacement.

In our experience, renaming types at compile-time, and having multiple notions of a type in the program, has not proven problematic. Other approaches [MPG$^+$00] have cited runtime type dispatch operators (*e.g.* `instanceof` in Java) as a reason for performing type replacement, but we believe more study is needed to bring to light the problems of type renaming in such a context.

7.3 DLpop/update: A DLpop Supporting Updating

To implement the updating approach that we have thus far described, we modified our DLpop implementation (described in the previous chapter); the new version is called DLpop/update to differentiate the two. Because of the decisions we made in designing our updating approach, *no changes needed to be made to TAL/Load*, only to the compiler and DLpop library. This means that our trusted computing base remains unchanged in adding updating. Furthermore, changes to the compiler and library are themselves small.

We present the changes we made to DLpop to allow updating, mirroring the presentation of the previous chapter. We begin by describing the concrete syntax for a dynamic patch and how we compile patches to Popcorn code. Then we describe how we enable programs to be patched, in two parts. First, we describe the changes made to the DLpop library to perform relinking, and then explain how to compile programs to correctly participate in this process.

main3.patch:

```
implementation: main3.pop
interface: ifc_main3.pop
sharing: t
renaming:
```

ifc_main3.pop:

```
extern int main::Local::num;
static void _init() {
  New::num = main::Local::num;
}
int Stub::f (t v0) {
  return New::f(v0,0);
}
```

Figure 7.8: Patch description and interface code files for Figure 7.3 (page 91)

7.3.1 Patches

Our implementation of dynamic patches closely follows the abstract description of §7.1. The contents of a patch are described by a *patch description file* containing four parts: the implementation filename, the interface code filename, the shared type definitions, and the type definitions to rename. The first two parts describe the patch: its implementation in the first file, and the state transformer and stub functions in the second file. The final two parts are for type namespace bookkeeping. The shared type definitions are those types that the new file has in common with the old, while the changed definitions are in the renaming list, along with a new name to use for each. The compiler uses this information to syntactically replace occurrences of the old name with the new one.

To illustrate, the patch description file and interface code file for the abstract patch presented in Figure 7.3 (page 91) are shown in Figure 7.8. The implementation file is `main3.pop`, which contains the implementation code shown in Figure 7.3 (page 91). The interface code file, `ifc_main3.pop`, is shown in the figure. It defines the state transformer as `_init`, and the stub function for `f` as `Stub::f`.

There are a number of namespace conventions that we use within the interface code file. The interface code file may need to refer to different versions of the same variable, so we differentiate by a prefix. For some variable `x`, we may have:

- *The new version of* `x` (that is, `x` defined in the implementation file) has syntax `New::x`. For example, we see that the state transformer assigns to the current version of `num` by referring to it as `New::num`. Similarly, it calls the new version of `f` by referring to it as `New::f`.

- *The stub function for* `x` has syntax `Stub::x`. In the interface code file, we have defined `Stub::f` as the stub function for `f`.

- *The* `static` *variable* `x` from file *basename*`.pop` has syntax *basename*`::Local::x`. This convention only applies to locals defined in a previous version, or in some other file; local variables in the new implementation are prefixed with `New::`, as indicated above. In the example, we see a reference to the existing `num` variable, declared `static` in `main.pop`, using name `main::Local::num`.

main4.patch:

```
implementation: main4.pop
interface: ifc_main4.pop
sharing:
renaming:
  New::t=MD5(struct t {
               int a;
               int b;
               int c;
        })
```

ifc_main4.pop:

```
extern struct t { int a; int b; }
extern int main::Local::num;
static void _init() {
  New::num = main::Local::num;
}
int Stub::f (t v0) {
  New::t to = new New::t(v0.a,v0.b,0);
  return New::f(to);
}
```

Figure 7.9: Patch description and interface code files for Figure 7.4

- *The old version of* `x` has the syntax `Old::x`. A `static` variable *basename*`::Local::x` may also be prepended with `Old::`.
- Alternatively, an unprefixed `x` is also the old version of the variable `x`, if `x` is defined in the old implementation file. Otherwise, an unprefixed `x` refers the *newest* version of an externally defined variable. In particular, if that variable is updated by some other patch applied simultaneously with this one, `x` refers to the updated version.

As another example, Figure 7.9 shows the patch description and interface code files for the patch shown in Figure 7.4. In this case, the type `t` has changed in the new implementation, and so is no longer in the sharing list. Instead, the renaming list indicates the new name that `t` should have; this value is calculated from the MD5 hash of `t`'s new definition.[2] The stub function now converts existing data of (the old) type `t` to have type `New::t` before calling the new function `f`. Occurrences of the name `New::t` will be changed to *MD5(...)*, as indicated in the patch description file.

Compiling a Patch File to Popcorn

The patch file is prepared for compilation by first converting it into a normal Popcorn file. First, all definitions in the implementation file whose variables are in the sharing list are made into `extern`s, which will resolve to the old version's definitions at link time. Second, all of the defined variables (non-`extern`) in the implementation file are prefixed with `New::`. Third, the interface code file and the implementation file are concatenated together. Finally, all the mappings from the renaming list are applied to the file's type names. The results of applying this process to the patches shown in Figures 7.8 and 7.9 are shown in Figures 7.10 and 7.11, respectively.

There are some points worth highlighting. As a clarification about the interface code file namespace semantics, note that because the implementation file is concatenated with the

[2]The actual value is `N1Br_A6_Cqr3Nvf0zjX6hD7w_B_B`, but we put *MD5(...)* for readability and to indicate how the value is calculated; this will be our custom throughout the rest of the dissertation.

```
static int New::num = 0;
extern struct t {
  int a; int b;
}
int New::f (t T, int x) {
  New::num++;
  return (T.a * T.b * x);
}
extern int main::Local::num;
int Stub::f (t v0) {
  return (New::f(v0, 0));
}
static void _init () {
  New::num = main::Local::num;
}
```

Figure 7.10: Converting the patch file from Figure 7.8 into a Popcorn file. Changes or additions to the new implementation file are boxed.

interface code file during compilation, symbols defined in the implementation (prepended with `New::`) are implicitly available to the interface code. This is why, in the example in Figure 7.8, there is no need for an `extern` for `New::f` or `New::num`. In other words, when writing the interface code file, the programmer assumes access to the implementation file namespace.

Also, there are essentially two ways we could have compiled the state transformer function. The approach we took (call it the *copy-translate* approach) was for the new module to have its own copy of the updated state: the state transformer copies (and transforms) the old state to the new module. A slightly different approach (call it the *inherit* approach) would be for the new module to link against the state of the old code when the state has not changed type. While the *inherit* approach reduces copying and allows incremental updates, it does not support short-term rollback. We cover this point in more depth below.

7.3.2 DLpop/update Library

As with dynamic linking, we enable dynamic updating via special compilation, both for statically and dynamically linked files, in combination with a library implementing the updating API. We describe the implementation of API first, describing the 'protocol' that occurs between the library and the loaded file's `dyninit` function. In the next subsection, we describe how we compile files to participate in this protocol so that programs are correctly updated.

The DLpop/update API is the same as that of DLpop (see Figure 6.1 on page 69),

```
static int New::num = 0;
struct MD5(...) {
  int a; int b; int c;
}
int New::f (MD5(...) T) {
  New::num++;
  return (T.a * T.b * T.c);
}
extern struct t {
  int a; int b;
}
extern int main::Local::num;
static void _init () {
  New::num = main::Local::num;
}
int Stub::f (t v0) {
  MD5(...) to = new MD5(...)(v0.a,v0.b,0);
  return (New::f(to));
}
```

Figure 7.11: Converting the patch file from Figure 7.9 into a Popcorn file. Changes or additions to the new implementation file are boxed.

except in the semantics of `dlopen` and `dlopens`. In DLpop/update, if a module is loaded that has the same name as a module in the running program, then the existing module is replaced by the loaded one. The function `dlopens` is similarly changed.

Roughly speaking, the algorithm for `dlopen` in DLpop/update works as follows. Say we want to dynamically update some module, call it A:

1. Dynamically load the patch for A, provided as an argument to `dlopen`, via a call to load.

2. Create a hashtable for A's symbols and add it to the dynamic symbol table, along with a pointer to A's `dyninit` function. Unlike DLpop, all hashtables in the dynamic symbol table are stored along with the module's `dyninit` function, needed to perform relinking, described below. We leave the hashtable (and `dyninit` function) for the old version of A in dynamic symbol table, since we will need it later.

3. Link A by calling its `dyninit` function. As arguments, pass it a `lookup` function that looks for symbols in the entire dynamic symbol table (as in DLpop), an `update` function that adds symbols to the new hashtable, and `false` for the `no_init` flag (indicating that the `_init` function should be executed). The updating protocol expects the `dyninit` function to perform the following actions:

   - Look up any needed symbols using `lookup`.
   - Register A's symbols using the `update` function. As a result, the newly created hashtable will contain all of the up-to-date entries for the module A, as well as entries for any stub functions. So that old modules are properly relinked, `dyninit` should have registered any stub functions *first*, before overriding them with the new versions of those functions.
   - Execute the module's `_init` function. For a dynamic patch, this is the state transformation function.

4. If `dyninit` fails because an exception is thrown, then the new hashtable is removed from the dynamic symbol table, effectively rolling back the changes to revert to the old version. The exception is then rethrown to be handled by the caller of `dlopen`.

   This form of rollback is essentially the same as that of Dynamic ML [GKW97] or Argus. In our case, the key to rollback is that all global data is redefined in the patch file, as opposed to inherited from the existing version.[3] This way, the state transformer will only modify the new copy of the state, leaving the old copy untouched and thus safe to roll back to. Defining the data in the new file also assures that the old code is made garbage-collectible, as we describe below.

5. Relink the rest of the program. For every module other than A and the old version of A, `dyninit` is called, but with non-standard `lookup` and `update` functions. For `lookup`, lookups favor stubs rather than the new versions, so hashtable queries first look for an overridden stub function, otherwise returning the actual function if not

[3]Dynamic ML achieves a similar condition by using a two-space, copying garbage collector, and piggy-backing the state transformation process on top of garbage collection.

found. The compilation of dynamically updateable files assures that the `dyninit` function for code already in the program (*i.e.* old code) will only ever perform calls to `lookup` and not `update`. Therefore, for `update` we pass a function that does not change the dynamic symbol table but simply prints a warning that it was called.

The old version of A needs to be relinked as well, in case it is still active. For this case, the `lookup` function differs slightly from the one described above. If `lookup` finds the requested symbol in a new version's table, but at the wrong type, then it looks for the old version of that symbol in an older hashtable. This circumstance will only occur when the new version of a symbol changes type and does not, or cannot in the case of data, define a stub function. Because the old code is going to shortly be outmoded by the new code, we allow the code to use the old version of the symbol. By contrast, we do not allow relinking against old symbols for current code, effectively enforcing that current code always makes use of the most current data.

If an an exception is thrown by some `dyninit` function during relinking, *e.g.* because of a type error during symbol lookup, then all of the new hashtables are removed and the program is relinked again, restoring it to its previous state. This circumstance should never occur if the loaded patches are derived from a program that compiles statically.

6. Finally, the links in dynamic symbol table to the old A's hashtable should be made into *weak pointers*. Weak pointers do not keep data from being garbage collected if it is not reachable by some non-weak pointer elsewhere in the program. If an item pointed to by a weak pointer is collected, then the pointer is set to *null*, and a finalizer is called for the collected table. This finalizer fixes the links in the hashtable list for the previous and next elements (which were previously pointing to the collected table) to point to each other.

   Unfortunately, TAL does not currently support weak pointers (though its underlying garbage collector does), but adding them would be straightforward enough. Essentially, a 'weakness' qualifier could be added to the type system, so that pointers having this qualifier would require a null-check preceding every use.

   In lieu of implementing weak pointers, we could just leave all of the old tables in the list, but doing so does not accurately estimate the impact on the program heap of containing the patch code. Therefore, we decided to remove the old tables following an update, but to ensure via program construction that the removed code will not be active at the next update; we discuss how FlashEd was constructed in this regard in §9.1.1. If old code were to remain active, then it would need to be relinked; obviously, relinking cannot take place if the hashtable and `dyninit` for the old code is not in the list.

This procedure changes somewhat when dynamically linking multiple patches at the same time. In particular, linking occurs in three passes, as opposed to two passes for DLpop (see §6.2.1). In the first pass, all patches are dynamically loaded and their `dyninit` functions invoked; this first invocation is used to look up old symbols (*i.e.* those prepended with the `Old::` prefix) before they are potentially overwritten by calls to `update` during the latter two passes. These old variables are of use to the state transformer and stub functions appearing in the interface code file as we described above. The second two passes are like the first two passes in DLpop. That is, the second pass performs all of the `updates` for each module, and the third pass performs all of the remaining `lookup`s and calls the module's `_init` function. The process of relinking is the same as described above.

7.3.3 Compilation

In this subsection, we describe how we compile files to be updateable, fulfilling their role in the updating protocol described above. The translation for updateability is source-to-source, and is very similar to the translation for files participating in dynamic linking (see §6.2). The basic differences are:

1. *All* files, whether statically or dynamically linked, are compiled to have a GOT, and external references are indirected through that GOT. This allows statically linked files, including libraries, to be relinked following a dynamic update. In particular, a module is relinked by repopulating its GOT with new symbols from module updates.

2. The `dyninit` function must change to properly participate in the updating protocol:

    - As mentioned above, any stub functions should be registered before their corresponding replacement functions. This ensures that old files will relink against stub functions if they are present.
    - In addition to registering global symbols with `update`, `dyninit` should also register any `static` symbols. Doing so is not common practice in dynamic linking, since internal symbols are not of interest to (or are protected from) external files. However, to support dynamic updating, we need to allow the state transformer of a module's patch to have access to all global state, which includes `static` variables. As mentioned above, we prepend `static` variables with *filename*`::Local::` when storing them in the table; this avoids name clashes with static variables from other files having the same name.
    - As mentioned above, when the DLpop/update library dynamically links multiple patch files, it uses a three pass algorithm in which the first pass looks up old symbols (*i.e.* those prepended with the `Old::` prefix) before they are potentially overwritten during the later passes. Therefore, the first call to `dyninit` should only perform early lookups, while subsequent calls work as before.
    - Finally, once the state transformation function has been executed, any references to variables in replaced modules should be nulled out. This prevents the new module from having spurious pointers into the old module, thereby keeping it from being garbage-collected. This is explained in greater detail in §7.4.2.

To make these changes more concrete, we will look at some examples. We first look at how patch files are compiled to be both loadable and updateable, and then look at the how the updating translation differs for statically linked files.

```
static int cnt = 0;
extern struct t { int a; int b; }
static int g (int i) {
  t T = new t(cnt++, i);
  return GOT.f(T);
}
static void _init () {
  cnt = 5;
}
static exception exncon__2(string);
static int fn__3 (t a) {
  raise (new exncon__2("f"));
}
static struct GOT_t { int f (t); }
static GOT_t GOT = new GOT_t{f=fn__3};
static bool looked_up_old_flag = false;
static bool is_updated_flag = false;
static bool done_init_flag = false;
void dyninit_loadable<b,c> (a lookup <a>(b,string,<a>rep),
                            b lookup_closure,
                            void update <a>(c,string,<a>rep,a),
                            c update_closure,
                            bool no_init) {
  if (!looked_up_old_flag) {
    looked_up_old_flag = true;
    if (no_init) return;
  }
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "g", repterm@<int (int)>, g);
    update(update_closure,
           "loadable?Local?cnt", repterm@<*(int)>, &cnt);
  }
  GOT.f = lookup(lookup_closure, "f", repterm@<int (t)>);
  if (!no_init)
    if (!done_init_flag) {
      done_init_flag = true; _init();
    };
}
```

Figure 7.12: Compiling `loadable.pop` (from Figure 6.2, page 71) to be both updateable and loadable. Differences from Figure 6.3 (page 73) are boxed.

Dynamically Linked Files

To apply a patch to the running program, we first compile it to Popcorn, as described above, and then compile that file to be loadable and updateable. To show how compiling for loading and updating differs from compiling for just loading, Figure 7.12 illustrates the file `loadable.pop`, shown in Figure 6.2 (page 71), compiled to be both loadable and updateable. The parts in boxes indicate the changes in compilation for the file to be updateable, compared to the transformation shown for loading only, in Figure 6.3 (page 73).

Most of the file is the same as before. The only differences are reflected in the `dyninit` function, which has changed in two ways. First, there is now a preliminary code block used to look up functions that will be replaced by stubs. In this case, no stubs are present so no lookups take place. Notice that in the case that the `no_init` flag is set, the `dyninit` function will return immediately when within this code block. This is to correctly work with the three-pass algorithm, noted above. Here we want the `dyninit` function of each file we are linking to only execute this lookup code block and not do any updates; doing updates might overwrite symbol tables entries needed by early lookup calls in other patches being simultaneously applied. If we are only linking one file, the `no_init` flag will *not* be set, and so the rest of `dyninit` will execute. The second change is that the `dyninit` function now also calls `update` for each `static` variable; in this case for the variable `cnt`. We can see that the symbol name provided to `update` is not simply `cnt`, but `loadable?Local?cnt`.[4]

Now consider the translation for the patch file shown in 7.9. This file is first transformed into the Popcorn file shown in Figure 7.11, and then compiled to be loadable and updateable; the results of this compilation are shown in Figure 7.13. Changes from the code shown in Figure 7.11 are boxed. Some key things to notice in this figure are:

1. The early-lookup code queries the old address of `main?Local?num`. Both versions are used in the state transformation in the `_init` function, so the old address must be acquired before it is replaced with the new one in the call to `update`, just below.

2. The `update` function is used to export the new function and the stub function for `f`; the stub function is exported first, then the new version.

3. Following the call to `_init`, the GOT entry for the old `num` variable is "nulled out" by replacing it a dummy tuple `new (0)`. As a result, following initialization, the new code will not have any pointers into the old code.

Some additional implementation points are important. First, because we still compile code to use a global offset table, all external references require an extra indirection. But, as described in §6.4.2, we could change code to use runtime code generation to eliminate this indirection. Doing so would require changes to the relinking procedure, described below, since new code would have to be generated for all functions that used the updated symbols and then updated in the symbol table. As it is now, the addresses for existing symbols in the program are stable.

Second, the GOT only stores references to external symbols; references to locally-defined variables are direct. This means that only a subset of all function calls and data

[4]During translation to TAL, all occurrences of `::` are changed to `?`, so `loadable::Local::cnt` gets translated to `loadable?Local?cnt`.

```
static int New::num = 0;
extern struct t { int a; int b; }
static int New::f (t T, int x) {
  New::num++;
  return (T.a * T.b * x);
}
static int Stub::f (t v0) {
  return (New::f(v0, 0));
}
static void _init () {
  New::num = GOT.main__Local__num.1;
}
static struct GOT_t { *(int) main__Local__num; }
static GOT_t GOT = new GOT_t{main__Local__num=new (0)};
static bool looked_up_old_flag = false;
static bool is_updated_flag = false;
static bool done_init_flag = false;
void dyninit_main3<b,c> (a lookup <a>(b,string,<a>rep),
                         b lookup_closure,
                         void update <a>(c,string,<a>rep,a),
                         c update_closure,
                         bool no_init) {
  if (!looked_up_old_flag) {
    looked_up_old_flag = true;
    GOT.main1__Local__num =
      lookup(lookup_closure, "main1?Local?num", repterm@<*(int)>);
    if (no_init) return;
  }
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "main3?Local?num",
           repterm@<*(int)>, &New::num);
    update(update_closure, "f", repterm@<int (t)>, Stub::f);
    update(update_closure, "f", repterm@<int (t,int)>, New::f);
  }
  if (!no_init)
    if (!done_init_flag) {
      done_init_flag = true;
      _init(); GOT.main1__Local__num = new (0);
    };
}
```

Figure 7.13: Compiling the transformed patch file from Figure 7.11 to be loadable and updateable. Changes due to the dynamic transformation are boxed.

references made by a program will have to pay the indirection penalty. At the same time, this approach essentially requires updates to occur on the granularity of files, rather than individual data or procedures. Otherwise, the callers of an updated `static` procedure from the original file would still call the old version.

Statically Linked Files

Compiling statically linked files to be updateable differs somewhat from compiling them to participate in dynamic linking. As mentioned above, statically linked files must now be compiled just like dynamically loadable ones, defining a GOT, an early-lookup code block, and regular calls to `lookup`. This is because statically linked files must be relinked following an update, so the `dyninit` must have the same semantics. The file `main.pop` from Figure 6.2 (page 71) compiled to be updateable is shown in Figure 7.14; differences from the translation for dynamic linking, shown in Figure 6.4 (page 76), are boxed. Note that default functions need not be generated for the GOT; instead we can statically link in the actual functions needed. In the figure, we see that GOT is initialized as

```
static GOT_t GOT =
  new GOT_t{dlsym=&dlsym, dlopen=&dlopen, dlclose=&dlclose};
```

The symbols `dlsym`, `dlopen`, and `dlclose` will be resolved during static linking. However, they will be overwritten (with the same values) by the calls to `lookup` in `dyninit`, invoked at program startup. Overwriting these symbols is just an artifact of supporting relinking.

Another change is to the startup code generated to call `dyninit` for each statically linked file. Rather than just generate one call to `dlinit` for each statically linked module, we instead must generate three passes for linking, with each pass calling `dlinit` for each module. This mirrors the three-pass dynamic linking algorithm described above. If `dyninit` raises an exception during this process, it is caught by `dlinit`, which returns an error code to the caller. If, following the third pass, any call to `dlinit` returns an error code, then static linking has failed and the program halts.

7.4 Discussion

We have covered most of the details of our implementation of dynamic updating, but have deferred some discussion and a few details to this point. We first discuss how to properly update data that contains pointers to functions and data that are updated. During this discussion, we reveal the difficulty in updating exceptions and exception constructors. Next, we indicate how we ensure that old code is garbage collected. We finish by comparing our earlier implementation of dynamic updating by reference indirection to our current implementation.

7.4.1 Updating Pointers to Functions and Data

As mentioned in §7.2.1, data that contains pointers to changed definitions must be altered in the state transformer function, and is therefore the responsibility of the programmer. In this subsection, we look more closely at the process of updating pointerful data. Our

```
void pop_main () {
  handle_t h = GOT.dlopen("loadable");
  int g (int) = GOT.dlsym(h, "g", repterm@<int (int)>);
  g(3);
  GOT.dlclose(h);
}
static struct GOT_t {
  a dlsym <a>(handle_t,string,<a>rep);
  handle_t dlopen (string);
  void dlclose (handle_t);
}
static GOT_t GOT =
  new GOT_t{dlsym=&dlsym, dlopen=&dlopen, dlclose=&dlclose};
static bool looked_up_old__11_flag = false;
static bool is_updated_flag = false;
void dyninit_main<b,c> (a lookup <a>(b,string,<a>rep),
                        b lookup_closure,
                        void update <a>(c,string,<a>rep,a),
                        c upd_closure,
                        bool no_init) {
  if (!looked_up_old_flag) {
    looked_up_old_flag = true; if (no_init) return;
  }
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(upd_closure, "f", repterm@<int (t)>, f);
    update(upd_closure, "main?Local?num", repterm@<*(int)>, &num);
    update(upd_closure, "pop_main", repterm@<void ()>, pop_main);
  }
  GOT.dlsym = lookup(lookup_closure, "dlsym",
                     repterm@<a <a>(handle_t,string,<a>rep)>);
  GOT.dlopen =
    lookup(lookup_closure, "dlopen", repterm@<handle_t (string)>);
  GOT.dlclose =
    lookup(lookup_closure, "dlclose", repterm@<void (handle_t)>);
}
```

Figure 7.14: Compiling statically linked file `main.pop` from Figure 6.2 (page 71) to be updateable. Changes relative to the standard transformation for statically linked files (see Figure 6.4, page 76) are boxed. Some of the `main.pop` definitions are not shown due to space constraints.

```
int f (int x) {
  return x+1;
}
struct fnptr {
  int f(int);
}
static fnptr ptr_intfn = new fnptr(f);
int g (int x) {
  return ptr_intfn.f(x)+1;
}
void change_f(int intfn(int)) {
  ptr_intfn.f = intfn;
}
```

Figure 7.15: The file `fnptr.pop`, which uses function pointers.

discussion focuses solely on function pointers, but all of the ideas (and proposed solutions) apply equally to pointers to arbitrary data.

Consider code shown in Figure 7.15, for the file `fnptr.pop`. It defines a simple function `f`, and stores a pointer to that function in the variable `ptr_intfn`. Function `g` invokes `f` via `ptr_intfn` in calculating its result. Now consider what would happen if we were to patch `fnptr.pop` with a new version of `f` that subtracts, rather than adds, 1 to its argument. A first, but incorrect, attempt at a patch is shown in Figure 7.16.

There is no state transformer function: the new version of `ptr_intfn` is assigned the *new* version of `f` in the static initializer of the new code. However, if the program were ever to call `change_f` with a value other than `f`, then initializing the new `ptr_intfn` to `f` would be incorrect. A proper patch could be constructed if we knew that the program would only ever call `change_f` with a fixed number of values, and therefore `ptr_intfn` could only ever be `f` and those values, for instance two externally-defined functions `h` and `i`. Then we could construct a patch as shown in Figure 7.17 (using `fnptr2.pop` as above).

The state transformer function `_init` must check the old value of `ptr_intfn` to see if it matches one of the old values of `h` or `i` (designated `Old::h` and `Old::i`, respectively), and if so, assign the appropriate new version.[5] Otherwise the new version of `f` is assigned. As a defensive measure, rather than assuming it in the `else` case, we could have tested that indeed the old version of `ptr_intfn` contains the old version `f` before assigning the new version, throwing an exception otherwise. This exception would be caught by the dynamic linking code, and all of the changes would be rolled back, as described in §7.3.2.

In our experience thus far, this approach has been adequate for dealing with function pointers. In particular, we have written patches much like this in our dynamically updateable webserver, FlashEd, described in Chapter 9. However, this approach does have some drawbacks. First, we must know all of the possible values a function pointer could have to

[5]Note that the if `h` or `i` has not changed as part of a simultaneously applied patch, then `Old::h == h` and `Old::i == i`.

fnptr2.patch:

```
implementation: fnptr2.pop
interface:
sharing:
renaming:
```

fnptr2.pop:

```
int f (int x) {
  return x-1;
}
struct fnptr {
  int f(int);
};
static fnptr ptr_intfn = new fnptr(f);
int g (int x) {
  return ptr_intfn.f(x)+1;
}
void change_f(int intfn(int)) {
  ptr_intfn.f = intfn;
}
```

Figure 7.16: A first attempt at patching the code in Figure 7.15

reassign it the appropriate value. Second, pointers may be stored inside of abstract data, and thus inaccessible. Third, if we change a function definition in one file, we may have to update the state stored in another file, perhaps a file that has not (otherwise) changed. Finally, there may be function pointers to older versions on the stack, not accessible to the state transformer function. We consider each of these points in turn.

Function Pointers with Unknown Value

We imagine that in many cases it will be feasible to determine all of the possible values that a function pointer might have during a program's execution. In particular, either a by-hand source-code analysis, or a more sophisticated dataflow analysis could conservatively determine the possible values in global variables. However, making this determination *a priori* is impossible for programs for which not all of the source-code is available, as is the case with extensible systems. For example, the Linux kernel allows dynamically loading *modules* that implement protocol handlers, where these modules install function pointers into a protocol handler list. In such a system, a suitable state transformer function cannot be written (or can only with difficulty) without some help from the system.

One way to help is for DLpop/update keep track of the changes made to symbols during linking, and to provide a function for use in the state transformer functions that will map between the changed values:

```
b updated_sym<a,b>(a oldval, <a>rep oldtyp, <b>rep newtyp);
```

That is, `updated_sym` is given as arguments the global variable that may contain an updated value, along with the type representation for this variable, and the type representation for the expected type of the new version. If `oldval` is found to be in the updated list, fnptr2.patch:

```
implementation: fnptr2.pop
interface: fnptr2_patch.patch
sharing:
renaming:
```

fnptr2_patch.pop:

```
extern struct fnptr { int f(int); }
extern int Old::h(int);
extern int Old::i(int);
extern fnptr Old::ptr_intfn;
extern int h(int);
extern int i(int);
static void _init() {
  if (ptr_intfn.f == Old::h)
    New::ptr_intfn.f = h;
  else if (ptr_intfn.f == Old::i)
    New::ptr_intfn.f = i;
  else
    New::ptr_intfn.f = f;
}
```

Figure 7.17: A patch for `fnptr.pop`

then it will return the new version of it, provided the new version has the type specified. In our example above, the state transformer would change to become:

```
static void _init () {
  New::ptr_intfn.f =
    updated_sym(ptr_intfn.f, repterm@<int (int)>, repterm@<int (int)>);
}
```

The function `updated_sym` goes through the update list looking to see if the value stored in `ptr_intfn.f` is a function that has changed due to an update, and returns the new value. If, for example, `ptr_intfn.f` contained `Old::f` then the new value `f` would be returned. If `updated_sym` does not find that the old value has been updated, it attempts to cast `oldval` to have the new type, and returns this instead. If `oldtyp` $\neq$ `newtyp` then this cast operation will fail, implying that no new version for the changed function has been defined at the given type and some programmer error has occurred; we throw an exception to signal this. If a stub function has been defined at the given type, then it will be returned.

Using `updated_sym`, it is possible to update state containing pointers without knowing all possible values for those pointers in advance. The only thing that must be indicated in advance is the particular global variable that contains pointers. Furthermore, even in the case that all possible values for a pointer are known in advance, `updated_sym` makes it easier to write state transformer functions: compare the one-line function above with lengthy `if-then-else` test in the previous version.

We have implemented `updated_sym` and the changes required to correctly construct the updating list during linking in the DLpop/update library. To accommodate stub functions properly, the updating list to maps the original version of a function to both its stub and new version, but prefers the *stub* in the case that the new version has not changed type. In practice, using updated_sym is quite slow, due to the fact that the update list is potentially very long, and each comparison requires runtime checks for the call to checked_cast. Therefore, we are better off constructing an if-then-else transformation as opposed to using updated_sym if we know all possible values in advance.

Pointers in Abstract Data

TAL (and Popcorn) allows the definition of abstract data in two ways: with existential types [MP88], and with a module-level abstraction mechanism, similar to ML's use of opaque types in signatures [HL94, Ler94]. Using either mechanism, data can be constructed that contains pointers to updateable data, but because the abstract data's implementation is hidden, it becomes difficult to identify and update these pointers in the state transformer function. Note that because exceptions are implemented using existential types, updating exceptions with pointerful data components is similarly difficult.

In some cases, a sufficiently rich interface to the abstract data may avoid these problems. For example, the Popcorn Hashtable library implements hashtables with the abstract type table. Say a module A contains some global data of type Hashtable::table, and that this table stores function pointers. Because table is abstract, the new version of A cannot directly deconstruct the table to update the pointers. However, the Hashtable module defines an iterator function

```
void iter<a,b>(void f(a,b), <a,b>table t);
```

The function iter, iterates over the entire contents of the table and invokes the function f on each of its key/data pairs. If either the key or the data contained pointers, the new implementation of A could invoke iter to incrementally create a new hashtable whose data pointed to the new values.

A more general way to solve the problem is to use reference indirection. In particular, each function pointer in the abstract data is compiled to contain an extra level of indirection so that when data in the dynamic symbol table is updated, the update is automatically reflected in the abstract data. We explore this idea more deeply in Section 11.1.

Even reference indirection may be insufficient in some cases. For example, exceptions are implemented as a three-tuple in TAL, where the first argument is a pointer to the exception's tag (called the *exception constructor*). According to [Gle00], this pointer must be *read-only* to ensure type soundness. This implies that an update to an exception's constructor cannot be reflected in any extant exceptions. It may be that this restriction can be relaxed if certain circumstances are met; we plan to explore this in future work. One way to deal with this problem now is to update exception constructors by *renaming*, just like we do with types.

Updating Global Variables in Other Files

While updated_sym makes it possible to write state transformers that work independent of the values that a function pointer may have, there is more to do. In particular, if a module A stores a pointer to a function f in some other module B, and then B is updated to contain a new definition of f, then this change will not be reflected in A's global variable.

The reason is that no state transformer has been run for `A`'s code that might change this variable, since `A` did not change. Furthermore, `B` cannot necessarily run a transformer for `A`'s state because it may not be permitted access, or may not even know that `A` exists.

A reasonable approach to this problem is to designate for each module a special `re_init` function to be called every time the system performs an update. That is, when relinking the old modules, this function would be called from `dyninit`. We would use `re_init` to call `updated_sym` as shown above for all relevant global variables.

Updating Function Pointers on the Stack

The approach that we have outlined above allows a state transformer function to properly alter static data (or heap-allocated data reachable from static data) containing function pointers following an update. But pointers to functions may be stored on the stack (or heap-allocated data reachable from the stack) as arguments to functions; our approach does not apply to data on the stack simply because the stack is not first-class, directly manipulable by the state transformer function. How, then, do we deal with stack-allocated function pointers?

This question is part of a larger question concerning the *timing* of dynamic updates. That is, at what moments during program execution may a dynamic update legally take place? For instance, if a piece of code can be updated while it is running, then there will be a return address on the stack for the old code, which is akin to having a function pointer on the stack. In our system, we take the following approach with respect to update timing: programmers must construct their programs such that existing state on the stack can be safely ignored by the state transformer. We justify this position at length in §8.2.

7.4.2 Loaded Code and Garbage Collection

Because programs that require dynamic updating will run for quite a long time and potentially need many updates, we must ensure that old code is garbage-collected by the system once it is no longer needed. Otherwise, the memory footprint of the system could grow unreasonably large.[6] The problem of ensuring garbage collection is one of reachability: when a module is no longer needed by the program, it must not be reachable from datastructures that are still in use.

A loaded module may be referenced in two ways: either through the entries in the dynamic symbol table, or directly. In the first case, when a module is loaded, its symbols are stored in a hashtable as part of the dynamic symbol table. These symbols are mapped to pointers into the loaded module itself. As long as the hashtable for the module is within the dynamic symbol table, the module itself will be reachable.

In the second case, a module may refer *directly* to symbols in another module, rather than through the dynamic symbol table. This occurs in two ways: when the GOT of one module points to symbols in another, and when a new version of module refers to the corresponding symbols in its old version within the state transformer or stub functions. For example, in the patch shown in Figure 7.9, the `_init` function assigns `main::Local::num`

[6]Although with modern virtual memory systems, unreferenced memory should get paged out and pose no performance problem.

