A Typed Assembly Language for Java

Dennis Brylow
Qualifying Exam Part II
Purdue University*

July 26, 1999

1 Introduction

The JavaTAL project is an effort to provide infrastructure for a safe, fast system for transporting object-oriented programs from an untrusted source to a protected destination. A comparable mechanism is currently used by Java developers everywhere; Java is compiled to Java byte codes, a machine-independent, stack-based language, and is then run on Java Virtual Machines that try to guarantee certain safety properties for the code consumer. A very serious drawback to the Java Virtual Machine system is that it is shockingly slow, and continues to resist optimization techniques that have been successfully applied to other compilers for decades. An even more serious drawback to the JVM system is that it isn't truly safe either; dangerous errors have been found in the JVM language, and there still is no rigorous proof of the JVM's correctness, despite years of effort.

The JavaTAL project aims to construct a new machine-independent, assembly-like language, capable of encoding common object-oriented languages while still allowing aggressive optimizations to be performed. An integral part of this JavaTAL language will be type-soundness. Any program encoded into JavaTAL can be run through a simple verifier which has been rigorously proven to accept only type-sound programs. While type-safety is only the beginning of comprehensive software safety, it is a critical foundation upon which all other mobile code safety mechanisms can depend.

This paper does not describe a completed system. It is a snapshot of a design in progress. While many components of the system have been completed in the past seven months, the overall scope of this work is such that many more months of design, implementation, and testing remain. Figure 1 is a high-level road map of the JavaTAL project. This work builds upon the advances made by the TAL group at Cornell, (see Section 3 for references,) which currently has no support for objects. Our goal is to complete the design for a Typed Assembly Language capable of supporting "bounded existential types," which are a general description of objects and classes that transcend most commonly used object models. When our design is complete, we will be able to encode not only Java programs, but most other object-oriented programming languages as well.

Of the three major milestones in the JavaTAL project, we have passed the first, and are rapidly approaching the second. We have completed our prototype compiler for translating a Java subset

*Purdue University, Dept of Computer Science, W Lafayette, IN 47907, USA, brylow@cs.purdue.edu, phone: 765–494–7843.

into a Typed Assembly Language, as well as an interpreter to test the results of this compiler. After reaching this first milestone, "JavaTAL v0," we discovered that Java's name-based type system is not strong enough to prove type-safety at an assembly language level. The second milestone will be to borrow enough structure from the bounded existential types to prove type-safety for JavaTAL version 1. This is all but done; the current design appears to have all of the necessary safety properties, but time constraints and a rapidly improving type-system design have precluded completion of a formal proof.

The last major phase of the project will be to convert completely to the bounded existential type system. This final system, which we have temporarily dubbed "ObjectiveTAL," will support all of the relevant object capabilities of Java, but leave behind many of its failings.

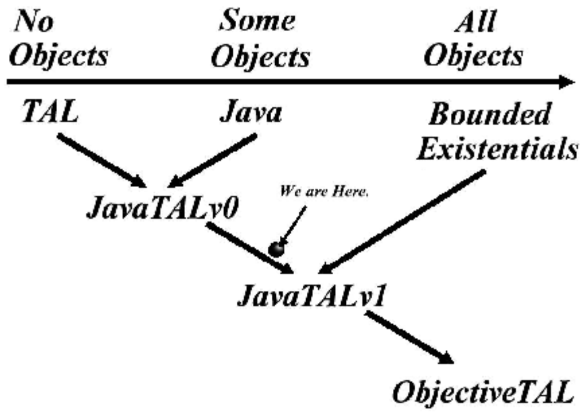

Figure 1: A Road map of the JavaTAL Project

Briefly, the major sections of this paper are as follows.

Section 2 is a broad survey of the mobile code arena. Although relatively new, the study of mobile code is already a large, multi-faceted area with many diverse open questions.

Section 3 is an in-depth examination of previous related work in the field of Typed Assembly Languages. Typed Assembly Language is a promising new idea in the mobile code arena; there are only a handful of prototype mobile Typed Assembly code systems in development, and none of these has progressed to the point of supporting object-oriented languages yet.

Section 4 is a status report on our JavaTAL system.

Appendices A, B, and C give detailed technical information about our completed compiler, and the draft design for JavaTAL version 1.

Appendix D, "Type Systems Explicated for Non-Theoreticians," is a short explanation of the type-theory notation commonly used by theorists. While the theoretical notation used in this paper is extensive, much of it can be conveniently mapped to concepts familiar to mainstream computer scientists who do not derive lambda calculus type judgments on a daily basis.

Contents

2 Mobile Code

As platform-independent languages like Java become progressively more prevalent in the computing world, questions of safety and security for mobile code become more pressing. This section examines several possible frameworks for supporting secure mobile code, particularly in Java. Proposed possibilities have ranged from redesigning or enhancing the Java Libraries or Virtual Machine to allow software interposition [WBDF97], to proof-carrying code with certifying compilers [NL96, Nec97, NL98], to Java Language changes to support information flow annotations [ML97, Mye99], to type-safe assembly language [MWCG97]. These widely disparate methods each have their advantages and disadvantages, and each aims to satisfy a particular class of safety concerns.

In order to implement a full scale, comprehensive mobile code system, the entire gamut of safety concerns must be covered adequately. This section discusses the advantages of the four techniques above, and explores the need to combine these techniques in order to address the diverse security requirements of an entire mobile code system.

2.1 What Does a Mature Mobile Code System Require?

Memory protection is a must for any mobile code system. Untrusted code cannot be allowed to run rampant on the system stack, or reference protected memory on the host system; without memory protection, any other safety properties we wish to discuss are invalid.

Any mature mobile code system will need a notion of secure services. In order for mobile code to perform any non-trivial tasks, it must be able to interact in a safe, regulated manner with the services provided by the host system.

But merely protecting memory and securing services will probably still not be enough. It would really be desirable to set down semantic limitations for code – to in some way ensure that the code "does what it is supposed to do." To be sure, semantic properties like termination are undecidable in the general case; but that does not mean that sufficiently sophisticated mechanisms could not be devised for stricter language subsets that would provide a mobile code system with some guarantees of behavior.

A large-scale mobile code system needs a mechanism for controlling the flow of information. The web of trust in a mobile-code system could rapidly become untenable without some mechanism for explicitly annotating where information is and is not permitted to flow.

And finally, a mobile-code system needs speed. So-called "Active Networks," which toss around mobile "agents" to manage traffic and services, are already starting at a performance disadvantage to traditional passive networks, because the active part of the active network requires additional bandwidth, computational power, and complexity of design. As the entire point of active networking is to tangibly improve overall network performance in certain tasks, the cost of mobile code must not outweigh the benefit.

2.2 The Need For Software Protection

In Wallach, et. al's "Extensible Security Architectures for Java," [WBDF97], the first claim the authors advance is the need for software security mechanisms instead of hardware protection. One clear overarching argument in favor of software protection is portability. Committing to a particular hardware security paradigm also constrains code to a particular architecture, or at best, a subset of the available platforms. This is inherently contrary to the hardware independence that has driven Java's success in the first place. It is even less practical in an environment where mobile code is desired, because assurances of safety must be given for each class of desired security, multiplied by each kind of hardware.

If we ignore the portability argument, (which is quite conceivable, considering the vast number of Internet users currently using the IA-32 architecture,) there is the issue of performance overhead for hardware security measures. Having instrumented a JVM, the authors estimated the time spent by normal programs crossing software "trust boundaries" to be 0.5% of overall runtime. If this percentage is multiplied by the factor of 1000 slowdown caused by an Intel x86 Task Switch, a program using hardware memory protection (instead of software protection) would spend 6-8 seconds crossing hardware trust boundaries for every second of productive computation [WBDF97, pp.116-117].

In essence, using the IA-32's built in hardware features for memory protection would be like enforcing the traditional sand box model – except that now there will be a twelve-person committee standing next to the sand box, which must debate and approve every grain of sand prior to its use. While the authors' ballpark estimate of slowdown caused by this naive scheme is plausible, it is difficult to believe that the selection of the Intel Task Gate, (quite possibly the most intricate and time-consuming micro-code sequence ever devised, [Int99]) is a representative model for the implementation of hardware memory protection for Java. However, those issues aside, there is certainly need for safety guarantees without hardware dependence.

2.3 Language Level Interposition

Most papers on software protection concentrate on memory protection, guaranteeing that software does not run amok in a shared memory space, or begin executing arbitrary, unrelated, or even supposedly protected code. Methods such as software fault isolation, proof-carrying code, and type-safe languages are all designed to enforce memory protection. Wallach and colleagues argue that while certainly necessary, memory protection alone is not sufficient to guarantee the security of a mature system. Comprehensive security requires the notion of secure services, subsystems for file I/O, GUI manipulation, network I/O, etc., that can be trusted not to compromise the entire system.

The authors outline three separate concepts for implementing secure services in software. All three methods are a form of interposition, a technique that routes calls to secure services through a trusted reference monitor of some sort. Also, all three methods assume the presence of digital signatures to identify the principals responsible for mobile code.

Capabilities, the first of the three software security mechanisms [WBDF97, p.119], are encapsulating entities that regulate access to services according to some security policy. The idea is that an untrusted code segment would be able to access only those secure services for which it was given a capability upon startup. In addition, untrusted code could request a capability from a central broker, which would dole out capabilities based upon the principal signature of the code, and some internal security policy.

The chief advantage of capabilities in Java is that they are easily implemented in the current language. The chief disadvantage is that all of the secure services openly available in the current Java runtime libraries would have to be hidden, so that the only permissible access path would be through the capability system.

Extended Stack Introspection, the second proposed software security mechanism [WBDF97, p.120], is a scheme by which calls to secure services are accepted or denied based upon a search of the call stack for certain "privileges". Untrusted code would acquire a privilege by calling an EnablePrivilege() method, which would consult a policy engine to determine if the privilege should be allowed. A DisablePrivilege() method would allow the privilege to be discarded. A CheckPrivilege() method would check the call stack for the privilege of a particular secure service.

The chief advantage of Extended Stack Introspection is that privileges are contained; unlike the capability system, one piece of untrusted code cannot pass a privilege to another piece of untrusted code from another principal. The chief disadvantage of this scheme is that a modified JVM would be required to properly handle any older code that was written prior to the adoption of the privilege system.

The final proposed software mechanism [WBDF97, p.122], Name Space Management, would interpose the Java Class-Loader to change the visibility of secure services from untrusted code. The new class-loader would use a "configuration," a mapping of old Java class names to new class names. In this way, all of the security decisions for a program are made prior to execution, at link-time.

The chief advantage of this system is that there is no runtime overhead from the security mechanism itself – only the overhead of the interposition itself. Also, third-party trusted subsystems could be distributed with untrusted applets. The chief disadvantage is that this mechanism may not be possible with newer versions of Java, as applets may be able to inspect all of the available class libraries to find the new name of the secure service they seek.

The authors include an analysis and comparison of the three methods using various criteria for performance, effectiveness, and compatibility [WBDF97, pp.123-127]. In the end, they conclude that none of the three methods are perfect – each has its respective advantages and disadvantages. Ultimately, the answer for providing secure services for Java code is probably a clever combination of the three mechanisms, but the exact nature of that merging remains an open question. In addition, none of these methods help at all with the prerequisite memory protection, and all require changes to the JVM or Java Libraries across all of the hosts in the mobile code system.

2.4 Proof-Carrying Code

In 1996, Necula and Lee proposed "Safe Kernel Extensions Without Run-Time Checking," [NL96], a technique for embedding a formal proof of correctness in mobile code, which could then be verified upon arrival at its destination. In 1997, Necula refined this mechanism called "Proof-Carrying Code," [Nec97], and showed what such a framework might look like. In 1998, Necula and Lee revealed a working, non-trivial implementation of this concept in "Design and Implementation of a Certifying Compiler," [NL98].

In a proof-carrying code system, the ultimate destination of mobile code, called the "code consumer" would publish a safety policy of some type. In this case, the safety policy would consist of three parts [NL96, p.3]: a verification condition generator, called a safety predicate, which would compute a first-order logic predicate for the code to be certified; a set of initial axioms to be used by the safety predicate; a precondition, to indicate the calling conventions under which the proof-carrying code will be invoked by the consumer. In essence, it is the function of the safety policy to describe completely what is considered "safe" behavior by the code consumer.

The code producer would construct a safety proof to accompany the mobile code to its destination. Upon arrival at the destination, the proof is checked against the safety policy of the consumer, and the code is validated.

The key benefit of this approach is that the mobile code is, in some senses, tamper-proof. While the code or the proof can be maliciously forged or altered in transit, the code consumer will only accept the code if upon arrival the proof still guarantees the safety policy, and the code still matches the proof. This alleviates the need for encryption, and other verification mechanisms to establish relationships of trust between code sources and destination. Furthermore, this system can actually detect many kind of compiler errors, and does not require the compiler to be in the trusted code base.

The authors have concentrated on fine-grained memory protection, but their scheme could be extended to far more complex notions of secure services. It is not difficult to see how a code consumer could setup facilities and security policies for handling a system of capabilities, or other semaphore-based access to protected resources [NL96, pp.5-6].

Necula and Lee have implemented their proof-carrying code system in the context of packet-filtering. Packet filters are an interesting case where other approaches to safe code are already in use. In addition to being able to compare the proof-carrying code system to other previous systems, working in this venue has the added bonus of restricting the complexity of possible programs. The packet filters are greatly limited in their memory access, and all code branches are forward. As expected, hand-coded assembly language packet filters carrying proofs were able to outperform other more costly schemes, such as the BSD packet filter architecture, Software Fault Isolation and implementations in a more restrictive, safe high-level language [NL96, p.10]. The start-up cost of verifying the proof is negligible when amortized across the speedup of assembly code with no run-time overhead of any kind.

The chief obstacle to this type of approach is the difficulties and overheads associated with the proof itself.

In the first place, Necula and Lee's experience thus far has shown that the size of the proof is usually 2 to 3 times the size of the actual code [NL96, p.8][NL98, p.342]. In a mobile code situation, a 200% to 300% overhead for transmission and buffering could be prohibitively large. Unfortunately, a proof can in theory be exponentially large in the size of the code, and the task of constructing a proof for a general program can be undecidable [NL96, p.13]. While this has not yet shown itself in the authors' testing, it could ultimately prove to be a serious flaw if such as system were deployed on a large scale.

In the second place, this kind of formal proof technique requires limitations on control flow. In the packet-filter application, all branches were limited to forward – in other words, no loops. If loops are permitted, then explicit invariants must be included, in order to precisely characterize the nature of the loop. This process is not readily automated, and may require the programmer to specify the invariant at the source level. This would be an annoyance to developers, although it should be noted that it is an excellent software engineering practice, particularly if the author wishes to guarantee or verify certain code properties.

These difficulties aside, Necula and Lee have implemented a certifying compiler for a type-safe subset of C, which is able to generate proofs, and verify them, without the tedious manual intervention characteristic of earlier theorem-proving systems [NL98].

2.5 Confinement: Sometimes Mobility is a Bad Thing

In the press for platform independence and mobile code, it is easy to forget that not everything in a mobile code system should be mobile. In addition to services that need to be secure, most systems also have information that must be secured. If the system uses capability-style secure services, what is to prevent semi-trusted code that has acquired a capability from passing it on to another entity that the code consumer would not have trusted in the first place? The problem of confining information and controlling its propagation is addressed in Myers and Liskov's "Decentralized Model for Information Flow Control," [ML97], and Myers's "JFlow" paper [Mye99].

JFlow is an extension to Java, allowing the developer to explicitly annotate permissible information flow. Such a system can be used not only to facilitate capability confinement, but also to confine all kinds of sensitive data.

The JFlow flow annotations appear in the form owner:reader appended to the types of variable declarations. This system allows multiple owners to specify by name all of the permitted readers. As a result, the effective reader set is the intersection of the readers specified by every owner. In other words, a reader can only access a variable if all of that variable's owners agree that the reader is permitted. In addition, principals can declassify data safely, thus loosening their own protections without weakening the policies of others.

Although JFlow allows dynamic checks to be used for granting authority, the normal mode of operation is to performs all checks statically, yielding no run-time overhead. This is a substantial improvement over comparable mechanisms, such as mandatory access control, and allows a finer granularity of control over information as well.

Of special interest is the decentralized nature of the JFlow model. Unlike many schemes, which require a centralized authority to dole out capabilities or their equivalents, the JFlow model allows principals to protect their private data, without having to establish relationships of trust. This is especially beneficial in a mobile code system, as any kind of central authority would require additional bandwidth, and in itself would become an additional security concern.

As Myers is quick to point out, [Mye99], there are still covert channels of information through which JFlow programs could leak information, such as inter-thread communication, and timing code. However, all of the obvious channels for circumventing secure services are covered.

2.6 The Need For Speed

Executing software in Java remains slow – shockingly slow. While many attempts have been made to speed up the process of verifying and executing Java byte codes, the only solid advances have come from returning to machine-level code, as in Just-In-Time (JIT) Compilation.

With this in mind, it is interesting to consider the recent advances in Typed Assembly Language, such as Morrisett's TAL [MWCG97]. The TAL language is a RISC-like assembly language, with annotations at basic block and allocation points that allow the code to be proven type-safe. In this way, typed assembly language is a particular kind of proof-carrying code, with the overhead of the proof being dramatically reduced.

The chief advantage of typed assembly language is speed. A program in typed assembly language can be aggressively optimized by the code producer well before it is sent to the code consumer. Upon arrival, the code requires only verification, and compilation down to machine code. The verification process can be proven to be correct, and guarantees type-safety and memory safety.

More importantly only the verifier and final translator need to be trusted by the code consumer, substantially reducing the size of the trusted code base over a purely Java-based system.

Typed assembly language can be targeted to particular architectures, thereby reducing the tremendous run-time overhead of interpreting Java byte code through a Java Virtual Machine. Unlike JIT compilers, typed assembly language need not start out fighting the stack-based target architecture of the JVM, which is not especially well emulated in today's largely register-based architectures.

An important step in Morrisett's compilation down to typed assembly language is Continuation-Passing Style (CPS) conversion. CPS conversion has the property of eliminating all procedure calls, replacing them with branch statements. The benefit of this can be enormous in modern architectures, in which an actual procedure call can be quite expensive in terms of execution cycles and cache management. The end result is again faster execution over call-based code, although some increase in code size can be expected for complex programs.

Unfortunately, typed assembly language by itself lacks many of the desirable properties of the approaches outlined earlier. Typed assembly does nothing to promote secure services, and does not even address matters of confinement of sensitive data. In fact, typed assembly really offers no guarantees of any kind as to the actual semantics of a program, merely that it is type-safe, memory safe, and will not "get stuck".

Even in the realm of type-safety, additional work remains to be done in order to support type-safe object-oriented programming. While Morrisett's TAL supports all of the standard types, polymorphic types, and existential types, it still lacks facilities for key object-oriented features like dynamic dispatches and inheritance.

2.7 The Best of All Worlds

It would seem that no mechanism proposed thus far is capable of providing more than a fraction of the desired properties for a large-scale, mature mobile code system. In order to achieve all of the security properties a mobile code system should possess, it seems natural that components of all of these methods must be employed.

Consider a typed-assembly language like TAL which was capable of completely implementing the Java Language. If one was willing to sacrifice guaranteed platform independence, a JavaTAL would be a laudable speed increase over Java byte code, while still offering assurances of type-safety and memory safety. Even if platform independence were still a key issue, any suitably RISC-like JavaTAL would be readily mapped onto actual instructions for most modern architectures, still leaving only the verifier and final translators as the trusted code base.

JavaTAL alone would not be sufficient to enforce any semantic policies. However, the addition of proof-carrying code components on top of it could serve such a purpose. Relieved from the burdens of type-safety and memory safety, the proof-carrying code segment of our hypothetical design would need only be concerned with purely semantic aspects of a mobile code segment – termination, or computational correctness. The addition of this PCC (Proof-Carrying Code) layer would increase the size of the mobile code segment, but would again incur a time penalty only at the initial analysis time, not during subsequent run-time.

Unfortunately, a proof of correctness is still not enough to guarantee secure services, without some rather awkward restrictions on the entire service structure of the mobile-code network. Java capabilities, however, remain an excellent option for implementing secure services. In addition to having very well understood security properties, introducing the capability model at the Java source level allows programmers to design service-level security in the most natural way possible. Since capabilities in Java would be nothing more than normal objects and classes, they would still translate without difficulty to the JavaTAL level. Capabilities introduce actual run-time overhead to our JavaTAL programs, but no more so than any other secure service mechanism that does not rely on changes to the Java Virtual Machine.

In order to address the confinement issues with Java capabilities, we could introduce the JFlow extensions to the language. JFlow would allow a developer to explicitly control dissemination of capability objects, while at the same time substantially tightening the information security of the system as a whole. In addition, the JFlow verifier could be modified to enforce the capability system, without requiring that all of the Java run-time Libraries be altered to hide the non-secure Java services encapsulated by the capabilities.

JFlow flow information is annotated to variables like types. Also like types, this flow information could be propagated all the way down to the JavaTAL level, where it could still be statically checked at the same time as the typed assembly language.

The end result is a multi-layered system addressing a wide array of mobile code concerns, with a minimum of run-time overhead, and a minimum of trusted code-base. On top of that, most of the security design mechanisms are at the Java source level, where developers are most comfortable with coding security features, and the vast majority of computational overhead takes place either at compile time, (optimization and proof construction,) or at startup time, (one-time initial checks of the proofs.)

2.8 What Remains to be Done?

As idyllic as the above proposal sounds, there are still many open questions buried within it.

As mentioned earlier, at present there is no typed assembly language that implements all of Java, or even most of it. Whether the verifier for such a language would be a “smaller trusted code base” than an entire Java Virtual Machine remains to be proven. The assertion that a RISC-like assembly language could be verified and mapped to machine code faster than a JVM with JIT compilation remains to be proven.