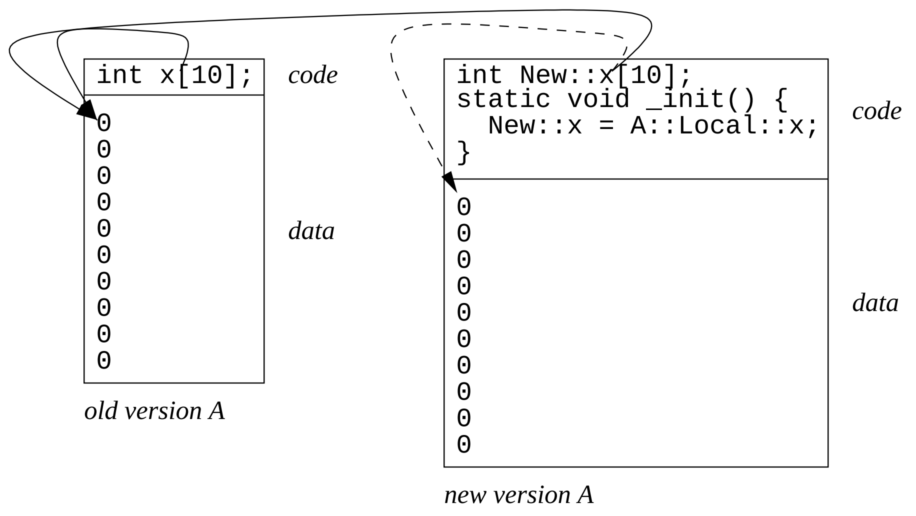


Figure 7.18: Copying an array by reference during a dynamic update

from the old version to `New::num`, the corresponding variable in the new version. When the patch file is translated to be dynamically loadable and updateable, as shown in Figure 7.13, the GOT entry for this old variable is filled in the early lookup block in `dyninit`.

Once the old version of a module is no longer needed, following an update, its hashtable is removed from the dynamic symbol table, meaning that the old module is no longer reachable from the dynamic symbol table. Furthermore, the code that remains in the program will have been relinked, so their symbol hashtables will point into the new code, rather than the old. The only remaining references to the old code will then be from the new code's GOT, in the case that old symbols were referenced in the new `_init` or stub functions. In this case of `_init`, we can null out these references once we have executed the `_init` function from `dyninit`. Figure 7.13 shows that following the call to `_init`, the GOT entry value for `main::Local::num` is replaced with a dummy tuple `new (0)`. In this case, nulling the GOT entry is sound because (1) the symbol `main::Local::num` is only ever referenced in the `_init` function, not the rest of the program; (2) the `_init` function will never be called again by the new file itself; and (3) it will never be called by code outside the new file, since the `_init` function is not exported by a call to `update`. If any of these conditions were untrue, then the nulling operation would be disallowed, and the compiler would issue a warning.

A similar nulling operation could take place if we were to allow stub functions to replace themselves when they are no longer needed, as described in §7.1.1. In particular, at the time the stub is replaced, the old module's symbols it used may be nulled in the GOT, as long as they are not in use by any other stubs. Making this determination may not be possible at compile-time, but may depend on what other stubs have been replaced. This would require a dynamic counter on each stub-referenced symbol.

Old code can be referenced directly by new code in one other way: an old variable can refer to data stored in the static data segment of the old module, and this variable can be 'copied' into a variable in the new module. This situation in shown in Figure 7.18.

Popcorn arrays are defined as pointers to a contiguous area of memory. In the example, the old version of a module `A` defines an array `x` containing ten elements, which are default-initialized. The Popcorn compiler allocates the contents of default-initialized, global arrays in the static data area; the array `x` is set to point to this allocated buffer. When the new version of module A is loaded, its state transformer function copies the contents of the old array `x` to its new version of `x`. This causes the new variable to point to the data in the old code. As a result, the entire old module is kept alive because its data is still in use by the new module.

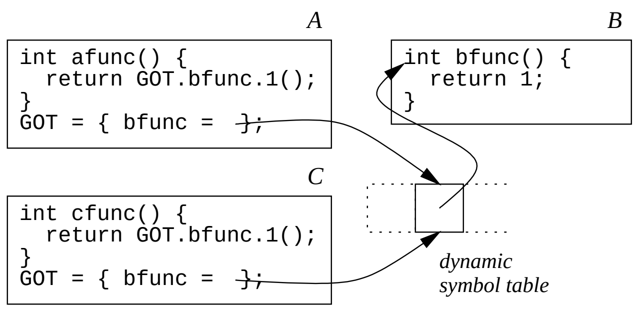
There are three ways around this problem. First, we could change the state transformer to instead copy the contents of the old array, one by one, into the contents of the new array. This is a reasonable approach as long as the old array has not been aliased to another variable; if it were, the alias would have to be redirected to the new version (*e.g.* by using technology described in §7.4.1, above). Second, we could change our implementation of `load` to separately allocate the code and the data segment of each module so that references to old data would not prevent old code from being reachable. Even better, we could separately allocate each static data definition, allowing garbage collection on a finer granularity (although this only reduces the impact of the problem without solving it). Third, we could avoid using statically allocated data altogether, and instead perform all initialization at start-time or load-time.

In practice so far, we have used a combination of the first and third approaches. We have observed that statically allocating large chunks of data does not make sense when we expect to patch the program, simply because the data will have to be copied between the old and new versions. On the other hand, *exception constructors* must be aliased between versions in order to maintain consistent semantics (for reasons described above). An exception constructor is essentially a unique tag that identifies a particular kind of exception. When an exception is caught, the exception's tag is examined to determine the exception's identity. So that exceptions maintain consistent identity between versions, this tag cannot change. Therefore, if a module defines an exception constructor, when a new version of the module is loaded, the new version must refer to the old version's constructor; but this keeps the entirety of the old code alive since it was allocated as a single block. As a result, performing more fine-grained allocation of static data during loading seems to be a necessary future enhancement.

7.4.3 Updating by Reference Indirection

Before settling on relinking as the method of code and data updating, for reasons enumerated in §7.2.1, we also implemented updating by reference indirection. For purposes of comparison, we summarize the highlights of that implementation here.

With reference indirection, rather than relink the rest of the program following an update, we compile the program to *notice* the update automatically. This is made possible by indirecting each GOT entry to point to the corresponding entry in the dynamic symbol table. When a patch is dynamically linked, the dynamic symbol table entry is modified to point to the new definition. As a result, the caller now effectively points to the new version, as illustrated in Figure 7.19. Both `afunc` and `cfunc` indicate that the `bfunc` field from the GOT structure should be extracted (`GOT.bfunc`), dereferenced (`GOT.bfunc.1`),



```
int afunc() {
  return GOT.bfunc.1();
}
GOT = { bfunc =  };
```

```
int bfunc() {
  return 1;
}
```

```
int cfunc() {
  return GOT.bfunc.1();
}
GOT = { bfunc =  };
```

Figure 7.19: Indirection via the dynamic symbol table

and finally called (`GOT.bfunc.1()`). This approach adds a level of indirection above that of relinking alone (two, instead of one).

To implement this approach, we must make a number of changes, both to file compilation and to the DLpop/update library. First, we change each module hashtable entry, as shown for module B in Figure 7.20; the entry in our current approach is shown in the left, and the modified form is on the right. In the indirection approach, an extra indirection is added to the module's symbol table so that rather than storing a symbol's value in the entry, we store a pointer to that value, as a one-tuple, instead. This makes rollback simpler, as we describe below. When B is updated, a new hashtable is added to the dynamic symbol table, as in DLpop/update. However, when a new or stub version of a symbol is added to the new table, it is made to share the old version's one-tuple (assuming the symbol has not changed type), and that one-tuple is changed to point to the new entry. This is shown in Figure 7.21. When the new version of a symbol changes type, an entirely new entry is added to the new hashtable, effectively overriding the old one.

Also shown in Figure 7.21 is that each time a one-tuple is changed, its old value is stored in a *rollback list*. Each entry in the rollback list contains two pointers, one to the changed one-tuple, and one to the old value (here a pointer to the old version of function `bfunc`). If an error occurs during linking or initialization, the new hashtables are removed, and all of the entries are processed to restore the old one-tuples. That is, for each rollback list entry, the one-tuple element is assigned the old value. If instead all linking operations complete normally, then the old hashtables are simply removed. Note that when using the indirection approach, there is no need to worry about weak pointers, since no relinking need occur.

The type of `lookup`, as passed to `dyninit`, must change so that the GOT of the loaded code can store a pointer to the symbol table entry, rather than a pointer to the symbol itself; *i.e.* `*(a)` is returned, rather than just `a`. In addition, all functions that access the table must pass a tupled type representation, even if they are only interested in the *contents* of the tuple. This includes the `update` function passed into `dyninit`, as well as the `dlsym` function in DLpop/update must be changed to the more unintuitive type

```
extern a dlsym<a>(handle_t h, string sym, <*(a)>rep typ);
```

The semantics of the function is the same; the difference is that the R-term provided as the third argument now requires an extra level of indirection in the provided type. So,

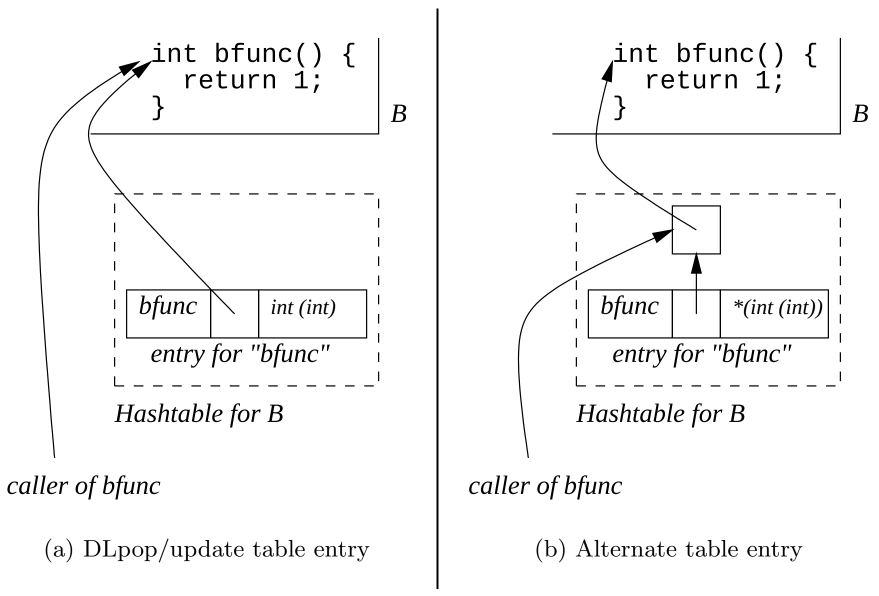


Figure 7.20: Implementation of per-module hashtable in dynamic symbol table, shown for module B (see Figure 7.19)

rather than

```
    int bar(int) = dlsym(h,"bar", repterm@<int(int)>);
```

as we saw in the example in Figure 6.2, we would have

```
    int bar(int) = dlsym(h,"bar", repterm@<*(int(int))>);
```

The reason for this can be readily seen when looking at the implementation for `dlsym` in DLpop/update:

```
a dlsym<a> (handle_t x, string name, <*(a)>rep typ) {
  <<string,entry>hashtable>list tail = null;
  if (x.mod_tab != null) {
    tail = x.mod_tab.tl;
  }
  return (find (x.mod_tab, tail, name, typ)).1;
}
```

The function `find`, as shown in Figure 6.5, requires that `typ` to be tupled so that the checked_cast operation works properly. If we could construct type representations dynamically (*i.e.*, type representations could be for non-closed types), we could construct a tupled type representation from a non-tupled one, and `dlsym` could retain its more intuitive type, having an implementation like:

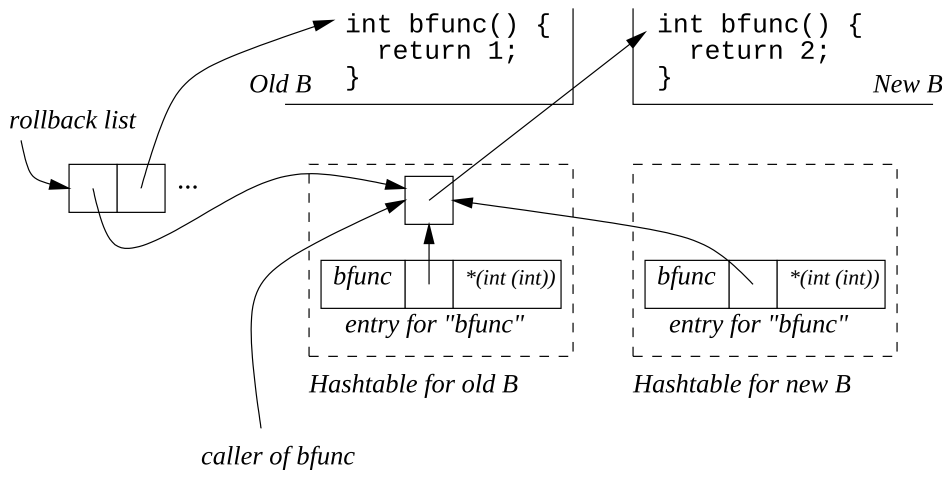


Figure 7.21: Dynamic symbol table and rollback list following a dynamic update of B

```
a dlsym<a> (handle_t x, string name, <a>rep typ) {
  <<string,entry>hashtable>list tail = null;
  if (x.mod_tab != null) {
    tail = x.mod_tab.tl;
  }
  return (find (x.mod_tab, tail, name, repterm@<*(a)>)).1;
}
```

We could effect a similar change by adding `typecase` and slightly altering the hashtable entry type and the `find` function. As we mentioned in §5.4.1, allowing dynamic construction and deconstruction of type representations adds a fair amount of complexity to the system; in fact, a complete implementation of λ_R in a system with named types is still a matter of research.

Evaluating our preferred approach of relinking in comparison to reference indirection, relinking is superior for a number of reasons. First, it has the performance benefit of only one, rather than two, indirections per external reference.[7] Second, the process of dynamic linking is simpler: there is no need for an extra rollback list, for an extra indirection in the dynamic symbol table, or for the type of `dyninit` to change. Furthermore, the presence of the `dyninit` function makes relinking simple and elegant. Third, the potential drawback of relinking, having to track all of the old code, is not really a drawback since all of the modules are tracked in the global symbol table anyway.

[7]In both cases, the use of runtime code generation could reduce the number of indirections by one, to zero and one for relinking and reference indirection, respectively.

Chapter 8

Building Updateable Systems

Building dynamically updateable software raises two concerns: how to enable programs to be dynamically patched, and how to best build and apply patches to those programs. In the previous chapter, we addressed the first concern, and in this chapter, we focus on the latter. In particular, we consider the questions of how to generate patches for updateable programs and how to ensure that they are applied at the correct time. In presenting answers to these questions, we complete the argument that our updating approach is flexible, robust, and easy to use; here, we support the latter two criteria. First, we consider two aspects of robustness:

1. A patch (or set of patches) that is *complete* addresses all of the changes made to the program from the old to new version. Completeness ensures that the programmer has not forgotten to address some aspect of the changed file. Patch completeness provides no guarantee that the changes are correct, but proving as much is, in general, undecidable.

   To ensure that patches are complete, we have written a tool that compares the old and new version of a file, identifies all of the relevant changes, and generates a patch that reflects those changes. In many cases, the tool is able to generate proper stubs and state transformer code; in the cases where this is too difficult, it leaves placeholders for the programmer.

2. The correctness of a patch cannot be determined independently of the time at which it is applied; in particular, choosing a poor time could cause race conditions during state transformation. We would like to provide a framework in which the programmer can determine that his patches are *well-timed*.

   Two past approaches [Lee83, FS91] have provided support for enforcing user-provided timing constraints at runtime. That is, the user can specify that an update should not be applied unless certain modules or functions are *inactive*. While in principle this adds flexibility as to properly timing dynamic updates, in practice this technology is difficult to use correctly and is hard to implement. Because the benefit of these mechanisms is largely unproven, and because they impose a potentially high overhead and implementation burden, we chose not to implement them. Instead, we require the programmer to construct the program to 'know' that it is updateable, in effect enforcing timing constraints at software construction time, rather than at runtime.

In addition to making the implementation more robust, the automated patch generator makes the system easier to use. After developing and testing of the next version of a

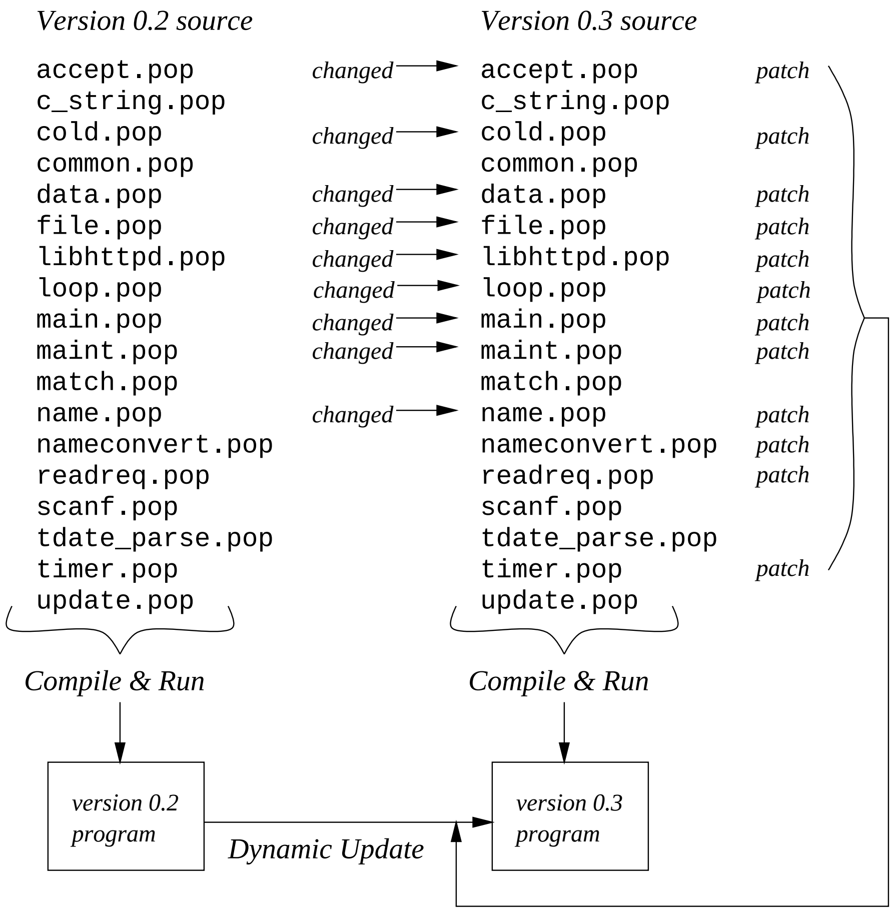


Figure 8.1: Building and maintaining an updateable program

program, the patch generator can quickly identify and (partially) generate patches for all the changed files, reducing the workload on the programmer.

This chapter is organized as follows. In the first section, we describe the process of developing updateable software, where the patch-building phase fits in, and how patches are generated. We focus most of our attention on the implementation of the automatic patch generator. In the second section, we look at the question of when, during program execution, patches should be applied to ensure they are executed properly. We consider two potential timing models, and justify our approach of requiring the programmer to construct updateable software to accept updates at appropriate times. We then describe a reasonable programming pattern for use in constructing updateable programs. In the next chapter, we describe how we applied the principles and techniques described in this chapter in constructing FlashEd, a dynamically updateable webserver.

8.1 Constructing Dynamic Patches

A typical way to develop software is as follows. Each version of a program is given a revision number, and the corresponding program source is associated with that revision, probably archived with revision control software like CVS [Fog99]. When changes need to be made, such as to fix bugs or add new features, the current version is modified to effect those changes. Once these changes have been thoroughly tested, the modified source is assigned a new revision number, compiled and tested, archived, and deployed. In short, to make a software change, we start with the current source, modify and test it, and then deploy the changed program as the new version.

Our approach to building dynamically updateable systems alters this process only slightly. Just as before, programmers make changes to the current sources, and then compile and test the result to create the new version. Once the new version is stable, rather than halting the existing version and then deploying the new one, patches are created that reflect the differences between the old and new versions of the software. In our system, much of these patch files can be generated automatically. The programmer only fills in the parts of the state transformer and stub functions that cannot be automatically generated. The patches are then dynamically applied to the old version of the software, thereby migrating it to the new version.

The development process is depicted in Figure 8.1. The current version 0.2 of some software[1] consists of a number of source files, which can be compiled and run. In moving to the next version, 0.3, many of these files are changed. The changes are tested by compiling and running the software and making sure it works. When testing is complete, patches are created for the changed files. In some cases, patches are needed for files whose contents did not change (*cf.* `nameconvert.pop`) due to changes in the definitions of types used by those files but defined elsewhere. These patches are then dynamically applied to the currently running version 0.2, resulting in a running program equivalent to version 0.3. The new version retains the state of the old version, and only negligibly interrupts (but does not cancel) service while the patches are applied. These are the benefits of dynamic updating; if we were to instead shut down the old version and restart with the new one, the running program's state would be lost, and any midstream processing would be forcibly canceled.

The maintenance process described here cleanly separates software development from patch development. Such a separation is possible because our notion of patch (and our implementation of it) is cleanly separated from the software itself. Furthermore, our patches can describe nearly arbitrary changes to the running program. In many other systems, patches are limited to certain forms, and so software development is similarly limited. For example, in Dynamic C++ classes [HG98], only changes to *instance* methods and data may be reflected dynamically; per-*class* (*i.e.* `static`) methods and data cannot evolve. As a result, the process of development is hampered by what may be expressible as a patch. For example, new static methods must be *added* to programmatically replace the old ones, but the old ones will remain, cluttering the code and obscuring its meaning.

On the other hand, there are times when writing a valid state transformer is not possible without further altering the source files. For example, it may be that the existing state respects one invariant, but the new version of that state respects another. If the old

[1] This is actually the source for the FlashEd updateable webserver, described in the next chapter.

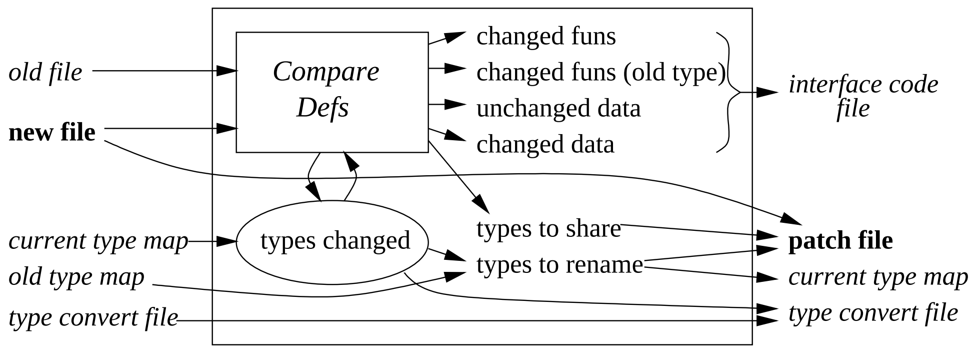


Figure 8.2: Structure of the automatic patch generator tool

state cannot be transformed to respect the new invariant, the new code could be changed to temporarily accept state having the old invariant, until it is no longer needed. In our experience, such changes are rare; we consider this issue more in the next chapter when describing our experience with FlashEd.

8.1.1 Automatic Patch Generation

A novel aspect of our approach is the mostly automatic generation of patch files. This feature was originally designed to make the system easier to use: it is very tedious to write state translation and stub functions by hand. It has also proven invaluable in minimizing human error, since it is less likely that a necessary state translation or stub function will be accidentally left out. As it turns out, a very simple syntactic comparison of files, informed by type information, can do a good job of identifying changes and partially generating patch code.[2] In this section, we explain the patch generation algorithm, and then present a couple of examples of its use.

The implementation of our patch generator is illustrated in Figure 8.2. As inputs, the patch generator takes the new file, the old file, a current *typename map*, the old file's *typename map*, and the current *type conversion* file. Only the new file is required, all other arguments are optional. The results of patch generation are the patch file, the interface code file, the updated typename map, and the updated type conversion file; if the new and old files differ then a patch file will always be generated, but the other outputs are generated only if needed. Descriptions of these patch file types may be found in §7.3.1.

Patch generation is broken into two stages, *identification* and *generation*. The identification phase is shown in the figure as the *Compare Defs* box. It takes the old and new files as inputs, along with a set of named types that are known to have changed. The set starts off as empty, or may be initialized by the contents of the current *typename map* file; this file is explained in more detail below. The algorithm works as follows. First, the old and new files are parsed and type-checked. Then, for each definition in the new file, the corresponding definition is looked up by name in the old file. In the case of type definitions (*i.e.* `struct` or `union` declarations), the bodies of the definition are compared;

[2]Note that our patch generator is similar in spirit to the *transformGen* database schema evolution system developed by Garland *et. al* [GKS86].

if found different, the name of the type is added to the set of changed types. In the case of value declarations, the bodies are also compared syntactically, taking into account the differences in type definitions; in particular, the syntax of a function may remain the same from the old to the new version, but the function has actually changed if a type definition mentioned in the body has changed. Furthermore, a function is considered to have changed if it references static data or functions that have themselves changed; the reason for this is explained shortly.

The results of the identification phase are a number of sets that describe the differences between the files. These sets are (1) the functions that changed, (2) the functions that changed but retained their old type, (3) the data declarations that *did not* change, (4) the data declarations that changed, (5) the named types that *did not* change, and finally (6) the named types that did change. These sets are used, along with some of the inputs, to generate the the patch file and some supporting files, including the interface code file, the type conversion file, and the typename map file. We cover each of these in turn.

Interface code file First, an interface code file is generated. If the old file contained any global data, then a state transformer (*i.e.* `_init`) function will be generated. For all data (global variables) that were unchanged, other than exception constructors (for reasons explained in the last chapter in §7.4.1), an assignment statement is created from the old to the new versions, like the one for `num` in Figure 7.2, to propagate the state. For those global variables that have changed type, appropriate code is generated to convert the data from the old to the new type, when possible. This code is generated by inductively examining the types of the old and new version. For cases when the type being considered is neither an array type nor a tuple type, and the types of the two versions are the same, the corresponding data is simply copied. For arrays, a loop is generated to translate the elements piecewise; for tuples, one translation statement is generated per tuple element. If the type considered is a named type that has the same name for the old and new version but a different definition, then a call to an appropriate *type conversion function* is made to convert between the two. Type conversion functions are generated automatically as well, as described below. Some values of different type can be translated with a cast, for example from a `boolean` to an `int` or an `int` to a `float`. More complicated translations are also possible (such as tuples with added fields), but we have not implemented them. If translation code cannot be generated automatically, a comment is inserted in the `_init` function to indicate that it must be inserted by hand.

The patch generator also generates default stubs for functions that have changed type. Two basic modes are possible. In the simplest mode, the generator creates a function body having the old type, and inserts a statement that raises an exception. This mode is useful when all patches for the running program are to be applied simultaneously, in which case no stub functions should ever be invoked, so the exception signals an unexpected error. The second mode is to automatically generate a call to the new version of the function, first translating the arguments appropriately, as shown in Figure 7.4 (page 92). Because we have, to this point, only applied all patches simultaneously, we did not implement this mode, although it would be straightforward to do so by reapplying existing code. For those functions that have changed in content but not in type, a comment is included in the `_init` function, but no stub is generated, since the old callers' type is not affected.

Typename map file During the identification phase, the patch generator keeps track of any type definitions that have changed, and generates new names for these types. The new name is determined by taking the MD5 hash of the pretty-printed type definition (which includes the type name), meaning that the same definition will always generate the same name. This allows development of patches by multiple programmers without the worry of choosing incompatible type names.

The mapping from old to new name is stored in the *typename map* file. This file is read in as each patch is generated, so that the global fact that a type changed informs the local process of patch generation for a particular file. The updated map file is written out upon generation completion. Furthermore, the typename map file for the old version may also be consulted so that types that have changed name as a result of earlier patches are properly named in the current set of patches.

Type conversion file Finally, *type conversion functions* are constructed to the extent possible for data conversion from old to new versions of a named type, and vice versa. These are used by the state transformation and stub code, as mentioned above. All type conversion functions are stored in a separate file; this file is read in at the start and written out upon patch generation completion, with new conversions functions for changed types not already covered. The type conversion file is dynamically loaded into the running program along with the other patches for their use.

For `struct` types, each field with an unchanged type is copied; each field that is added is given a default value; and each field that has changed type is translated. In the case that a translation is not possible, a 'placeholder', consisting of the string `"XXX FILL"`, is left for the programmer to fill in the appropriate value. Currently we support translation between like types (*i.e.*, `int` and `float`), and `struct` and `union` types (by calling the appropriate type conversion function).

For `union` types, we deconstruct the value of the old union type, and by cases construct a value of the new type. If a field has been removed in the new version, then a `default` case will be needed to deal with values of this variant specially; we put a placeholder in the `default` branch mentioning the missing field. For a field that is unchanged, the value is simply reconstructed, using the tag from the new type. For a field that has changed type, the new value is reconstructed with a value translated to the new type. The translation similar to the one for `struct`'s, using a placeholder if translation is not possible. Note that both for `unions` and `structs`, if the data may be null then a null-check is added.

For type conversion functions, and for patch generation in general, much of the automation is built around definition names, and therefore is not as helpful when these names change. For instance, adding or removing fields from a structure definition, or changing the types of some fields, will be detected by the patch generator as a changed type, and it will generate the majority of a routine to convert between elements of the old type and the new one. However, if the new version changes its name, then the patch generator will think of this as a new type, as opposed to a modified version of an existing one, and therefore will not generate any conversion routines. This difficulty of changing names arises in other areas, in particular file synchronization (*e.g.* [TM96, BP98], *etc.*); we should be able to apply its solutions to our patch generation system. For example, some problems could be alleviated by permitting the user to inform the generator of *old* → *new* relationships old version `old_foo.pop`:

```
?struct t {
  int a; int b;
}
t someTs[];
int f (t T) {
  return T.a + T.b;
}
```

new version `new_foo.pop`:

```
?struct t {
  int a; int b; int c;
}
t someTs[];
int f (t T) {
  return T.a + T.b;
}
```

Figure 8.3: The old and new versions of example file `foo.pop`

between definitions having different names.

Example

To illustrate the patch generator in action, consider the following example. Figure 8.3 illustrates the old and new versions of some file `foo.pop`. The new version has changed in two ways: the type of the structure `t` has changed to include an additional field `c`, and as a result the function `f` has now changed type, since it takes a value of the new type `t`, rather than the old `t`. Providing these two files as input to the patch generator results in four output files, shown in Figure 8.4. They are the patch description file `new_foo.patch`, the interface code file `new_foo_patch.pop`, the typename map file `TYPENAME_MAP`, and a file containing the type conversion functions, `convert_patch.pop`.

The patch generator observes that the type `t` changed, so it generates a name for the new version of `t` from the MD5 hash of its definition. It stores this mapping in the `TYPENAME_MAP` and indicates it in the `renaming` list of the patch description file. The `TYPENAME_MAP` file should be used as input for other patches in the same program that reference `t`, so that even if those files do not change themselves, they will be considered to have changed since the definition of `t` is different.

The interface code file `new_foo_patch.pop` defines a state transformer function `_init` to translate the array `someTs`, and a stub function for `f`, since `f` now takes a value of the new type `t`. For the array translation, a loop is generated that piecewise translates the elements of the array by calling the type conversion function `t__old2new`; this function is defined in `convert_patch.pop`, explained below. The stub function simply raises an exception indicating that an existing caller has not been properly updated. Note that the array conversion is not entirely correct: the new version of the array `someTs` needs to be allocated before the copying can take place. Retaining more information during the identification phase concerning how globals are allocated, either statically or dynamically, would allow this translation to be more precise.

Finally, the file `convert_patch.pop` contains the type conversion functions for translating values to and from the old and new versions of `t`. As with interface code files, the old and new versions are differentiated by their prefix: new versions are prepended by `New::`, and old versions have no prefix. Note that in this case, a default value of 0 is generated for the added field; we could conceivably have inserted a comment to remind the user to use a more appropriate value if necessary. So that this file is properly compiled, we treat it as if it were a patch, referring to it from the patch description file `convert.patch`:

```
interface: convert_patch.pop
renaming:New::t=MD5(?struct t {
                            int a;
                            int b;
                            int c;
                        })
```

By indicating `convert_patch.pop` as the `interface` file, the namespace is dealt with properly. Furthermore, we specify the mapped name for the new version of `t` in the `renaming` list.

The function `t__old2new` translates an element from the old type `t` to the new one. When the function is called, a new value of type `t` is allocated and initialized with the fields it shares with the old value. Since the new `t` has an added field, the patch generator also inserts a default value for that field. The function `t__new2old` translates in the reverse direction, dropping the value in the new field. In general, functions *x*`__old2new` are useful in state transformation, while *x*`__new2old` functions are useful in stub functions, for returning an value of old type to an existing caller.

To illustrate how `union` types are translated in the type conversion file, consider the following example. Some existing file defines the `union` type `tree` for describing elements of a binary tree:

```
union tree {
  void Leaf;
  *(tree,tree) Node;
}
```

We choose to change this definition to allow arbitrary numbers of children, by using an array rather than a pair:

```
union tree {
  void Leaf;
  tree[] Node;
}
```

The conversion code generated for this change is:

```
extern union tree {
  void Leaf; *(tree,tree) Node;
}
extern union New::tree {
  void Leaf; New::tree Node [];
}
New::tree tree__old2new (tree from) {
  New::tree to;
  switch (from) {
    case Leaf:
      to = new New::tree.Leaf;
    case Node (x):
      to = new New::tree.Node("XXX FILL");
  }
  return (to);
}
tree tree__new2old (New::tree from) {
  tree to;
  switch (from) {
    case Leaf:
      to = new tree.Leaf;
    case Node (x):
      to = new tree.Node("XXX FILL");
  }
  return (to);
}
```

For both conversion functions, the argument is deconstructed and examined by cases. For the `Leaf` case, a new `Leaf` value is constructed for the corresponding type. For the `Node` case, the patch generator does not know how to automatically translate between arrays and tuples (though we could imagine reasonable translations), and so it leaves a 'placeholder' `"XXX FILL"` for the programmer to fill in the appropriate value.

The automatic patch generator is a key element of our implementation for two reasons. First, it greatly reduces the workload on the programmer, taking care of many of the tedious aspects of generating patch files. In our experience so far, the code generated requires few alterations; we present our experience in §9.2.2. Second, it guarantees that patches are *complete*: it identifies all of the changes between two versions of a file, and either generates the needed transition code, or leaves placeholders reminding the programmer to do so. Together, these two advantages improve the likelihood that a patch is complete, reducing the possibility for error and thus improving overall robustness.