First-order logic proof-carrying code may not become notably smaller when relieved of the burdens of type-safety and memory safety. Furthermore, it has not been shown that proof-carrying code can prove anything useful about an object-oriented language without restricting the language to a trivial subset. The Halting Problem remains undecidable as ever, and it seems unlikely that a proof-carrying code system will ever be able to make sweeping guarantees about program termination or computational correctness without strict limitations on the expressiveness of the source language.

While the JFlow system is a marvelous vehicle for limiting the flow of secure information, its author is quick to point out that many clever covert information channels remain uncovered. In addition, although the TAL group has shown it is possible to retain type information through all of the compilation stages, it does not necessarily follow that JFlow’s type-like annotations can also be correctly pushed through to the assembly language level.

Overarching all of these open questions about particulars is the bottom line. In the end, would all of this be worthwhile? Could the system described above really beat the Java Virtual Machine in both functionality and reliability? How would its speed and bandwidth compare to a mobile code system based on the JVM?

Finally, one wonders if at the end of the day support for object-oriented, mobile code design can really supplant "unsafe", unstructured, hardware-level "bit-fiddling" in a real arena such as an Active Network. It would seem that the only way to find out is to build both and compare.

2.9 The Key Mobile Code Question

A wide variety of security issues are being discussed in the area of mobile code. As shown in Figure 2, most of the proposed mechanisms for facilitating mobile code address particular safety concerns, while leaving others completely untouched.

Mechanism	Memory Safe	Type Safe	Semantic Safe	Secure Services	Data Confinement
Capabilities				Yes	
PCC	Yes	Yes	Yes		
JFlow					Yes
TAL	Yes	Yes			

Figure 2: Mobile Code mechanisms vs. desired properties

It seems clear that a comprehensive, large-scale mobile code system will require components from many of these different mechanisms. A key open question in this area is to what extent can these mechanisms be combined to provide, fast, powerful, and safe mobile code?

This section has presented an overview of several proposed code safety mechanisms which, in combination, possess a much wider spectrum of desirable properties than any mechanism alone. Fortunately, as orthogonal as the many security concerns are, so are the solutions. It would seem that there is hope to successfully combine many of the current branches of disparate mobile code research into a single, powerful system with the capabilities and assurances desired by developers everywhere.

Toward that end, the next section will examine, in depth, the properties of current Typed Assembly Language research. Of particular interest is the extent to which Typed Assembly Language can already provide safe mobile code, and in what areas it still requires improvement to be considered fast and powerful.

3 Typed Assembly Language

In 1997, Morrisett et al. presented a translation from System F (a variant of the polymorphic lambda calculus,) into Typed Assembly Language[MWCG97]. Using this is a foundation, Morrisett and his colleagues have produced an entire line of Typed Assembly Language research, tackling issues like type-safe linking[GM99], type-safe memory management[CWM99] and type-safe stack management[MCGW98] for a substantial subset of the Intel x86 instruction set. Morrisett's group has also constructed a working prototype compiler for a safe subset of the C language as proof that their concept is a practical, realizable system for producing mobile code[MCG$^+$99]. This section of the paper will explore in depth the advances that have been made in the TAL arena, as well as what advances remain to be made.

Figure 3 summarizes the strengths of each of the flavors of TAL. TALx86 is the culmination of this Typed Assembly Language line of research, lacking only the explicit deallocation feature of the Capability Calculus to be the final word in functional Typed Assembly Language. However, none of the current flavors of TAL support encoding of objects, or dynamic dispatch; as we will see in the third section of this paper, this requires more than simply adding another type to TAL.

Flavor	Memory Safety	Type Safety	Stack Manipulation	Separate Compilation	Heap Deallocation	Support for Objects
TAL (vanilla)	Yes	Yes				
STAL	Yes	Yes	Yes			
MTAL	Yes	Yes		Yes		
Capability Calc	Yes	Yes			Yes	
TALx86	Yes	Yes	Yes	Yes		

Figure 3: Flavors of TAL vs. desired properties

3.1 Wherefore Typed Assembly?

There are a variety of reasons to study Typed Assembly Language even without mobile code concerns. From a compiler standpoint, many compilers require type information during intermediate passes in order to perform sophisticated optimizations. Maintaining type information throughout the compilation process can help verify the correctness of the individual transformations; many compiler bugs can be caught by type-checking each intermediate form. Furthermore, the restrictions brought about by the type system do not appear to interfere with most of the low-level optimizations used by the cleverest of compilers[MWCG97].

Secondly, as mentioned in the previous section, compiling to Typed Assembly removes the compiler from the trusted code base. One does not have to rely upon the correctness of the compiler, its optimizations, or any kind of authentication scheme in order to verify that the resultant code is type-sound. Only the type-checker itself must be trusted, as well a comparatively trivial final pass to map the assembly language down to machine code. This also means that the assembly code can be safely hand-tuned, or run through any number of additional low-level optimizations, with the confidence that the type-checker will only verify type-safe code.

3.2 TAL

Morrisett's original TAL paper outlines five translations between six typed calculi as a process for getting from System F to TAL. These translation steps will be summarized below, as a prelude to presenting our own adaption, called "JavaTAL", in the next section of this paper.

The initial typed calculus, called "Lambda F", is a call-by-value variant of the "System F" polymorphic lambda calculus, and is sufficiently expressive to model most functional languages. It contains types for integers, functions, and tuples (like C structs, or Pascal records,) as well as polymorphic types. (The need for these will become apparent later.)

The desired result is that each basic block of assembly code, and each memory allocation point, will be annotated with typing information. This typing information will allow a simple, provably correct type checker to examine the annotated code, and quickly accept or reject the program as type sound. For example, a typical basic block label might be annotated with the following type information:

$l\_swap : code\{r1 : int, r2 :< int, int >\}$

which would indicate that the opcodes below this label, which comprise function "swap," expect register r1 to contain an integer, and register r2 to contain a record with two integer fields. Of course, a real TAL label will have more complicated syntax, because many functions require more elaborate type information than simple combinations of integers, as we shall soon see.

3.3 Compiling to TAL

The first step in compilation is a type-preserving Continuation-Passing Style (CPS) conversion. CPS conversion is a well-understood transformation for replacing all non-void functions with e-quivalent void counterparts. The end effect of this process is that the Typed Assembly Language will contain no "call" or "return" instructions; all control transfers will be handled by unconditional jumps or conditional branches. While this method incurs additional runtime overhead from managing the continuation structures, it removes much of the overhead caused by procedure calls in modern architectures.

Next, the intermediate representation is "closure converted", which means that all functions referencing free variables outside of their scope are rewritten to accept an environment parameter containing those free variables. In this way, each function becomes closed, which has the effect of breaking each function body into its own "basic block", separated in variable scope from each other basic block. This transformation is, again, provably type-sound.

It is at this step in the conversion that existential types first appear, in order to abstract the environment parameters of the closures. Consider a function, "f", which after CPS conversion is a function which takes a single integer argument and has a void return type. (Of course, in reality, after CPS conversion the function should also have a second parameter that is a continuation function, but this is omitted for the clarity of this example.) Function f has type:

$f : \forall[].(int) \mapsto void$

where the $\forall[]$ indicates polymorphic abstraction, which will be explained later.

After closure conversion, the environment will be bundled in with the function, and its type abstracted:

$$f : \exists\beta. < \forall[].(\beta, int) \mapsto void, \beta >$$

This can be read as, "There exists an environment type β such that function f has the type of a two-element tuple, where the first element is a void-type function taking two parameters, (of type β and int, respectively), and the second element is an environment of type β." The fact that β is hidden behind the existential, rather than being spelled out explicitly, acts to prevent the function in this closure from being called with anything but the correct environment bundled along with it in the closure. When the existential is "unpacked" later, β becomes instantiated to a new, fresh type, so only the correct environment will have the proper type to be passed as the first parameter to the function. In this way, the type system guarantees the integrity of the closures, by mandating that the newly transformed function bodies are only called with the same environment they would have had prior to the transformation.

Immediately following closure conversion, all of the basic blocks are "hoisted" to the top level of the code, thereby eliminating all nested blocks. This transformation has no bearing on the types of the program, as it is simply re-ordering independent basic blocks.

Once this is done, all of the tuple declaration points in the intermediate representation are expanded into explicit allocation points. Thus, for each struct or record in the program, instructions are generated for allocating and initializing the appropriate amount of space in a type-sound fashion.

Finally, the last intermediate representation is translated down to TAL itself, which is not especially difficult because the code looks pretty much like assembly language at this point anyway. Throughout each of the five transformations, type information is preserved or correctly transformed for each variable and function. The TAL program can be checked to ensure that each arithmetic operation is applied only to numeric operands, that each jump instruction targets executable code with the correct number and type of arguments, etc.

TAL is a small, RISC-like instruction set, containing opcodes for load, store, move, conditional branch, and unconditional jump. In addition, there is a macro instruction for memory allocation, "malloc", which would be expanded into a simple move/addition instruction pair during the final translation from TAL to machine code. Also, there is an "unpack" instruction, which acts to unpack the existentially typed closures generated earlier in the closure conversion phase. In the final translation to machine code, the unpack instructions generate no opcodes – they can be thought of as directives to the type-checker, with no real computational effect or instruction cycles.

During type-checking, each line of TAL is checked for type-soundness. The line

$$mul \quad r1, r2, r3$$

passes the type-check only if it can be statically determined that registers r2 and r3 do indeed both contain data of the proper type to be multiplied together. Register r1 will then have the appropriate result type in subsequent checks of lines farther down the basic block. A store operation,

$$st \quad r1[3], r2$$

type-checks only if register r1 contains a tuple type with at least four entries, and the fourth entry of that type corresponds to the type of the source register, r2.

Each basic block of TAL code has a tag indicating the types required in the registers in order for a jump to that basic block to be type-sound. For example:

$l\_fact: \quad code[]\{r1 :<>, r2 : int, r3 : \exists\beta. < \forall[].(\beta, int) \mapsto void, \beta >\}$

demarcates a basic block with name, "l_fact", which expects register r1 to be empty, (or rather, specifies that anything in r1 is thrown away,) r2 to contain an integer, and r3 to contain a closure of the type discussed earlier. The type checker simply needs to statically verify that the type environment at each jump point suits the type constraints at the target label.

Morrisett et al. go on to prove "Subject Reduction", (that a well-typed program remains well-typed after any step of execution,) and "Progress" lemmas (that a well-typed program can continue taking valid execution steps until it completes execution,) based on the structure of the type derivations and possible cases of the actual instructions. Combined, these lemmas prove that a TAL program that passes the type-check cannot get stuck, and will perform only type-sound operations during its execution.[MWCG97] (Of course, this makes no guarantees about other semantic properties, like algorithmic correctness, or termination, but it's a major first step.)