8.2 When to Apply Patches

So far we have concentrated entirely on *how* dynamic updates can be realized, and *what* well-formed updates will consist of. However, an equally important question is *when* updates should be performed. To understand the question of timing, and why it is important, we consider two models of updating, the *interrupt model* and the *invoke model*. We explain that while most past dynamic updating approaches use the interrupt model, it makes determining appropriate update times no less difficult even though it ostensibly provides greater flexibility, and it requires greater implementation complexity than the invoke model, our model of choice.

new_foo.patch:

```
implementation: new_foo.pop
interface: new_foo_patch.pop
renaming:
  New::t=MD5(?struct t {
           int a;
           int b;
           int c;
         })
```

TYPENAME_MAP:

```
New::t=MD5(?struct t {
         int a;
         int b;
         int c;
       })
```

new_foo_patch.pop:

```
#include "core.h"
extern ?struct t {
  int a; int b;
}
extern t someTs [];
extern New::t t__old2new (t);
static void _init () {
  int idx__0;
  for (idx__0 = 0;
       idx__0 < size(someTs);
       ++idx__0)
    New::someTs[idx__0] =
      t__old2new(someTs[idx__0]);
}
prefix Stub {
  int f (t v0) {
    raise (new Core::InvalidArg(
      "Stub?f (int (New::t))"));
  }
}
```

convert_patch.pop:

```
extern ?struct t {
  int a; int b;
}
extern ?struct New::t {
  int a; int b; int c;
}
New::t t__old2new (t from) {
  if (from == null)
    return (null);
  else {
    New::t to = new New::t{b=from.b,
                           a=from.a,
                           c=0};
    return (to);
  }
}
t t__new2old (New::t from) {
  if (from == null)
    return (null);
  else {
    t to = new t{b=from.b,a=from.a};
    return (to);
  }
}
```

Figure 8.4: The patch and supporting files generated for `foo.pop`

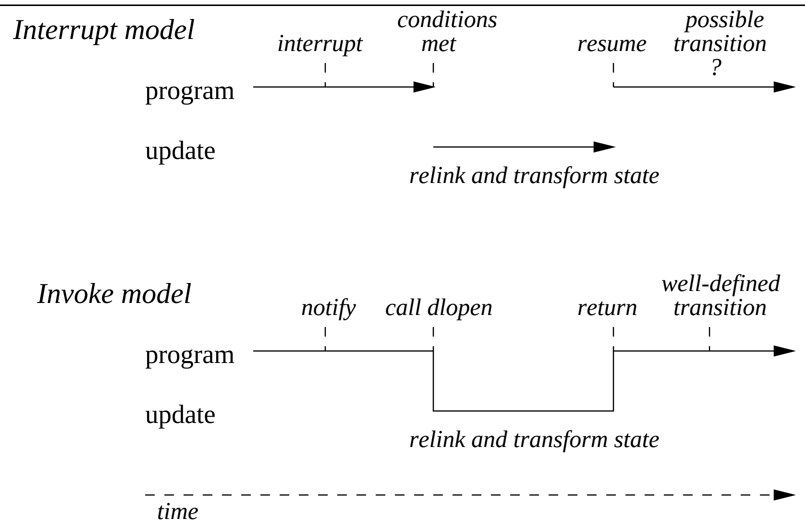


Figure 8.5: Two models for updating a single-threaded program

8.2.1 Interrupt Model

In general, it is possible for a well-formed update to be applied at a bad time, resulting in incorrect state. For example, consider the file f and its patch, shown in Figures 7.1 and 7.2, respectively. Here the patch state translation function S copies the current value of `num` to the new version. The new code then uses this new version of `num`. If this patch is applied while `f` is *inactive* (that is, `f` is not currently running, and not on the stack) then everything will be fine. However, if (the old version of) `f` begins execution just before the patch is applied, it will increment the *old* version of `num` *after* it has been copied by S. The result is the new version of `num` will not reflect the call of `f`.

In part, the above scenario occurs because we assume that a program could be updated at any moment during its execution. This implies an *interrupt*-driven model of updating: the program is interrupted at some point during its execution, the update takes place, and then the program is resumed. This model can be further generalized. Rather than performing the update at the moment of interruption, the update can be delayed until certain conditions are satisfied. For example, in DYMOS [Lee83], the programmer specifies *when-conditions* along with the patches to update as in

```
update P, Q when P, M, S idle
```

This specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M`, and `S` do not have activations in any thread stack.

This so-called *interrupt model* is visualized in the top portion of Figure 8.5. During its execution, the program is interrupted, then after some time the necessary conditions are satisfied and the program context-switches to perform the update atomically.[3] Control

[3]This does not mean that the update cannot be performed in parallel with program execution, although this is frequently the case in existing systems, only that the update must *appear* atomic to the program. This implies the use of synchronization.

then returns to the running program, and at some point the program *transitions* to using the new code. For example, if procedure `Q` was running when the update took place, the old `Q` would continue to run and the new `Q` would be invoked sometime later. In some systems procedures may not be updated while they are active [GKW97, MPG+00, SF93], so the transition to new code always occurs immediately upon program resumption.

Being able to enforce timing conditions at runtime adds to update flexibility, but specifying those conditions so that updates are correct is not necessarily straightforward. In fact, Gupta *et. al* have shown that the problem of correct timing is, in general, undecidable. To show this, they developed a model for dynamic updating and defined a notion of *update validity* [Gup94, GJB96]. In their model, a running program P is a pair (Π, s), where Π is the program code and s is the program state, encapsulating the notion of the stack, heap, and machine registers. An *update* to P is a pair (Π', S), where Π' is the new program code, and S is a state transformer function that maps the old state to a new state; this is analogous to our notion of dynamic patch as described in §7.1. Applying the update yields a new program (Π', s') where $s' = S(s)$. An update is *valid* if and only if the new program's state s' eventually becomes *reachable*. Reachability is defined as follows. A state s, relative to code Π, is *reachable* if and only if a program (Π, s_{Π_0}), where s_{Π_0} is a legal initial state, can evaluate to (Π, s) at some time for some inputs. The authors show that, in general, determining that a change is valid is undecidable (by relating to the halting problem). However, they show that if certain conditions are met as to *when* a change may take place and *what* state transforming functions S may be used, then validity can be proved formally; we describe these conditions below. Bloom *et al* develop a similar, but more complicated, model for Argus [Blo83, BD93].

Because no automated means of generally determining a valid update time is possible, previous researchers have developed techniques to identify program patterns that have valid update points. Perhaps the most advanced was developed by Gupta *et. al*; it compares the old and new versions of C code (not including functions, stack allocation, or heap allocation) and identifies, based on a syntactic analysis, program points that would preserve update validity. This analysis is quite conservative, and can only handle restructurings of the same algorithm, not changes to program functionality. Lee [Lee83] describes a way to decompose a valid update into a set of smaller valid updates. A directed graph is constructed such that each node in the graph represents a function to be replaced, and an edge from f to g implies that *g should be updated before or with f*. The strongly connected components of the graph then represent functions that must be updated together. Lee does not formalize why one procedure should be updated before another; in some cases this is easy to determine (*e.g.* if the types of functions change), but in others it is not straightforward. Furthermore, a valid update must be known before it can be deconstructed, but no guidance is provided in finding such an update. Many systems simply impose the restriction that updates may only occur to inactive code [GKW97, MPG+00, SF93], but this does not guarantee that race conditions will not occur.

Enforcing timing constraints at runtime is expensive, in terms of performance and implementation complexity. For the system to test update constraints at runtime, there must be some way to identify the set of active procedures. If some procedure required to be idle is in the set, then the program continues to execute, updating the active set as it goes, with each change to the set testing whether the idle conditions have been met.

PODUS [FS91] relies on the fact that its programs must be single-threaded and therefore the active set is effectively the stack; the set is checked by extra code that checks the stack depth upon procedure return. In DYMOS, which supports multi-threading, each function call requires a synchronized access to some global structures to store the fact that the function is active; this can be quite costly.

Whether the problems we have described with identifying valid dynamic timing constraints arise in practice is uncertain. However, none of the systems mentioned above presents any analysis that says otherwise. Some simple cases are considered, but no realistic application experience is presented. Therefore, we are led to believe that while flexibility is *potentially* increased by timing enforcement mechanisms, using these mechanisms may or may not result in actual gains, calling into question the loss in performance and increase in implementation complexity.

8.2.2 Invoke Model

The problem of timing can be greatly simplified by requiring the program to be coded from the outset so that updates are only permitted at well-understood times. This transfers the timing enforcement issue from runtime to software construction time: rather than assuming, as in the interrupt model, that a program will not be aware that it is updateable, and thus updates may conceptually occur at any time, we instead require the program to be coded to perform its own updating by *invoking* the updating procedure. This model, which we call the *invoke model*, is illustrated in the bottom half of Figure 8.5. Here, the program is somehow notified that it should perform an update, and so it calls the update procedure (*i.e.* `dlopen`) at the next appropriate moment to apply the appropriate patches. Once the patches have been applied, the update procedure returns and the program continues where it left off. If it was properly constructed, it should transition to the new code at a well-understood time.

Choosing the appropriate update time differs in the two models. In the interrupt model, we have the old and new version's files, and we compare the two to determine times *not* amenable to updating, resulting in a list of constraints. Therefore, the update times are negatively determined (*i.e.* we determine when the update may *not* be performed) and relative to the particular update at hand. On the other hand, the invoke model requires choosing a time that should be amenable to *any* future update. This requires the programmer to think abstractly about future updates, and to create conditions that should not interfere with those updates.

Example

Based on our experience, we have found a general approach to structuring applications so that updates are well-timed when using our system. This is not the only possible program structuring, but we believe it works well.

The problem with arbitrary update times is two-fold:

1. Running procedures might be manipulating state we want to transform; the interaction between the state transformer and these procedures could result in race conditions. To prevent this, we essentially want to delay state transformation until

running code has completed *transactions* manipulating global state. The notion of transaction has been formalized in the database community as being a series of operations that must occur all-at-once (as far as the rest of the program is concerned) or not at all. Since most programming languages (including Popcorn) do not support formal transactions, we consider a more informal notion. In particular, a program transaction is a computational sequence that performs some self-contained piece of work.

2. We want the transition to the new code to be well-defined. If functions have activation records on the stack, and these functions are updated, then the old code will run until the functions exit and are re-entered. Unless the program is structured in a reasonable way, running functions may not exit (and re-enter the new code) in a timely fashion, leading to potential problems. For example, old code could continue to operate on old copies of global state, thus not properly communicating computation to the new code.

We can solve both of these problems by requiring that the program *unwind* the stack at update-time. That is, any code that is currently executing must exit, without performing any meaningful (*i.e.* state-manipulating) computation. Once all active functions have exited, the program restores the stack to its former state by calling into the new code, and resumes its computation on the transformed state. This way, any piece of code that was executing, including the event loop itself, can be updated in a timely manner.

It is important to identify program transactions when implementing this unwinding in the program. In particular, updates should only be applied when there are no active transactions. This allows the stack to be safely unwound and restarted, since all meaningful work has been completed. Event-based programs are easily restructured to unwind and restart computation. In particular, each event-handler essentially implements a transaction, so that an update notification can be processed at the start of the event loop, when there are no transactions outstanding. The loop can then be exited and restarted, using the new code. We take this basic approach with FlashEd, as we will describe in detail in the next chapter.

This approach can apply to multi-threaded programs as well. First, each thread can be notified that an event is pending. The threads then complete any outstanding transactions, and unwind their stacks. They 'check in' with the main program thread, at which time the update is applied. Finally, the main thread notifies the remaining threads that they may restart, at which point they begin using the new code.

Discussion

The fact that the invoke model fixes the moment(s) of update is both an advantage and a disadvantage. On the one hand, our confidence in an update's correct timing is likely to be greater because we know *exactly* when updates will occur and thus can determine how they will interact with the system. On the other hand, choosing an update time may be difficult since it must accommodate updates of unknown composition, and if an update time is chosen poorly, we may be limited in the updates we can perform correctly (at least until we can update the program to accept updates at other times). However, our experience with the program structure described above, which derives from our approach in FlashEd, has been that that *a priori* choosing a reasonable update time has not affected what changes a patch can express, and has greatly improved our confidence in its correctness. That is, the advantage of fixed timing is significant and the disadvantage of fewer times for update is minimal.

Furthermore, other valid program structurings may exist. For example, we can use the same methods proposed by proponents of the interrupt model, with some modification, to determine that the chosen update point is reasonable. For example, we could use Gupta *et. al*'s syntactic analysis to determine if our chosen update-time is reasonable for an update we are about to perform. If it is not, we may be able to change the update to make it reasonable, or perform a different update prepare the system. For example, if we discovered that a potential race condition exists due to a thread accessing some state we are about to update, we could first update that thread to acquire a mutex before accessing the state, and then apply our update, using the same mutex during state transformation.

To be fair, determining proper update timing is the largest unexplored area of this work. We have had good results with single-threaded, event-driven programs, but have little practical knowledge in the way of multi-threaded programs. We are encouraged that updating multi-threaded programs is not onerous with the invoke model, as this model is successfully employed by multi-threaded Erlang [AVWW96] programs. We believe there is ripe opportunity to apply formal methods to proving that an update is valid, and hope to pursue formalizing update validity in the invoke model in future work. In particular, we plan to prove that the 'unwind' structure that we have described above is sound, and hope that the analysis will reveal other possible program structurings. One possibility is to consider how formal transactions could be added to the language to support proper update timing. Past work on adding transactions to programming languages would be a useful starting point (*e.g.* Wing *et. al* [WFMN92]).

Chapter 9

FlashEd: an Updateable Webserver

To show that our dynamic updating infrastructure is useful in practice, we built a substantial application: a dynamically updateable webserver, based on the high-performance webserver Flash [PDZ99]. The original Flash consists of roughly 12,000 lines of C code and performs competitively with the highest performance web servers available today (as per measurements in [PDZ99]). Our version, called *FlashEd* (for *Ed*itable *Flash*), is a port of the majority of Flash's functionality to Popcorn, although we omit some advanced features. Our most advanced version of FlashEd is roughly 8700 lines of Popcorn code.

In addition to illustrating the updating system in use, building FlashEd informed and justified its design. As we built FlashEd, we patched a publicly running server with significant new features. In doing so, we realized areas in which the updating system could be improved. For example, we developed the automated patch generator following our attempts to construct patches for FlashEd by hand, a process we found was too tedious and error-prone. Many prior systems lack a serious application to inform their design in this way, making their claims less grounded in experience.

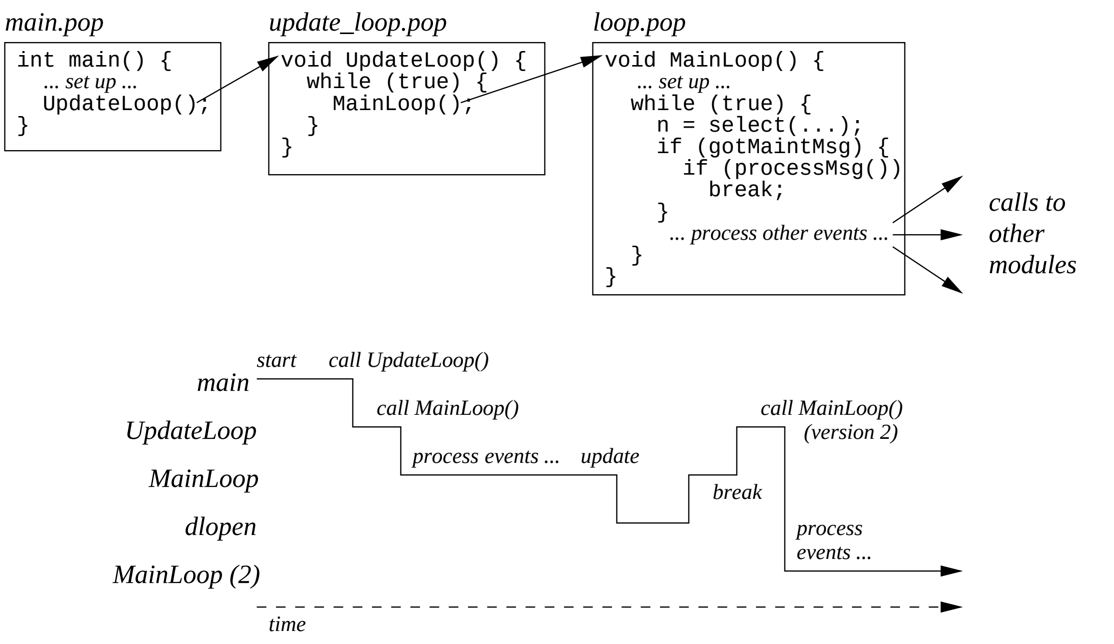
In this chapter, we focus on two things. In the first section we describe how we constructed FlashEd to be updateable, structuring it for use with the invoke model. In the second section, we describe how we generated and tested patches for FlashEd, and how these patches were tested and applied in practice. The goal is to illustrate that our updating system is flexible enough to handle non-trivial changes, and is easy to use. In the next chapter, we measure FlashEd's performance to assess the overhead imposed by our updating system.

9.1 Building FlashEd to be Updateable

Because we use the invoke model in applying dynamic updates, we needed to enable FlashEd (relative to Flash) to receive update notices, and to respond by applying the indicated patches at an appropriate time. In addition, to ensure that FlashEd runs non-stop, we had to alter how some errors were handled. We describe these two changes in detail below.

9.1.1 Update Timing

Flash's structure is well-suited to the invoke model, and in particular to the *unwind* program pattern we defined in the previous chapter. It is constructed around an event loop, implemented in the function `MainLoop` that is called from `main()` following initialization,



```
int main() {
  ... set up ...
  UpdateLoop();
}
```

```
void UpdateLoop() {
  while (true) {
    MainLoop();
  }
}
```

```
void MainLoop() {
  ... set up ...
  while (true) {
    n = select(...);
    if (gotMaintMsg) {
      if (processMsg())
        break;
    }
     ... process other events ...
  }
}
```

Figure 9.1: Structure of FlashEd and FlashEd update procedure

that does three things. First, it calls `select` to check for activity on existing client connections and the connection listen socket. Second, it processes any client activity (either writing more data from previously requested files or processing new requests). Finally, it accepts any new connections.

Part of the implementation of FlashEd is shown in Figure 9.1. As in Flash, the event loop is in the `MainLoop` function, in the file `loop.pop`. While the overwhelming majority of `MainLoop` is unchanged from the C version, we added a *maintenance command interface* to support update notification and patch application.

Maintenance Command Interface

The maintenance command interface allows a separate application running on the same machine to connect to the webserver and send textual commands. The `select` checks for connection requests from the maintenance listen socket, and if one is found, it calls a function `processMsg` to accept the connection and process the request. If the request is an update request, as indicated by the command `update` *filelist*, where the *filelist* is a non-empty list of dynamic patch filenames, then `dlopens` is called to dynamically update the corresponding modules in the program.

To implement the unwind pattern, we need to clear the stack by exiting the `MainLoop` function and then re-entering it. Among other things, this allows the `MainLoop` function itself to be updated. In particular, `MainLoop` contains an infinite loop for handling events, so even if we were to update `loop.pop` to include a different version of `MainLoop`, the program counter would never leave the old version of `MainLoop`, and thus would never enter the new version. To properly unwind and restart the stack, we created an *updating* *loop* in a file outside of `loop.pop`; in the figure, this is shown as the function `UpdateLoop` in the file `update_loop.pop`. The updating loop is just an infinite loop that continually calls `MainLoop`. If an update request completes successfully (*i.e.* in the case that `processMsg` returns `true`), then we break out of the event loop and exit `MainLoop`. During the update, `update_loop.pop` will be relinked so that it will call the new version of `MainLoop`. So that program execution proceeds correctly, we had to make some slight changes to the 'set up' code in `MainLoop` so that it could be executed more than once.[1]

The entire updating sequence is shown at the bottom of Figure 9.1. The webserver starts in `main`, which performs initialization and setup, finishing by calling `UpdateLoop`. `UpdateLoop` enters its infinite loop, and calls `MainLoop`. The webserver then processes events until it receives an update request, so that it calls `dlopen` (by way of `processMsg`). Once the update finishes successfully, control returns to `MainLoop`, which immediately breaks out and returns to `UpdateLoop`. Finally, `UpdateLoop` will call `MainLoop` again, which because of relinking during the update, will go to the new version of `MainLoop` to begin processing more events. Of course, the existing state, modified as necessary by the state transformer function, is preserved between loop invocations.

The odd use of the extra file `update_loop.pop` owes to the fact that our implementation dumps old hashtables from the symbol table once their code has been updated. As described in §7.3.2, this action is not acceptable in general; old tables should be retained with weak pointers in case their code is still active at the time of the next update and they need to be relinked. The use of `update_loop.pop` structures the system so that old code will not be needed following an update. In particular, if we were to put the updating loop at the end of `main`, this loop would not be relinked following the first update. After `main.pop` is first updated, the old hashtable for `main.pop` would be thrown out. However, the return address on the stack for the call to `MainLoop` would still point into the original `main.pop`. When `main.pop` is updated for a second time, the original version of `main.pop` is not relinked since its symbol table was dumped (only the second version is). Therefore, when control returns from `MainLoop`, it goes back to the first `main.pop`, which jumps into the wrong `MainLoop` function, the one from the second version. We ensure this situation does not arise by *never* updating the file `update_loop.pop`. This is a short-term workaround to allow us to examine how well old code can be garbage-collected; using weak pointers is the proper long-term solution.

9.1.2 Fatal Error Handling

The second change to Flash was in how fatal errors were handled. Flash contained many places where errors were detected and program execution aborted by calling `exit`. Such aborts are not acceptable in a non-stop program, but on the other hand, many of the detected conditions indicate that the program is not functioning properly. To keep the program running but make sure it does so properly, we did two things. First, we changed the cases in which `exit` was called to throw an exception instead. The `MainLoop` function

[1]In DYMOS, infinite loops can be updated by having the jump at the end of the loop go to the newest (*i.e.* the updated) loop header; Erlang implements this same functionality with a tail-recursive call to a new looping function. Adding such features to our system would be possible with some modification to the Popcorn compiler and our updating library.

catches any unexpected exceptions and prints diagnostics. Second, we reset the program state to be sure that it is consistent. For example, `MainLoop` shuts down existing connections upon receiving an exception, and restarts the loop. In addition, if an exception is thrown from a module that maintains state, that state is reset before the exception is thrown.[2] Thus the program can continue service until it can be repaired online, albeit with the loss of some information and connections.

9.2 Updating FlashEd in Practice

Having built the webserver to be updateable, the real test is to actually update it. We wanted to learn how flexible our system is in practice, and how easy it is to develop and apply correct patches. To do this, we decided to construct FlashEd *incrementally* and to deploy it publicly. Together, these actions simulate the process of maintaining a service-providing, non-stop system as it evolves. Furthermore, by holding ourselves accountable publicly, we were not tempted to shut the system down out of convenience, but instead had to use the updating infrastructure in all circumstances.

This exercise was extremely fruitful. We learned how to construct patches effectively, test them 'off-line', and apply them. Furthermore, because we used code that originated with a different developer (*i.e.* the Flash sources in C), we were not tempted to write an application that by its nature would work well with our system. On the other hand, the updates to the system were contrived by us, and neatly partitioned into functional components. This partitioning most likely did not match the actual development process for Flash, and therefore may not represent some of the difficulties that could arise in writing patches. Nonetheless, our experience shows that our approach is effective, and also provided firsthand evidence of how it could be improved. For example, it was the exercise of building patches for FlashEd that led to the development of the automated patch generator.

9.2.1 Update Chronology

Our FlashEd implementation has undergone a number of changes, resulting in four versions. Our initial implementation, version 0.1, lacks some of the C version's features (such as CGI and directory listings) and performance enhancements (such as pathname translation caching and file caching). It consists of eighteen source files, and roughly 6200 lines of Popcorn code. We deployed version 0.1 as the host of the project home page, and then made modifications to support pathname translation caching for version 0.2. After testing version 0.2 built statically, we then constructed patches, tested them, and applied them dynamically. We repeated this process for version 0.3, which adds file caching, and version 0.4 which adds directory listings. This last version consists of 21 source files and about 8700 lines of Popcorn code.

A brief chronology showing the evolution of our public server is shown in Figure 9.2. We started version 0.1 at `http://flashed.cis.upenn.edu` on October 12, 2000, to host

[2]While resetting the local module's state makes sense for FlashEd, because of the way the state is partitioned between modules, it might make sense in general to reset the whole program's state through some kind of generic interface. An appropriate policy is application-dependent.

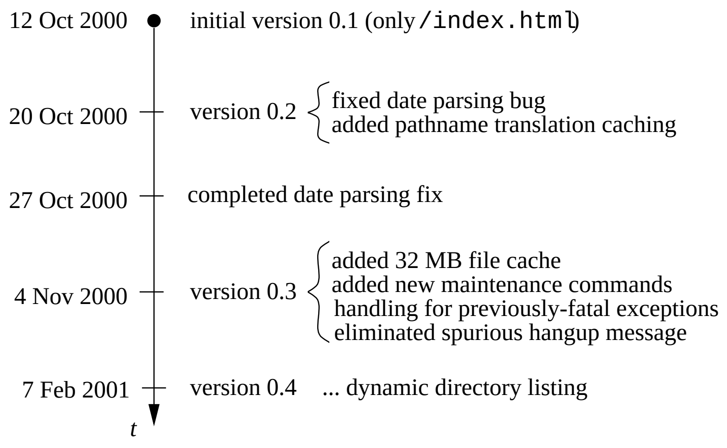

12 Oct 2000
initial version 0.1 (only /index.html)
20 Oct 2000
version 0.2
fixed date parsing bug
added pathname translation caching
27 Oct 2000
completed date parsing fix
4 Nov 2000
version 0.3
added 32 MB file cache
added new maintenance commands
handling for previously-fatal exceptions
eliminated spurious hangup message
7 Feb 2001
version 0.4
... dynamic directory listing
t

Figure 9.2: Timeline of major *FlashEd* updates

To	changed		LOC	total	interface LOC	
version	files	types	changed source	patches	auto	by hand
0.2	11	3	433	16	1324	48
0.3	9	2	813	14	1261	99
0.4	7	1	1557	12	1214	99

Table 9.1: Summary of changes to versions 0.2 through 0.4 of FlashEd

the FlashEd home page. We applied patches for version 0.2 on October 20, for version 0.3 on November 4, and for version 0.4 on February 7. All patches were tested off-line on a separate copy of the server under various conditions, and when we were convinced they were correct, we applied them to the on-line server. Even so, we found a mistake in the first patch—a flag had not been properly set—and applied a fix on October 27. In addition, we applied roughly five small patches for debugging purposes, such as to print out the current symbol table.

9.2.2 Patch Construction

A description of the changes between each version and the patches required are summarized in Table 9.1. The first three columns of the table show the changes to the source code made from the previous version, including the number of changed or added source files (not including header files), the number of changed type definitions, and the number of changed or added lines of code. The last three columns describe the patches, including the total number of patches generated (not including the type conversion file), the total lines of generated code for the patch interface code files, and those lines that were added or changed by hand.

There are two things to notice in the table. First, the number of patches generated exceeds the number of changed source files; this is because type definitions used by those files were changed, meaning that some of the functions in those files that make use of those types were also effectively changed. Second, the number of lines of interface code automatically generated far exceeds the amount modified or added by hand. This is not to say that the process of modifying the automatically-generated files was simple (it was not in some cases), only that a large portion of the total work, much of it tedious, could be done automatically. For example, many of the generated lines include `extern` statements that refer to the old and new versions of changed definitions; these would have had to be placed by hand otherwise. Most importantly, using the patch generator guaranteed that the patches were *complete*—all of the changes were identified automatically, even though some changes needed to be addressed by the programmer.

The alterations to the generated files usually were of the following forms:

1. *Translate pointerful data.* The global array `allHandlers` stores the *handler* function for each connection, indexed by the connection number. Each handler function takes as an argument the connection it is handling; a connection has type `httpd_conn`. This type changed between versions one, two, and three, and so the type of handler functions also changed. Therefore, the state transformer needed to initialize the new array to have functions that correspond to their old versions, using techniques described in §7.4.1, resulting in a patch similar to that shown in Figure 7.17 (page 114).

2. *Complete type conversion functions.* In a number of spots, the patch generator could not provide suitable values for new or changed fields in `union` and `struct` definitions. For example, each `httpd_conn` contains an `expireFunc` field, which stores the function to be called on a timeout, taking the `httpd_conn` itself as an argument. When `httpd_conn` changes for any other reason, this field changes type since the `expireFunc` takes an `httpd_conn` argument. The conversion routine cannot insert meaningful default values for functional types, so it leaves a placeholder instead. In the current implementation, there is only one timeout function, `IdleTimeout`, so choosing the correct value is very simple.

3. *Initialize new state or functionality.* In each of the versions, a new entity was added: a pathname translation cache for version 0.2, a file cache for version 0.3, and in version 0.4 a separate helper program to perform the directory listings. In a statically linked executable, each of these entities would be initialized in `main`, preceding event processing by `MainLoop`. When updated dynamically, this initialization code in `main` is never executed, and so instead must be executed as part of the state transformer function. We copied the relevant code there, but once had to add to it significantly, as described below.

There were three cases in which these alterations were non-trivial. The first was in the patch from version 0.2 to version 0.3, which adds a file cache. In all versions of FlashEd, each `httpd_conn` contains a field `dataEnt`, which has struct-type `DataEntry`. Elements of type `DataEntry` contain information about the file that a particular URL refers to, including its modification date, size, contents, *etc.* In versions earlier than 0.3, this entry was constructed from scratch for each request and then thrown out after the connection closed. In version 0.3, this entry is inserted into the file cache after it is first created, and then shared among concurrent connections that have requested the same file thereafter. The difficulty is that when the connection is complete, the code for releasing `DataEntry` values expects these values to respect certain invariants. Constructing the entries out of context to respect these invariants is not trivial.

Therefore, we had to slightly change the *source* of version 0.3, specifically the routine `ReleaseDataEntry`, to deal specially with those `DataEntry` values that are problematic. This would prevent them from confusing the invariants maintained by the file cache software. Fortunately the changes to be made were fairly simple in this case, but it may be that more complicated changes would be necessary in other cases. This would seem to disprove our assertion that software and patch development are separate processes. Most often we were able to create patches without making changes to the source. However, we could consider the development process for updateable software an iterative one: develop the next version of the software statically, then test it; develop the patches and test them; if during patch development any changes needed to be made to the source, go back and test the static program. More experience is needed to understand whether this process could be burdensome in reasonable cases, and/or whether the 'pollution' of the source code to accommodate one-time patches will be excessive.

Another patch component that was not totally straightforward was that for version 0.4, which adds directory listing support. Listings are acquired in a parallel computation using a 'slave' process so that the parent process can continue to service connections. Once the listing has been gathered, the slave notifies the parent and transmits the information to it. At start-time, the slave process is forked off, and it waits for instructions from the main program.

The difficulty is that the slave process is a separate executable (rather than a module in the FlashEd program) and it is not clear from the current program state where it is located. In particular, Flash (and thus FlashEd) was coded to expect that all of its executables would be stored in the same directory. However, after Flash has been invoked, the directory that Flash resides in is lost, particularly because we change the current working directory to be that of the pages we serve. But the update needs to be able to find that directory so it can initialize the slaves.

Therefore, we had to code the patch to 'guess' where the executables might be. It looks to see if the original `argv[0]`, saved in global state for other reasons, was an absolute path; if so it can extract the directory from there. If not, it calls `getcwd` to try the current directory as long as the user did not specify some other directory to serve the documents from. Finally, it tries an absolute path provided by the programmer in the patch. If none of these work we throw an exception, which will cause the update to be rolled back.

This case implies that being able to parameterize patches would be useful. For example, we could load the patch and provide to its initialization function the directory location. The difficulty is that the type of the user-defined `init` function would differ depending on what arguments it needed. We could deal with this problem by specifying the user-defined `_init` function *generically*, using R-types. For example, it could have the type

`void _init(`$\exists\alpha.\ (R(\alpha) \times \alpha)$` array)`

That is, `_init` takes an arbitrarily-sized array of parameters, so that each parameter is coupled with its type. The `_init` function would have to use `checked_cast` to extract these parameters at the expected types. To facilitate a parameterized `_init`, we would have to change `dlopen` to be able to take lists of parameters along with the modules to open, and then pass these along to `dyninit` after loading the module, so that `dyninit` in turn could pass them to `_init`.

In the final case of a non-trivial patch, we had to initialize some global data of a file (call it p) that required not only the old version of that data as input, but also some data in another patch (call it q). The difficulty occurs if the external data needed by p is from the *new* (patched) version of q. This implies that the q's state transformer must be run before p's. This is possible by ordering the patches provided to `dlopens`. In our case, we could just as well use the old version of p's data, and therefore were immune to issues of ordering. How much this could end up as a problem in practice requires more experience.

Overall, patch construction was not difficult, given that the automatic generator did most of the work. Furthermore, we did not come across any cases in which we could not construct a reasonable patch. However, it is worth asking how scalable our approach is. Certainly, thanks to the automatic identification of changes, it is more scalable than existing approaches that lack this support. However, many non-stop systems consist of tens or even hundreds of thousands of lines of code; would it be more difficult to identify tricky invariants when altering the automatically-generated patches for these systems? Our experience so far has been promising, but more experience is needed to answer this question.

9.2.3 Testing Patches

Because, as we have seen, patch construction is not trivial, we needed to test our patches on an off-line version of the server before applying them to the public server. We applied patches under no-load and extreme-load conditions to make sure they would work in various circumstances when applied to the public server. In addition, not only did we apply the patches to our test executable, compiled from the existing sources, but we also made sure they would work with the actual public executable with all of the patches we had applied publicly. This was important because during the FlashEd development process, both the TAL implementation and our patch generator changed to fix some bugs and tweak various features, so the old executable and patches we built from the source differed somewhat from the ones that were deployed.

Testing patch application under no-load conditions is easy: just start the server, apply the patches, and see if the new functionality works. Of course, this approach fails to fully exercise the state transformers, since there is little state to transform; in particular, there are no existing connections being served. To test the patches under extreme-load conditions we used benchmarking software—the same software used to measure the updating overhead on FlashEd's throughput, described in §10.2.1—to push the server to its limits before applying the patches. Doing so often revealed flaws in our initial state transformer code. For example, when writing the patch to add file caching, we neglected to check for *null* when transforming the `dataEnt` field of existing `httpd_conn` values in the connection array since for valid connections, no `dataEnt` field should be *null*. However, once a connection is closed, its `httpd_conn` structure is kept in the connection array after nulling its fields (including `dataEnt`; a separate bitmap is used to identify which `httpd_conn`s are valid). In our initial version of the state transformer, we assumed all `httpd_conn`s in the connection array were valid, but then added the null check when this assumption was proven false.