3.4 STAL: Stack-based TAL

The original TAL deals in heap-allocated data, and heap-allocated activation records, but has no provisions for explicit, type-sound stack operations. This is a substantial shortcoming, in light of the fact that the vast majority of modern compilers rely on stack allocation for all but dynamically allocated data structures. Furthermore, the CPS conversion required for the heap-based translation is a complex transformation that can backfire for certain classes of circumstances. (This is discussed in greater detail in the final section on design decisions for JavaTAL.) Clearly, the utility of a Typed Assembly Language is increased if it supports either heap-based or stack-based allocation, so Morrisett and colleagues proposed "STAL", a stack-based extension to TAL[MCGW98].

First, the register abstraction is extended to include a "sp" Stack Pointer register. A new class of types is devised to support typing of the Stack Pointer. Not surprisingly, these are called stack types.

If a code fragment were to allocate three locations on an empty stack, and store integers in the first and last, the Stack Pointer would be typed as follows:

$sp: \quad int :: ns :: int :: nil$

The type "nil" indicates the end of the stack; the type "ns" or "nonsense" indicates uninitialized (and therefore untyped) stack locations.

Because the type rules for TAL do not cope with stack types in the load and store operations, new pseudo-opcodes are added to STAL – salloc, sfree, sld, and sst – to allocate, free, load and store things on the stack. In the final translation to machine code, these instructions would be mapped to the expected opcodes, with the stack pointer register used as the appropriate source or destination. Common stack operations, like push and pop, can be easily typed using combinations of salloc/sst and sld/sfree.

The obvious thing to do now would be to append the following type to a basic block:

$label : \forall[].\{r1 : int, sp : \sigma, ra : \forall[].\{sp : \sigma\}\}$
where σ would be a stack type.

This type annotation appears to specify a function that takes a single integer as parameter in r1, a particular stack type "σ" in sp, and a return address in register ra that mandates the stack be returned to its initial state when the function returns. However, this has several critical flaws.

First of all, once the stack type σ is actually spelled out, there is no mechanism to prevent this function body from popping items off of the stack, and storing whatever it likes in higher stack frames, as long as it uses the correct types. More importantly, once σ is finalized, this function can only be called from jump points that have identical stack shapes. In fact, this function cannot even call itself recursively, because it cannot push its own return address on the stack before jumping to this label.

Finally, a clear example where parametric polymorphism can save the day comes into view. If the label is given the polymorphic type,

$$\mathit{label} : \forall[\beta].\{r1 : \mathit{int}, sp : \beta, ra : \forall[].\{sp : \beta\}\}$$

then the type system essentially says that, "for any stack type β, this function returns with stack type β." Because β is abstracted in the function body, this basic block can make no type-safe alterations to higher stack frames, because it cannot create values with the proper type. Furthermore, β can be instantiated to any stack type prior to jumping to this label, so this function can be called with any kind of stack, because it promises not to touch anything that it did not push on itself.

Support for exceptions in original TAL is not difficult, because all that needs to be done to CPS converted code is to add an exception continuation parameter to each function. Unfortunately, simply adding an exception return address to an STAL function is not sufficient, because the two possible return addresses could specify completely different, conflicting stacks. However, the addition of compound stacks can overcome this deficiency.

The key observation is that when an exception occurs, proper behavior is to pop superfluous data off the stack, and jump to an exception routine. The ramification of this is that the stack type for the exception block is necessarily a suffix of the normal return address's stack type. Thus, we can express the type of the normal return stack as the compound stack type, "$\sigma_1 \circ \sigma_2$", where σ_1 is stack items to be disposed of if there is an exception, and σ_2 is the stack items in common between the two possible return addresses.

Of course, for compound stacks types to actually be useful, there must be a type-safe way to pop off all the σ_1 items, even though σ_2 could be buried an arbitrary and statically unknowable number of layers deep in the stack. The solution is to introduce pointers into the stack, and permit registers to store stack locations. At face value, this seems like a dangerous scheme, but it can be proven that stack pointers are not out of date as long as they are a tail of the current stack whenever they are used.

The final, correct typing scheme for a stack-based function call with an exception handler is:

$$\begin{aligned}\mathit{label} : \forall[\sigma_1, \sigma_2].\{&r1 : \mathit{int}, sp : \sigma_1 \circ \sigma_2, ra : \forall[].\{sp : \sigma_1 \circ \sigma_2\},\\ &ep : \mathit{ptr}(\sigma_2), re : \forall[].\{sp : \sigma_2\}\}\end{aligned}$$

where register "ra" contains the return address, (which expects the same stack as the function entry point,) "ep" contains the pointer to the portion of the stack the exception handler expects, and "re" contains the return address to use if an exception occurs.

3.5 MTAL: Modular TAL and Type-Safe Linking

One of the chief drawbacks of the Typed Assembly Language family that has been presented so far is that all type checking is done under the closed-world assumption. All of the code must be available to the checker, and any changes require all of the code to be re-examined. However, in the normal world of compiling, it is desirable to break large programs into separate modules, both as a matter of software engineering principle and the simple convenience of separate compilation after updates.

Cardelli proposed a high-level calculus for compilation units in 1997[Car97]. Findler and Flatt [FF98] described how the concept of "interfaces" could be extended to link-level modules in the context of their MzScheme units. Glew and Morrisett then formalized and codified these concepts in the context of TAL in "Type-Safe Linking and Modular Assembly Language"[GM99].

The new typed module language, called MTAL, (pronounced, "metal",) allows an interface to be calculated for each TAL module file. These interfaces specify exactly the set of imports and exports the module claims. Intuitively, two modules can be safely linked together if their import interfaces agree on all overlapped items, and their export sets are disjoint. A new module is formed, whose import set is the union of the submodule imports, minus the submodule exports, and whose exports are the union of the submodule exports. A complete program is formed when there are no more outstanding imports.

The MTAL calculus allows this to be done in a provably safe fashion.

On top of that, the MTAL calculus offers several handy features to augment it's modules. Using abstraction in the interfaces as in previous flavors of TAL, it is possible to encapsulate implementations in their modules. Other modules are constrained by the type system to use such imported types only in the fashion proscribed by the interface.

Glew and Morrisett point out that the MTAL system not only is applicable to dynamic linking, but probably can be extended to dynamic loading. However, dynamic loading has many additional caveats, centered mostly around possible failure modes should an attempted dynamic load fail. Their work in this area has not been completed, largely because it is difficult to evaluate the many design decisions involved in a prototype implementation.

3.6 Typed Memory Management

The next shortcoming of vanilla TAL is in the area of memory management. While TAL programs are free to allocate space on the heap, there is no mechanism for safely freeing memory. The incarnations of TAL presented thus far depend upon a trusted conservative garbage collector to clean up after them. Clearly, there is a need for type-safe memory management if fast, efficient code is to be run under this system.

The solution is a Capability Calculus, presented in "Typed Memory Management in a Calculus of Capabilities."[CWM99]

The basic idea is to have the type-safe code allocate dynamic space in chunks called, "regions." Individual basic blocks are assigned static "capabilities," which govern their access to specific memory regions. A capability is thought of as an unforgeable pointer to a memory region, but in reality these capabilities are no longer present at runtime.

It is not difficult to envision a naive scheme in which memory is allocated and deallocated in a LIFO order, with stacks of capabilities acting not unlike stack-based activation records. However, such a system lacks the flexibility to support real programs, because software frequently does not deallocate heap data in a LIFO fashion. Furthermore, it is desirable to place matching region allocation and deallocation points in separate basic blocks, as part of the normal course of compiler optimizations.

The fundamental information that a typed memory management system must track can be summed up in the question, "when is it safe to free an object?" The answer is that it is safe when there are no other references to the given object. Thus, Crary and colleagues employ capabilities to track aliasing, that is, how many references there are to a given object in memory. In this way, responsibility for deallocation can be formally assigned to particular basic blocks; many common memory leaks, caused by failure to voluntarily deallocate, can be statically detected before runtime.

If a code segment has a capability corresponding to a region, "r", the capability appears as $\{r\}$ in the analysis. This capability is annotated as either "r^1", indicating a unique capability, or "r^+" to indicate a capability for which duplicates may exist. If a block has the capability $\{r^1\}$, then it may deallocate that region, and discard the capability.

The difficulty, of course, is in recovering a unique capability r^1 after a block has already duplicated a reference and upgraded to a r^+ capability. Once again, the tricky typing problem is solved through abstraction, this time using bounded quantification. There is a fair amount of additional syntax necessary to append static capabilities to basic blocks (which will not be covered here for the sake of brevity,) but the basic idea is as follows. For a block with type:

$$\forall[\rho_1 : Rgn, \rho_2 : Rgn, c < \{\rho_1, \rho_2\}].(c, ...(c, ...) \mapsto 0) \mapsto 0$$

a calling block holding capability r^1 could instantiate "c" with r^1, and instantiate ρ_1 and ρ_2 with r^+. This would have the effect of allowing the outer nested function to have aliases to the region "r", while ensuring that the continuation function would regain the unique capability r^1. The bounded quantification abstraction allows the calling routine to hide certain information from the callee about its capability – i.e. that there is only one copy of "r" which can be deallocated – while still revealing other information – such as a duplicate of region "r" being available to the callee.

Crary et al. prove that their capability calculus is type-sound, and claim that the system is sufficiently flexible to type voluntary memory management in real programs. In addition to guaranteeing that programs return all dynamic memory to the system before exiting, this capability calculus can be applied to other interesting problems.

One common operating systems problem is caused by the passing of buffers in and out of kernel memory space. Many kernels demand time-consuming copy operations to move data between protection layers, in order to guarantee the integrity of kernel memory. With modifications, this system could be used to pass unduplicatable capabilities into the kernel, essentially passing memory space into the kernel with a statically verifiable guarantee that the user process cannot touch the memory space until the kernel returns the capability. In fact, all kinds of statically checkable locking mechanisms could be devised with such a system.

3.7 TALx86: TAL for the common man

The 1999 paper "Talx86: A Realistic Typed Assembly Language"[MCG+99], presented a realistic subset of the Intel x86 instruction set which makes use of most of the properties discussed in this section so far.

TALx86 supports separate compilation of modules, stack allocation, parametric polymorphism, arrays through singleton types, and abstract data structures like linked lists through the use of sum types and recursive types. In short, it is powerful enough to completely support a powerful, type-safe C-like language, which the authors have dubbed “Popcorn”. (Popcorn has most of the features of C, except for the “dangerous” pointer arithmetic and type casting features, plus some additional features like exceptions.)

At this point, there are only a few salient features missing from TALx86. Alias based optimizations remain difficult to type, as they require substantially more complicated flow analyses or other mechanisms to guarantee semantic properties of the code. The explicit deallocation mechanisms described in the previous subsection have not been incorporated into TALx86’s type system, because they add quite a bit of complexity. Finally, support for true object abstractions has not been added.

It seems difficult to encode the nature of object-oriented programs in a low-level language like TALx86 without either committing to a very narrow, particular object model, or inducing gigantic runtime overheads. However, with the addition of bounded existential types, and a few other mechanisms to support object inheritance hierarchies, we believe a practical solution can be engineered. It is toward this goal, efficiently encoding general support for object oriented programs in a typed assembly language, that our own work has been directed.

4 JavaTAL

This section will present a Typed Assembly Language designed to demonstrate support for object oriented code. As our chosen source language happens to be Java, we have dubbed this variant of TAL, "JavaTAL." (OOTAL seemed a most unsatisfactory moniker.)

In many ways, JavaTAL's features remain very similar to the original RISC-like TAL[MWCG97], diverging only where necessary to provide object support, or simplify the presentation. Appendix A presents the overall compilation process from a subset of Java to JavaTAL, with special attention to the salient object-related design issues. The next few subsections detail the actual syntax of JavaTAL, with explanations for key design decisions.

Not surprisingly, it is the object-oriented enhancements to the TAL type-system that have proven most problematic. After exploring several type representations, we concluded that many of the problems inherent in encoding objects into Typed Assembly Language have been documented by theoreticians encoding objects into the Lambda Calculus. Unfortunately, some of these problems have not been solved in the Lambda Calculus yet, either. As a result, we have only a design for a Typed Assembly Language at this point – not an actual proof of type-soundness. Furthermore, there are significant obstacles to generalizing our design for object-oriented languages other than Java. However, the lack of known solutions to these problems is more a reflection of the newness of the area than an indication of unsolvability. We are confident that a system with the desired properties can be designed, and optimistic that the latest proof techniques will be able to prove its correctness.

In the latter half of this section, we present our new type systems, reasons why object-oriented programs are so very difficult to prove type-sound, and vectors along which we believe the solutions may lie.

4.1 JavaTAL version 0

Instruction	Meaning
`ld` r_d, $r_s[i]$	Load into r_d the i^{th} entry of the array starting at address r_s.
`st` $r_d[i]$, r_s	Store the contents of r_s into the i^{th} entry of the array starting at address r_d.
`mov` r_d, v	Move value into r_d.
`add` r_d, r_s, v	Add v and r_s, store result in r_d.
`sub` r_d, r_s, v	Subtract v from r_s, store result in r_d.
`cmp` r_d, r_s, v	Functionally identical to `sub`. Used to distinguish between numerical subtraction and pointer comparison.
`malloc` r_d, τ	Allocate an object of type τ on the heap, and set r_d to point to it. This is a macro that will be expanded into simple instructions during the final mapping from JavaTAL to machine code.
`jmp` v	Continue execution with the instruction at the address specified by v.
`bnz` r, v	If register r contains 0, branch control to address at v.
`halt`	End execution.
`print` r	Print contents of register r. Used for debugging.

Figure 4: JavaTALv0 instruction set

Once the major steps of compilation are complete, we are ready to map the last MiniJava intermediate representation to JavaTAL. Appendix A describes all of the standard compilation steps necessary to transform the initial MiniJava programs into simple, assembly-like MiniJava statements. Figure 4 summarizes the JavaTAL assembly language instruction set.

The JavaTAL instructions are for a generic load/store register machine, and are not meant to represent a particular processor or architecture. These instructions are relatively easy to map to most modern processors, which means that the final translation software will be substantially easier to verify than a complex compiler or proof-generator.

4.2 JavaTALv0 Compiler Design Decisions

In order to actually build objects at runtime, we have chosen a standard two-level vector approach, as illustrated in Figure 5.

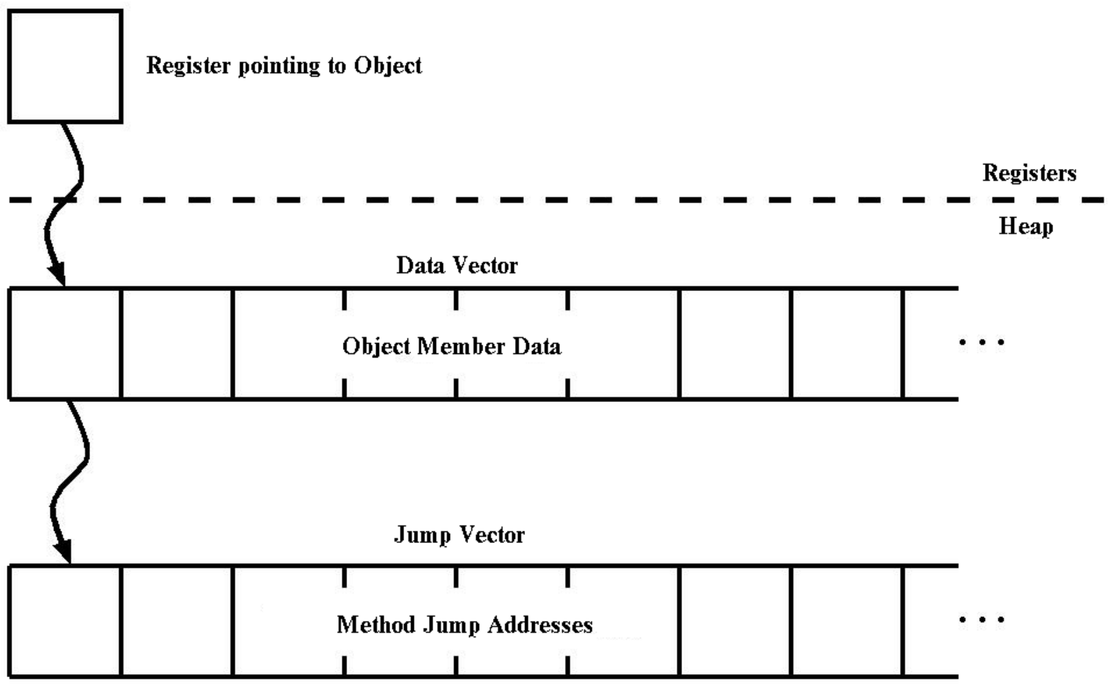

Figure 5: Runtime object layout

In this design, a pointer to an object is in fact a pointer to an array. The elements of the array are the member variables of the object, as defined by the object's class. Under MiniJava, these elements can be base types, which each fit in one array element, or other classes, which are simply pointers to other arrays. The very first element of this data vector is reserved, and points to a second vector often called the "jump vector", or the "virtual table". The jump vector contains the addresses of the basic blocks corresponding to the object's methods. When a dynamic dispatch is executed, the normal protocol is to load the first element of the data vector into a register. Then, the correct jump address is loaded from the jump vector, and the jump takes place.

The two-level vector approach has several advantages in the MiniJava context. At runtime, all objects of a given class can point to the same jump vector, which saves heap space by not duplicating jump vectors. (Note that this is not required; a type system could allow objects of the same class to have different jump vectors, or even to change jump vectors while running. This behavior, called "mode-switching"[AC96], is not allowed in Java, but could be allowed in more flexible object-oriented languages.)

```
interface l_I1
     <
       <
       ( r0:l_I1 r1:int r2:l_Result r3:l_ContInterface ) ; m
       >
     >
class l_Ex1 implements l_I1
     <
       <
         (                       ; Method m
           r0:l_l_Ex1 r1:int r2:l_Result r3:l_ContInterface
         )
       >
       l_Result          ; rstack
       l_Result          ; tempstack
       int               ; x
       l_C1              ; v5
       int               ; v7
       int               ; v6
       int               ; v4
       int               ; v8
       int               ; v3
       l_ContFor_Ex1_m_2 ; Cont_K_Ex1_m_2
     >
```

Figure 6: Ex1 class hierarchy information.

The jump vectors for a MiniJava program need not be constructed at runtime, as they are well understood by compile time, and entirely known by link time. As a result, the actual jump vectors are assembled and explicitly listed at in our JavaTAL programs. At the verification stage, the jump vectors can be checked for correctness – that they have the correct number of jump addresses, and each jump address points to a basic block with the type promised by the class or interface definition.

At runtime, the jump vectors are located on the heap after the basic blocks of executable code, where they are referenced when objects are created and initialized.

While we employ a two-level vector design for this current incarnation of MiniJava, this scheme could be expanded to an n-level vector design if MiniJava were expanded to include inheritance from classes. One can envision a system of inheritance where jump vectors contain pointers to other jump vectors containing the methods of their parent classes. For the time being, two levels and single interface inheritance are sufficient to show the viability of the design.

```
interface l_ContInterface
     <
       <
       ( r0:l_ContInterface )           ; apply
       >
     >
class l_ContFor_Ex1_m_2 implements l_ContInterface
     <
       <
         (                              ; Method setCont
           r0:l_l_ContFor_Ex1_m_2 r1:l_Ex1 r2:l_C1 r3:int
            r4:l_Result r5:int r6:l_Result r7:l_ContInterface
         )
         (                              ; Method apply
           r0:l_l_ContFor_Ex1_m_2
         )
       >
       l_Ex1                            ; that
       int                              ; a
       l_Result                         ; resstack
       l_ContInterface                  ; Continuation_K
       l_Result                         ; rstack
     >
                                        ; End hierarchy information
                                        ; Begin jump vectors
```

Figure 7: Ex1 continuation class hierarchy information.

Because the typing information in the program will reference classes and interfaces by name, it is helpful to verify the integrity of the object hierarchy before examining the program. For this reason, we include a layout of the class hierarchy at the top of each JavaTAL program. Interfaces and classes are enumerated, as well as the methods and members for each class. The verifier will check that all referenced classes and interfaces are properly defined, and that all of the subtyping and inheritance relationships are legal.

The code example given in the appendix of the paper would generate the class hierarchy infor-

```
l_Ex1 : ; Jump vector for class Ex1.
     <
       l_Ex1__m
     >
l_ContFor_Ex1_m_2 : ; Jump vector for class ContinuationFor_Ex1_m_2.
     <
       l_ContFor_Ex1_m_2__apply
       l_ContFor_Ex1_m_2__setCont
     >
```

Figure 8: Jump vectors for Ex1.

mation shown in Figures 6 and 7, and the jump vectors shown in Figure 8.

Each entry of class hierarchy information describes the exact layout of the corresponding run-time objects in memory. Each entry in the jump vector information lists the exact labels of the corresponding method's basic block. At link time, the labels in the jump vectors will be replaced with the actual addresses. Any item prepended by "l_" is an actual label that can be looked up elsewhere in the JavaTAL file.

4.3 Typing the Leap of Faith problem: a Solid False Start

With the JavaTAL language and compiler completed, all that remained was to devise the JavaTAL type system and type-checker. Unfortunately, this proved to be a more challenging task than it had first appeared. This subsection describes the chief difficulty in typing JavaTAL, and details two possible type systems for JavaTAL. While both systems would result in type-soundness for JavaTAL, we have discarded both for either imposing unacceptable restrictions on the expressiveness of JavaTAL, or being too cumbersome to result in an elegant proof of type-soundness.

The central problem in typing JavaTAL is the dynamic dispatch mechanism. When the target class of a method call is known to the caller only by its interface, we know intuitively that the method call is still OK. In order for any class to legally implement an interface, it must have all of the proper methods specified by the interface. Unfortunately, it is not possible to encode the "big picture" into a type system. The JavaTAL type-checker will examine the JavaTAL programs line by line, and must be certain it has the correct class when it reaches the jump point. We call this problem the "Leap of Faith", because without a type system, we simply take it on faith that the jump address is correct.