Our testing with FlashEd, while non-trivial, was not rigorous to the level of industry standards. For our dynamic updating system to be of practical use, rigorous testing techniques should be possible, both for testing the patching process and the resulting executable. In general, standard testing techniques can be used to test the patched executable. For example, we can employ *coverage testing* tools like χSUDS [xSU], which identifies tests that exercise control-flow paths through a program. Using such tools is possible because, following patch application, the running program is accurately represented by current source code. We would also like to apply coverage techniques to the state transformers, which are used only at patch-time. One way to do this would be to compile patches to be linked in *statically* with the prior version of the program, allowing the coverage tool to identify paths in those patches in conjunction with the rest of the program. Having done this, a coverage tool would be able to identify the 'shape' that program state must be in to exercise various paths in the state transformers. Looking into further into patch testing would be interesting and useful future work.

9.3 Lessons Learned

Running the server revealed which aspects of the system work well and which do not. For instance, we learned soon after we deployed the server that our version of the TAL verifier was buggy—it only checked a subset of all of the basic blocks in loaded files. Since the verifier is part of the trusted computing base, it could not be updated. Ultimately we had to shut down the system and recompile it with the new version of the verifier. We did this on February 7, 2001, and deployed version 0.4 at that time. In the future, we could accommodate changes to the verifier, or other trusted code, by allowing linking without verification.

We also made a human error when first compiling the server: we forgot to enable the exporting of `static` variables when compiling the library code. This problem became apparent when we attempted to dynamically update the dynamic updating library. The library was not properly removing old entries from the dynamic symbol table, and so we wanted to patch the library to fix the problem, as well as clean up the existing symbol table. However, since the symbol table is declared `static`, it was not available for use by the patch. As a result, any update to the library is effectively precluded since the state cannot be properly transferred.

On the whole, however, the system has been both flexible and extremely easy to use. Never were we unable to express a change that needed to be made. Furthermore, the system was flexible in the ways it could be used. For example, on a number of occasions we loaded code that would print out the dynamic symbol table (by calling an existing function in the updating library) to make sure that symbol names referenced in our patches, particularly the ones chosen for static variables, matched the ones present in the table. We also loaded code to print out the state of the file and translation caches, to make sure that things were working.

Having the verifier to check patches as they are being loaded has greatly enhanced system robustness. For example, we tried to apply some patch files that were incorrectly generated; the implementation file path mentioned in the patch description file pointed to the wrong directory. As a result, some of the type definitions were incorrect, and this fact was caught by the verifier. Once we applied a patch whose state translation function failed to account for null instances; the updating library caught the `NullPointer` exception and rolled back the changes made to the symbol table. Using an unsafe language, such as C, would have resulted in our non-stop system stopping with a core dump.

Chapter 10

Performance

To this point, we have argued that our system for dynamic updating is flexible, robust, and easy to use, having presented how the system works and how it has been used in practice. In this chapter we support the claim that our approach imposes only a low overhead. We present the performance of our system, in two parts. First, we look at the component overheads, in terms of space and time, imposed by dynamic linking and updating—that is, `load`, DLpop, DLpop/update—over and above statically linked programs. Second, we look at the updating overhead imposed on application performance. We consider our webserver FlashEd, comparing its throughput with updating enabled to its throughput without.

Measurements were performed on an isolated benchmarking cluster consisting of four machines connected by 3Com SuperStack 3000 Fast Ethernet (100 Mb/s) switch. Each machine is a dual-300 MHz Pentium-II with split first level caches for instruction and data, each of which is 16 KB, 4-way set associative, write-back, and with pseudo LRU replacement. The second level 4-way set associative cache is a unified 512 KB with 32 byte cache lines and operates at 150 MHz. These machines receive a rating of 11.7 on SPECint95 and have 256 MBs of EDO memory. We run the fully patched version of RedHat Linux 6.2, which uses Linux kernel version 2.2.17.

10.1 Dynamic Updating Component Costs

The execution time overhead imposed by dynamic linking, relative to Popcorn programs that use static linking only, occurs at three points in time: runtime, dynamic load-time, and program start-time. At runtime, each reference to an externally defined symbol must be indirected through the GOT. At load-time, the running program must verify and copy the loaded code with `load`, link it by executing its `dyninit` function, and then relink the rest of the program when doing dynamic updating. At startup, statically linked code must construct the initial dynamic symbol table and register the program type interface.

10.1.1 Runtime Overhead

Runtime overhead is incurred through the use of a global offset table (GOT), which stores references to symbols external to the module, so that function calls and variable references to external symbols are indirected through the table. Non-updateable programs only use a GOT in dynamically linked modules, while updateable programs require all modules to each contain a GOT, whether they are statically or dynamically linked.

```
Without GOT          With GOT

     External variable assignment var = 1
movl $0x1,%eax       movl GOT,%eax
movl %eax,var        movl n(%eax), %esi
                     movl $0x1,%eax
                     movl %eax,(%esi)

        External function call func()
call func            movl GOT,%eax
                     movl n(%eax), %eax
                     call *%eax
```

Figure 10.1: Code for accessing external values with and without a GOT

timer	function call		assignment	
overhead	w/o GOT	with GOT	w/o GOT	with GOT
33	3	5	1	3

Table 10.1: The overhead of per-GOT references (measurements are in cycles).

For the essentially unoptimized Popcorn compiler, the code sequences for function calls and variable references with and without a GOT are shown in Figure 10.1. When using a GOT, each access requires two additional instructions: one to move the GOT address into a register and the other to get the address of the proper GOT field. With a slight change to TAL, the overhead could be optimized to a single instruction. In particular, we could combine the first two instructions to be instead `movl GOT+`n`,%eax` (with the target being `%esi` rather than `%eax` in the variable case). Arithmetic of this sort is not currently allowed in TAL, but could be easily implemented with relocation offsets supported by both ELF and COFF object files: `GOT+`n compiles to a relocation for `GOT` with an offset n. This trick is employed by `gcc`.

We measured the elapsed time due to the added instructions on one of our cluster machines; elapsed time was measured with the cycle counter, using the `rdtsc` instruction. The results are shown in Table 10.1. The first column shows the overhead of using the cycle counter, for each architecture. This was determined by "measuring nothing;" that is, we read and stored the value of the cycle counter twice with no intervening computation. The remaining values were obtained by measuring the desired code sequence and then subtracting the cycle counter overhead from the result. We present the median of 53 measurements for each code sequence, in effect throwing out the outlying cases (such as when the code or GOT pointers were not hot in the cache).

In both cases, the added instructions resulted in a 2 cycle overhead. It is tempting to assess this overhead out of the context of a real program. That is, we could say that dynamic updating adds $(5 - 3)/3 = 66\%$ overhead per function call. However, this is an unlikely worst-case because:

1. A called function will perform some computation that most likely dwarfs the 2 cycle overhead added by the call. The more computation performed, the less the 2 cycles will impact it.

2. Only references to *external* symbols will incur the overhead; references to functions and variables in the same module do not use the GOT. Therefore, how a program is structured into multiple source files will affect the total overhead.

Therefore, to better assess the impact of the added cycles, we need to examine the impact on *application performance.* As shown in §10.2.1, FlashEd application performance suffers less than a 2% overhead under a variety of conditions.

A second GOT overhead arises for imported abstract values. Recall that imported values of abstract type, by nature, cannot have a default GOT value, and therefore must be stored in a potentially-null field of the GOT. As a result, in principle, each GOT access for an abstract value requires a null check. However, we have yet to see this overhead occur in practice. Most modules do not export abstract values, but instead use "constructor" functions that produce abstract values; an exception in our current code base is the Popcorn `Core` library, which defines `stdin`, `stdout`, and `stderr` to have abstract type `FILE`. However, these cases typically define the abstract type to allow a null value (a sort of abstract `option` type), meaning that a null-check would have occurred anyway.

10.1.2 Load-time Overhead

At load-time (that is, when loading a patch or new module), there are three basic operations: *loading* the module, *linking* it, and *relinking* the rest of the program. We look at the cost of each of these operations in turn.

Loading is performed with TAL/Load's load primitive, whose implementation is shown in Figure 5.11 (page 61). The two major operations of load are disassembly and verification. Verification itself is performed in three phases: consistency checking, link-checking (these two are collectively labeled *type-check* in the figure) and interface checking (labeled $t = \mathit{typeof}(vs)$? in the figure). Following verification, load must instantiate (load) the code and data of the object file in the program address space, as described in §5.4.2, and construct a tuple of the exported items conforming to load's type argument. Following module loading, the DLpop library will link the module(s) by, among other things, calling the returned `dyninit` function.

We measured each of these component costs for patch files to version 0.3 of Flashed; the trends are summarized in Figure 10.2, drawing from the data presented in Table 10.2. All of these files are applied at once, due to mutually-recursive references among them, and so the three-pass linking algorithm is used. The total time to apply all of these patches was about $16.2s$, and the cost of relinking the program following the update was about $0.81s$; this time includes all of the old versions of the modules being replaced and the other modules as well.

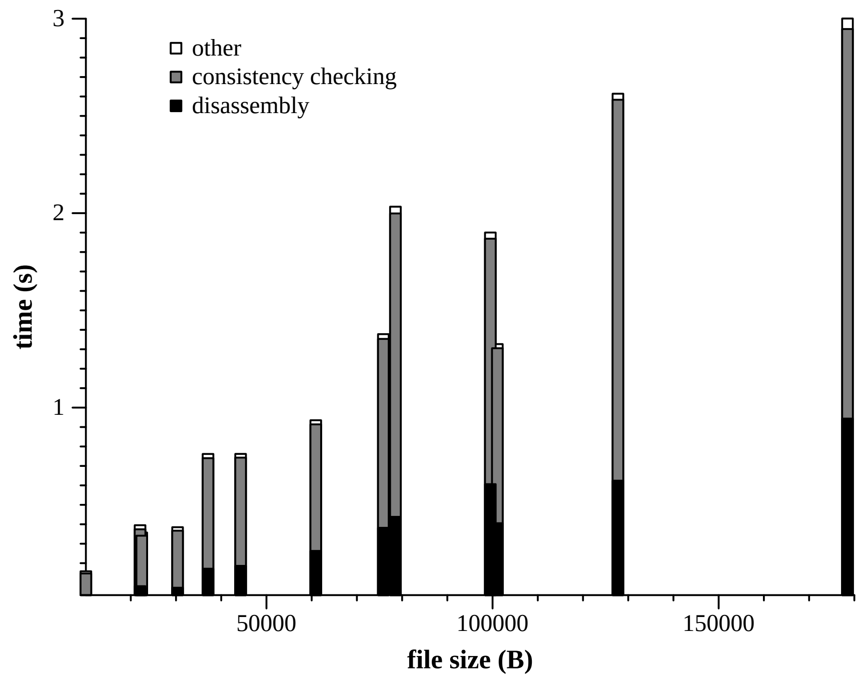


Figure 10.2: The component costs of dynamic linking relative to file size.

In the figure, the X-axis is the total size of the types and object files for each patch, and each bar sums the total time to perform dynamic linking for that patch. We explicitly show disassembly and consistency checking, which dominate the total overhead, and combine all other overheads as 'other.' We can see that the total time is dominated by verification in general, consistency-checking in particular, averaging 72%, while disassembly is the second-largest overhead, averaging 25%. According to [GM00], verification is generally linear in the size of the files being verified, which we find to be essentially true here.

All load-time operations are itemized in Table 10.2. The files are listed in order of their size, corresponding to the bars in the figure left to right. The total time for all operations is shown in the rightmost column, in seconds, and the percentage of that total time for each operation is shown in the adjacent columns. In the table it is clear that in all cases, verification is the dominant cost, while linking (including relinking) and loading are relatively inexpensive.

In many contexts, loading times of this magnitude are not problem. For example, 16 seconds of pause time is less intrusive than an OS reboot. In the case of the webserver, an infrequent pause is at worse just inconvenient to the user, not harmful to the system. However, in other contexts, there may be good reason to want shorter update times; because `dlopen` executes atomically, no other work gets done during updating, and therefore a long pause to update the program is tantamount to a temporary loss of service. We have identified three means of reducing the load-time cost. First, verification times, particularly consistency-checking, could very well be improved. For example, proof-carrying code [Nec97] has demonstrated small verification times, albeit with a different type system,

File info			load %					link	total
Patch name	Size (bytes)		Disasm	Verification			Loading	%	time
	`.o`	`.to`		*cchk*	*lchk*	*ifchk*			(secs)
`convert`	5340	4731	22.48	70.45	4.28	1.54	0.53	0.72	0.158
`read_master`	13676	8373	20.65	74.03	2.90	1.33	0.35	0.73	0.395
`timer`	13748	8677	18.24	77.12	2.35	1.24	0.31	0.74	0.358
`accept`	19776	10570	19.13	76.19	2.42	0.72	0.41	1.12	0.385
`nameconvert`	25176	11903	22.52	74.65	1.46	0.62	0.24	0.51	0.762
`file`	29816	14471	24.44	73.05	1.18	0.38	0.32	0.63	0.762
`dir_master`	43120	17787	28.13	69.60	0.82	0.38	0.33	0.74	0.935
`readreq`	52436	23415	27.71	70.50	0.68	0.22	0.25	0.63	1.38
`main`	58708	19812	21.55	76.74	0.50	0.21	0.20	0.80	2.03
`data`	71036	28474	31.91	66.43	0.56	0.14	0.27	0.70	1.90
`cold`	76144	24915	30.56	67.82	0.61	0.19	0.37	0.44	1.32
`libhttpd`	96360	31366	23.88	74.94	0.29	0.15	0.24	0.51	2.61
`loop`	137016	41472	31.44	66.76	0.33	0.85	0.27	0.35	3.00
		average %	24.82	72.18	1.41	0.61	0.31	0.66	

Table 10.2: Time to load and link patches for FlashEd 0.3 − 0.4

and even TAL's implementors recognize that further gains could be made [GM00]. Furthermore, disassembly has not been optimized. Second, verification could be performed in parallel with normal service. After verification completes, only linking remains, which has negligible overhead. Finally, in the case of a completely trusted system (as is FlashEd, for example), we can safely turn off the consistency-checking phase during verification, since it can be run for each loaded module on some other machine. Leaving on link-checking and interface-checking still ensures that the loaded code meshes with the running program at the module level (link checking caught the bug described in §9.3), but trusts that the contents of the loaded module are well-formed. Since consistency-checking is the most time-consuming operation, we greatly reduce our total update times as a result. Breaking up the verification operation onto server and client machines has been explored for Java in [SGGB99].

10.1.3 Start-time Overhead

At start-time, before execution begins, each statically linked module's `dyninit` function is executed to create the initial dynamic symbol table and program type interface for the program. The costs of these operations depend on the number of symbols and type definitions exported by each module, and which libraries are used.

We measured this cost for Flashed 0.3, which consists of thirty-two source and library files. For each module, the cost of registering the symbols for each module is on the order of milliseconds; registering the program type interface is similarly cheap. As a result, the total time is roughly 0.13 seconds, which is negligible, for two reasons. First, it is on the order of time taken to perform other startup operations, such as reading the program from disk. Second, and more importantly, we expect that programs using dynamic linking and

	Dynamic Linking		Dynamic Updating	
	file is linked		file is linked	
	statically	dynamically	statically	dynamically
uses a GOT		√	√	√
exports globals	√	√	√	√
exports `static`			√	√

Table 10.3: Breakdown of space overhead components based on when files are linked and whether they use dynamic linking or dynamic updating.

updating will be long-running, making even larger start times inconsequential.

10.1.4 Space Overhead

Compiling Popcorn files to be used by DLpop and DLpop/update adds some additional code and annotations to the source file, resulting in an extra space overhead to both the object file and the types file. The majority of the additional space is used only during linking, and therefore has little effect on the system's runtime performance. We first break down the component space overheads, justify why they are not a problem in practice, and then look at the particular overheads that occur with FlashEd.

Per-symbol Overheads

For the object file, space overheads arise *per symbol*, and are therefore only loosely related to a file's size. Overall, per-symbol overheads are due to both the `dyninit` function and the use of a GOT. Overhead is added per exported symbol in all cases, per `static` symbol when using dynamic updating, and per imported symbol only when a GOT is used. These points are summarized in Table 10.3. Of all these overheads, only the GOT is used at runtime, and could affect performance due to both the additional space it takes up, and the space of the added instructions required to use it, by altering the program's cache locality characteristics. We have not attempted to measure this effect, but expect it to be minimal, given that our overall application performance is impacted by only about 2% (as shown in §10.2).

Per-symbol overheads are shown in Table 10.4; the overheads pertaining to imported symbols are only relevant when a GOT is used (otherwise imports add no overhead). For both imported and exported symbols, DLpop imposes three space costs: the string representation of the symbol name l,[1] its type representation t, and the instructions in the `dyninit` function that perform its linking—7 instructions (about 30 bytes) per exported symbol and 9 instructions (about 33 bytes) per imported symbol.[2]

[1]Popcorn strings have a length field and an extra pointer (for easier translation to/from C-style strings), adding 2 words to a C-style representation.

[2]Import overheads may be reduced; they are currently large due to the compiler's simplistic approach to assuring left-to-right evaluation order.

		symbol name	`dyninit` code	type rep	GOT slot	dummy value	per ref	total
import	function	$8 + l$	33	t	4	11	5	$56 + l + t + 5r$
	data	$8 + l$	33	t	4	≥ 8	5	$\geq 53 + l + t + 5r$
export		$8 + l$	30	t				$38 + l + t$

Key:

l	symbol name length
t	size of a symbol's type representation
r	number of times the symbol is referenced

Table 10.4: Per-symbol object file overheads due to dynamic linking and updating

When using a GOT, each imported symbol requires a GOT slot and a default value, as well as the extra instructions to reference the symbol via the GOT. Each default function simply throws an exception, resulting in 4 instructions (about 11 bytes). Each default value pointer points to a dummy value of appropriate type initialized to 'null'. Here, the cost is the pointer (4 bytes) plus the size of the 'null' value, which is an additional 4 bytes in the case of integers, characters, *etc.*, but will be more for strings or values of structured type. Finally, each call or reference that uses the GOT adds two instructions (see Figure 10.1), resulting in 5 extra bytes.

The size of type representations t can be large, between 128 and 200 bytes for functions. Function type representations encode not only the types of their arguments and returned values, but also the calling convention. We mitigate the cost somewhat by sharing type representations among elements of the same type. However, because the calling convention is uniform, we would further improve the overhead by sharing type components among representations. Doing this would require being able to construct type representations from smaller components, which we do not currently support.

In addition to the object file overhead, the types file requires added type information for the new entities in the object file, notably annotations for: the various control point labels in the `dyninit` function, the strings and type representations stored in the static data segment, and the dummy values. Because the information in the types file is discarded following load-time verification, we do not analyze its overheads in detail.

Impact on Performance

These space overheads could affect runtime performance in three ways, but we argue that in practice, performance is only negligibly impacted:

1. *The additional space will occupy more memory in the application,* particularly in the application heap, and thereby could cause the application to page. We believe this will not be a problem for two reasons. First, memory is relatively inexpensive, so a larger physical memory requirement will not be cost-prohibitive. Second, much of the additional overhead will not contribute to the program's memory footprint, and can be paged out between updates. That is, the majority of the space overhead is

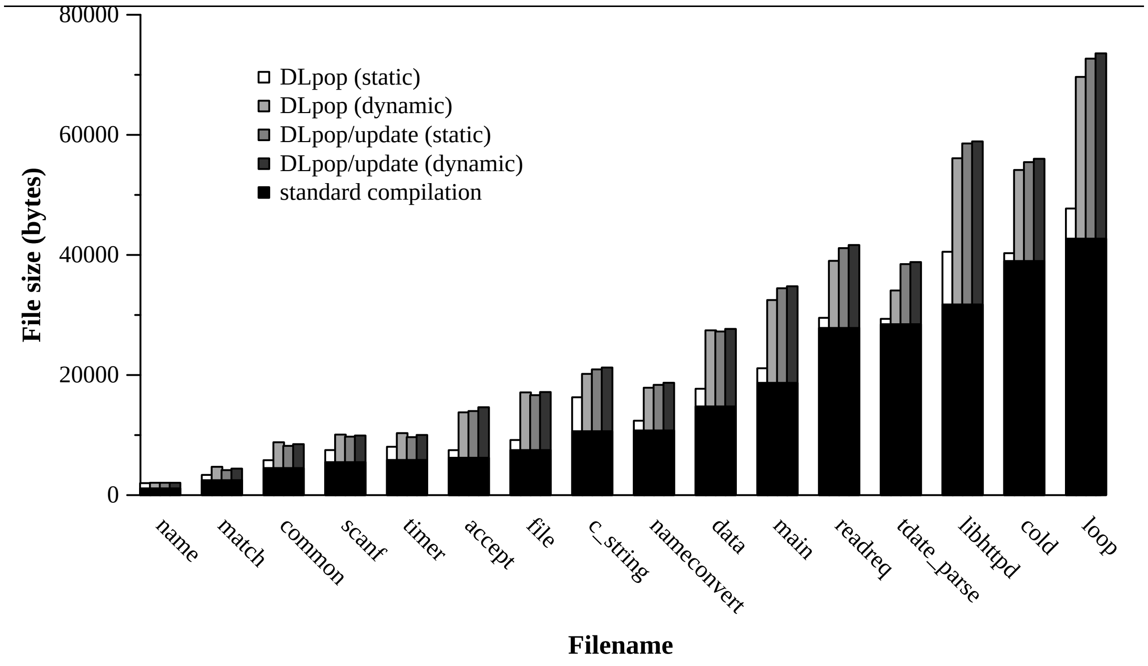


Figure 10.3: Space overhead for FlashEd 0.1 object files, compiled for loading or updating

only used at link-time, meaning that it will not be needed during normal execution. The link-time-only space overheads are due to the `dyninit` function, the symbol names and type representations, and the dummy functions; this constitutes all but the 5 bytes of overhead per symbol reference and 4 bytes per symbol definition to use the GOT. On the other hand, these elements are needed each time an update is performed, due to our strategy of relinking, so they cannot be permanently paged out. Using reference indirection (as described in §7.4.3), rather than relinking, would solve this problem, but it would introduce an extra level of indirection.

2. *The program's cache locality could be negatively influenced.* We do not expect this to be a problem for the same reasons as point 1. That is, the majority of the space overhead is only in use during linking, and therefore should not occupy space in the cache.

3. *Because dynamic linking and/or updating may be used in a distributed environment (*a la *Java), the extra space could result in added network transmission time when loading a file across the network.* As it turns out, types files and type representations are highly compressible (up to 90% using `gzip`), and therefore need not contribute to excessive network transmission time.

Overheads in Practice

Having considered the source of space overheads abstractly, we consider how these overheads manifest in practice. Figure 10.3 shows the measured increase in size for the object

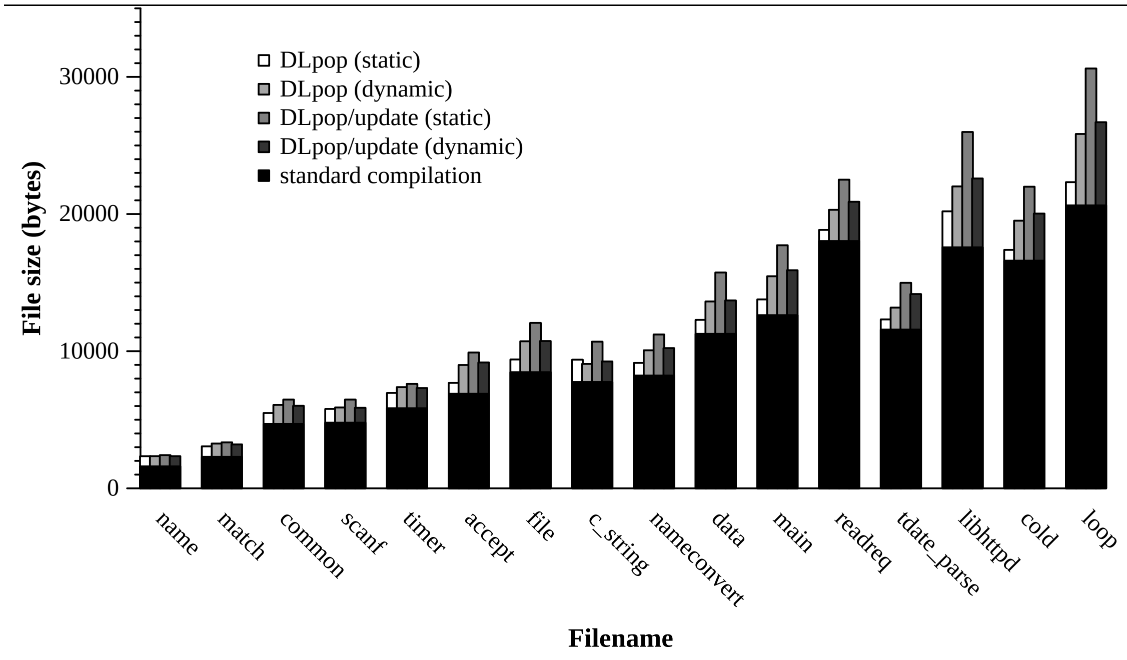


Figure 10.4: Space overhead for FlashEd 0.1 types files, compiled for loading or updating

files of FlashEd 0.1 due to support for dynamic linking and updating; Figure 10.4 shows the same increases for the types files.

In each figure, the X-axis shows the Popcorn file being compiled, and the Y-axis shows the size of the compiled file. Each file has a cluster of bars; the black part of each bar indicates the size of the file compiled 'normally,' without any support for dynamic linking or updating. Each bar in the cluster shows the space added to the file based on how it was compiled. In particular, the *DLpop (static)* bar indicates the added space for compiling a file to export its symbols to dynamically linked files; *DLpop (dynamic)* indicates the added space for compiling a file to be dynamically loadable; *DLpop/update (static)* is the file compiled to be statically linked but updateable; and *DLpop/update (dynamic)* is the file compiled to be loadable and updateable. These correspond to the columns shown in Table 10.3.

In the worst case, dynamic updating support increases the object file size by 135% (for `accept.pop`), and at best increases it by 36% (for `tdate_parse.pop`). However, considering percentages is somewhat misleading since the overhead is not really related to the file size, but the number of symbols it imports and exports, and the number of times imported symbols are referenced. We can see this by comparing `cold.pop`, which is itself roughly 39K and adds about 17K of information due to dynamic updating, and `libhttpd.pop`, which is about the same size (31K) but adds about 27K. File `cold.pop` has a few very large functions, importing 42 symbols but only exporting 18 symbols, while `libhttpd.pop` has many small functions and data, exporting 48 symbols and importing 43 symbols.

In general, dynamic updating adds greater overhead to both statically and dynamically linked files than does dynamic linking only. The difference is slight in dynamically loadable files (just the exported local symbols), but more pronounced in statically linked files, due to the `lookup`s required for updating. An interesting phenomenon is that while the object file size *increases* when updateable files are made loadable (*i.e* compare the *DLpop/update (static)* bar with the *DLpop/update (dynamic)* bar in Figure 10.3), the types file size *decreases* (compare the same bars in Figure 10.4). This is because statically linked files initialize GOT members with `extern` values resolved by the static linker, meaning that the MTAL interface file size (which is stored in the types file) increases over the dynamically loadable version, which defines its own dummy functions.

For the particularly interested reader, Table 10.5 presents the underlying information for these graphs. The table should be read row-wise. The 'no loading' column serves as the base case (0 overhead), and the rightmost columns indicate the space overhead of employing a particular compilation approach. The second two columns show the size, in bytes, of the object file and types file generated by compiling a Popcorn file normally. In each column that follows, the additional overhead for various compilation strategies is shown in bytes, relative to the 'no loading' case, for both the object file and the types file, and below it as a percentage of the 'no loading' size. The eight rightmost columns are grouped for dynamic linking and dynamic updating, considering both files that are statically linked and those that are dynamically linked.[3]

For example, the `accept` row indicates that compiling `accept.pop` normally results in an object file of size 6224 and a types file of size 6893. Compiling `accept.pop` to be involved in dynamic linking, but to be linked statically (*i.e.* to export its symbols to dynamically linked files), adds 1268 bytes to the object file and 794 bytes to the types file, as shown in columns 4 and 5. These additions constitute a 20% and 12% overhead, to the object and types files respectively, compared to normal compilation. Compiling `accept.pop` to be both updateable and dynamically loadable (columns 10 and 11) adds 8412 bytes to the object file and 2283 bytes to the types file, compared to normal compilation, constituting a 135% and 33% overhead to those files, respectively.

10.2 Application Performance

We have argued that the primary impact on dynamic updating performance is the added level of indirection per external symbol reference, due to the use of a GOT by our dynamic linker. Of course, how much this overhead affects *application performance* depends on the application. In particular, it depends on the ratio of time the application spends referring to external symbols as compared to performing other work. The higher this ratio, the greater the impact. In this section, we consider the performance of one particular application, the FlashEd webserver, to understand the overall effect of our dynamic updating overheads.

[3]Note that the measurements for dynamic linking include extra logic due to current support for updating (*i.e.*, they are compiled to use the three-pass, rather than the two-pass, algorithm, *etc.*). This causes them to be somewhat larger than they would be if compiled as described in Chapter 6.

File (`.pop`)	No loading		Dynamic Linking +b/%				Dynamic Updating +b/%			
			Static		Dynamic		Static		Dynamic	
	.o (b)	.to (b)	.o	.to	.o	.to	.o	.to	.o	.to
`accept`	6224	6893	1268 20%	794 12%	7560 121%	2100 30%	7780 125%	3009 44%	8412 135%	2283 33%
`c_string`	10644	7755	5652 53%	1625 21%	9548 90%	1315 17%	10296 97%	2938 38%	10608 100%	1494 19%
`cold`	38992	16599	1304 3%	786 5%	15160 39%	2915 18%	16468 42%	5393 32%	17024 44%	3434 21%
`common`	4512	4696	1308 29%	796 17%	4292 95%	1385 29%	3692 82%	1775 38%	3980 88%	1321 28%
`data`	14772	11264	2944 20%	1019 9%	12672 86%	2361 21%	12480 84%	4473 40%	12912 87%	2437 22%
`file`	7504	8471	1676 22%	924 11%	9604 128%	2252 27%	9148 122%	3592 42%	9668 129%	2270 27%
`libhttpd`	31756	17573	8772 28%	2625 15%	24348 77%	4442 25%	26812 84%	8410 48%	27156 86%	5020 29%
`loop`	42720	20631	5016 12%	1694 8%	26932 63%	5207 25%	29976 70%	9982 48%	30856 72%	6066 29%
`main`	18700	12628	2428 13%	1147 9%	13792 74%	2836 22%	15752 84%	5096 40%	16084 86%	3275 26%
`match`	2488	2295	880 35%	767 33%	2232 90%	970 42%	1680 68%	1057 46%	1932 78%	907 40%
`name`	1128	1604	872 77%	739 46%	932 83%	742 46%	932 83%	817 51%	932 83%	742 46%
`nameconvert`	10772	8221	1624 15%	924 11%	7116 66%	1844 22%	7592 70%	2996 36%	7944 74%	2005 24%
`readreq`	27844	18032	1684 6%	810 4%	11176 40%	2275 13%	13296 48%	4473 25%	13812 50%	2862 16%
`scanf`	5496	4785	2008 37%	1003 21%	4588 83%	1111 23%	4244 77%	1687 35%	4432 81%	1086 23%
`tdate_parse`	28468	11573	888 3%	743 6%	5616 20%	1604 14%	10008 35%	3411 29%	10348 36%	2594 22%
`timer`	5856	5842	2200 38%	1113 19%	4472 76%	1539 26%	3804 65%	1772 30%	4164 71%	1471 25%

Table 10.5: Space overhead for FlashEd 0.1 compiled for loading or updating

10.2.1 FlashEd Performance

To measure server performance, we used `httperf`, version 0.8, a a freely available webserver benchmarking system [MJ98]. `httperf` is a single, highly-parameterizable executable process that acts as an HTTP client. It can generate HTTP loads in a variety of ways, being able to simulate multiple clients by using non-blocking sockets. To ensure that the server is saturated, multiple `httperf` clients can be executed concurrently on different machines. Throughput is measured by sampling the server response at fixed intervals and then reporting the average and standard deviation at the end of the run; sampled throughput can also be printed during the test. In version 0.8, the sample time is fixed (at 5 seconds), and a run is only concluded after performing a set number of requests. We changed the code to allow the sample time to vary, and to allow fixed-time tests, ensuring an equal number of samples across various runs.

To understand the cost of updating, we compared statically compiled and updateable versions of the first three versions of FlashEd. In addition, we compared the difference between a version that was statically compiled with updating enabled, to the version that was actually patched on-line. We suspected that the latter case might have worse performance due to a larger memory footprint or poorer cache locality. In particular, it will retain the original version of the code in the text segment along with any new code, which is loaded into the heap.

Using the various versions of FlashEd, we ran two kinds of tests. In both cases, one machine ran the FlashEd server, and three machines ran `httperf` clients. In the first case, we ran a *log-based* test, used to simulate 'typical' client activity. Each client uses an identical *filelist* containing a list of files to request, and a corresponding weight for each file. Each request is determined pseudo-randomly, based on its weight. After the test has run for a specified time, all clients are halted, and relevant data from each is collated. While other performance metrics are potentially interesting, here we focus on *server throughput.* For the log-based test, throughput was measured in terms of the bytes served by the webserver per unit time. We also ran URL-based tests, to reduce the amount of variability, in which the `httperf` clients request the same URL constantly for the duration of the test. Throughput is reported here in terms of connections per second.

Log-based Test

For the log-based test, we used a filelist obtained from the WebStone benchmarking system [Web]; the file is shown in Figure 10.5. The first column indicates the URL and the second column indicates the weight to assign to it; the comment in the third column indicates the file's size in bytes. The WebStone documentation claims that this list is a fair representation of server load, at least for file-based traffic. To reduce the variability in the sampled numbers, we used 90 second sample-times. We ran each test for close to 32 minutes, yielding 21 samples. Because we observed skewed distributions in many cases, we report the median, rather than the mean, and use the quartiles to describe the variability.

Figure 10.6 shows the results of our measurements. The X-axis varies with server version; the first three columns show the throughput for FlashEd 0.1, 0.2, and 0.3, respectively, and the fourth column shows the throughput for Flash (compiled using `gcc` version egcs-2.91.66 with flag `-O2`) as a point of reference (we compare the performance

```
/file500.html   350     #500
/file5k.html    500     #5125
/file50k.html   140     #51250
/file500k.html  9       #512500
/file5m.html    1       #5248000
```

Figure 10.5: Filelist used in the log-based test.