Figure 9 shows a standard dynamic dispatch sequence in JavaTAL. Typing information has been included in the comments. This code segment does not exhibit the Leap of Faith problem, because the proper class type of the message receiver is known. When the type-checker reaches the "jmp" line, it looks up the "code" clause of the destination, and sees that the required type for the "self" pointer is in r0.

Figure 10 shows the Leap of Faith problem. The class of the receiver is known only by an interface type. As a result, when the type-checker arrives at the final "jmp" instruction, it cannot

```
; Prepare for jump sequence to label1()
        ; Initial Types = [ r0:l_ClassC  ... ]

ld      r1,     r0[0]       ; Load address of Virtual Table into temp.
        ; Types = [ r0:l_ClassC
        ;           r1:Virtual(l_ClassC)  ...]

ld      r2,     r1[1]       ; Load address of Jump Vector into temp.
        ; Types = [ r0:l_ClassC
        ;           r1:Virtual(l_ClassC)
        ;           r2:Label(l_ClassC__label1) ]

jmp     r2                  ; Make the Jump.
...
l_ClassC__label1:
        code (  r0:l_ClassC ...)
...
```

Figure 9: Dynamic dispatch in JavaTAL

ascertain whether or not the receiver class in "r0", (with type `l_InterfaceForC`,) is a certain match for the type of the jump address, (`r0:l_ClassC`). For example, if instructions were inserted just before the jump to load r0 with a different object of another class that implements the interface, it could be potentially disasterous, (certainly not type-sound,) for our program to make the jump based on faith.

Our first design of type system used rules that were specifically setup to recognize the load-load-jump pattern of a standard dynamic dispatch. While this would guarantee type-soundness, it limits JavaTAL to a fixed sequence of instructions for every dynamic dispatch. In essence, it is like having a macro instruction for dynamic dispatch, which is one of the failings of Java byte codes we were trying to overcome in the first place. A fixed load-load-jump sequence would prevent any attempts to reorder, regroup, or otherwise optimize the instructions involved. Furthermore, this solution would not scale well in relation to other object-oriented problems; it appeared that we would need to add additional macro instructions to JavaTAL to support many additional object-oriented features in MiniJava. As the purpose of this work is to design a sufficiently low-level language to allow aggressive optimization, each additional macro instruction in JavaTAL would be a step away from our goal.

As we examined the Leap of Faith problem in greater depth, it became apparent that all the type-checker needed to know at the jump point was that the two prerequisite loads had taken place somewhere earlier, and that the destination registers had not been overwritten by intervening instructions. The problem could be solved if the type system were somehow made aware of flow information – the flow of data in and out of registers.

It is not difficult to design a flow analysis system that can determine whether or not the registers

```
; Prepare for jump sequence to label1()
        ; Initial Types = [ r0:l_InterfaceForC ]

ld      r1,     r0[0]          ; Load address of Virtual Table into temp.
        ; Types = [ r0:l_InterfaceForC
        ;           r1:Virtual(l_InterfaceForC) ]

ld      r2,     r1[0]          ; Load address of Jump Vector into temp.
        ; Types = [ r0:l_InterfaceForC
        ;           r1:Virtual(InterfaceForC)
        ;           r2:Label(InterfaceForC__label1) ]

jmp     r2                     ; Make the Jump.

...

l_ClassC__label1 :
        code (  r0:l_ClassC    ; 'this' pointer.
             )
...
```

Figure 10: The Leap of Faith problem in JavaTAL

contain the proper "self" pointer and jump vector upon reaching a jump point. Furthermore, such a system supports all manner of code motion, and places no undue restrictions on optimizations. Unfortunately, it is extremely difficult to encode flow analysis into the rules of a type system. Our initial attempts resulted in extremely cumbersome side-conditions on many of the typing rules. Thus, while the notion of a type system super-charged with our particular style of flow analysis was novel, it does not appear to be a practical solution to the Leap of Faith problem. We have abandoned flow analysis rules in favor of pursuing a more elegant proof of type-soundness.

4.4 Bounded Existential Types to the Rescue

Our attempts to design a type system for JavaTALv0 revealed the many difficult sticking points in typing objects in a low-level language. Armed with that knowledge, we looked to previous work in the programming language community, to see how others had dealt with the problem at higher level representations. This subsection covers what we found, and how it can be applied to the very different representation of JavaTAL.

In Abadi and Cardelli's work on encoding objects into variations of the lambda calculus, they used bounded existential types to model abstraction [AC96]. Like the existential types discussed earlier in the section on MTAL, bounded existential types hide information about an object through abstractions. Unlike simple existential types, bounded existential types are able to advertise an interface through a bound. By saying that an object has type "there exists some β such that β is less than or equal to I," ($\exists(\beta \leq I).\beta$), it is revealed that whatever mystery type β is, it must at least implement all of interface I's proscribed methods.

The use of recursive, bounded existential types has been studied extensively in the context of typed lambda calculi, such as "System $F^{\omega}_{<:}$" [BCP97], and "$Ob_{1<:\mu}$" [AC96]. The bottom line is that it is a clever way to package up a method with its environment in such a way as to prevent them from being separated before the method is called.

Recursive, bounded existential types have several desirable properties for encoding objects. In addition to supporting all of the operations normally performed on objects in MiniJava, these types could allow method update, jump vector switching, multiple tiers of data abstraction, and other object-oriented features.

First and foremost, we need to be able to solve the Leap of Faith problem. Figure 11 shows the familiar JavaTAL dynamic dispatch, with a bounded existential type for the interface. Following popular convention, (such as [BCP97],) we will use the syntax "`Some(Beta<:I).Beta`" to represent "$\exists(\beta \leq I).\beta$".

```
; Prepare for jump sequence to label1()
        ; Initial Types = [ r0:Some(Beta <: l_InterfaceForC).Beta ]

unpack  r0                      ; Instantiate Existential
        ; Types = [ r0:Beta ]
        ; Ordering = < Beta <: l_InterfaceForC >
        ; Classes =
           < Beta := Classes(l_InterfaceForC)[r0:Beta / r0:l_InterfaceForC] >

ld      r3,     r0[0]           ; Load address of Virtual Table into temp.
        ; Types = [ r0:Beta
        ;           r3:Virtual(Beta) ]

ld      r4,     r3[0]           ; Load address of Jump Vector into temp.
        ; Types = [ r0:Beta
        ;           r3:Virtual(Beta)
        ;           r4:Code( r0:Beta ) ]

jmp     r4                      ; Make the Jump.

...

l_ClassC__label1 :
        code (  r0:l_ClassC     ; 'this' pointer.
             )
...
```

Figure 11: The Leap of Faith with Bounded Existential types.

The new JavaTAL instruction, "unpack" can be thought of as a type-checker directive, which will generate no actual instructions when JavaTAL is mapped down to machine code. The unpack instruction tells the type-checker to do several things. First, r0 is given a fresh type, which we call "Beta" here. It is important to note that this type is "fresh", that it does not occur anywhere else in the program. Thus, in spite of the fact that there could be several existential types listed in the program as Beta, the type-checker will differentiate each of these by their unpack instructions.

Next, the type-checker adds information about this new Beta into its environments for tracking object types and subtyping relationships. More specifically, it makes note that this Beta is less than or equal to (a subtype of) the type of its specified bound. In the environment for tracking object types, a copy of the bounding interface is made, with "r0:Beta" substituted for each occurrence of the original interface in the jump vectors.

In this way, when the type-checker reaches the jmp instruction, it sees that the desired type for the jump address is an object of type Beta in r0. Because this Beta can only come from an unpacked object, the type system ensures that the jump destination corresponds to the proper self-pointer in r0. This fact, combined with lemmas about object well-formedness and proper construction of the class hierarchy, will allow us to prove the type-soundness of this dynamic dispatch by concluding that the variable Beta is really "l_ClassC". Type-soundness removes the need for faith from the Leap of Faith problem.

Another desirable property of bounded existential types is that they solve the Leap of Faith problem without hindering the simpler optimizations in JavaTAL. For example, when a message send is known to have only one possible destination, it is eminently desirable to eliminate the overhead of the dynamic dispatch with a jmp directly to the address. (A comparable optimization in C++ has been shown to improve overall execution speed by as much as 24% in some experiments [CG94].) Our previous typing systems did not support this, because they needed to identify components of the dynamic dispatch in order to allow any jump. Under the new type system, the direct jump is trivial to type, as in Figure 12.

```
; Prepare for jump sequence to label1()
        ; Initial Types = [ r0:l_ClassC ]

jmp     l_ClassC__label1        ; Make the Jump.

...

l_ClassC__label1 :
        code (  r0:l_ClassC     ; 'this' pointer.
             )
...
```

Figure 12: Direct jump to a singleton receiver class.

At present, JavaTAL does not include primitive support for "instanceof" or type-casting, in spite of the fact that MiniJava includes these object-oriented features. The instanceof operator can be transformed into a dynamic dispatch at the MiniJava level, and thus can be swept under the carpet during one of the intermediate passes of our compiler. Type-casting can be handled similarly, but both of these source-to-source transformations can theoretically generate additional classes on the order of n^2, where n is the number of classes in the original program. Ultimately, we would like to have efficient support for these operations at the JavaTAL level. The "typecase" construct in [AC96] is supported by bounded existential types, so it is very likely that it will not be a problem to incorporate into later versions of JavaTAL.

4.5 The Trouble with Bounded Existentials

There are two important practical problems with bounded existential types, both of which remain unsolved.

The first difficulty is in determining subtype information between bounded existential types with different bounds, as noted by [GP98] and [Pie92]. These types support generic object encodings to the point that undecidable problems can be encoded as subtyping statements. As it happens, the algorithm only diverges in certain rare, pathological cases. However, it is extremely difficult to prove indelible properties about a system as a whole when certain key components can conceivably diverge.

Our solution to this quandary is to include conclusive, decidable subtyping information at the top of our JavaTAL files. In the case of MiniJava, it is a trivial task to verify the integrity of the static class hierarchy, and lookup subtyping relations in at most O(n) time during type-checking. (This is because MiniJava inheritance trees can be represented as directed, acyclic graphs, with each node having at most one parent.)

To support more general object-oriented languages, more elaborate methods will be required to efficiently encode subtyping proof information at the head of our mobile code files. A major segment of our future work will be to design a system for encoding class hierarchy information for general, recursive, bounded existential object types, without the help of the restrictive Java class system. This problem remains a major obstacle to building a general object-oriented Typed Assembly Language, because it is not clear how to effectively encode subtype proofs in our JavaTAL files in a compact, fully expressive manner. In the mean time, we cheat and rely on the Java name-based type system.

The second problem with bounded existentials is a matter of efficiency. The simplified existential types presented in the previous section are an abbreviated version of bounded existentials, adapted to the Java name-based type system. While these "demi-existentials" are sufficient to solve the Leap of Faith problem, they are much weaker cousins to the real bounded existentials presented in [AC96] and [BCP97]. The full-blown bounded existentials would have the form, "$\exists(\beta \leq I).\langle\beta\langle\langle\Gamma_0, ..., \Gamma_n\rangle\tau_1, ..., \tau_m\rangle\rangle$". The two-level vector representation of an object becomes a three-level vector, with the first level of the vector containing a pointer to the head of the structure, and a pointer to the second level vector.

While this extra level of indirection in the type system solves many otherwise intractable problems with encoding general objects, it adds an extra level of indirection to every dynamic dispatch at runtime. Even worse, it adds an extra level of indirection when accessing any member variable of an object. In short, the price we pay for general, flexible object support is a huge hit in runtime speed, as well as increased memory overhead in every runtime object.

As one of the goals of this work is to optimize object-oriented programs at a lower level, the addition of an extra level of indirection to every operation is an unacceptable backward step. Unfortunately, the three-level vector scheme is the only type-sound method that theoreticians have found for encoding general objects. For this reason, JavaTALv1 uses only the "demi-existentials" described in the previous section. One of the primary goals of the JavaTAL project is to overcome this limitation in future versions of our system, in order to provide efficient support for general object types.

4.6 Design for JavaTALv1

The Leap of Faith problem is under control with the help of demi-existentials, but there remain many design issues to be resolved before full-blown bounded existential types can be employed in JavaTAL. In order to better understand the obstacles presented by bounded existential types, we have pressed ahead with the design of a type system for JavaTALv1, even though this is only an intermediate milestone in the life of the JavaTAL project. This subsection deals with the improvements that must be made to the TAL system in order to accommodate JavaTAL.

Morrisett's TAL type-checker [MWCG97] maintains three separate environments while verifying a TAL program. The first of these is "Ψ", which maps labels to heap types, where heap types are either code blocks or tuple types. The second of these is "Δ", which keeps track of type contexts, the lists of type variables currently in use. The third of these is "Γ", which can be thought of as the current register types.

Our JavaTALv1 design adds two additional environment variables to these. First, there must be an ordering environment to encode the class subtyping information listed at the top of the JavaTAL file. We call this environment "Ω", ("Omega" is for "Ordering",) and define it to contain entries of the form "`C<:I`" to explicitly list valid subtype relationships. When program verification begins, Ω is built up from the static class hierarchy information listed at the top of the program; however, Ω may be extended with fresh subtyping relationships by `unpack` instructions for bounded existentials during the type-check process.

Secondly, we add a label lookup environment, "Λ". ("Lambda" is for "Label Lookup".) The Λ environment maps class and interface label names to actual tuple types. Λ is not strictly necessary, because the type system could eliminate class names altogether, and use the actual tuple types where ever the names occur. This is quite cumbersome in practice, however, and keeping the original class and interface names around is helpful for debugging. When program verification begins, Λ is constructed from the hierarchy information at the top of the JavaTAL file, but Λ serves an additional purpose when `unpack`ing bounded existential types. The type-checker can extend Λ with a tuple type for the new, fresh type variable it is instantiating, and thereby make available to subsequent steps the information represented by the bounded existential's interface bound.

Class types in JavaTALv1 will be represented in the same fashion as the initial version. All classes are a tuple of members, (the data vector,) with the first position containing a tuple of jump addresses representing methods, (the jump vector.)

The structures of interfaces are specified at the head of the JavaTAL file in the same way as the initial version. However, all items of interface type will be given bounded existential types, where the bound is the interface. Thus, member variables that start out as interface I in MiniJava will become $\exists(\beta \leq I).\beta$, which is spelled out in printable characters as "`Some(Beta<:I).Beta`".

Appendix B contains the current draft of the grammar for parsing JavaTAL version 1 files. The grammar remains a draft because the proof of type-soundness for JavaTALv1 is not yet complete. It is possible that unforeseen developments during the proof phase of the project could require changes to the JavaTAL grammar or JavaTAL file format.

The JavaTAL draft grammar is capable of accepting more general JavaTAL files than have been described in this paper. For example, there are provisions in the grammar for “flagged types”, as used by Morrisett to track tuple initialization. While our type-checker will track initialization information in the same way as the TAL compiler [MWCG97], such types only appear in the middle of basic block analysis for our system. However, as we add optimizations to our compiler, it is likely that we will want to break up basic blocks in a manner that exposes type information that is currently available only at intermediate steps. For that reason, our grammar is designed to accept more detailed JavaTAL files than our current prototype compiler ever generates.

Appendix C contains the current draft of the type rules for JavaTAL version 1. These rules are a draft for two important reasons. First, time constraints and a rapidly improving design have prevented us from completing the proof of type-soundness. While these type rules are based on other type-sound systems, we have not yet shown conclusively that our new system is sound. The final proof may require alterations to the draft rules presented here. Secondly, this entire design remains extremely Java-centric. Given our goal of developing a general system for supporting object-oriented languages, there is much work that remains to be done for this system to be disentangled from the particulars of MiniJava.

For example, many traces of MiniJava’s name-based class system remain evident in JavaTALv1’s label system. A class specifies its parent by name, a jump vector is tied to its class by name, specific basic blocks are tied to their jump vectors by name. This greatly reduces the overall expressiveness of the current JavaTAL system, preventing such things as mode-switching and structural subtyping. Recursive, bounded existential types have the power to encode many features not present in Java [AC96], when not hamstrung by Java’s name-based type system.

4.7 Summary and Future Work

The current JavaTALv1 system requires only a type-soundness proof to demonstrate that Java programs can be encoded in a very low-level, type-sound language with the help of recursive, bounded existential types. Therefore our most immediate plan for future work is to complete a Subject Reduction proof for JavaTAL, in much the same style as Morrisett et al’s Subject Reduction proof for TAL[MWCG97].

The JavaTALv1 design is only the beginnings of a real Typed Assembly Language for object-oriented programs. Once version 1 is complete, there are many more challenges to be pursued even before arriving at a sufficiently general system to begin testing with real programs. When the crutch of Java’s named-based class hierarchy has been removed, the bounded existential type system acquires theoretically undecidable subtyping relationships in the general case. In place of Java’s class system, a more general (yet still decidable) system for expressing class information must be devised. Furthermore, this new class system must be compact, easy to encode and decode, and capable of supporting a wide variety of high-level, object-oriented languages besides Java.

Many features must be added to the JavaTAL design before it will be ready for full-scale testing. Visibility modifiers, exceptions, mode-switching, explicit memory management, multiple inheritance, arbitrarily long inheritance chains, and structural subtyping are but a few possible features that have yet to be incorporated into any type-sound, low-level system such as this.

Most importantly, the questions still remains whether or not this system can actually offer an improvement over current mobile code schemes. At the time of writing, there is no real data on the overhead our type of system incurs in downloading, verifying, or running mobile code. It remains to be seen whether or not type-safe mobile code is capable of performing real, useful tasks.

Much has been said, in this paper and elsewhere, on the optimizability of Typed Assembly Language. Yet, no such optimizing, type-sound compilers are available. The key question out of this line of work that we would ultimately like to answer is, "Can we make type-sound Java programs run faster than in the Java Virtual Machine?" While the Virtual Machine is not currently a very challenging target to beat, there is a very real question of precisely what optimizations can be applied in the framework of a Typed Assembly Language. Much of the compiler research of the last two decades has gone into developing techniques for writing optimizing compilers that produce fast code. The constricted nature of the platform-independent Java system baffles many of the best speed-up techniques. Can JavaTAL allow us to once again take advantage of many of these well-known optimizations? Will JavaTAL enable us to apply new optimizations, such as using flow information to replace the types for single receiver class dynamic dispatches off of objects with interface types?

Compilers continue to scale up to match the ever increasing complexity of computer hardware. The additional structure provided by strong type systems such as JavaTAL's may be the best solution to prevent compiler complexity from spiraling out of control. In the end, the lessons we hope to learn from the JavaTAL project may be applicable to all manner of future compilers.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[BCP97] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. In *Proceedings of TACS'97, International Symposium on Theoretical Aspects of Computer Software*, pages 415–438. Springer-Verlag (*LNCS* 1281), 1997.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 1997.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of POPL'94, 21st Annual Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 262–275, January 1999.

[DTM95] Amer Diwan, David Tarditi, and Eliot Moss. Memory-system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, August 1995.

[FF98] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ICFP '98, Third ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill Book Company, 1992.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 250–261, January 1999.

[GP98] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75–96, 28 February 1998.

[Int97] Intel. *Intel Architecture Optimization Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1997.

[Int99] Intel. *Intel Architecture Software Developer's Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 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. Presented at 1999 ACM Workshop on Compiler Support for System Software, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *ACM Workshop on Types in Compilation*, pages 95–118, Kyoto, Japan, March 1998.

[ML97] Andrew Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 129–142, October 1997.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language (extended version). Technical Report TR97-1651, Cornell University, 4130 Upson Hall, Cornell University, Ithaca, NY 14853-7501, November 1997. A shorter version was presented at POPL'98.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[Nec97] George Necula. Proof-carrying code. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, October 1996.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 305–315, 1992.

[SF94] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 1–12, 1994.

[WBDF97] Dan Wallach, Dirk Balfanz, Drew Dean, and Edward Felten. Extensible security architectures for java. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 116–128, October 1997.

A Compiling Java into JavaTAL

Our current prototype compiler uses a small subset of Java as its source language. Called "Mini-Java," our language subset has been selected to exhibit all of the major object-oriented features of Java, without the syntactic complexity that makes real Java compilers unwieldy test beds.

MiniJava contains interfaces and class, and allows single inheritance among interfaces. Classes must inherit from a single interface. Base types include integer, boolean, and void. MiniJava includes assignment statements, if-statements (with else clause,) message send (also called function call,) and a print statement. Permissible expressions include addition, subtraction, comparison (less-than only,) type-cast, and instanceof. MiniJava does not contain any visibility modifier other than "public", and methods have no local variables.

The code example in Figure 13 shows a small MiniJava program which calculates and prints the integer "11".

```
class Te1 {
          public static void main(String[] a)
                    { System.out.println( new Ex1().m(5) ); }
          }
interface I1        { int m(int a); }
interface IC1       { int p(int b); }
class Ex1 implements I1
          {
          int x;
          public int m(int a)
                    {
                    x=(new C1().p(a+1))+4;
                    return x;
                    }
          }
class C1 implements IC1
          { public int p(int b) { return b+1; } }
```

Figure 13: Minijava class "Ex1".

The first compilation step is to break the complex statements and expressions into smaller, simpler chunks. The result of this is that each intermediate computation is named. For example, the assignment statement

```
x = w * (y + z);
m = (new N().p(x));
```

becomes

```
          v1 = y + z;
          x = w * v1;
          v2 = new N();
          m = v2.p(x);
```

where “v1” and “v2” are automatically generated variable names of the correct type.