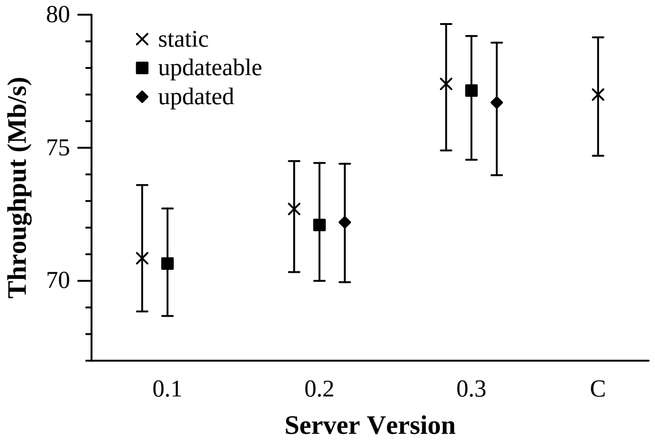


Figure 10.6: Flash and FlashEd throughput (Mbits/sec) for the log-based test

of Flash and FlashEd below). The Y-axis shows throughput in Mb/s (note that it does not start at 0). For each version of FlashEd, we measured the server's performance when it was compiled with and without updating support (labeled *static* and *updateable* in the figure, respectively), as well as when it was patched on-line (labeled *updated* in the figure); for example, the *updateable* FlashEd 0.3 was compiled directly from the version 0.3 sources, while *updated* FlashEd 0.3 was compiled from the version 0.1 and then patched twice dynamically. For each server we show the median throughput, with the quartiles as bars; because we have 21 trials, the range between these bars serves as a 98% confidence interval, as per [PG81].

The overhead due to updating is the difference in performance, per server version, between the medians of the static and updated/updateable versions. In all cases, this overhead is is between 0.3% and 0.9%, which is negligible when compared to the measured variability. For FlashEd 0.2, the updated code is slightly faster than the statically linked, updateable code, while the reverse is true for version 0.3. The fact that the relative and absolute locations of the code in an updated program is different than the updateable one may be one source of difference, since the same modules will be affected differently by cache policy. In addition, because the heap sizes are the same but the updated program uses some of its heap to store update code, the updated version garbage collects more often. However, in general this difference is well within the confidence interval of the numbers and may be due to experimental variation rather than deterministic difference.

URL Test

The log-based test characterizes 'typical' performance, but has two shortcomings. First, the actual activity seen by the server is variable from sample to sample, since the URLs requested in aggregate by the three clients may differ during each sample. We extended the sample time to 90 seconds to mitigate this problem, but each value in Figure 10.6 has a (relatively-speaking) sizeable variability. In particular, the *semi-interquartile range* (SIQR), which is the difference between the high and low quartiles divided by two, is roughly 3% of the median, as compared to a SIQR of $< 1\%$ of the median for the tests we are about to describe. The second problem is that the log-based test provides less of a sense of 'worst-case' overhead, because some of the files requested during the test are quite large, and thus the I/O time dominates the overhead imposed by updating. To address the problem of per-sample variability, we ran tests that request the same URL repeatedly. To examine how I/O dominates updating overhead, we considered a variety of URL file sizes, from 500 B files to 500 KB files. For each URL, we used a 10 second sample time, and ran each test for just over 5 minutes, totalling 31 samples. Again, we calculated the median and the quartiles.

The results are shown in Figure 10.7, which has the same format as Figure 10.6. The first thing to notice here is that the variability is much decreased; in particular the SIQR is typically less than 0.5% of the median, and except in the case of 500k files, the inter-quartile ranges rarely overlap for each cluster of points. The range of the Y-axis differs for each graph, with none starting at 0. The error bars for the 500k files have the same size as the rest but appear more significant because the scale of the Y-axis is smaller.

To understand the trends exhibited in this graph, we graphed the overhead due to updating, shown in Figure 10.8, where the X-axis is URL file size—shown for 500 B, 1 KB, 10 KB, and 500 KB files—using a logarithmic scale for presentation purposes, and the Y-axis is percent overhead from the non-updateable (static) FlashEd. There are three basic trends:

1. The overhead for updateability decreases as the size of the file increases. For the 500 byte file, we see as much as a 2.3% overhead, while for the 500 kilobyte file the overhead is 0. This is most likely because the added I/O time overwhelms the extra processing cost.

2. In general, the relative overhead due to updating decreases as the version number of FlashEd increases. This can be seen in the Figure by comparing all of the "updateable" lines (whose points are marked with boxes) and comparing all of the "updated" lines (whose points are marked with $\times$).

   There are two explanations for this phenomenon. First, because the processing time per request decreases for each file, while the network transfer time remains the same, the impact of updating is decreased. Second, there are fewer external references made for version 0.3 than for versions 0.1 and 0.2. Because the runtime penalty of an extra indirection occurs only when references are to definitions not in the current file, the fewer these kinds of references, the lower the overhead. To discover the relevant fraction of references, we modified the Popcorn compiler to insert counters for global references, whether to data or functions, differentiating between references

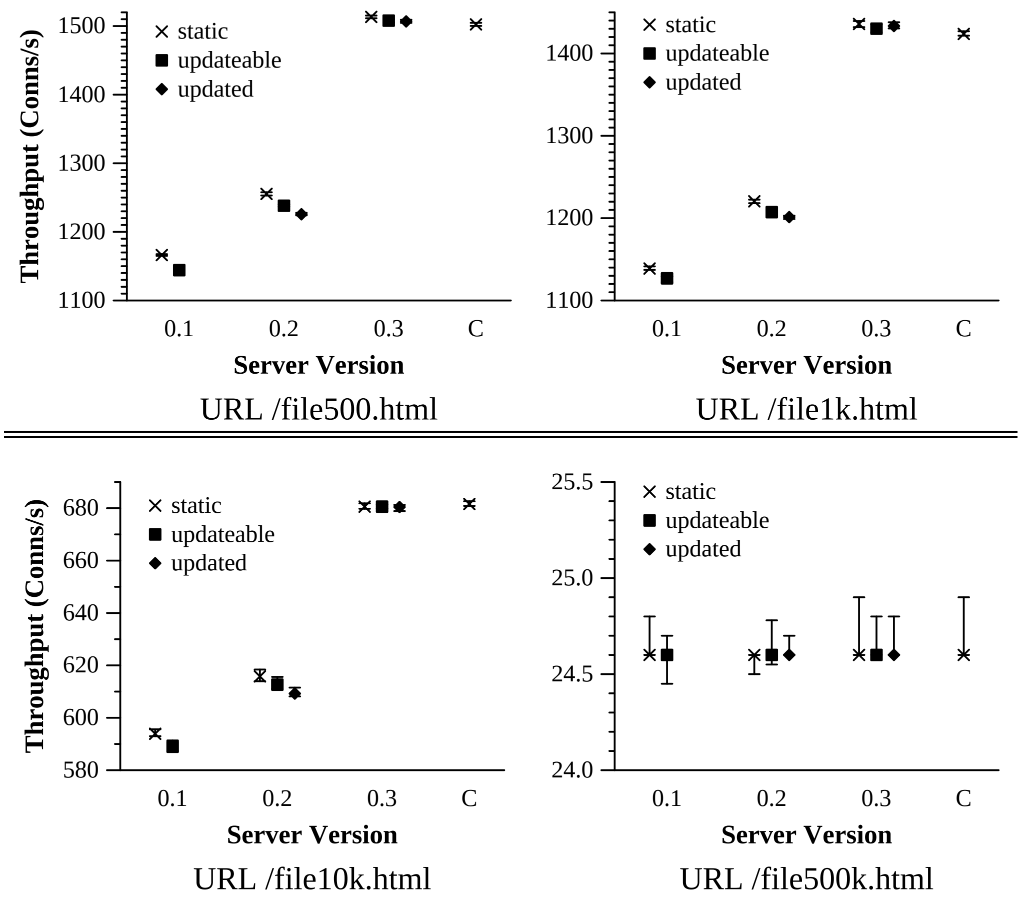


Figure 10.7: Flash throughput for URL-based tests

to external and static variables. We then re-built the three versions of FlashEd and ran our tests on each of them. For versions 0.1, 0.2, and 0.3 of the server, the percentage of dynamic references to external definitions was 70%, 71%, and 62%, respectively, meaning that version 0.3 incurs a lower penalty for indirection, relative to its non-updateable version.

3. The relative overhead of updated and updateable versions is inconsistent; that is, sometimes the updated version performs better and sometimes the updateable one does. This follows the same pattern as the log-based test.

C Flash

As a point of reference, we compared the performance of the fully-optimized, C version of Flash with FlashEd; the results of the comparison were shown in Figures 10.6 and 10.7. In general, its performance was quite similar to FlashEd version 0.3. In particular, the performance of Flash and FlashEd for the larger files is nearly identical, while the performance

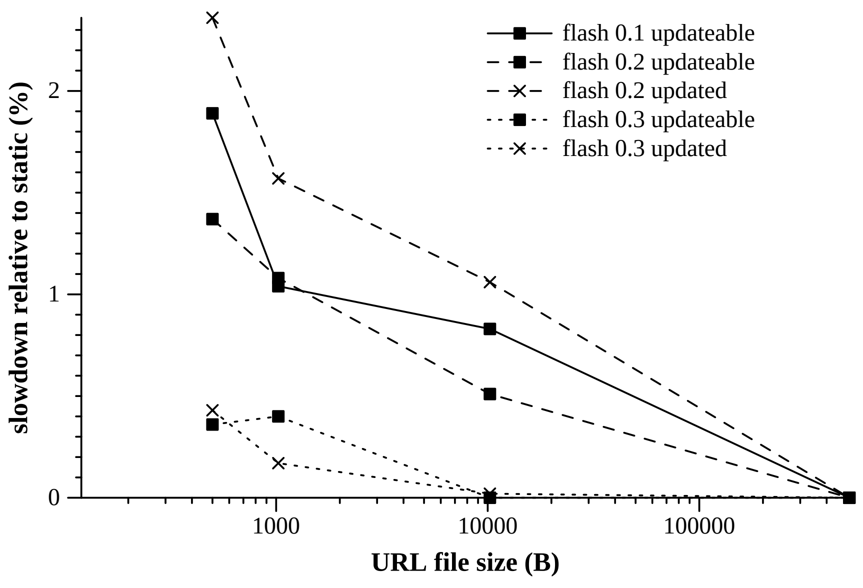


Figure 10.8: Correlating the overhead of updateability with URL file size

of Flash for the smaller files and for the log-based test is slightly worse.

We expected Flash to consistently outperform FlashEd, so we are surprised that for version 0.3 of FlashEd, the reverse is sometimes true. Of course, much of the cost of file processing is due to I/O; a more CPU intensive task would certainly favor the C implementation. In any case, we can draw the conclusion that TAL, and PCC in general, is a viable platform for medium-performance, I/O-intensive applications.

Chapter 11

Future Work

While our work is a significant advance in the state of the art, there are many areas of future study. In this section, we consider these areas more closely. First, we look at the relevance of our approach for other kinds of programming languages, notably functional and object-oriented languages. In particular, we look at the difficulty of encoding closures and classes to be updateable using our approach. Second, we look more closely at the question of update validity, and briefly formulate an idea to facilitate reasoning about the stateful properties of modules, in isolation from the rest of the program. Third, we look at a prime application domain for our technology, Active Networks, and discuss further work that is to be done to realize our technology in active networks. Finally, we consider some other features worth supporting, including unchecked updates, secure linking, and updating abstract types.

11.1 Functional Languages

The core reason for the additional complexity in dealing with function and data pointers, as described in §7.4.1, comes from our notion of dynamic patch. In our approach, only updates to code are performed 'automatically,' while updating data, which may contain pointers to updateable definitions, is left to the state transformer function.

This approach is not unreasonable in an imperative language such as Popcorn, which makes infrequent use of function pointers. However, applying our approach to functional programming languages may result in far greater difficulty. This is because these languages make extensive use of function pointers, or more specifically *closures*, which are function pointers combined with an *environment*, and so writing a state transformer to find all of these pointers would be very difficult in general, and essentially impossible if pointers are hidden inside of inaccessible, abstract data.

We first consider an alternate approach to updating function pointers, and then consider how the developed technique could be applied to closures.

11.1.1 Pointers to Updateable Definitions

We might consider a notion of patch that says if program data contains references to updateable definitions, then this data should be updated 'automatically' to point to an updated definition, rather than requiring the programmer to do it in the state transformer function, as described in §7.4.1. This 'automatic' approach is taken by Dynamic

ML [GKW97], for the functional language Standard ML [MTHM97]. In the case of Dynamic ML, however, the cost is a more complicated, less trustworthy implementation.

One possibility for implementing such a semantics without adding to the TCB would be to introduce, at compile-time, an extra indirection when a pointer is treated as data. In particular, rather than storing the pointer itself into a variable, we instead store a pointer *to that variable's GOT entry.* When the pointer is extracted from the variable, it must be dereferenced an extra time at some point before it is used; the closer it is dereferenced to when it is actually used, the better, so that it notices the most recent changes to the GOT following relinking.

Consider once again the example in Figure 7.15 (page 112), which manipulates function pointers. We want to compare how this file is normally compiled to be updateable with how we would compile it using reference indirection for function pointers. Figure 11.1 shows this file transformed to be loadable and updateable in standard manner, as described in Chapter 7; there are no surprises here. Figure 11.2 is the same transformation, but with the added indirection for function pointers through the GOT; the differences from Figure 11.1 are shown boxed. The first thing to notice is that the code from `fnptr.pop` has been changed so that declarations having type `int (int)` have been changed to have type `%(int (int))`. This includes the declaration of the `fnptr` struct type, and in the argument to `change_f`. As a result, the actual use of `ptr_intfn` in function `g` requires an extra dereference, just before it is called. The `%()` syntax indicates a *read-only tuple*, and is analogous to the `*()` syntax for writable tuples. Read-only tuples are created by making a regular tuple, and then casting it to be read-only; in the figure this is shown for the field of the variable `ptr_intfn` as the code `(:%(int (int)))(new (fn__3))`. The reason that tuples are made read-only is explained shortly.

The first time a function is referenced by name, we use the *address of its GOT entry*, rather than the entry itself, delaying its dereference to the actual use; we call this GOT address a *tupled function pointer*, since if the original function had type `x (x)`, the GOT entry address will have type *(x (x)). If the function is defined in the file itself, it normally does not have a GOT entry, and so we need to make one. In the case of the example, a GOT entry is created for the function `f` (and is properly initialized to `f`). Furthermore, we need to initialize the global variable `ptr_intfn` to the GOT's entry in the `dyninit` function: `ptr_intfn.f = &GOT.f`.

For tupled function pointers to work properly with polymorphism, we had to allow type applications to go 'under the tuple.' For example, consider the polymorphic list operations `fold_right` and `append` which can be used to code the function flatten:

```
extern b fold_right<a,b> (b f(a,b), <a>list x, b accum);
extern <a>list append<a> (<a>list x, <a>list y);
<a>list flatten<a> (<<a>list>list x)
  return (fold_right(append@<a>, x, null));
```

Compiling this file to be loadable and updateable yields (portions omitted):

```
static int f (int x) {
  return (x + 1);
}
struct fnptr { int f (int); }
static fnptr ptr_intfn = new fnptr(f);
static void change_f (int intfn (int)) {
  ptr_intfn.f = intfn;
}
static int g (int x) {
  return (ptr_intfn.f(x) + 1);
}
static bool looked_up_old_flag = false;
static bool is_updated_flag = false;
void dyninit_fnptr<b,c> (a lookup <a>(b,string,<a>rep),
                         b lookup_closure,
                         void update <a>(c,string,<a>rep,a),
                         c update_closure,
                         bool no_init) {
  if (!looked_up_old_flag) {
    looked_up_old_flag = true;
    if (no_init) return;
  }
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "fnptr?Local?ptr_intfn",
           repterm@<*(fnptr)>, &ptr_intfn);
    update(update_closure, "g", repterm@<int (int)>, g);
    update(update_closure, "f", repterm@<int (int)>, f);
    update(update_closure, "change_f",
           repterm@<void (int (int))>, change_f);
  }
}
```

Figure 11.1: Transforming `fnptr.pop` (Figure 7.15) in the standard manner to be loadable and updateable.

```
static int f (int x) {
  return (x + 1);
}
struct fnptr { %(int (int)) f; }
static fnptr ptr_intfn = new fnptr((:%(int (int)))(new (fn__3)));
static void change_f (%(int (int)) intfn) {
  ptr_intfn.f = intfn;
}
static int g (int x) {
  return (ptr_intfn.f.1(x) + 1);
}
static exception exncon__2(string);
static int fn__3 (int a) {
  raise (new exncon__2("f"));
}
static GOT_t GOT = new GOT_t{f=&f};
static struct GOT_t { int f (int); }
static bool is_updated_flag = false;
static bool looked_up_old_flag = false;
void dyninit_fnptr<b,c> (a lookup<a>(b,string,<a>rep), b
                         lookup_closure,
                         void update<a>(c,string,<a>rep,a),
                         c update_closure,
                         bool no_init) {
  if (!looked_up_old_flag) {
    looked_up_old_flag = true;
    if (no_init) return;
  }
  if (!is_updated_flag) {
    is_updated_flag = true;
    update(update_closure, "fnptr?Local?ptr_intfn",
           repterm@<*(fnptr)>, &ptr_intfn);
    update(update_closure, "g", repterm@<int (int)>, g);
    update(update_closure, "f", repterm@<int (int)>, f);
    update(update_closure, "change_f",
           repterm@<void (%(int (int)))>, change_f);
  }
  ptr_intfn.f = &GOT.f;
}
```

Figure 11.2: Transforming `fnptr.pop` (Figure 7.15) to automatically notice updates to function pointers. Differences from Figure 11.1 are boxed.

```
static struct GOT_t {
  b fold_right <a,b>(%(b (a,b)),<a>list,b);
  <a>list append <a>(<a>list,<a>list);
}
static GOT_t GOT = ...
static <a>list flatten<a> (<<a>list>list x)
  return (((:%(b <a,b>(%(b (a,b)),<a>list,b)))(&GOT.fold_right)).1(
          (:%(<a>list <a>(<a>list,<a>list)))(&GOT.append)@<a>,
          x,
          null));
```

The key is the boxed part: the type application is applied to the tuple `&GOT.append` (cast to be read-only), rather than `GOT.append`, which is the function itself. We have made modifications to the TAL verifier and Popcorn type-checker to allow this sort of 'deep' type application to occur. Allowing a type application to occur under the tuple is only sound if the tuple's value cannot be changed, which is the reason we must use read-only tuples. Consider the following code:

```
a   id<a>(a   x) { return x; }
int inc  (int x) { return x+1; }
void foo() {
  *(a <a>(a))  f = new (id);
  *(int (int)) g = f@<int>;
  g.1 = inc;        // should be illegal
  *(int) x = new (43);
  f.1(x);           // error!
}
```

We have the polymorphic `id` function from α to α, for all types α, and the increment function `inc` from `int` to `int`. In the body of `foo`, we create a tupled function pointer `f`, initialized to the `id` function, having type `*(a <a>(a))`. We then *alias* this tuple in `g`, but with refined type `*(int (int))`; we are OK so far. The problem occurs on the third line, in which we change the contents of `g` to the function `inc`, and because `g` is an alias of `f`, change the contents of `f` as well, constituting a type error. That is, `f` should contain a function of type `a<a>(a)` but now contains a function of type `int (int)`. As a result, invoking `f` on the tuple `new (43)` increments the pointer by 1 (not the contents of the tuple, but the tuple itself). Therefore, coercions under writable tuples are unsound.

We can explain somewhat informally why applying a type-application under a read-only tuple is sound.[1] Say we have three type modifiers $Source()$ (read-only), $Sink()$ (write-only), and $Reference()$ (read-write). It is known that the following subtyping relations are sound (they were formulated in Reynolds's language Forsythe [Rey96]):

$$\frac{s \leq t}{Source(s) \leq Source(t)} \qquad \frac{s \leq t}{Sink(s) \leq Sink(t)}$$

$$Reference(t) \leq Sink(t)$$

[1]Many thanks to Benjamin Pierce for helping me work this out.

$$Reference(t) \leq Source(t)$$

Furthermore, the following is rule is known to be sound (but intractable) [Mit86]:

$$\forall\alpha.t \leq t[a/s]$$

where $t[\alpha/s]$ denotes the capture-avoiding substitution of all occurrences of α in t with s. The rule states that the universal type may be used wherever a particular instantiation of that universal type may be used; this assumes type-erasure semantics so that the underlying representation of the $\forall$ type and the instantiated type is the same.

Using these rules, we can correctly can approximate type instantiations 'under the tuple' by passing a source containing a polymorphic function. Given the standard application and subtyping type-checking rules:

$$\frac{e_1 : t' \rightarrow t \quad e_2 : t'}{e_1\ e_2 : t} \qquad \frac{e : s \quad e_2 : s \leq t}{e : t}$$

and the functions:

$$\begin{array}{ll} f: & Source(\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int} \\ g: & Source(\forall\alpha.\alpha \rightarrow \alpha) \end{array}$$

we can type the term $f\ g$ as follows:

$$\cfrac{\cfrac{\forall\alpha.\alpha \rightarrow \alpha \leq \alpha \rightarrow \alpha[\alpha/\texttt{int}]}{\forall\alpha.\alpha \rightarrow \alpha \leq \texttt{int} \rightarrow \texttt{int}}}{Source(\forall\alpha.\alpha \rightarrow \alpha) \leq Source(\texttt{int} \rightarrow \texttt{int})}$$

$$\frac{g : Source(\forall\alpha.\alpha \rightarrow \alpha) \qquad Source(\forall\alpha.\alpha \rightarrow \alpha) \leq Source(\texttt{int} \rightarrow \texttt{int})}{g : Source(\texttt{int} \rightarrow \texttt{int})}$$

$$\frac{f : Source(\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int} \qquad g : Source(\texttt{int} \rightarrow \texttt{int})}{f\ g : \texttt{int}}$$

In place of the polymorphic subtyping rule, we propose to add a simpler rule that is tailored to type application under sources:

$$\frac{e : Source(\forall\alpha.t_1)}{e[t_2] : Source(t_1[\alpha/t_2])}$$

Intuitively, this rule is used as a hint to the type-checker as to where to apply the polymorphic subtyping rule. This simplifies type-checking, and may be key for decidability. Formally proving that TAL is sound with this addition is future work. We have focused on type application, but really type application is just one instance of the more general problem of 'deep coercions,' which deserves future study in its own context.

```
// a closure is a function pointer and abstract environment
extern abstype <a,b>fn[c] = *(b f(c,a), c);

// make a closure out of a function pointer and environment
extern <a,b>fn make_fn<a,b,c>(b f(c,a), c);

// apply closure f to argument x
extern b apply<a,b>(<a,b>fn f,a x);
```

Figure 11.3: The interface to Popcorn's `Fn` module

11.1.2 Closures

Using this approach greatly simplifies the process of constructing dynamic patches, and solves the problem of how to update pointerful data that is part of an abstract type.[2] This latter point, while useful in general, is particularly helpful for allowing *closures* to be updated. Typed encodings [MMH96] typically package a function pointer and its environment inside an existential type:

$$\exists e.(e \times \tau_1 \rightarrow \tau_2) \times e$$

Here, e is the type of the environment, and a closure consists of a 1) function that takes an environment e along with its argument type τ_1 returning type τ_2, and 2) an actual environment. Not only could the function part of the closure be updated, but relevant definitions pointed to by the environment could be updated as well.

To see how we can apply the indirection approach to closures, consider the encoding of closures provided in Popcorn's `Fn` library. The key portions of its interface is shown in Figure 11.3.

The existential type declaration `fn` is the type of closures, and is the same as the type presented above except that it is polymorphic:

$$\forall[a, b].\exists c.(c \times a \rightarrow b) \times c$$

The function `make_fn` constructs a closure from a function pointer and an environment, and the function `apply` applies a closure to an argument; this function unpacks the existential package and then calls the function pointer with its environment and the provided argument:

```
// apply closure f to argument x
b apply<a,b>(<a,b>fn f, a x) {
  with f[c] = f do
    return f.1(f.2,x);
}
```

[2]We defer further explanation as to why we have not yet adopted this approach to §11.1.3, below.

By applying our transformation to the `Fn` interface, we get:

```
extern abstype <a,b>fn[c] = *(*(b (c,a)) f, c);
extern <a,b>fn make_fn<a,b,c>(f *(b (c,a)), c);
extern b apply<a,b>(<a,b>fn f,a x);
```

Now closures store a tupled function pointer, and similarly `make_fn` requires a tupled pointer as its argument. The `apply` function becomes:

```
// apply closure f to argument x
b apply<a,b>(<a,b>fn f, a x) {
  with f[c] = f do
    return f.1.1(f.2,x);
}
```

An extra dereference is added just before the invocation. This way, if the implementation of the function changed, the dereference will point to the new version. The environment's contents are made updateable as well (by transforming the environment before it is provided as an argument to `make_fn`).

11.1.3 Limitations

While the approach described here captures the essence of functional languages, and would simplify patch construction in Popcorn, we clearly need experience with a real functional source language to see how all of its other features interact with the transformation. There are some problems with the approach as well. For Popcorn, we have mostly implemented the transformation for indirecting function pointers described here, but not adopted it for several reasons:

1. While it seems that intuitively the transformation is complete—that is, that the program will 'notice' all pointerful data has been updated—we have not proven that this is so. This requires some more work.

2. The transformation changes the types of functions. This results in two problems. First, because of the translation that we have been using, both files compiled to be updateable and files compiled to only be loadable can be freely intermixed in a program. More importantly, the same libraries, compiling to be updateable, can be used by code that is statically linked, if at a slight performance penalty. As a result, we need only one set of system libraries. If we change the types of functions, this is no longer the case. On the other hand, having multiple sets of libraries, in our case versions that are and are not updateable, is not uncommon. For example, many libraries on UNIX systems have historically been compiled with and without profiling information; which library to use is determined at link-time, so as not to inconvenience the programmer.

   Second, changing the function type may invalidate type representations used in conjunction with that function. For example, consider the following source code:

```
int f(int g(int), int x) =
  dlsym(h,"f",repterm@<int (int (int), int)>);
f(g,1);
```

This code looks up a function `"f"` using some handle `h`; the function `f` takes another function `g` as its argument. When translated, this code would be

```
int f(%(int (int)) g, int x)) =
  dlsym(h,"f",repterm@<int (int (int), int)>);
f(&GOT.g,1);
```

There is now a mismatch between the actual type of `f` and the type passed to `dlsym`; the former has tupled its function pointer argument but the type representation argument to `dlsym` has not been similarly tupled. It is not obviously clear that automatically tupling functional parameter types in type representations is correct; this will require more investigation.

3. The DLpop library itself uses function pointers, *e.g.* in calling the `dyninit` functions of loaded code, and compiling it to indirect these pointers is problematic. This is due to the disconnect between the compiler and the library: if the compiler is going to add indirections for function pointers when compiling DLpop, then the library must be aware of this.

   To avoid this problem, we thought of compiling the DLpop library in the standard manner, not having its function pointers indirected. The problem here is that it calls other library routines that take function pointers as arguments, *e.g.* `Hashtable::iter` which takes a function to operate on each element of the provided hashtable. It would then call this function with a normal function pointer, but the `Hashtable` module would have been compiled to expect a tupled function pointer.

4. This discussion has been focused on function pointers, but the same problems arise with pointers to data. That is, if we were to take the address of some top-level definition (say, defining a variable of type `int`) and store it in an array, and that definition were later updated, the pointer in the array would point to the old definition. It may be that identifying pointerful data and transforming code that manipulates it (just as we done here for function pointers) is straightforward, but we have not looked into it.

11.2 Object-oriented Languages

As a first step at understanding how our approach might be applied to object-oriented (OO) languages, we can consider how OO features are *encoded* in functional languages, and then draw on our previous discussion. A popular encoding of OO features in λ-calculi is the Pierce-Turner object encoding [PT94]. It is framed in the language $F^{\omega}_{\leq}$, an extension of the polymorphic λ-calculus that includes records, subtyping, and type operators. TAL's type system includes these features (though subtyping is more restricted), implying that

TAL should be a sufficient target for OO languages. While the encoding of [PT94] is functional, Pierce later showed that it applies with only slight alteration to imperative settings as well [Pie93].

Objects are encoded as records having existential type, where the type of the state is abstracted, and the record consists of two fields, one containing the state and the other containing the methods on that state. For example, a single-dimensional `point` object, having methods `getX` and `setX`, has type:

$$\begin{aligned}\texttt{point} = \exists R.\{\ & \texttt{state} : R,\\ & \texttt{methods} : \{\ \texttt{getX} : R \to \texttt{int},\ \texttt{setX} : R \to \texttt{int} \to R\ \}\ \}\end{aligned}$$

To use an element of type `point`, we have to unpack the existential, and then call the desired method with the object state and any additional arguments. For example, to invoke the `getX` method on some object p, we would do:

```
let {R, r} = p in
  r.methods.getX r.state
end;
```

The subtyping features of $F^{\omega}_{\leq}$ allow for structural notions of subtyping. For example, a different version of points, `cpoint`, might define the same methods as `point`, but additionally include methods to set and get a color. This version could be used in places where regular points are used, by virtue of the fact that its `methods` record contains the same functions as that of `point`. From these basic notions, [PT94] builds up more traditional abstractions of OO programming, including classes, inheritance, and references to `self` (*a.k.a.* `this`).

Objects encoded in this manner are quite similar to closures, and we could use the same techniques as described in the previous section to make the objects updateable. However, real-world object-oriented languages have features that are difficult to encode in this way. For example, Java's scoping allows one object to access the fields of an other object of the same class. Also, Java's use of `instanceof` implies that objects must be identifiable by their class at runtime. One way to do this is to use 'identity tags,' similar to the ones used in exceptions in Popcorn (Glew [Gle00] describes one such tagging mechanism). We have found tag-based identity to be problematic in updating exceptions (see §7.4.1). In general, as with functional languages, more experience is needed with real OO source languages, both to determine which OO features interact poorly with our transformation, and more fundamentally to see if TAL is a suitable target for OO languages.

11.3 Update Validity and State Visibility

As we explained in §8.2, allowing code to change arbitrarily can result in incorrect behavior if timing is not considered. While we believe that the invoke model simplifies finding correct update points, much work could be done to aid the programmer in this process. In particular, we hope that formalizing the conditions relevant to updating may enable automated determination and proof of valid update points. Previous formal work [GJB96, Lee83, FS91] can serve as a starting point for this investigation.

We could simplify the process of reasoning about a program and its updates by being able to consider individual modules in isolation. More specifically, we could consider how information about a module's state could be made visible to other modules in the program. If we knew precisely the ways in which other program modules access and manipulate a module's state, we could better how to update that module. We have thought a little about this notion of *state visibility* and present some of the details here. In particular, we look at classes of program state, consider the visibility of that state, and indicate how the state may be correctly updated.

11.3.1 Globally Visible State

State is globally visible when it is directly accessible to the rest of the program, defined as a non-abstract type. In this case, the state is visible to the entirety of the program: other modules may manipulate that state at any time and in any manner they desire. When programming in a modular style, globally visible state can sometimes be avoided in favor of particular functions to manipulate state, as with an ADT or module-protected state, explained next.

11.3.2 Module-protected State

Unlike globally visible state, module-protected state is only visible to functions in its module. Thus, we know exactly which functions may manipulate the state, and can insert mapper procedures for those functions to alter the state. For example, say we have a module `Routing` that administers the node routing table:

```
static <*(host,host)>list routing_table = null;
host lookup(host dest) {
  lookup dest in routing_table returning the result
}
```

Suppose we want to change the type of the routing table to include a route metric, stored as an integer. We might naively think that we could simply translate the old state to the new inside the `init` function at load time. However, if code in the module were currently manipulating that state, it might become inconsistent. Past approaches have sought to prevent inconsistency by forbidding updates from occurring when the module to be updated is running. However, our goal has been to find ways such that these constraints are not necessary. One way is to write stub functions to translate the state before executing any new functions:

```
static <*(host,int,host)>list New::routing_table = null;
extern <*(host,host)>list Routing::Local::routing_table = null;
static bool done_translation = false;
host Stub::lookup(host dest) {
  if (!done_translation) {
    do the translation from the old to the new
  }
  return New::lookup(dest);
}
```

This way, any code in the old module that was operating on the table when the new version was loaded will complete its operation on the old representation, and the next operation on the table will perform the translation. To eliminate the check to see if the translation had been done, we could have the stub update itself, as described in §7.1.1.

There is some concern about maintaining mutual exclusion. For example, if multiple threads may access the router table, it may be that after an update, one thread is using the old version when another thread enters the new version. This means that the two threads could potentially operate on the different versions of the table. However, we point out that if the code could allow more than one thread to access its state, then we would expect appropriate protections, *i.e.* mutexes, to be in place. The translation code could similarly make use of these mutexes to alter the state safely. Stating this more formally would be useful future work.

11.3.3 Thread-maintained State

Thread-maintained state is local to a particular thread. In Erlang, this is essentially the only kind of available state, as no mutation is possible: *server* threads are crafted as tail-recursive functions that carry their state as arguments. The only way clients may access thread-local state is via some communication with the thread, such as through message passing [AVWW96] or events [Rep99]. Again, this isolates the data, making it simpler to reason about changes. One example of updating a server thread in Erlang is found in [Arm97]; we could easily apply this technique in our context.

11.3.4 Abstract Data

Data having abstract type may only be manipulated by certain functions, while the rest of the program must treat it *abstractly*, providing essentially the same sort visibility as module-protected state. Dynamic ML [GKW97] exploits this property by allowing the implementation of abstract data throughout the program and its manipulation functions to be changed together. The drawback is that transformation of state may never be incremental, requires complex runtime support, and forbids manipulation functions from being in use during the update.

Because we implement type updating, abstract or otherwise, by renaming, we would allow different versions of abstract types and their data to coexist, such that newer versions of the ADT would have a different name. Say, for example, module `SymbolTable` implements some abstract type `t` to represent symbol tables, and module `Client` maintains some data of type `SymbolTable.t`. Later, we upload some new code `NewSymbolTable` that improves on the implementation of `SymbolTable`. `NewSymbolTable` will also contain routines that translate data of type `SymbolTable.t` to `NewSymbolTable.t`. To update `Client` to use the new symbol table, we would use a technique exactly like that for module-protected state, using these translation routines.

A potential drawback of this approach is that `NewSymbolTable` may only code its conversion routines using functions exported from `SymbolTable` (as proposed in [Gup94] for object-oriented programs). This has the advantage that the 'security' of the old implementation is preserved. However, this could limit the ability to perform conversions, since the data representation is not accessible. Dynamic ML [GKW97] takes the approach of

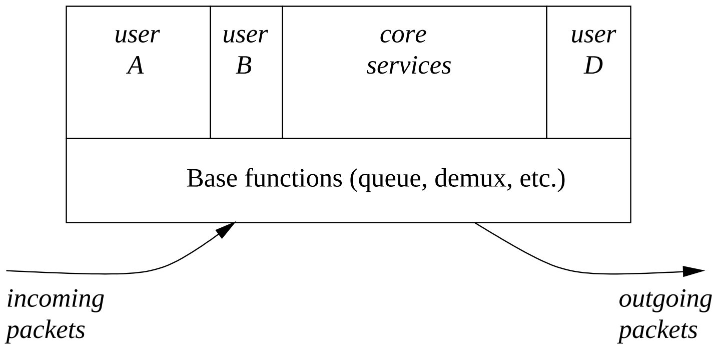


Figure 11.4: An Active Router supporting user-extensions

allowing the updating code have access to the representation of the old version. With some further work to our system, we could allow this as well. We touch on this idea below.

11.4 Active Networks

Active networks (AN) are networks whose elements are, in some way, programmable. Many prototype active network systems provide this programmability via router extensibility: routers can be extended with loaded code to implement new functionality, and in some cases extensions can be unloaded as well. Systems that take this approach include ALIEN [Ale98], PLANet [HMA$^+$99], Netscript [YdS96], and CANES [MBC$^+$99], to name a few. In these systems, code extensions are *plug-ins*, constrained to interface the system in a predefined manner. For example, in ALIEN, extensions receive packets by registering with the queuing machinery in the so-called *Core Switchlet*. In PLANet, extensions are *services* to be called by packet programs; as such, they are constrained to match the interface expected by the PLAN interpreter. More on this notion and the limitations of plug-in extensibility can be found in [HN00].

In general, the usefulness of these systems can be enhanced by using dynamic software updating. In particular, rather than being limited to loading and unloading extensions matching a predefined interface, any component of the running system can be potentially altered. However, updating modules in an active network router is more difficult than the examples we have been considering so far. The central reason is that the exact makeup of the code in the system is not known to all code updaters. In particular, a router is likely to have the shape shown in Figure 11.4.

It consists of some basic functions and core services, which are provided by the router's owner, as well as pockets of code that have been loaded by various users. In a router that supports dynamic software updating, a user may wish to update his piece of the code, but does so *unaware of some or all of the clients of that code.* That is, if user A wished to update a repository of information maintained by a service that he loaded, say by changing its type, he may inadvertently break clients of that code. To support updates without complete knowledge of the code making up the system requires (at least) the following characteristics:

1. The system must be able to enforce boundaries between user code. In particular, a user A should not be able to modify or access bindings in user B's code without permission. This implies proper presentation of the namespace during linking, based on a user's credentials. Our approach to dynamic linking pays off here, since the dynamic symbol table can be implemented (even updated) to do this, and the type namespace can be controlled by using *type heap masks* during `load`(see §5.5 and §11.5.2, below).

   Similarly, we must make sure that user code is safe. It could be that one user loads some code expecting to use some service, but at an old interface. Since we verify the code before running it, the TAL verifier would detect that the code is using an incorrect interface, and the load could be aborted.

2. An updated module must provide a way for old clients to call new code at the old type. We provide this characteristic with stub functions. Global data may not change type if other code that is not updated refers to it. This is ensured during the relinking process; when an old client's `dyninit` function is called, it will perform a lookup for that global variable at the old type, which will result in an error that causes rollback to occur.

3. Because clients may store pointers to updated code, there must be a way to update those pointers. We currently meet this requirement through the use of the `update_syms` and `re_init` functions, which are called during relinking (see §7.4.1). Alternatively, if we move to the indirection approach for pointerful data described above (§11.1), changes will be reflected automatically.

4. Changes to named types must be supported in isolation. In particular, if a service-provider redefines a named type, it should be easy for a client to use that type. In a type-replacement strategy, this is completely straightforward. Since we use type-renaming, there must be a way to communicate the new name for the type to other clients. However, this is simple since it can be calculated from the MD5 hash of the implementation.

5. The whole system must be structured so that updates are valid. Here, the idea of state visibility, explored above, is handy, since it considers updates to modules in isolation of the entire system.

Clearly, many of the mechanisms are in place to realize code updating in an active router. However, two areas need future study. First, updating abstract types is difficult because a new name for that type cannot be generated automatically, since a client may not know the implementation of the type. Second, we need to better understand how to restrict the structure of a system so that its evolutionary ability is not compromised, and so that updates are reflected correctly. For instance, services should not, in general, export global data, but use module-protected state instead. If some other user's code directly accesses the global state, then the state will not be updateable unless the client is updated along with the state. In addition to restrictions to support validity described in 8.2, there may be other practical restrictions on an active router's structure.

11.5 Other Improvements

We conclude this chapter by considering some other potential improvements to our system.

11.5.1 Unchecked Updates

While our dynamic updating system has proven to be very flexible, there are some changes that it cannot reflect dynamically. Most fundamentally, because we only allow updating with verifiable native code, it is impossible to update the trusted computing base. Although we expect changes at that level to be rare, they also can be critical, as we saw with the need to update the verifier so that it verifies all parts of a patch (see §9.3). Although there are some technical difficulties (not to mention robustness issues), we could relax this limit by providing a lower-level interface to the dynamic loader, circumventing the PCC-only one we use in general.

11.5.2 Namespace Management and Security

While our system is designed to support fine-grained management of symbols and types for security purposes, we have yet to explore this possibility. We speculate on how we could implement secure symbol management here.

Part of the benefit of having symbol management and linking outside of the TCB is that it allows greater flexibility: we can use different mechanisms, as the situation calls for, for implementing linking policy. For example, we could assign each user a cryptographic key for authentication purposes before any code is loaded. When a symbol is requested by the `lookup` function in `dyninit`, a policy check can be made to determine whether the user has sufficient privilege to acquire that symbol. If not, the `SymbolNotFound` exception can be raised (thus not leaking information to the user about the existence of the symbol, as would be the case if something like `InsufficientPrivilege` was raised instead). Better yet, different values of a requested symbol may be provided, based on privilege. For example, unprivileged users requesting the `open` symbol may instead get a version of `open` that only works for files in the `/tmp` directory. Furthermore, because these operations are implemented as TAL code, rather than as part of the TCB, they can be safely replaced in an updateable system if the need arises. For example, we might want to switch to using a different cryptosystem if flaws are found in the current one.

Extending `load` to allow for type heap masks, as described in §5.5 and formalized in §5.3, can make type management possible as well. The idea is that each module has a different type heap mask for each level of privilege, therefore making more or fewer named types available. For this to work, we need to be able access and construct relevant masks as the program's type heap changes with newly loaded code. Abstractly, we require the following operations: we must *obtain* one or more relevant type heap masks from each module that we load, and *combine* these masks to create various notions of the program type interface to be passed to `load`. There is no requirement that type heap mask manipulation be trusted, so different alternatives can be crafted for different situations.

One way determine what mechanisms would be appropriate is to consider to what extent type heap masks need be manipulated dynamically. The minimum would be to have the compiler generate various type heap masks relevant to a particular module and add them to its static data; these could be extracted following the call to load. At runtime, we use a provided function, `combine_masks`, that takes two type heap masks as arguments and returns the result of merging those masks (as in using the operator $\oplus$ defined in Figure 5.10 on page 58). Which masks to combine, and what masks to generate, would be subject to security policy. Because the masks to generate are determined statically, this scheme is only suitable when policy need not change at runtime.

We might also consider having load generate the module's program type interface and return it (along with its exported values). To manipulate that mask, we could use a function, `remove_typedef`, which takes as arguments a mask and type name and returns the mask minus the mapping for the given type name. We could also provide a function `make_abstract`, having the same signature, that instead causes the given mapping to be treated as abstract instead. The benefit of this approach is its dynamicity. That is, each time load is called, the type interface returned is the most permissive for the program. Then, a policy determined at runtime can *thin* that environment depending on the privilege of the user loading code. This general approach is taken in a number of systems, *e.g.* [HK99, AHI$^+$00].

Even more dynamic approaches are possible. We could provide functions that allow the direct construction of masks by adding bindings. For example, we could provide the function `add_binding`, that takes a mask, a type name, and a type representation, and returns the mask with the binding from the given type name to the given type. However, it is not clear that this level of flexibility is useful, simply because a mask must always be a 'sub-heap' of the program type interface, maintained within the TCB. Adding constructed bindings to a mask is therefore probably not any more useful than combining masks returned from load, or from removing bindings from those masks. More experience on this is needed before we can really determine what abstractions will be sufficient.

11.5.3 Updating Abstract Types

As described in §4.2.5, Popcorn supports module-level abstract types. In particular, structures and unions can be declared `abstract`, meaning that only the code in the local file may see the type's implementation; this is enforced by the TAL verifier. As a result, no dynamically linked file will be able to see the implementation of an abstract type. In general, this behavior is desirable, but it also prevents us from loading new code to "update" (by replacement) the implementation of the abstract type.

It is possible that the proposal we outlined above for namespace security can apply to abstract types as well. In particular, the verifier can maintain the least restrictive type environment (that allows breaking the abstraction), while the linker code will pass in a more restrictive environment for all those cases except ones in which the type's implementation is to be updated. Some recent work also provides insight into this question of updating abstract types [Sew01].

Chapter 12

Conclusions

In this dissertation we show that updating the code and data of a running program—that is, *dynamic software updating*—can be achieved in a *general-purpose* manner that is *flexible*, *efficient*, *robust* and is *easy to use*. To demonstrate this thesis, we have described the design and implementation of a dynamic updating system and argued that it has the desired characteristics:

- **Flexibility** Our system permits changes to programs at the granularity of individual definitions, be they functions, types, or data. Furthermore, we allow these definitions to change in arbitrary ways; most notably, functions and data may change type, and named types may change definition. The system permits updates to occur at any time, even while the code being updated is active, providing the programmer with greater control. Our approach uses an imperative, C-like language, and should thus be widely usable.

- **Robustness** In our system, dynamic patches consist of Typed Assembly Language (TAL) [MWCG99]. As a result, a patch cannot crash the system or perform many incorrect actions since it can be proven to respect important safety properties. Our implementation builds on top of basic dynamic linking, keeping the implementation simple. Furthermore, we have developed a means for loading dynamic patches that does not unduly expand the trusted computing base, improving our confidence in system safety. Our use of an automated patch generator ensures that patches are completely specified. The programmer is free to ensure that updates are well-timed, as there are no system-imposed timing restrictions. Finally, if a problem occurs during linking or state transformation, the update will be rolled back to the previous state.

- **Efficiency** Our system imposes only the modest runtime overhead associated with dynamic linking. However, because we use TAL, programs and patches consist of native code, giving obvious performance benefits as compared to interpreted systems like Java.

- **Ease of use** Construction of patches is largely automated and clearly separated from the typical development process. When a new software version is completed, a tool compares the old and new versions of the source files to develop patches that reflect the differences. Although total automation is undecidable, our tool can nonetheless generate useful patch code for a majority of cases, leaving placeholders for the programmer in the other (infrequent) cases. Because patches encapsulate all issues relating to dynamic change, they can be cleanly separated from the normal development code, simplifying software maintenance.

Throughout the dissertation, we supported these assertions by describing the system design, implementation, and component performance characteristics. In addition, we drew from our experience in building a non-trivial, dynamically updateable application, the FlashEd webserver. We explained how a publicly deployed version of FlashEd was developed by dynamically updating it in significant ways over a period of months, and in so doing, showed that the system is flexible and robust, and supported the assertion that it is easy to use. In addition, we measured FlashEd's performance and showed that the updating system imposes only a negligible overhead (less than 2%) on FlashEd's performance.

Our work represents a significant advance in the state of the art. In particular, no prior general-purpose updating system adequately meets all four of the evaluation criteria. These criteria are well-chosen: if a system is not flexible enough, it may not be able to express a desired change dynamically, requiring a service interruption; if the system is not robust enough, incorrect patches will crash the system or cause it to misbehave; if the system does not simplify building and maintaining updateable software, it will have limited scope and magnify the chance of user error; and if the system imposes a high overhead, it will be useless for a large potential clientele, that which runs high-performance server systems. In addition, our work stands as a significant advance in the *validation* of dynamic updating systems. As far as we are aware, no prior general-purpose approach has performed as detailed a performance assessment, and no prior study has published its experience with as significant an application.

12.1 Contributions

As well as the overall conclusion described above, the research presented in this dissertation makes a number of specific contributions that revolve around the design and evaluation of our dynamic software updating system:

1. We have developed the first complete framework for safe dynamic linking of verifiable native code. The system that we have built is the first to enable dynamic linking of native code in a way that is both safe and flexible enough to support a variety of dynamic linking strategies.

2. We have defined and implemented a novel notion of dynamic patch that cleanly separates the concerns of program and patch development. This simplifies the development process and makes program code more maintainable, since it does not become 'polluted' with code to support dynamic patching.

3. We have employed a novel approach to dealing with changes to type definitions by renaming them. This approach works well in practice, and avoids the implementation complexity of true type replacement, as employed in systems like Dynamic ML [GKW97] and Dynamic Java classes [MPG$^+$00].

4. We have developed a tool that mostly automatically generates patches, given two versions of a program. This tool greatly simplifies the process of developing dynamic updates and ensures that updates are completely specified.

5. We show that verifiable native code (VNC) technology, and in particular Typed Assembly Language, is flexible enough to support dynamically updateable programs. The use of VNC increases the robustness of both the running program and its dynamic patches.

6. We have built a sizeable updateable application: an updateable webserver. As far as we know, ours is the largest application described in the general-purpose dynamic updating literature to be updated in non-trivial ways over a lengthy course of time.

7. We show by direct measurement that dynamic updateability can impose a low overhead. Ours is one of the few systems to have documented performance data.

Appendix A

Proofs for Formal Properties of TAL/Load

This chapter presents the complete proof of soundness (*i.e.* type-safety) for the load-calculus, presented in §5.3. Our presentation of the proof is bottom-up, starting with properties of the system needed for the final proof. We start with properties of type environments, then properties of heaps, then properties of type derivations, and finally the proof of type-safety.

A.1 The load-calculus

Since the load-calculus was presented incrementally in §5.3, we summarize the syntax, static semantics, and operational semantics here. The definitions of heap linking, type interface linking, and the operations on type environments are not repeated here (they are the same as in the main text; see Definition 5.3.1 on page 48, Definition 5.3.2 on page 58, and Figure 5.10 on page 58, respectively). Note that this formulation includes the type heap mask, defined in S5.5.1.

A.1.1 Syntax

types	τ	$::=$	$\texttt{int} \mid n \mid \tau \to \tau \mid \tau\ \texttt{ref} \mid \alpha \mid \forall\alpha.\tau$
type environments	X	$::=$	$\{n_1 = \chi_1, \ldots, n_n = \chi_n\}$
type env values	χ	$::=$	$\top \mid \tau$
type interfaces	Θ	$::=$	(X_I, X_H)
expressions	e	$::=$	$i \mid L \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau]$
		$\mid$	$\texttt{reveal}\ e \mid \texttt{hide}_n\ e$
		$\mid$	$\texttt{ref}\ e \mid \texttt{assign}\ e_1 e_2 \mid !e$
		$\mid$	$\mathsf{load}[\tau]\ e_0\ e_1\ e_2\ e_3$
values	v	$::=$	$i \mid L \mid \lambda x{:}\tau.e \mid \texttt{hide}_n\ v$
heaps	H	$::=$	$\{L_1 = v_1, \ldots, L_n = v_n\}$
programs	P	$::=$	(Θ, H, e)
heap types	Φ	$::=$	$\{L_1 : \tau_1, \ldots, L_n : \tau_n\}$
type contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha$
contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$

$$\boxed{\begin{array}{l} i \in \mathcal{Z} \\ n \in \mathsf{TypeNames} \\ L \in \mathsf{Labels} \\ x \in \mathsf{Vars} \\ \alpha \in \mathsf{TypeVars} \end{array}}$$

A.1.2 Operational Semantics

The operational semantics are based on deterministic rewriting rules, expressing a call-by-value evaluation order. The two rules for load, below, make use of the type environment operators defined in Figure 5.10 (page 58), as well as the type interface linking operator `link`, Definition 5.3.2 page 58, and the heap linking operator `merge`, Definition 5.3.1, page 48. Please refer to §5.3 for more details.

$$\boxed{(\Theta, H, e) \mapsto (\Theta', H', e')}$$

$$\frac{\begin{array}{c} X_H \vdash \hat{i} : \tau \\ H\ \texttt{merge}\ H_i \Rightarrow H' \\ X_H \leq X^h \qquad X_I^i \mid (X_H - X^h) \\ (X_I, X^h)\ \texttt{link}\ (X_I^i, X_H^i) \Rightarrow (X_I', X_H'') \end{array}}{\begin{array}{l} ((X_I, X_H), H, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \mapsto \\ ((X_I', X_H'), H', e_2\ e_i) \end{array}} \left(\begin{array}{c} \hat{h} = X^h \\ \hat{i} = ((X_I^i, X_H^i), H_i, e_i) \\ X_H' = X_H'' \oplus X_H \end{array}\right) \quad \textit{(load-success)}$$

$$(\Theta, H, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \quad \mapsto \quad (\Theta, H, e_3) \qquad \textit{(load-failure)}$$
otherwise

$$(\Theta, H, (\lambda x{:}\tau.e)\ v) \quad \mapsto \quad (\Theta, H, e[v/x]) \qquad \textit{(beta)}$$

$$(\Theta, H, \texttt{reveal}(\texttt{hide}_n\ v)) \quad \mapsto \quad (\Theta, H, v) \qquad \textit{(reveal)}$$

$$(\Theta, H, \texttt{ref}\ v) \quad \mapsto \quad (\Theta, H \uplus \{L = v\}, L) \qquad \textit{(ref)}$$
where $L \notin \texttt{dom}(H)$

$(\Theta, H, !L) \quad \mapsto \quad (\Theta, H, v)$ *(deref)*
where $H(L) = v$

$(\Theta, H, \texttt{assign}\ L\ v) \quad \mapsto \quad (\Theta, H[L = v], v)$ *(assign)*

$(\Theta, H, (\Lambda\alpha.e)[\tau]) \quad \mapsto \quad (\Theta, H, e[\tau/\alpha])$ *(tapp)*

(congruence)

$$\frac{(\Theta, H, e) \mapsto (\Theta', H', e')}{\left\{\begin{array}{c} (\Theta, H, e\ e_2) \mapsto (\Theta', H', e'\ e_2) \\ (\Theta, H, v_1\ e) \mapsto (\Theta', H', v_1\ e') \\ (\Theta, H, \texttt{hide}_n\ e) \mapsto (\Theta', H', \texttt{hide}_n\ e') \\ (\Theta, H, \texttt{reveal}\ e) \mapsto (\Theta', H', \texttt{reveal}\ e') \\ (\Theta, H, \mathsf{load}[\tau]\ e\ e_1\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ e'\ e_1\ e_2\ e_3) \\ (\Theta, H, \mathsf{load}[\tau]\ v\ e\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ v\ e'\ e_2\ e_3) \\ (\Theta, H, \texttt{ref}\ e) \mapsto (\Theta', H', \texttt{ref}\ e') \\ (\Theta, H, !e) \mapsto (\Theta', H', !e') \\ (\Theta, H, \texttt{assign}\ e\ e_2) \mapsto (\Theta', H', \texttt{assign}\ e'\ e_2) \\ (\Theta, H, \texttt{assign}\ v\ e) \mapsto (\Theta', H', \texttt{assign}\ v\ e') \\ (\Theta, H, e[\tau]) \mapsto (\Theta', H', e'[\tau]) \end{array}\right\}}$$

A.1.3 Static Semantics

The judgments for the static semantics are presented bottom-up: type well-formedness ($\Delta \vdash \tau$), type environment well-formedness ($\vdash X$), heap type well-formedness ($X \vdash \Phi$), expression well-formedness ($X; \Phi; \Delta; \Gamma \vdash e : \tau$), heap well-formedness ($X \vdash H : \Phi$), and program well-formedness ($X_P \vdash (\Theta, H, e) : \tau$).

$\boxed{\Delta \vdash \tau}$

$$\Delta \vdash \texttt{int} \qquad \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \frac{n \in \texttt{dom}(X)}{X; \Delta \vdash n}$$

$$\frac{\Delta \vdash \tau' \qquad \Delta \vdash \tau}{\Delta \vdash \tau' \to \tau} \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \texttt{ref}\ \tau} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall\alpha.\tau}\ (\alpha \notin \Delta)$$

$\boxed{\vdash X}$

$$\frac{X; \cdot \vdash \tau \quad (\text{for each } \tau \in \texttt{rng}(X))}{\vdash X}$$

$\boxed{X \vdash \Phi}$

$$\frac{X; \cdot \vdash \tau \quad (\text{for each } \tau \in \texttt{rng}(\Phi))}{X \vdash \Phi}$$

$\boxed{X;\Phi;\Delta;\Gamma \vdash e : \tau}$

$$\frac{\begin{array}{c} X;\Phi;\Delta;\Gamma \vdash e_0 : \mathtt{int} \\ X;\Phi;\Delta;\Gamma \vdash e_1 : \mathtt{int} \\ X;\Phi;\Delta;\Gamma \vdash e_2 : \tau' \to \tau \\ X;\Phi;\Delta;\Gamma \vdash e_3 : \tau \end{array}}{X;\Phi;\Delta;\Gamma \vdash \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3 : \tau}$$

$$X;\Phi;\Delta;\Gamma \vdash i : \mathtt{int} \qquad X;\Phi;\Delta;\Gamma \vdash x : \Gamma(x) \qquad X;\Phi;\Delta;\Gamma \vdash L : \Phi(L)\ \mathtt{ref}$$

$$\frac{X;\Phi;\Delta;\Gamma \vdash e : n}{X;\Phi;\Delta;\Gamma \vdash \mathtt{reveal}\ e : \tau}\ (X(n) = \tau) \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \mathtt{hide}_n\ e : n}\ (X(n) = \tau)$$

$$\frac{X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e : \tau \qquad X;\Delta \vdash \tau'}{X;\Phi;\Delta;\Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau} \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e_1 : \tau' \to \tau \qquad X;\Phi;\Delta;\Gamma \vdash e_2 : \tau'}{X;\Phi;\Delta;\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \forall\alpha.\tau \qquad X;\Delta \vdash \tau'}{X;\Phi;\Delta;\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

$$\frac{X;\Phi;\Delta;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \mathtt{ref}\ e : \tau\ \mathtt{ref}} \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \tau\ \mathtt{ref}}{X;\Phi;\Delta;\Gamma \vdash\ !e : \tau} \qquad \frac{\begin{array}{c} X;\Phi;\Delta;\Gamma \vdash e_1 : \tau\ \mathtt{ref} \\ X;\Phi;\Delta;\Gamma \vdash e_2 : \tau \end{array}}{X;\Phi;\Delta;\Gamma \vdash \mathtt{assign}\ e_1\ e_2 : \tau}$$

$\boxed{X \vdash H : \Phi}$

$$\frac{X;\Phi;\cdot;\cdot \vdash H(L) : \Phi(L) \quad \text{(for each } L \in \mathtt{dom}(H))}{X \vdash H : \Phi}$$

$\boxed{X_P \vdash (\Theta, H, e) : \tau}$

$$\frac{\begin{array}{c} \vdash X_I \uplus X_H \qquad X_I \uplus X_H \vdash \Phi \\ X_I \uplus X_H \vdash H : \Phi \qquad X_I \uplus X_H;\Phi;\cdot;\cdot \vdash e : \tau \end{array}}{X_P \vdash ((X_I, X_H), H, e) : \tau}\ (X_H \mid X_P)$$

A.2 Properties of Type Environments

All of the lemmas (and their corollaries) developed in this section are for the purpose of proving the load case of the subject reduction, in Section A.5.

Lemma A.2.1 (Type Environment Equalities) *Suppose A, B, C, D are type environments, then*

1. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
2. $(A \oplus B) = (B \oplus A)$
3. *if* $A \mid B$ *then* $A \oplus B = A \uplus B$
4. *if* $A \precsim C$, $B \precsim D$, $A \diamond B$ *and* $C \diamond D$, $A \mid D$, $B \mid C$, *then* $(A \oplus B) \precsim (C \oplus D)$
5. *if* $B \precsim A$ *then* $(A - B) \uplus B = A \oplus B$.
6. *if* $A \leq B$ *then* $A \precsim B$
7. *if* $A \leq B$ *then* $A \oplus B = A$
8. *if* $A \precsim B$ *then* $A \diamond B$
9. *if* $B \leq A$ *and* $C \mid (B - A)$ *then* $C - B = C - A$

Proof of 4 This fails if for some n, $(A \oplus B)(n) = \top$ and $(C \oplus D)(n) = \tau$. Assume $A(n) = \top$. Then $C(n) = \top$, if $n \in \mathtt{dom}(C)$ as $A \precsim C$. Furthermore $n \notin \mathtt{dom}(D)$ as $A \mid D$. So $(C \oplus D)(n) = \top$ if anything. Analogous reasoning if $B(n) = \top$.

Proof of 5 If $n \in A$ and $n \notin B$ then trivially $(A \oplus B)(n) = ((A - B) \uplus B)(n)$, likewise if $n \in B$ and $n \notin A$, and if $A(n) = B(n)$. Suppose $A(n) = \top$ and $B(n) = \tau$, then $(A \oplus B)(n) = \tau$ and $(A - B)(n)$ is undefined so $(A - B) \uplus B(n) = \tau$. The reverse case, where $A(n) = \tau$ and $B(n) = \top$ cannot happen by assumption.

Proof of 9 As C does not include names in B that are not in A, then removing the names from C that are in B is the same as removing only the ones from A.

Lemma A.2.2 (Type Environment Merge) *If* $\vdash X_A$ *and* $\vdash X_B$ *then* $\vdash X_A \oplus X_B$.

Lemma A.2.3 (Type Environment Weakening) *Suppose* $X \oplus X'$ *is well-defined.*

1. *If* $X; \Delta \vdash \tau$ *then* $(X \oplus X'); \Delta \vdash \tau$
2. *If* $X \vdash \Phi$ *then* $X \oplus X' \vdash \Phi$
3. *If* $X; \Phi; \Delta; \Gamma \vdash e : \tau$ *then* $X \oplus X'; \Phi; \Delta; \Gamma \vdash e : \tau$.
4. *If* $X \vdash H : \Phi$ *then* $X \oplus X' \vdash H : \Phi$

Proof

1. Proof is by induction on $X;\Delta \vdash \tau$. If $\tau = \alpha$ then $(X \oplus X');\Delta \vdash \alpha$. If $\tau = \texttt{int}$ then $(X \oplus X');\Delta \vdash \texttt{int}$. If $\tau = n$ then n is still in the domain of $X \oplus X'$ (though its range might change), so $(X \oplus X');\Delta \vdash n$. The remaining cases follow by induction.
2. We are given that for each $\tau \in \texttt{rng}(\Phi)$, $X \vdash \tau$. It follows by 1 that $X \oplus X' \vdash \tau$.
3. Proof is by induction on $X;\Phi;\Delta;\Gamma \vdash e : \tau$. This follows trivially or by induction for every rule except: If e is an abstraction or a type application, then Part 1 is also needed to verify the type added to the context. If e is $\texttt{reveal}\ e'$ or $\texttt{hide}_n\ e'$ then we note that because $X(n) = \tau$, then by the definition of $X \oplus X'$, $(X \oplus X')(n) = \tau$ as well, and the rest follows by induction.
4. We are given that for each $L \in \texttt{dom}(H)$ that $X;\Phi;\cdot;\cdot \vdash H(L) : \Phi(L)$. It follows by 3 that $X \oplus X';\Phi;\cdot \vdash H(L) : \Phi(L)$.

Corollary A.2.4 *Suppose $X \uplus X'$ is well-defined.*

1. *If $X;\Delta \vdash \tau$ then $X \uplus X';\Delta \vdash \tau$*
2. *If $X \vdash \Phi$ then $X \uplus X' \vdash \Phi$*
3. *If $X;\Phi;\Delta;\Gamma \vdash e : \tau$ then $X \uplus X';\Phi;\Delta;\Gamma \vdash e : \tau$.*
4. *If $X \vdash H : \Phi$ then $X \uplus X' \vdash H : \Phi$*

Lemma A.2.5 (Type Environment Redundancy Elimination) *If $X \leq X'$ and $X \oplus X'' \vdash \tau$ then $X \oplus (X'' - X') \vdash \tau$*

Proof

(Sketch) Any name in X'' that is also in X' will also be in X as that type environment contains all of the name of X'. Therefore subtracting out the redundant names will not interfere with type well-formedness.

A.3 Properties of Heaps

The lemmas (and their corollaries) developed in this section are used in the proof of subject reduction for the `load` and `ref` cases.

Lemma A.3.1 (Heap Weakening) *If $X \vdash \Phi$, $X \vdash H : \Phi$, $X;\Phi;\cdot;\cdot \vdash e : \tau$ and given some $L' \notin \texttt{dom}(\Phi)$ and some type τ' such that $X;\cdot \vdash \tau'$, then*

1. *$X \vdash (\Phi \uplus \{L' : \tau'\})$*
2. *$X \vdash H : (\Phi \uplus \{L' : \tau'\})$*
3. *$X;(\Phi \uplus \{L' : \tau'\});\cdot;\cdot \vdash e : \tau$*

Proof

1. We must show that for all $\tau \in \texttt{rng}(\Phi \uplus \{L' : \tau'\})$, $X \vdash \tau$. If $\tau \in \texttt{rng}(\Phi)$, then this is true by inversion of $X \vdash \Phi$. If $\tau \notin \texttt{rng}(\Phi)$ then $\tau = \tau'$, and we are given that $X;\cdot \vdash \tau'$.

2. This follows trivially by assumption, since we have not changed the domain of H.
3. Proof by induction on $X; \Phi; \cdot; \cdot \vdash e : \tau$. Follows trivially or by induction. In the abstraction and type application cases, we need to use 1 for the introduction of the new type, and for $e = L$, we have $\Phi(L) = \tau = (\Phi \uplus \{L' : \tau'\})(L)$.

Corollary A.3.2 *If $X \vdash \Phi$ and $X \vdash H : \Phi$, and given some Φ' such that $X \vdash \Phi'$ and $\Phi \mid \Phi'$, then*

1. $X \vdash (\Phi \uplus \Phi')$
2. $X \vdash H : (\Phi \uplus \Phi')$
3. $X; (\Phi \uplus \Phi'); \cdot; \cdot \vdash e : \tau$

Lemma A.3.3 (Heap Redundancy Elimination) *If $X \vdash H : \Phi$, and there exists some $L' \in \mathtt{dom}(\Phi)$ s.t. $L' \notin \mathtt{dom}(H)$, then $X \vdash H : \{L : \tau | L : \tau \in \Phi, L \neq L'\}$.*

Proof

We must show that for all $L \in \mathtt{dom}(H), H(L) : (\{L : \tau | L : \tau \in \Phi, L \neq L'\})(L)$. But this is obvious, since we have only removed a label from Φ that was not in H.

A.4 Properties of Type Derivations

Lemma A.4.1 (Type in Type Substitution) *If $X; \Delta, \alpha \vdash \tau$ and $X; \Delta \vdash \tau'$ then we have $X; \Delta \vdash \tau[\tau'/\alpha]$*

Proof

Proof by induction on $X; \Delta, \alpha \vdash \tau$

Case 1: $X; \Delta, \alpha \vdash n$ or $X; \Delta, \alpha \vdash \mathtt{int}$ follows trivially.

Case 2: $X; \Delta, \alpha \vdash \alpha$ since by assumption $X; \Delta \vdash \alpha[\tau'/\alpha]$.

Remaining cases follow by simple induction.

The following lemma is used in the `ref` case of subject reduction.

Lemma A.4.2 (Regularity)

If $X; \Phi; \Delta; \Gamma \vdash v : \tau$, $\vdash X$, $X; \Delta \vdash \Gamma$, and $X \vdash \Phi$, then $X; \Delta \vdash \tau$.

Proof

The proof proceeds by induction on the derivation $X; \Phi; \Delta; \Gamma \vdash e : \tau$.

Case 1: $X; \Phi; \Delta; \Gamma \vdash i : \mathtt{int}$. Follows directly that $X; \Delta \vdash \mathtt{int}$.

Case 2: $X; \Phi; \Delta; \Gamma \vdash L : \Phi(L)$. By assumption $X \vdash \Phi$, and by inversion $X; \cdot \vdash \Phi(L)$.

Case 3: $X; \Phi; \Delta; \Gamma \vdash y : \Gamma(y)$. By assumption $X; \Delta \vdash \Gamma$, so $X; \Delta \vdash \Gamma(y)$.

Case 4: $X; \Phi; \Delta; \Gamma \vdash \lambda x{:}\tau'.e : \tau' \rightarrow \tau$. By inversion $X; \Delta \vdash \tau'$. As a result, $X; \Delta \vdash \Gamma, x{:}\tau'$, as τ' is well-formed. Therefore, by induction $X; \Delta \vdash \tau$. Thus, $X; \Delta \vdash \tau' \rightarrow \tau$.

Case 5: $X;\Phi;\Delta;\Gamma \vdash e_1\ e_2 : \tau$. By induction $X;\Delta \vdash \tau' \to \tau$, and by inversion $X;\Delta \vdash \tau$.

Case 6: $X;\Phi;\Delta;\Gamma \vdash \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3 : \tau$. By induction.

Case 7: $X;\Phi;\Delta;\Gamma \vdash \texttt{hide}_n\ e : n$. By the rule side-condition $X(n) = \tau$, thus $X;\Delta \vdash n$.

Case 8: $X;\Phi;\Delta;\Gamma \vdash \texttt{reveal}\ e : \tau$. By the rule side-condition $X(n) = \tau$, and by assumption $\vdash X$, so $X;\cdot \vdash \tau$. By weakening, $X;\Delta \vdash \tau$.

Case 9: $X;\Phi;\Delta;\Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}$. By induction $X;\Delta \vdash \tau$, so $X;\Delta \vdash \tau\ \texttt{ref}$ follows directly.

Case 10: $X;\Phi;\Delta;\Gamma \vdash !e : \tau$. By induction $X;\Delta \vdash \tau\ \texttt{ref}$, and by induction again $X;\Delta \vdash \tau$.

Case 11: $X;\Phi;\Delta;\Gamma \vdash \texttt{assign}\, e_1 e_2 : \tau$. By induction.

Case 12: $X;\Phi;\Delta;\Gamma \vdash e : \forall\alpha.\tau$. By induction.

Case 13: $X;\Phi;\Delta;\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]$, so by inversion $X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau$ and $X;\Delta \vdash \tau'$. By induction $X;\Delta,\alpha \vdash \tau$ and by type in type substitution, $X;\Delta \vdash \tau[\tau'/\alpha]$.

The following two lemmas are used in the proof of substitution, also below.

Lemma A.4.3 (Weakening) *If $X;\Phi;\Delta;\Gamma \vdash e : \tau$ and $x \notin \mathtt{dom}(\Gamma)$ and $\alpha \notin \mathtt{dom}(\Delta)$, then $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e : \tau$, and $X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau$. Moreover, the latter derivations have the same depth as the former.*

Lemma A.4.4 (Permutation) *If $X;\Phi;\Delta;\Gamma \vdash e : \tau$ with Γ' is a permutation of Γ and Δ' is a permutation of Δ, then $X;\Phi;\Delta;\Gamma' \vdash e : \tau$, and $X;\Phi;\Delta';\Gamma \vdash e : \tau$. Moreover, the latter derivations have the same depth as the former.*

Lemma A.4.5 (Substitution) *If $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e : \tau$ and $X;\Phi;\Delta;\Gamma \vdash e' : \tau'$ then $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : \tau$.*

Proof

Proof is by induction on $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e : \tau$.

Case 1: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash i : \texttt{int}$

Therefore $e[e'/x] = i$, and $X;\Phi;\Delta;\Gamma \vdash i : \texttt{int}$.

Case 2: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash L : \Phi(L)$.

Therefore $e[e'/x] = L$, and $X;\Phi;\Delta;\Gamma \vdash L : \Phi(L)$.

Case 3: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash y : (\Gamma, x{:}\tau')(y)$.

If $y = x$ then $y[e'/x] = e'$. By assumption, $X;\Phi;\Delta;\Gamma \vdash e' : \tau'$, and the result follows from $\tau = \tau'$. Otherwise, $y[e'/x] = y$ and $X;\Phi;\Delta;\Gamma \vdash y : \Gamma(y)$.

Case 4: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash \lambda y{:}\tau''.e : \tau'' \to \tau$

Follows by induction (with Weakening and Permutation): $X;\Phi;\Delta;\Gamma, y{:}\tau'' \vdash e[e'/x] : \tau$. Therefore, $X;\Phi;\Delta;\Gamma \vdash (\lambda y{:}\tau''.e)[e'/x] : \tau'' \to \tau$.

Case 5: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash e_1\ e_2 : \tau$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e_1[e'/x] : \tau'' \to \tau$ and $X; \Phi; \Delta; \Gamma \vdash e_2[e'/x] : \tau''$. Therefore, $X; \Phi; \Delta; \Gamma \vdash (e_1 e_2)[e'/x] : \tau$.

Case 6: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \mathsf{load}[\tau'']\ e_0\ e_1\ e_2\ e_3 : \tau$.

Follows by induction; we have:

- $X; \Phi; \Delta; \Gamma \vdash e_0[e'/x] : \texttt{int}$
- $X; \Phi; \Delta; \Gamma \vdash e_1[e'/x] : \texttt{int}$
- $X; \Phi; \Delta; \Gamma \vdash e_2[e'/x] : \tau'' \to \tau$
- $X; \Phi; \Delta; \Gamma \vdash e_3[e'/x] : \tau$

so therefore $X; \Phi; \Delta; \Gamma \vdash (\mathsf{load}[\tau'']\ e_1\ e_2\ e_3)[e'/x] : \tau$.

Case 7: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \texttt{hide}_n\ e : n$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[e'/x] : \tau''$, so $X; \Phi; \Delta; \Gamma \vdash (\texttt{hide}_n\ e)[e'/x] : n$

Case 8: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \texttt{reveal}\ e : \tau$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[e'/x] : l$, so $X; \Phi; \Delta; \Gamma \vdash (\texttt{reveal}\ e)[e'/x] : \tau$.

Case 9: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \texttt{ref}\ e : \tau\ \texttt{ref}$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[e'/x] : \tau$, so $X; \Phi; \Delta; \Gamma \vdash (\texttt{ref}\ e)[e'/x] : \tau\ \texttt{ref}$.

Case 10: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash\ !e : \tau$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[e'/x] : \tau\ \texttt{ref}$, so $X; \Phi; \Delta; \Gamma \vdash (!e)[e'/x] : \tau$.

Case 11: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \texttt{assign}\, e_1 e_2 : \tau$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e_1[e'/x] : \tau\ \texttt{ref}$ and $X; \Phi; \Delta; \Gamma \vdash e_2[e'/x] : \tau$. Therefore, $X; \Phi; \Delta; \Gamma \vdash (\texttt{assign}\, e_1 e_2)[e'/x] : \tau$.

Case 12: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash \Lambda\alpha.e : \forall\alpha.\tau$. Follows by induction (with Weakening): $X; \Phi; \Delta, \alpha; \Gamma \vdash e[e'/x] : \tau$, so $X; \Phi; \Delta, \alpha; \Gamma \vdash \Lambda\alpha.e[e'/x] : \forall\alpha.\tau$.

Case 13: $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash e[\tau'] : \tau[\tau'/\alpha]$. Follows by induction: $X; \Phi; \Delta, \alpha; \Gamma \vdash e[e'/x] : \forall\alpha.\tau$, so $X; \Phi; \Delta; \Gamma \vdash (e[e'/x])[\tau'] : \tau[\tau'/\alpha]$.

Lemma A.4.6 (Type Substitution) *If* $\vdash X$ *and* $X; \Phi; \Delta, \alpha; \Gamma \vdash e : \tau$ *and* $X; \Delta \vdash \tau'$ *then* $X; \Phi; \Delta; \Gamma \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Proof

Proof is by induction on $X; \Phi; \Delta, \alpha; \Gamma \vdash e : \tau$. Most cases are trivial or by induction. Selected cases:

Case 1: $X; \Phi; \Delta, \alpha; \Gamma \vdash \mathsf{load}[\tau'']\ e_1\ e_2\ e_3 : \tau$.

By induction: $X; \Phi; \Delta; \Gamma \vdash e_0[\tau'/\alpha] : \texttt{int}$, $X; \Phi; \Delta; \Gamma \vdash e_1[\tau'/\alpha] : \texttt{int}$, $X; \Phi; \Delta; \Gamma \vdash e_2[\tau'/\alpha] : (\tau'' \to \tau)[\tau'/\alpha]$, and $X; \Phi; \Delta; \Gamma \vdash e_3[\tau'/\alpha] : \tau[\tau'/\alpha]$. By type in type substitution $X; \Delta \vdash \tau''[\tau'/\alpha]$ so therefore $X; \Phi; \Delta; \Gamma \vdash (\mathsf{load}[\tau'']\ e_0\ e_1\ e_2\ e_3)[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Case 2: $X; \Phi; \Delta, \alpha; \Gamma \vdash \texttt{hide}_n\ e : n$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[\tau'/\alpha] : \tau''[\tau'/\alpha]$. As $X(n) = \tau''$ and $\vdash X$ then $X; \cdot \vdash \tau''$. Therefore $X; \Phi; \Delta; \Gamma \vdash (\mathtt{hide}_n\ e)[\tau'/\alpha] : n[\tau'/\alpha]$ since τ'' must be closed.

Case 3: $X; \Phi; \Delta, \alpha; \Gamma \vdash \mathtt{reveal}\ e : \tau[\tau'/\alpha]$.

Follows by induction: $X; \Phi; \Delta; \Gamma \vdash e[\tau'/\alpha] : n[\tau'/\alpha]$, so $X; \Phi; \Delta; \Gamma \vdash (\mathtt{reveal}\ e)[\tau'/\alpha] : \tau[\tau'/\alpha]$.

The following lemma is used in the proof of progress, to develop type soundness.

Lemma A.4.7 (Canonical Forms) *If* $X; \Phi; \cdot \vdash v : \tau$ *and*

- $\tau = \mathtt{int}$ *then* $v = i$*.*
- $\tau = \tau_1 \to \tau_2$ *then* $v = \lambda x{:}\tau.e$*.*
- $\tau = n$ *then* $v = \mathtt{hide}_n(v')$ *for some* v'*.*
- $\tau = \tau\ \mathtt{ref}$ *then* $v = L$*.*
- $\tau = \forall\alpha.\tau$ *then* $v = \Lambda\alpha.e$*.*

Proof

Proof is by examination of the last step of the typing derivation $X; \Phi; \cdot; \cdot \vdash v : \tau$. Most rules either require the expression to be a non-value or require a non-empty context. The remaining rules produce each of the types at the correct values.

A.5 Type Soundness

Type soundness is proven via subject reduction and progress, as is standard.

Lemma A.5.1 (Subject Reduction) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto (\Theta', H', e')$ *then* $\vdash (\Theta', H', e') : \tau$

Proof

$\vdash (\Theta', H', e') : \tau$ is proven by showing that, for some Φ'

- (Type environment well-formedness) $\vdash X'_I \uplus X'_H$
- (Heap typing well-formedness) $X'_I \uplus X'_H \vdash \Phi'$
- (Heap well-formedness) $X'_I \uplus X'_H \vdash H' : \Phi'$
- (Expression well-formedness) $X'_I \uplus X'_H; \Phi'; \cdot; \cdot \vdash e' : \tau$

For brevity, we refer to these points in the proof as *TEWF*, *HTWF*, *HWF*, and *EWF*, respectively, and except when otherwise noted, we assume that $\Phi' = \Phi$ and that *TEWF*, *HTWF*, *HWF* hold by assumption. The proof is by induction on the typing derivation $\vdash (\Theta, H, e)$ (for some Φ), and on $(\Theta, H, e) \mapsto (\Theta', H', e')$.

Case 1: (beta) $(\Theta, H, (\lambda x{:}\tau.e'')v) \mapsto (\Theta, H, e''[v/x])$.

As e is an application, by inversion $X_I \uplus X_H; \Phi; x{:}\tau' \vdash e'' : \tau$, and $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$. *EWF* follows by substitution: $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e''[v/x] : \tau$.

From the evaluation rule:

a. $\hat{h} = X^h$

b. $\hat{i} = ((X_I^i, X_H^i), H_i, e_i)$

c. $X_H \vdash \hat{i} : \tau'$

d. $(X_I, X^h)\,\texttt{link}\,(X_I^i, X_H^i) \Rightarrow (X_I', X_H'')$

e. $X_H' = X_H'' \oplus X_H$

f. $H\,\texttt{merge}\,H_i \Rightarrow H'$

g. $X_H \leq X^h$

h. $X_I^i \mid (X_H - X^h)$

As merging is well-formed for heaps (by fact *f*) and linking is well-formed for type environments (by fact *d*):

i. $X_I \diamond X_I^i$

j. $X^h \mid X_H^i$

k. $X_H^i \precsim X_I$

l. $X^h \precsim X_I^i$

m. $H \mid H_i$

As the loaded program is well-formed (by fact *c*), we have for some Φ_i:

n. $\vdash X_I^i \uplus X_H^i$

o. $X_I^i \uplus X_H^i \vdash \Phi_i$

p. $X_I^i \uplus X_H^i \vdash H_i : \Phi_i$

q. $X_I^i \uplus X_H^i; \Phi_i; \cdot; \cdot \vdash e : \tau'$

r. $X_H^i \mid X_H$

Since the running program is well-formed, we have for some Φ:

s. $\vdash X_I \uplus X_H$

t. $X_I \uplus X_H \vdash \Phi$

u. $X_I \uplus X_H \vdash H : \Phi$

v. $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \mathsf{load}[\tau']\ h\ i\ e_2\ e_3 : \tau$

By inversion of this last expression's typing judgment:

w. $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash h : \texttt{int}$

x. $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash i : \texttt{int}$

y. $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_2 : \tau' \to \tau$

z. $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_3 : \tau$

Figure A.1: Facts used in *load-success* case of the proof of Subject Reduction

Case 2: (load-success) $((X_I, X_H), H, \mathsf{load}[\tau']h\ i\ e_2\ e_3) \mapsto ((X_I', X_H'), H', e_2\ e)$

We must establish each of *TEWF*, *HTWF*, *HWF*, and *EWF*. Some useful facts are shown in Figure A.1. We define Φ as the least Φ that satisfies fact *u*, and Φ_i as the least Φ_i that satisfies fact *p*. Finally, we define Φ' as $\Phi \uplus \Phi_i$, which is well-defined as $\texttt{dom}(H) = \texttt{dom}(\Phi)$, $\texttt{dom}(H_i) = \texttt{dom}(\Phi_i)$, and $H \mid H_i$ by fact *m*.

To prove well-formedness of the new program we must establish:

- *(TEWF):* $\vdash X_I' \uplus X_H'$

  $X_I' \uplus X_H' = ((X_I \oplus X_I^i) - (X^h \uplus X_H^i)) \uplus ((X^h \uplus X_H^i) \oplus X_H)$
  By definition.

  $= ((X_I \oplus X_I^i) - (X^h \uplus X_H^i)) \uplus (X_H \oplus X_H^i)$

  Since $((X^h \uplus X_H^i) \oplus X_H) = ((X^h \oplus X_H^i) \oplus X_H)$
  By Lemma A.2.1 (3).

  $= ((X^h \oplus X_H) \oplus X_H^i)$
  By associativity and commutativity.

$= (X_H \oplus X_H^i)$

By Lemma A.2.1(7), given fact g $(X_H \leq X^h)$.

$= ((X_I \oplus X_I^i) - (X_H \oplus X_H^i)) \uplus (X_H \oplus X_H^i)$

Since $X_H \leq X^h$, this will be true as long as for all $n \in \mathtt{dom}(X_H - X^h), n \notin \mathtt{dom}(X_I \oplus X_I^i)$. This is true by fact h and fact s.

$= ((X_I \oplus X_I^i) - (X_H^i \oplus X_H)) \uplus (X_H^i \oplus X_H)$

By commutativity.

$= (X_I \oplus X_I^i) \oplus (X_H^i \oplus X_H)$

This is true by Lemma A.2.1 (5). For this Lemma to apply, we must show that $(X_H^i \oplus X_H) \precsim (X_I \oplus X_I^i)$. This is true by Lemma A.2.1 (4). For this Lemma to apply, we show $X_H^i \precsim X_I$ (by fact k), $X_H \precsim X_I^i$ (see below), $X_H^i \diamond X_H$ (by fact r), $X_I \diamond X_I^i$ (by fact i), $X_H^i \mid X_I^i$ (by fact n), and $X_H \mid X_I$ (by fact s).

To show $X_H \precsim X_I^i$, we must show that for all $n \in \mathtt{dom}(X_H) \cap \mathtt{dom}(X_I^i)$, $X_H(n) \leq X_I^i(n)$. We know $X_H \leq X^h$. For $n \in \mathtt{dom}(X_H)$ such that $n \in \mathtt{dom}(X^h)$, if $n \in \mathtt{dom}(X_I^i)$ then $X_H(n) \leq X_I^i(n)$ by fact q. Otherwise $n \in \mathtt{dom}(X_H - X^h)$, but then we know $n \notin \mathtt{dom}(X_I^i)$ by fact h.

$= (X_I \oplus X_H) \oplus (X_I^i \oplus X_H^i)$

By associativity and commutativity.

By fact s, $\vdash (X_I \oplus X_H)$, and by fact n, $\vdash (X_I^i \oplus X_H^i)$. So by lemma A.2.2, we have our result: $\vdash (X_I \oplus X_H) \oplus (X_I^i \oplus X_H^i)$.

- *(HTWF):* $X_I' \uplus X_H' \vdash \Phi'$

  This is equivalent to $X_I' \uplus X_H' \vdash (\Phi \uplus \Phi_i)$. Consider some $L : \tau \in \Phi'$; there are two possibilities:

  a. $L : \tau \in \Phi$. By fact t, $X_I \uplus X_H \vdash \Phi$, so $X_I \uplus X_H \vdash \tau$. As $X_I \mid X_H$, this is equivalently $X_I \oplus X_H \vdash \tau$. By A.2.3 (type environment weakening), $(X_I \oplus X_H) \oplus (X_I^i \oplus X_H^i) \vdash \tau$, which we have shown in the proof of *TEWF* is equivalent to $X_I' \uplus X_H' \vdash \tau$.

  b. $L : \tau \in \Phi_i$. By fact o, $X_I^i \uplus X_H^i \vdash \Phi_i$, so $X_I^i \uplus X_H^i \vdash \tau$. By similar weakening as above we may conclude $X_I' \uplus X_H' \vdash \tau$.

- *(HWF):* $X_I' \uplus X_H' \vdash H' : \Phi'$ This is equivalent to $X_I' \uplus X_H' \vdash (H \uplus H_i) : (\Phi \uplus \Phi_i)$. Consider some $L \in H'$; there are two possibilities:

  a. $(L = v) \in H$. By fact u, $X_I \uplus X_H \vdash H : \Phi$, so $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau$, where $\Phi(L) = \tau$. By A.2.3 as in HTWF, $(X_I \oplus X_H) \oplus (X_I^i \oplus X_H^i); \Phi; \cdot; \cdot \vdash v : \tau$.

  b. $(L = v) \in H_i$. $X_I' \uplus X_H'; \Phi'; \cdot; \cdot \vdash v : \Phi'(L)$ follows by similar reasoning.

- *(EWF):* $X_I' \uplus X_H'; \Phi'; \cdot; \cdot \vdash e_2\ e : \tau$

  We know $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_2 : \tau' \to \tau$ and $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_3 : \tau$. By the same weakening argument as above, $X_I' \uplus X_H'; \Phi'; \cdot; \cdot \vdash e_2 : \tau' \to \tau$, and $X_I' \uplus X_H'; \Phi'; \cdot; \cdot \vdash e_3 : \tau$. Therefore, we may conclude our well-formedness result.

Case 3: (load-failure) $((X_I, X_H), H, \mathsf{load}[\tau']\ h\ i\ e_2\ e_3) \mapsto ((X_I, X_H), H, e_3)$

EWF follows directly as $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_3 : \tau$.

Case 4: (reveal) $((X_I, X_H), H, \mathtt{reveal}(\mathtt{hide}_l\ v)) \mapsto ((X_I, X_H), H, v)$

During the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \mathtt{reveal}(\mathtt{hide}_n\ v) : \tau$ we must have concluded that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \mathtt{hide}_n\ v : n$ (where $(X_I \uplus X_H)(n) = \tau$), and again $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau$, which proves *EWF*.

Case 5: (ref) $((X_I, X_H), H, \mathtt{ref}\ v) \mapsto ((X_I, X_H), H \uplus \{L = v\}, L)$

TEWF follows by assumption. We show *HTWF* and *HWF* as follows. Consider the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \mathtt{ref}\ v : \tau$: by inversion $\tau = \tau'\ \mathtt{ref}$ and $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$, for some Φ. We may assume that $L \notin \mathtt{dom}(\Phi)$ by heap redundancy elimination since $L \notin H$ (by the side-condition on the evaluation rule). Therefore we choose $\Phi' = \Phi \uplus \{L : \tau'\}$.

To show *HTWF*, we must show that $X_I \uplus X_H \vdash \Phi'$. Consider an arbitrary $L' \in \mathtt{dom}(\Phi')$:

- if $L' \in \mathtt{dom}(\Phi)$ then $X_I \uplus X_H \vdash (\Phi \uplus \{L : \tau'\})(L')$ by assumption and heap weakening.
- if $L' = L$ then to show $X_I \uplus X_H \vdash (\Phi \uplus \{L : \tau'\})(L')$, we must show that $X_I \uplus X_H \vdash \tau'$. This follows because by the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \mathtt{ref}\ v : \tau$ we must have previously concluded that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$. By Lemma A.4.2, $X_I \uplus X_H \vdash \tau'$.

To show *HWF*, we must show that $X_I \uplus X_H \vdash (H \uplus \{L = v\}) : \Phi'$. Consider an arbitrary $L' \in \mathtt{dom}(H \uplus \{L = v\})$:

- if $L' \in \mathtt{dom}(H)$ then $X_I \uplus X_H \vdash H(L') : (\Phi \uplus \{L : \tau'\})(L')$ by assumption and heap weakening.
- if $L' = L$ then to show $X_I \uplus X_H \vdash H(L) : (\Phi \uplus \{L : \tau'\})(L')$, we must show that $X_I \uplus X_H \vdash v : \tau'$. But this follows by assumption, as noted above.

Finally, to show *EWF*, we note that $X_I \uplus X_H; (\Phi \uplus \{L : \tau'\}); H \uplus \{L = v\} \vdash L : \tau'\ \mathtt{ref}$.

Case 6: (deref) $((X_I, X_H), H, !L) \mapsto ((X_I, X_H), H, H(L))$.

By inversion, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash L : \tau\ \mathtt{ref}$, and furthermore that $\Phi(L) = \tau$. By program well-formedness, $X_I \uplus X_H \vdash H : \Phi$, which implies that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash H(L) : \Phi(L) = \tau$, which is the desired result.

Case 7: (assign) $((X_I, X_H), H, \mathtt{assign}\ L\ v) \mapsto ((X_I, X_H), H[L = v], v)$.

TEWF and *HTWF* follow by assumption for $\Phi' = \Phi$. To show *HWF*, we must show that $X \uplus X_H \vdash H[L = v] : \Phi'$. Consider some $L' \in H[L = v]$:

- if $L' \neq L$, then $X_I \uplus X_H \vdash (H[L = v])(L') : \Phi'(L')$ follows by assumption (since H has not changed at these labels).
- if $L' = L$, then we must show that $X_I \uplus X_H \vdash v : \Phi'(L)$. But by inversion $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash L : \tau\ \mathtt{ref}$ which implies (again by inversion) that $\Phi'(L) = \tau$. Also by inversion $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau$, which gives the desired result.

Finally, for *EWF* we must show that $X_I \uplus X_H; \Phi'; \cdot; \cdot \vdash v : \tau$. This follows trivially by inversion.

Case 8: (tapp) $(\Theta, H, (\Lambda\alpha.e)[\tau]) \mapsto (\Theta, H, e[\tau/\alpha])$, of type $\tau'[\tau/\alpha]$.

By inversion, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash (\Lambda\alpha.e) : \forall\alpha.\tau$ and $X_I \uplus X_H; \cdot; \cdot \vdash \tau'$. Doing this again we get $X_I \uplus X_H; \Phi; \alpha; \cdot; \cdot \vdash e : \tau$. We may now apply type substitution to conclude $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$

Case 9: (congruence rules) Follow by induction of $(\Theta, H, e) \mapsto (\Theta', H', e')$.

The following lemma is to establish the more useful Lemma of *Progress*, defined as a corollary.

Lemma A.5.2 *If $X_I \uplus X_H$, $X_I \uplus X_H \vdash \Phi$, $X_I \uplus X_H \vdash H : \Phi$, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau$, and e is not a value, then there exists an $((X_I', X_H'), H', e')$ such that $((X_I, X_H), H, e) \mapsto ((X_I', X_H'), H', e')$.*

Proof

Proof by induction on $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau$ and $((X_I, X_H), H, e) \mapsto ((X_I', X_H'), H', e')$. We will only consider the expression typing rules in which e is not a value:

Case 1: (app) $e = e_1\ e_2$

Three cases:

- e_1 is not a value
  By induction, there exists an $((X_I', X_H'), H', e_1')$ such that $((X_I, X_H), H, e_1) \mapsto ((X_I', X_H'), H', e_1')$. By congruence, $((X_I, X_H), H, e_1\ e_2) \mapsto ((X_I', X_H'), H', e_1'\ e_2)$.
- e_2 is not a value
  By induction, there exists an $((X_I', X_H'), H', e_2')$ such that $((X_I, X_H), H, e_2) \mapsto ((X_I', X_H'), H', e_2')$. By congruence, $((X_I, X_H), H, e_1\ e_2) \mapsto ((X_I', X_H'), H', e_1\ e_2')$.
- e_1 and e_2 are values.
  By canonical forms, $e_1 = \lambda x : \tau'.e$. So by beta reduction, $((X_I, X_H), H, e_1\ e_2)$ steps to $((X_I, X_H), H', e[e_2/x])$.

Case 2: (load) $e = \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3$

If either of the first two arguments is not a value, then by induction there exists a $((X_I', X_H'), H', e')$ such that $((X_I, X_H), H, e) \mapsto ((X_I', X_H'), H', e')$. Therefore, by congruence, load can take a step.

Otherwise, e_0 and e_1 are values and by canonical forms, some integers h, i. If the conditions for load-success hold (i.e. h is the representation of a type environment and i is the representation of well-typed program that is link-compatible with $\hat{h}$ and the current type environment) then $((X_I, X_H), H, \mathsf{load}[\tau']\ h\ i\ e_2\ e_3) \mapsto ((X_I', X_H'), H', e_2\ e)$. If not, the load-fail step rule applies and $((X_I, X_H), H, \mathsf{load}[\tau']\ h\ i\ e_2\ e_3) \mapsto ((X_I, X_H), H, e_3)$.

Case 3: (reveal) $e = \texttt{reveal}\ e_1$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 must be a value of type n, so by canonical forms, $e_1 = \texttt{hide}_n(v)$, and thus $((X_I, X_H), H, \texttt{reveal}(\texttt{hide}_n v)) \mapsto ((X_I, X_H), H, v)$ by the reveal reduction.

Case 4: (ref) $e = \texttt{ref}\ e_1$.

If e_1 is not a value, congruence rule applies. Otherwise, $((X_I, X_H), H, \texttt{ref}\ v) \mapsto ((X_I, X_H), H \uplus \{L = v\}, L)$ by the ref reduction.

Case 5: (deref) $e = !e_1$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 must be a value of type $\tau'\texttt{ref}$, so by canonical forms, $e_1 = L$. By inversion, $L \in \texttt{dom}(H)$, and by the deref reduction, $((X_I, X_H), H, !L) \mapsto ((X_I, X_H), H, H(L))$.

Case 6: (assign) $e = \texttt{assign}\ \ e_1\ e_2$.

If e_1 and/or e_2 are not values, congruence rule applies. Otherwise, e_1 is a value of type $\tau'\texttt{ref}$, so by canonical forms, $e_1 = L$. By inversion, $L \in \texttt{dom}(H)$, and by the assign reduction, $(\Theta, H, \texttt{assign}\, Lv) \mapsto (\Theta, H[L = v], v)$.

Case 7: (tapp) $e = e_1[\tau]$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 is of type $\forall\alpha.\tau$, so by canonical forms, $e_1 = \Lambda\alpha.e'$, so by tapp reduction $(\Theta, H, e_1[\tau]) \mapsto (\Theta, H.e'[\tau/\alpha])$.

Corollary A.5.3 (Progress) *If* $\vdash (\Theta, H, e) : \tau$ *and e is not a value, then there exists a* (Θ', H', e') *such that* $(\Theta, H, e) \mapsto (\Theta', H', e')$.

We say that a term is *stuck* if it is not a value and if no rule of the operational semantics applies to it. Type safety requires that no well-typed term can become stuck:

Theorem A.5.4 (Type Safety) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto^* (\Theta', H', e')$ *then* (Θ', H', e') *is not stuck.*

Proof

Proof is by induction on the number of steps of execution $(\Theta, H, e) \mapsto^* (\Theta', H', e')$ using Progress to show there is a new state and Subject Reduction to show that that new state is well typed.

Appendix B

Related Work

In this section, we present a survey of work related to dynamic software updating. We begin by looking two enabling technologies of dynamic update: state transfer and dynamic linking; for the latter area we also describe past work in static linking and component-based systems. Next, we examine systems directly targeted at modifying a running program; this discussion serves as a supplement to the analysis in §2.2. We also briefly describe the related area of dynamic reconfiguration of software architectures. Other surveys of dynamic updating approaches can be found in Gupta [Gup94], and Segal and Frieder [SF93].

B.1 State Transfer

We use the term *state transfer* to refer to systems that capture a program's state and then later restart the program with the saved state. State transfer can be used to implement dynamic updating by restarting a *new* program with an old program's state; problems with doing this are given in §1.1.2. State transfer is typically used to implement process migration [Smi88], checkpointing [Pla97] and general-purpose persistence [PJW96]. We briefly consider each area.

B.1.1 Process Migration

Process migration loosely describes techniques used to move running processes across the network to different machines, either for load balancing, or to access remote resources. An excellent survey of techniques can be found in Smith [Smi88]. Process migration systems typically either require the program to manually save its own state, or use an automated *checkpointing* technique to save and restore the state.

B.1.2 Checkpointing

Checkpointing technology is used to seamlessly capture a program's state and save it for later restore. The simplest checkpointing approaches are architecture-specific: they copy the pages of the program's heap and stack, plus the state of the registers, and store them in a file. The same program can then be easily restarted on the same kind of machine, since the code, addresses of data, *etc.* will be the same. The `libckpt` library [PBKL95] is a simple, elegant approach to this kind of checkpointing implemented in user-space. Unfortunately, a user-space library cannot capture OS-resident data, like mappings from file descriptors to open sockets, pipes, *etc.* However, some operating systems implement checkpointing, *e.g.* EROS [SSF99], and so can capture all relevant process data. With either user-space of OS-based approaches, the drawback is that the saved state is both program- and architecture-dependent.

Some work has been done in architecture-independent checkpointing. For example, both Hofmeister [Hof93] and Ramkumar and Strumpen [RS97] have developed means of instrumenting a program to capture its state *abstractly.* This is achieved by performing a source-to-source translation of the original program so that at programmer-identified points the program will capture its global state. In addition, some extra code is added before and after function calls to capture (and later restore) the stack. While this approach solves the architecture-specificity of checkpointing, it is still program-dependent. In particular, the layout of the stack matches the expected call sequence of the original program. Furthermore, languages like C, which are not strongly-typed, can thwart mechanisms for properly capturing (and restoring) pointers, a problem observed in the garbage collection community that led to so-called *conservative* collection [BW88].

B.1.3 General-purpose Persistence

General-purpose persistence (GPP) (*cf.* [PJW96]) is an approach whereby some of a program's data is made stable during its execution. If the program crashes while it is running, it can be restarted using the saved state. GPP generalizes database systems in that stable data need not be stored or extracted explicitly from the stable store using a special interface. Instead, program data is manipulated in normal fashion, and the system ensures it is made persistent automatically. Different approaches identify persistent data in different ways. For example, data can be marked syntactically (*e.g.* instead of declaring `int x;`, the programmer would declare `stable int x;`). Another approach is *persistence by reachability*—data reachable by any number of pointer dereferences from a *persistent root* is made persistent. Nettles and O'Toole [NO93] describe a way in which garbage collection can be used to elegantly identify and save reachable persistent data. To make sure that saved data is consistent, GPP is typically coupled with database-style transactions. The Argus system, described below, employs GPP and transactions to aid its updating approach.

B.2 Linking

Many systems use dynamic linking to add new code to a running program. While this is not as flexible as dynamic *updating*, dynamic linking does support extensible and adaptable systems. In this section, we look at research in dynamic linking, starting with its origins in static linking, and finishing with its common use in component-based systems.

B.2.1 Static Linking

The problems we face in developing a dynamic linker/updater are a superset of those faced by traditional, static linkers. A number of recent papers have presented formal descriptions of linking and considered when the operation is safe. Cardelli [Car97] developed a small module system based on the simply-typed lambda calculus and proved that linking in the system was type-safe. Glew and Morrisett [GM99] expanded Cardelli's approach in the setting of TAL, a type-safe assembly language. Finally, Flatt and Felleisen [FF98] defined program units, which are first-class software components, and showed that the linking of units is type-safe.

Incremental linking [QL91] is analogous to dynamic update done at compile-time, rather than runtime. The motivation here is to update an executable *in place* with modules that have changed since the last time the program was linked, rather than reconstruct the entire executable from scratch, to reduce link-time. This requires the redirection of pointers from the old code to the new.

B.2.2 Dynamic Linking

The term "dynamic linking" has been used to mean two different things in the literature. In one instance, it refers to the action by which parts of a conceptually static program are loaded incrementally. In this case, the executable file has "holes" that are filled in at program start-time or just before they are used at runtime by loading modules. Most operating systems implement this approach to allow for dynamically loaded libraries (DLL's), with the interest of reducing executable size (since the library code is not stored in the executable itself) and facilitating changes to the library without requiring application recompilation. Dynamic linking may also refer to the operation by which new modules may be loaded into the running program, based on the execution of the program itself. In this case the program has no holes, it is extended with new functionality. Dorward [DSS90] refers to these two operations as incremental and dynamic linking, respectively; we adopt that convention here.

Franz [Fra97] presents an overview of the history of dynamic and incremental linking, and five approaches that may be used to implement incremental linking: load-time indirection, load-time rewriting, runtime rewriting, load-time compilation, and dynamic compilation. In the first two cases, all modules are loaded just before program execution. In load-time indirection, modules are compiled so that references to other modules are made indirectly via a link table. Linking then reduces to filling the link table in each module appropriately; this is essentially the mechanism we use in DLpop (see Chapter 6). In load-time rewriting, rather than pay the penalty of the indirection for each external call at runtime, each of the call-sites for an external reference is overwritten with the correct address, as in most static linkers; our alternate implementation of DLpop using runtime code generation mimics this approach (see §6.4.2). Runtime-rewriting is a lazy implementation of this approach: it waits until a reference is accessed before it is rewritten. The last two approaches consider pushing some elements of compilation, either code generation or the entire compilation process, until runtime. This prevents the need for fixing up links, as the links are not generated until runtime.

Ho and Olsson [HO91] present one of the earliest approaches to dynamic linking for C. Their system lacks type-safety (loaded symbols may be coerced to incompatible types at runtime) and program integrity (program modules may be unloaded while still in use, leaving dangling pointers). Their interface is much like the modern C dynamic linking library for UNIX, DLopen [Lin95], which is described informally in 6.1.

Dynamic linking is provided in a number of object-oriented languages, most notably Java [AG96], and C++, either via special means [DSS90, HG98] or with DLopen. These systems preserve type-safety, although in the case of C++ the lack of strong typing weakens this property. Both of the specialized C++ systems require newly loaded classes to be related to (are subtypes of) classes in the running program; Java does not impose this restriction. Java dynamic linking is incremental. Newer implementations of Java allow class unloading if code is unreachable, although reachability conditions are more conservative than one might guess [Jav98].

Many functional languages provide dynamic linking, in particular Scheme [Sch, Dr.], Haskell [PHL97], and OCaml [Rou96] (it has been proposed for SML [App94, GKW97] but not implemented as far as I can tell). The type-safety of the running program is preserved during dynamic linking. Most implementations do not implement dynamic unlinking, although allowing code to be garbage-collected would be likely straightforward. Dr. Scheme [Dr.] allows the dynamic linking of Units [FF98], described in §6.4.2; while the current Units implementation is dynamically typed (and thus has weaker type-safety guarantees), statically typed implementations are possible.

Other than the TAL/Load approach developed here, only one other approaches has been designed for dynamically linking of verifiable native code (VNC). VNC is native code coupled with annotations that ensure its safety can be formally verified. Two existing flavors of VNC are TAL [MWCG99] (used by our updating approach) and PCC [Nec97]. Duggan proposes Typed Modular Assembly Language (TMAL) [Dug00]. TMAL defines a number of extensions to the underlying assembly language to express dynamic linkages similar to TAL/Load; the drawback is a larger TCB. See §6.4.2 for details.

Peterson *et. al's* Haskell implementation HUGS [PHL97] and Appel's "hot-sliding" approach for SML [App94] both claim to support dynamic update, but really support *plug-in extensions*, since bindings are fixed at static- and dynamic-linking time. As a result, the program must make extensive provision for incorporating new code, and may still be limited in some cases, as described in §1.1.3. This approach is not unlike other special-purpose updating approaches, as in SPIN [BSP$^+$95], and dynamic protocol architectures, including active networks [HN00], the Click Modular Router [MKJK99], and others.

There is a lot of interest in so-called *software components*, defined by systems like COM [com01] and CORBA [cor01]. Components are more application-independent than typical dynamically-loadable libraries in that they contain information to allow an application to query their contents. More on components (and COM in particular) can be found in §6.4.2.

B.3 Dynamic Updating

Research into dynamically updating running programs goes back as many as thirty years. Dynamic updating (our term) appears under a number of monikers in past work, including dynamic (software) reconfiguration, dynamic code replacement, dynamic code improvement, and on-line software version change. Some work has examined the formal foundations of dynamic update, and a number of systems have been described and/or implemented. We look at both avenues of work below, presented chronologically.

B.3.1 DYMOS

While some earlier systems exist, Lee presented one of the first systems to support dynamic updating, called DYMOS (for Dynamic Modification System) [Lee83]. Though nearly twenty years have passed, DYMOS remains one of the most flexible dynamic updating systems ever envisioned. Designed for a multi-threaded variant of Modula called StarMod [Coo80], DYMOS permits changes to module definitions—including type, function, and data definitions—and loop bodies (to allow infinite loops to be updated).

The DYMOS environment is *integrated*, so that with each updateable program is associated a command interpreter, a source code management system, a StarMod compiler, and a supporting runtime environment. In fact, this approach is not too different from the so-called *top-level environments* available in typical functional programming languages, like Scheme (*cf.* Scheme 48 [Sch]), or ML (*cf.* Objective Caml [Ler00]). Like these systems, a DYMOS user can define and compile modules, but unlike them, the user can *update* existing code. Updates are performed using a special command that specifies which modules to update, and when they should be updated. The latter condition is specified as a list of modules and/or functions that should not be active when the new code is installed in the running program; see §8.2.1 for more on this. Implementing the runtime environment to enforce these so-called *when-conditions* can be costly, in terms of performance and implementation complexity.

DYMOS aimed to provide a flexible updating system with particular concern for update correctness. Part of the benefit of the integrated approach is that the system has intimate knowledge of the running program, both its executable image and its source-code makeup. As a result, attempted changes to the program can be type-checked in the context of the rest of the program. This approach achieves the effect of our use of Typed Assembly Language, but with a substantially larger trusted computing base. In addition, the system's runtime enforcement of when-conditions provides programmers with some mechanisms for ensuring that updates are applied at the correct time. As a theoretical aid to the use of these enforcement mechanisms, Lee defined an algorithm for partitioning an update into several smaller updates (see §8.2.1); unfortunately, no criteria were presented for ensuring proper timing of *any* update, whether large or small. This deficiency of determining reasonable update timing remains in large part in the field today.

B.3.2 Dynamic Module Replacement in Argus

Argus is a programming language for building reliable distributed applications [Lis88], based on the CLU programming language. In Argus, a distributed application consists of one or more *guardians*, which are essentially like UNIX processes, that can communicate with each other using an RPC-like interface. Unlike typical UNIX processes, guardians are equipped with mechanisms that allow them to be restarted following a crash. In particular, the programmer identifies some portion of the guardian's state as *stable*. Stable state is made *persistent* during program operation so that it can be recovered later (such as by storing it on disk), and the state is kept *consistent* through the use of database-style transactions.

Like other distributed programming environments supporting reconfiguration (such as

Conic and PolyLith; see below), the connections between guardians can be rerouted dynamically. Bloom describes the design of a dynamic updating system [Blo83, BD93] so that a guardian's code can be replaced while it runs. To maximize flexibility while preserving type-safety, Bloom allows collections of guardians, called *subsystems* to be replaced simultaneously. We allow groups of modules to be updated in a running program simultaneously for the same reason; that is, if only single modules/guardians could be replaced, functions used by other modules/guardians in the program could not change type.

Guardians to be replaced must be coerced to a *quiescent* state, meaning that no transactions are being computed; the system can either abort running transactions or wait for them to complete. New guardians are then started using the old guardians' state, possibly with some alteration. To notice these new guardians, old guardians must be redirected. This is done by requiring each RPC call to perform an extra lookup to find the most recent version of the function called, at some performance cost.

Another, more application-level approach to replacement in Argus was proposed by Day [Day87, BD93]. Here, replacement of individual guardians takes place by using standard Argus facilities of crash recovery. To replace a guardian, the old guardian is crashed and the new version is restarted with the existing state of the old one. To be able to start a new guardian with old state, the old state is encoded at crash-time and then decoded when the replacement guardian starts up; this approach is used in PolyLith (see below). So that existing guardians are properly redirected to the new guardian, it is expected that RPC's will handle a special exception. When a guardian makes an RPC call and this exception is raised, it will try to relocate the function it was attempting to call. Upon finding the new guardian, it directs its call there. This approach has the disadvantages that more work is required of the user, and that only individual guardians can be replaced.

Perhaps the largest lesson of Argus is that fault-tolerance mechanisms for ensuring consistent state (*i.e.* transactions and general-purpose persistence) can be leveraged to provide a form of dynamic updating, but at the cost of a greater implementation burden.

B.3.3 Conic

Conic [MKS89, MK85] is a distributed programming system that allows distributed components of an application to be reconfigured. Like Argus applications, Conic applications consist of a number of processes distributed throughout the network that communicate through a well-defined entry points. A process's communication channels are separate from the process's code, so channels can be redirected at runtime by a configuration manager. The Conic language ensures that connections are always well-typed. However, while Conic can redirect connections between processes, it it does not provide system-level support to update the code of a running process, as is possible in Argus. This therefore requires Conic applications to be coded to capture and transfer their state, similar to, but not as flexible as, the application-level approach proposed by Day for Argus [Day87].

B.3.4 PODUS

PODUS [FS91, SF93] (Procedure-Oriented Dynamic Update System), developed by Mark Segal and Ophir Frieder, provides for the incremental update of procedures in a running program. Multiple versions of a procedure may coexist, and updates are automatically delayed until they are *syntactically* and *semantically sound* (as determined by the compiler and programmer, respectively).

Updates are only permitted for non-*active* procedures. Syntactically active procedures are those that are on the runtime stack and/or may be called by the new version of a procedure to be updated. Semantically related procedures are defined by the programmer as having some non-syntactic dependency. Thus, if a procedure A is currently active, and is semantically related to procedure B, then B is considered semantically active. Procedures whose code does not change from version to version are never considered active. By forbidding updates to active code, PODUS loses some flexibility, since long-running event-loops or top-level functions can never be replaced.

If a procedure may not be updated because it is active, then the system waits until it is no longer active. Because PODUS programs must be single-threaded, the set of running procedures is effectively the stack; extra code is inserted by the compiler to check the stack depth upon procedure return and thus determine when procedures are no longer active. Once all updates have been applied, the entire program runs using the most recent version.

Procedures whose interface changes as a result of dynamic update are made available to old code via *interprocedures*; these are analogous to our stub functions. Interprocedures translate the arguments required by the old interface to those required by the new one, call the new function, and then translate the result to be returned to the old code. Similarly, *mapper procedures* translate static data that must be migrated between versions; the combination of all mapper procedures is essentially equivalent to our state transformer function. Unlike interprocedures, which are invoked with each procedure call, mapper procedures are invoked only at update-time.

PODUS is implemented by overloading a segmented virtual memory infrastructure. All procedure calls are notated by sparse addresses which contain version information. Versions numbers (among other things) are used index the segment table; thus, multiple versions of a program are in separate segments of the program. Interprocedures may be used to map calls from one segment to another. This approach is quite similar to using *reference indirection* (see §7.2.1), except it is more flexible: rather than using a single, well-known indirection table to point to the symbols of the current version of the program, the segment register is used to index a table of indirection tables, one per program version. However, while segmented virtual memory provides more flexibility, this comes at the cost of performance, since segmented addressing typically requires system calls (a user-space simulation would avoid this cost but impose others).

B.3.5 Reconfigurable PolyLith

PolyLith [Hof93] is a distributed programming system for C that supports *reconfiguration*. Three forms of reconfiguration are supported: structural, geometric, and modular. The first two reconfigurations concern the number of individual processes (termed *modules*) that make up a distributed program, and connections between them, while the third considers the implementation of the individual processes. A structural change alters the process-level makeup of the program, either by redirecting communication channels or by adding or deleting processes. A geometric change is tantamount to process migration. These two changes are provided by Conic, described above. A modular change is one in which a process's program is altered; in PolyLith terminology, dynamic software updating is a methodology for modular change.

Because PolyLith processes use a special library for communication channels, structural changes can occur without altering the processes themselves. That is, the bindings between processes can be altered seamlessly. However, geometric and modular changes require process-level participation to perform state capture and restore. As with our system, modular changes may only occur at *reconfiguration points* specified in advance by the programmer, basically implementing the *invoke* updating model (see §8.2.2. Like the application-level approach in Argus, updating occurs by starting a new and/or relocated version of the program with the captured state. PolyLith therefore has the deficiencies of state transfer regarding updating (see §1.1.2), but can support machine-independent process-migration (up to certain flexibility limitations, see below).

PolyLith provides automatic support for state capture and restore but also requires programmer aid. In particular, a compile-time transformation is used to automatically capture the program stack at the time of update. Capturing the heap and static data require assistance from the programmer. Similarly, an automatic transformation is used to generate code to restore the stack when a program is restarted. The major drawback here is that unless the structure and semantics of the stack remain unchanged in the new version of a process's program, the automatically-generated code will have to be altered by hand; drastic changes to program structure would make hand alterations nearly impossible. Furthermore, the transformation does not properly deal with pointerful data stored on the stack, like function and data pointers. As a result, this code also needs to be inserted by hand. On the other hand, programs could be structured so that stack capture is almost totally avoided, so that only heap and data capture is required; we advocate for such a structure in our system in §8.2.2.

No specific performance data for PolyLith is available, but we note that the stack capture routines result in a number of extra checks added at runtime, even when the stack is not being restored. Of course, if the program is well-structured, such checks will be infrequent.

B.3.6 On-line Software Version Change

Gupta and Jalote present a framework for doing dynamic update implemented as a state transfer between programs written in C [GJ93]. In subsequent work, they establish formal groundwork for reasoning about dynamic update in general [Gup94, GJB96].

Their implementation is based on state transfer: the running program's state is captured and the new code is started with the existing state. State capture occurs by copying the stack and data area *as is* to the new process, meaning the mechanism is platform-dependent. Like PODUS, the system allows the user to code interprocedures to migrate between different interfaces, and uses a state transformation function for global state. As with PODUS, no procedures on the runtime stack may be updated. Unlike PODUS (and DYMOS), no framework is given for automatically delaying an update should there be active procedures (so far as I could tell).

The advantage of platform-dependent state transfer is that it can be easily implemented in the framework of standard tools; *e.g.* no special compiler support is required, even a general-purpose checkpointing library (*e.g.* `libckpt` [PBKL95]). It has the standard drawbacks as well (§1.1.2). Moreover, because stack and heap data are copied as is, all procedures and data in the new program must be at the same addresses as before. This prevents the straightforward addition of new global data.

Perhaps Gupta's most important contribution is a formal framework for understanding whether a dynamic update is *valid*, and a proof that determining update validity is, in general, undecidable. The basic framework is described in §8.2.1. In light of this framework, Gupta considers different programming language styles, including imperative languages without procedures, imperative languages with procedures, and object-oriented languages. Unfortunately, while the basic result of undecidability is interesting and useful, the formalism does not extend to realistic languages very easily. Like the theoretical work of Lee for DYMOS and Bloom for Argus, not enough is known about validity to be of much use to the programmer in writing updateable programs. Therefore, an improved formal framework is still much-needed future work.

B.3.7 Erlang

Erlang is a dynamically typed, concurrent, purely functional programming language designed for building long-running telecommunications systems [AVWW96]. Erlang provides language-level and library support for dynamic updates to modules in running programs. Updates are permitted to running code, but the update does not take effect until it is called from an external source. Like our approach, calls *within* the old module call the old rather than new code. Erlang does not provide looping constructs; instead, as in many functional languages, loops are programmed via tail-recursion. So that these 'loops' can be updated, any call to procedure within the caller's module that is fully-qualified (*i.e.* function `iter` in module `M` syntactically specifies its recursive call as `M.iter()` rather than simply `iter()`) then the new version of the function is called, if it is available. Only two versions of code can be available in the system at any given time; the current old version of code must be explicitly deleted (if any exists) before new code may be loaded, and certain library routines may be used to detect if the old code is still in use.

The main strengths of Erlang are the simplicity of this interface and implementation; one implementation is described in [Hau94]. Erlang employs reference indirection to realize updating (see §7.2.1): all fully-qualified calls are indirected through an indirection table, and the table entries are redirected at load time, while intra-module calls are directly addressed. This requires very little compiler support, and is not too much beyond simple dynamic linking.

Like our approach, Erlang uses the *invoke* model, rather than the *interrupt* model. Reasoning about updates in Erlang is made more straightforward by two key language features: 1) all data is write-once (no mutation), and 2) all thread-communication occurs via message passing. In effect, only one thread will ever "change" long-lived data (by passing a modified copy to its recursive call), and all other threads may only access this data in some distilled form via message passing. In this way, essentially *all* function calls to other modules are stateless: the state carried around by a thread is in its argument list, and the only way to get at state managed by another thread (called a *server* in Erlang jargon [Erl97]) is to pass it a message and receive its response (which is separate from function call). In the jargon of information leakage (see §11.3), we call this *thread-protected state*. Of course, while reasoning about global state is now simpler, there is still room for inconsistency. As a result, programming patterns for update, such as barrier synchronization, are used to ensure update correctness. Some such patters are described in the Erlang specification [Erl97].

The main drawback of Erlang is its lack of load-time checking. As a result, problems due to a type-incorrect change will be delayed until later in program execution, at which time it may be difficult to recover and/or determine the reason for the problem.

B.3.8 Dynamic ML

Dynamic ML [GKW97] is a proposed implementation of ML that enables distributed, agent-based programming via the replacement of module components at runtime, including both signatures and structures. If a structure is replaced, then it must match the signature (interface) of the old version. If definitions have been added to the module that should be made visible, then these are revealed by subsequently replacing the module's signature with a wider one.[1] Additionally, the programmer is allowed to respecify the implementation of abstract (non-transparent) type definitions in the new module. Multiple versions of a module are not allowed to coexist, so if the ADT implementation changes, then the user must provide a series of conversion routines to translate the present values of the type to use the new implementation. The formal semantics [WKG98] dictate that all conversion is performed "atomically" during a copying garbage collection initiated by the update. As live values whose implementation has changed are copied to to-space, they are filtered through the conversion routines; if an error should occur during conversion, resulting in a thrown exception, then the update is aborted, reverting to old representations in from-space. Finally, because only one version of a module can exist in the program, no updates may be performed on active modules.

This approach has a number of advantages. First, the system employs automated mechanisms to ensure correctness and simplify the creation of updates. In particular, updates are coded as functors between the old and new implementation, with the advantage that functors are already specified in the language (and will thus be familiar to the programmer), and that type-safety is guaranteed by the compiler. Furthermore, the compiler will guarantee that type conversion functions are present, and thus updates are provably *complete*. Using copying garbage collection as a means to type translation and rollback elegantly leverages a standard implementation requirement. Furthermore, a more sophisticated collector (*e.g.* replicating collection [ON94]) would allow state transformation to occur concurrently with program execution.

However, the sorts of changes that may be effected to implementation are limited. In particular, there are many instances in which a module might export a translucent (*i.e.* non-abstract) type such that its implementation is known to the client. These sorts of types may not be changed, since doing so could break the assumptions compiled into client code. Furthermore, running code cannot be updated, and update timing is not under programmer control.

[1]Conversely, if components are to be deleted then the signature must be replaced first, followed by the structure.

B.3.9 Dynamic Classes for C++

Hj'almt'ysson and Gray have designed and implemented mechanisms for the dynamic update of classes in C++ [HG98]. Their implementation requires the programmer to specially code classes that may be dynamically replaced using a proxy class `Dynamic`. `Dynamic` allows objects of multiple versions of a dynamic class to coexist: it maintains a pointer to the most recent version of a class, directing constructor calls to that class, while instance methods are executed by the class that actually created the object. The implementation is simple and portable, and demonstrates the appeal of object-oriented code replacement: by using instance methods, an instance's operations are consistent throughout its lifetime, even if a newer version of its class is loaded later.

While the approach is simple and portable, it lacks some flexibility in two key ways. First, `static` methods may not be replaced. This is due in part to the library-based implementation; adding a compilation approach similar to ours might mitigate this problem. As a result, the system would appear to be severely restricted as all conceptually global data must be anticipated at deployment. Second, existing instances of a changed class cannot be changed; the model requires that existing instances die naturally. This may prevent some useful program restructurings.

B.3.10 Dynamic Java

Dynamic Java [ACR98] proposes alterations to the Java runtime to allow for the runtime replacement of classes in a Java program. Like Dynamic ML modules, no two versions of a class may be in use simultaneously; however, the authors specify ways by which code may be updated incrementally rather than all at once: whenever an object of the old class is referenced, it is converted to suit the new class. Other references to the old object are also pointed to the new version lazily, and mutual exclusion is used to make sure that the same object is not converted by two different threads. These are implemented by extending all objects and classes with a pointer to their respective substitute version. No mention is made in the proposal about the semantics of correctness for updates.

B.3.11 Dynamic Java Classes

The Dynamic Virtual Machine (DVM) is a modified Java Virtual Machine (JVM) that implements Dynamic Java Classes [MPG^{+}00]. In essence, the DVM implements two new trusted functions for the default Classloader that allow users to replace Java classes dynamically. The DVM requires that classes be replaced logically at load-time: any existing instances of the old class must be made compatible with the new class before they are accessed. This prevents arbitrary syntactic and semantic changes from being made to a class since old clients may use the class incorrectly.

To more easily ensure that updates are correct, the majority of the updating functionality is integrated within the Java runtime. This has two consequences. First, the DVM TCB is made larger by the extra implementation, reducing assurances of security. Second, the changes defined by the system are implementation-specific; they will not quickly transfer to other JVM's, as would a library and/or compilation-based approach. Moreover, the implementation seems to rely on a byte-code interpreter due to problems with just-in-time (JIT) compilers used in more modern virtual machines. Consequently, the DVM's performance is far below current standards for Java.

B.3.12 DITools

DITools (Dynamic Interposition Tools) [SNC00] is a software library that exploits the presence an indirection table per ELF object file, used to store references external to that file extant in shared libraries. The DITools library overrides the standard dynamic linker to customize how these indirection tables are filled in, allowing users to *rebind* the definition normally associated with a reference, or *interpose* a function in between the reference and its original definition. DITools is designed chiefly to customize third-party executables, *e.g.* to enable performance monitoring, or service customization, and therefore lacks much infrastructure germane to dynamic updating, including the patch generation facilities and specialized compilation.

B.3.13 Guarded Software Updating

Tai *et. al* propose an approach to upgrading the software for long-lived, deep-space missions called *guarded software updating* (GSU) [TTA$^+$99, TTA$^+$00]. Their proposed system is designed for the X2000 avionics architecture, but should generalize to other multi-processor architectures in principle. GSU's chief contribution is the use of a transparent, message-based protocol to test new software while it runs, gradually building up trust that the new software is correct. When a replacement software process begins execution, the old version continues to run in *shadow mode*, meaning that its messages are only logged, while the new software version's messages are actually sent. If an erroneous message from the new software (or a message from a different process influenced by the new software) is detected, the system switches over to using the old version. So that this switch over is semantically consistent, GSU employs checkpointing technology [Pla97] to checkpoint the state of the old version and and other on-board processes when their state is known to be consistent. When the switch over takes place, the relevant processes are rolled back as necessary to use the checkpointed state.

Erroneous message detection occurs transparently. Normal calls to send or receive messages instead go through a middleware library that performs checkpointing and logging, when necessary, and checks messages for accuracy. Message accuracy is checked using user-provided *acceptance tests*, which are predicates indicating whether a message is well-formed. General error detection is assumed to be localized in exchanged messages; that is, no other forms of possible error are checked for.

Few details about the process of loading and executing the new software version are provided. The process appears to be roughly as follows:

1. Load the new software process image onto an idle processor.
2. Start the process with the checkpointed state of the process to be replaced.
3. Switch the old process into shadow mode.

Based on this updating procedure, we can see some potential drawbacks regarding the flexibility of the system. The new version must be able to work with the state of the old version just as it is, preventing a number of possible changes.[2] A similar restriction appears to be placed on the external behavior of the new process, with respect to the messages it sends. That is, in order for GSU to work, the new and old processes must exhibit the same message behavior based on the same stimulus, within the realm defined by the acceptance test. In other words, the new software is restricted to using roughly the same message protocols as the old version, precluding changes to those protocols.

Despite these disadvantages, GSU is successful in many ways. Because the old and new versions run on separate processors, the only performance overhead is the additional code in the message sending and receiving primitives. Also, constraining the form of updates can make the system easier to use by reducing complexity. Most importantly, GSU is unique in its ability to test and roll back software updates once they are on-line. In particular, while our system, Dynamic ML, and Argus permit short-term rollback (that is, immediately following an update), GSU can roll back an update long after it has been in operation. The techniques to allow longer-term rollback should be applicable to other software updating approaches. For example, we should be able to program the GSU error detection and recovery protocol as a library in our system with the addition of either a general- or special-purpose checkpointer.

B.3.14 DynInst

DynInst (for Dynamic Instrumentation) [BH00] defines an API for patching a running program with dynamically generated code. Unlike the template-based approaches of other runtime code generation systems [CLM+98, HJ99], DynInst allows more arbitrary changes as determined at runtime. In particular, the user may specify what are essentially abstract syntax trees which are compiled and inserted into the running program at specified points. The enabling technology is the Metric Description Language (MDL) [HMG+97], which uses machine-specific techniques to insert the code into the running system. In essence, the programmer identifies a *point* in the program, say the first instruction inside a procedure body, and a *snippet*, which is a piece of code to insert. A process called the *mutator* attaches to the process and inserts the snippet at the desired point by replacing the existing instruction with a branch to a *base trampoline*, which contains code common to all snippets. This trampoline saves the registers and then jumps to a *mini-trampoline* that contains the actual snippet code. Once this code completes, it returns to the base trampoline which restores the registers and executes a copy of the replaced instruction before returning to the main program.

DynInst's main advantage is that a program can be updated without requiring special compilation.[3] Furthermore, a performance penalty is only paid for the code that actually has been updated; this includes the cost of the added branch, and the cost to save and restore the appropriate registers.

[2]The authors state that as a benefit of their approach, no special-purpose engineering need be performed on the new software version. We therefore infer that no transformation is expected.

[3]The debug flag needs to be turned on so that points can be properly identified, but this is still standard compilation.

On the other hand, the methodology used in DynInst has a couple of disadvantages. First, while the API defined is portable, much of the implementation code is architecture-specific; this is because changes are defined at the system level rather than at the source-code level. Furthermore, the implementation of update by trampolining can be quite difficult and error-prone (compared to a compiler-inserted indirection, for example).. Second, there is no guarantee that an inserted snippet will interact well with the running program; in particular, type-unsafe code can easily be inserted which will crash the system. Third, there is little support for long-term evolution of the system. For example, alterations must be done by hand, although the authors state that they ultimately expect high-level analyses to target the the API. In addition, there is little concern about maintaining consistent state. This lack of support is largely by design: DynInst is intended for analysis purposes, so that a running program can be instrumented to count certain events, or to debug anomalous behavior.

B.3.15 Dynamic Architectures

In the realm of software engineering, there is an extensive body of work examining *system architectures*. A system architecture provides a view of the system's implementation, considering the interactions among high-level components, such as modules or objects, or even more heavyweight entities such as processes or machines. A number of tools exist for constructing and analyzing system architectures, and for facilitating their implementation.

Dynamic architectures model systems that may change during execution, and thus identify the mechanisms needed to implement such a system. Oreizy [Ore96] identified the additional features that a dynamic architecture must have above a static one, including modification operations, modification constraints, and legal modification times. Dynamic architectures typically allow reconfiguration, in the style of Conic or PolyLith, but not state-preserving update of existing components.

Bibliography

[5ES] The 5ESS Switch. `http://www.lucent-ssg.com/switching/products/configurations/switch.shtml`.

[AA98] Joe Armstrong and Thomas Arts. A practical type system for Erlang. Technical report, Erlang User Conference, 1998.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Preliminary version in Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, June 1992.

[ACR98] Jesper Andersson, Marcus Comstedt, and Tobias Ritzau. Run-time support for dynamic Java architectures. In *Proceedings of ECOOP Workshop on Object-Oriented Architectures (OOSA)*. AITO, July 1998.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[AHI$^+$00] K. G. Anagnostakis, M. W. Hicks, S. Ioannidis, A. D. Keromytis, and J. M. Smith. Scalable resource control in active networks. In Hiroshi Yashuda, editor, *Proceedings of the Second International Working Conference on Active Networks*, volume 1942 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, October 2000.

[Ale98] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks.* PhD thesis, University of Pennsylvania, September 1998.

[App94] Andrew Appel. Hot-Sliding in ML, December 1994. Unpublished manuscript.

[ARI] ARINC, Inc. `http://www.arinc.com`.

[Arm97] Joe Armstrong. The development of Erlang. In *SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM, June 1997.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, second edition, 1996.

[BD93] Toby Bloom and Mark Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, March 1993.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[BL01] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and distributed online games. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*. IEEE, April 2001.

[Blo83] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, March 1983.

[BP98] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[BSP$^+$95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, Colorado, 1995.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

[Cac] The Cactus project, University of Arizona. `http://www.cs.arizona.edu/cactus/`.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 266–277. ACM, January 1997.

[CLM$^+$98] Charles Consel, Julia Lawall, Renaud Marlet, Gilles Muller, Jacques Noyé, Scott Thibault, and Nicolae Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), September 1998.

[CLN$^+$00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, June 2000.

[COF] Microsoft portable executable and common object file format specification. `http://msdn.microsoft.com/library/specs/msdn_pecoff.htm`.

[com01] Microsoft COM technologies, 2001. `http://www.microsoft.com/com/default.asp`.

[Coo80] R. P. Cook. *Mod—a language for distributed programming. *IEEE Transactions on Software Engineering*, 6(6):563–571, 1980.

[cor01] OMG's CORBA website, 2001. `http://www.corba.org/`.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Third ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.

[Day87] Mark S. Day. Replication and reconfiguration in a distributed mail repository. Master's thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, April 1987.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, April 1997.

[Dr.] Dr. scheme homepage. `http://www.cs.rice.edu/CS/PLT/packages/drscheme/`.

[DSS90] Sean C. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding new code to a running C++ program. In *Proceedings of the USENIX C++ Technical Conference*, pages 279–292, April 1990.

[Dug00] Dominic Duggan. Sharing in Typed Module Assembly Language. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[Ens] The Ensemble distributed communication system. `http://www.cs.cornell.edu/Info/Projects/Ensemble/`.

[Erl97] Erlang design principles, February 1997. `http://www.erlang.org/doc/doc/doc/design_principles/part_frame.html`.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of SIGPLAN International Conference on Programming Language Design and Implementation*, pages 236–248. ACM, June 1998.

[Fog99] Karl Franz Fogel. *Open Source Development with CVS.* Coriolis Group, 1999.

[Fra97] Michael Franz. Dynamic linking of software components. *IEEE Computer*, 30(3):74–81, March 1997.

[FS91] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, September 1991.

[GBHC00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Symposium on Operating Systems Design and Implementation*, 2000.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pages 63–92, 1971.

[GJ93] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software—Practice and Experience*, 23(9):949–964, September 1993.

[GJB96] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *Transactions on Software Engineering*, 22(2):120–131, February 1996.

[GKS86] D. Garlan, C. W. Krueger, and B. J. Staudt. A structural approach to the maintenance of structure-oriented environments. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 160–170, Palo Alto, CA, 1986. ACM.

[GKW97] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, The University of Edinburgh, December 1997.

[Gle00] Neal Glew. *Low-Level Type Systems for Modularity and Object-Oriented Constructs*. PhD thesis, Cornell University, January 2000.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 250–261. ACM, January 1999.

[GM00] Dan Grossman and Greg Morrisett. Scalable certification for Typed Assembly Language. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[Gup94] Deepak Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[Hau94] Bogumil Hausman. Turbo Erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[HG98] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[HJ99] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. *Journal of Higher-Order and Symbolic Computation*, 12(4), 1999. An earlier version appeared in Partial Evaluation and Semantics-Based Program Manipulation, January 22-23, 1999.

[HK99] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Third ACM International Conference on Functional Programming*, pages 86–93. ACM, September 1998.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 123–137, January 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[HMA+99] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, March 1999.

[HMG+97] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonçalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212, November 1997.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[HN00] Michael Hicks and Scott Nettles. Active networking means evolution (or enhanced extensibility required). In Hiroshi Yashuda, editor, *Proceedings of the Second International Working Conference on Active Networks*, volume 1942 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, October 2000.

[HO91] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 21(4):375–390, April 1991.

[Hof93] Christine Hofmeister. *Dynamic Reconfiguration*. PhD thesis, Computer Science Department, University of Maryland, College Park, 1993.

[HW00] Michael Hicks and Stephanie Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, April 2000.

[jav96] Basics of java class loaders, 1996. `http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html`.

[Jav98] Clarifications and amendments to the java language specification, May 1998. `http://java.sun.com/docs/books/jls/unloading-rationale.html`.

[JRL00] John R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 2000.

[Lee83] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, April 1983.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[Ler00] Xavier Leroy. *The Objective Caml System, Release 3.00*. Institut National de Recherche en Informatique et Automatique (INRIA), 2000. Available at `http://caml.inria.fr`.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1997.

[Lin95] DLOPEN(3). *Linux Programmer's Manual*, December 1995.

[Lis88] Barbara Liskov. Distributing programming in Argus. *Communications of the ACM*, pages 300–312, March 1988.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MBC$^+$99] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an active network. In *Proceedings of the Thirty-seventh annual Allerton Conference on Communication, Control and Computing*, September 1999.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[MH97] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[Mit86] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions (summary). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 308–319, New York, 1986. ACM.

[MJ98] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[MK85] Jeff Magee and Jeff Kramer. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, April 1985.

[MKJK99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[MKS89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[MPG$^+$00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the Fourteenth European Conference on Object-Oriented Programming*, June 2000.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Second ACM International Conference on Functional Programming*. ACM, June 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.

[NO93] Scott Nettles and James O'Toole. Implementing orthogonal persistence: A simple optimization using replicating collection. In *Proceedings of the Third International Workshop on Object Orientation and Operating Systems*, pages 177–181, Asheville, NC (USA), 1993.

[OG97] M. Oehler and R. Glenn. HMAC-MD5 IP Authentication with Replay Prevention. Internet RFC 2085, February 1997.

[ON94] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. In *Proceedings of the Conference on Lisp and Functional Programming*. ACM, June 1994.

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, August 1996.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX Winter Technical Conference*, 1995.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference*, pages 106–119, Monterey, June 1999.

[Pes00] David Pescovitz. Monsters in a box. *Wired*, 8(12):341–347, 2000.

[PG81] J. W. Pratt and J. D. Gibbons. *Concepts of Nonparametric Theory*. Springer-Verlag, 1981.

[PHL97] John Peterson, Paul Hudak, and Gary Shu Ling. Principled dynamic code improvement. Technical Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.

[Pie93] Benjamin C. Pierce. Mutable objects. `http://www.cis.upenn.edu/~bcpierce/papers/mutable.ps`, 1993.

[PJW96] *First International Workshop on Persistence and Java*, September 1996. `http://www.sun.com/research/forest/UK.Ac.Gla.Dcs.PJW1.pj1.html`.

[Pla97] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

[QL91] Russell W. Quong and Mark A. Linton. Linking programs incrementally. *Transactions on Programing Languages and Systems*, 13(1):1–20, January 1991.

[Rep99] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloquium sur la programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.

[Rou96] François Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.

[RS97] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[Sch] Scheme 48. `http://s48.org`.

[Seg] Mark Segal. Personal communication.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 236–247, London, January 2001.

[SF93] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, pages 53–65, March 1993.

[SFPB96] Emin Gun Sirer, Marc E. Fiuczynski, Przemyslaw Pardyak, and Brian N. Bershad. Safe dynamic linking in an extensible operating system. In *First Workshop on Compiler Support for System Software*, Tucson, February 1996.

[SGGB99] Emin Gun Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, December 1999.

[Smi88] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review, SIGOPS*, 22(3):28–40, 1988.

[SNC00] Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In *Proceedings of the USENIX Annual Technical Conference*, March 2000.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[TAL99] Typed assembly language homepage, 1999. `http://www.cs.cornell.edu/talc/`.

[TISC95] Tool Interface Standards Committee. Executable and Linking Format (ELF) specification, May 1995.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Canberra 0200 ACT, Australia, 1996. `http://samba.anu.edu.au/rsync/`.

[TTA$^+$99] Ann T. Tai, Kam S. Tso, Leon Alkalai, Savio N. Chau, and William H. Sanders. On-board guarded software upgrading for space missions. In *Proceedings of the Eighteenth Digital Avionics Systems Conference*, 1999.

[TTA$^+$00] Ann T. Tai, Kam S. Tso, Leon Alkalai, Savio N. Chau, and William H. Sanders. On low-cost error containment and recovery methods for guarded software upgrading. In *Proceedings of the Twentieth International Conference on Distributed Computing Systems*, pages 548–555, April 2000.

[Web] Mindcraft—WebStone benchmark information. `http://www.mindcraft.com/webstone`.

[Wei00] Stephanie Weirich. Type-safe cast. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming Languages*. ACM, September 2000.

[WFMN92] Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles. Extensions to Standard ML to support transactions. In *1992 ACM Workshop on ML and its Applications*, pages 104–118, 1992.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH*, April 1998.

[WKG98] Chris Walton, Dilsun Kirli, and Stephen Gilmore. An abstract machine for module replacement. Technical report, Laboratory for the Foundations of Computer Science, The University of Edinburgh, June 1998.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[xSU] Telcordia software visualization and analysis toolsuite (χsuds). `http://xsuds.argreenhouse.com`.

[YdS96] Yechim Yemini and Sushil da Silva. Towards programmable networks. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, September 1996.