Next, all methods are converted to type “void”. In order to do this, an extra parameter is added to each method; the parameter is used to store the return value of the original method. In Java, all parameters are call-by-value, and thus cannot be changed at the top level. Thus, base return types must be encapsulated in objects if the return value is to escape the scope of the method. The example class “Ex1” from above now appears as below, in Figure 14.

```
interface I1
          {
          void m ( int a , Result result ) ;
          }
class Ex1 implements I1
          {
          Result result ;
          int x ;
          C1 v5 ;
          int v3, v4, v6, v7 v8 ;
          public void m ( int a , Result result )
                    {
                    v5 = new C1 ( ) ;
                    v7 = 1 ;
                    v6 = a + v7 ;
                    v5 .  p ( v6 , result ) ;
                    v4 = result .  fint ;
                    v8 = 4 ;
                    v3 = v4 + v8 ;
                    x = v3 ;
                    result .  fint = x ;
                    }
          }
```

Figure 14: Class Ex1 in Intermediate Form #1.

Notice the manner in which an object “result” of type “Result” is used to transport return values across methods. The Result class contains one field of each type in the program, both base types and user-defined classes, in order to serve as the carrier for all return values.

Unfortunately, there is a problem with this naive approach. While this transformation appears intuitively to be sound, the use of a single object for returning values fails for certain kinds of recursive methods when all variables are class members. There are two possible solutions. The addition of local variables to methods would make it possible to properly preserve the return values during recursion; however, we did not wish to add to JavaTAL the complications that accompany local variables. (If the STAL [MCGW98] stack extensions were incorporated into JavaTAL, type-sound local variables would take care of the whole matter.) The other solution is to create a stack of results for recursive procedures, so that each result does not overwrite the result above it before it is used. The resultant mechanism is rather inelegant, as it requires shuffling around several pointers before and after each method call. While the current JavaTAL compiler invokes the ResultStack mechanism for every call point, a clever compiler need only do this for certain recursive functions. In addition, much of the inefficient or stylistic boilerplate that appears in the intermediate representations can be optimized away at the JavaTAL level.

A.1 CPS Conversion

Like Morrisett et al.'s TAL compiler[MWCG97], the next stage in compiling to JavaTAL is CPS conversion.

In recent years, CPS conversion has become a controversial topic in the programming language community. On the one hand, CPS is a well understood compiler technique[FWH92], which has been employed in many successful optimizing compilers, such as the SML/NewJersey compiler[App92]. While CPS had been accepted in the compiler community for years as nebulously helpful, Sabry and Felleisen have shown that CPS-based program analysis does not provide any more information for optimizations than can be had from sufficiently powerful direct analyses[SF94]. Furthermore, Diwan et al. have shown that the loss of locality caused by allocating all activation records (continuations) on the heap can cause a serious degradation in memory-system performance when compared with non-CPS, stack-based allocation[DTM95]. Modern architectures such as the Intel Pentium II use pipeline stall-preventing branch prediction hardware that depends on a call/return, stack-based execution model[Int97]. At best, CPS-converted programs fail to take advantage of this hardware; at worst, CPS may actually hinder hardware optimizations like branch prediction.

In the absence of a clear consensus on CPS conversion's positive or negative effects on overall program performance, we have elected to use CPS conversion in our prototype compiler for several practical reasons. One of the most useful results of CPS conversion that everything ends up in tail form. In other words, every intermediate control flow point becomes explicitly named. This is especially helpful in the context of Typed Assembly Language, as it breaks the program into non-nested methods with clearly specified interfaces. Each "basic block" in JavaTAL is a segment of uninterrupted control flow; because each basic block will come from a MiniJava method, the complete environment of the basic block is spelled out at the top of the corresponding method.

The second practical reason for choosing CPS is that all message sends will occur in tail position, that is to say, at the end of a basic block. Without subsequent instructions to return to in the basic block, there is no need to store a return address. Tail form calls can be mapped into jump instructions, rather than call instructions, which are substantially cheaper[Int99]. CPS conversion introduces additional method calls to the program, so the benefit of jump instructions versus call instructions does not necessarily translate immediately into overall program speedup. However, the presence of jump instructions is a valuable step toward our goal of optimizing certain dynamic dispatches to use direct jumps.

The use of CPS conversion also greatly simplifies memory management in our prototype system. All data – activation records and user data alike – is allocated on the heap, so our programs need not worry about deallocation. It is assumed that a trusted, conservative garbage collector will clean up after our prototype programs. There do not appear to be any inherent barriers in our system that would prevent using the region-based stack allocation technique used by STAL[MCGW98].

Java's class-based, object-oriented nature presents some interesting problems when it comes to CPS conversion. The process of splitting complex methods into simpler, basic block-like methods generates many additional methods; the first problem is determining where the newly generated methods go in the class hierarchy. It would be helpful to have continuation methods added to the original class from which they are spawned, as this would allow the code in the continuation method to access all the member variables in the same object as the original method. However, it is also the nature of continuations to be anonymous to the methods in which they are invoked. The most natural invocation for a continuation is "k.apply()", where "k" is the continuation object, and "apply" is the method containing the actual continuation code. In Java, this constraint implies that all continuations inherit from a uniform continuation interface, and that a given object can contain at most one continuation.

It would be possible to design an intermediate representation for the CPS programs that sidesteps the limitations of Java. However, we wanted this prototype compiler design to use MiniJava in all of its intermediate forms leading to JavaTAL, in order to make use of the existing Java type-checker. Also, our intermediate translations could then be compiled and run at any stage to check correctness. As a result, each continuation generated during CPS conversion causes the creation of a new class. At runtime, continuations will be passed around as actual objects instantiated from the continuation classes.

Continuation classes all inherit from a uniform interface, called "ContinuationInterface". This interface contains one method, "apply()", which allows any continuation object to be applied anonymously at any tail point in the program.

Actual Continuation classes, which our compiler automatically names using the original class name, original method name, and a unique number, contain two methods. The first method is the apply() method, which contains a basic block of code lifted from the original complex method. The second method, the setContinuation() method, is tailored to be called from the point just after a continuation is created. SetContinuation() is used to initialize all of the data fields in the continuation class to point to the appropriate fields in the original class. In this way, code executing in the scope of the continuation class can access and modify the member variables of the original class – the environment in which the continuation code would have executed in prior to CPS conversion.

The example code in Figures 15 and 16 is the CPS version of the example from above, along with the recursive "stack hack". Note the use of the "that" pointer in the ContinuationFor_Ex1_m_2 class to access the member variables of class Ex1, where the instructions originally resided before CPS conversion.

At first glance, it appears that our initial attempt to optimize dynamic dispatches has resulted in at least twice as many dynamic dispatches. However, it should be noted that all setContinuation() methods can be immediately inlined in the resulting JavaTAL code, because they represent the normal parameter shuffling that takes place before a normal procedure call is executed. The setContinuation() methods remain in place (un-inlined) for the MiniJava intermediate representations

```
interface I1
          { void m ( int a , ResultStack resstack ,
                    ContinuationInterface Continuation_K ) ; }
class Ex1 implements I1
          {
          ResultStack rstack, tempstack ;
          int x ;
          C1 v5 ;
          int v7, v6, v4, v8, v3 ;
          public void m ( int a , ResultStack resstack ,
                    ContinuationInterface Continuation_K )
                    {
                    v5 = new C1 ( ) ;
                    v7 = 1 ;
                    v6 = a + v7 ;
                    tempstack = new ResultStack ( ) ;
                    tempstack.tail = rstack ;
                    rstack = tempstack ;
                    Continuation_K_Ex1_m_2 = new ContinuationFor_Ex1_m_2 ( ) ;
                    Continuation_K_Ex1_m_2.setContinuation ( this , v5 , v6 ,
                              rstack , a , resstack , Continuation_K ) ;
                    }
          ContinuationFor_Ex1_m_2 Continuation_K_Ex1_m_2 ;
          }
interface ContinuationInterface
          { void apply ( ) ; }
```

Figure 15: CPS Converted class Ex1.

only to satisfy Java's rigid object model.

```
class ContinuationFor_Ex1_m_2 implements ContinuationInterface
        {
        Ex1 that ;
        int a ;
        ResultStack resstack ;
        ContinuationInterface Continuation_K ;
        public void setContinuation ( Ex1 that , C1 MessageSend_Target , int v6 ,
                ResultStack rstack , int a , ResultStack resstack ,
                ContinuationInterface Continuation_K )
                {
                this.that = that ;
                this.a = a ;
                this.resstack = resstack ;
                this.Continuation_K = Continuation_K ;
                MessageSend_Target.p ( v6 , rstack , this ) ;
                }
        ResultStack rstack ;
        public void apply ( )
                {
                this.rstack = that.rstack ;
                that.v4 = rstack.fint ;
                this.rstack = that.rstack ;
                that.rstack = rstack.tail ;
                that.v8 = 4 ;
                that.v3 = that.v4 + that.v8 ;
                that.x = that.v3 ;
                resstack.fint = that.x ;
                Continuation_K.apply ( ) ;
                }
        }
```

Figure 16: Continuation for class Ex1.

B JavaTAL Grammar

Figures 17 and 18 show the current draft of the grammar for parsing JavaTAL version 1 files.

```
                Goal ::= ( TypeDeclaration )* ( HeapElement )* "main" ":"
                           CodeBody <EOF>
     TypeDeclaration ::= InterfaceDeclaration
                     |   InterfaceExtends
                     |   ClassDeclaration
InterfaceDeclaration ::= "interface" ClassOrInterface SpecialTupleType
    InterfaceExtends ::= "interface" ClassOrInterface "extends"
                            ClassOrInterface SpecialTupleType
    ClassDeclaration ::= "class" ClassOrInterface "implements"
                            ClassOrInterface TupleType
                Type ::= BooleanType
                     |   IntegerType
                     |   ClassOrInterface
                     |   TupleType
                     |   RegisterFileType
                     |   ExistType
         BooleanType ::= "bool"
         IntegerType ::= "int"
    ClassOrInterface ::= Label
           TupleType ::= "<" ( Type )* ">"
    RegisterFileType ::= "(" ( RegisterDecl )* ")"
           ExistType ::= "Some" "(" Label "<" ":" ClassOrInterface ")" Label
        RegisterDecl ::= Register ":" Type
    SpecialTupleType ::= "<" "<" ( RegisterFileType )* ">" ">"
         HeapElement ::= VirtualTable
                     |   CodeBlock
        VirtualTable ::= Label ":" "<" ( Label )* ">"
           CodeBlock ::= Label ":" "code" RegisterFileType CodeBody
            CodeBody ::= CodeSequence
                     |   Jmp
                     |   Halt
```

Figure 17: Draft of Grammar for JavaTAL version 1

```
        CodeSequence ::= Instruction CodeBody
         Instruction ::= Add
                      |   Bnz
                      |   Cmp
                      |   Ld
                      |   Malloc
                      |   Mov
                      |   Print
                      |   St
                      |   Sub
                      |   Unpack
                 Add ::= "add" Register "," Register "," Value
                 Bnz ::= "bnz" Register "," Value
                 Cmp ::= "cmp" Register "," Register "," Value
                  Ld ::= "ld" Register "," Register "[" IntegerLiteral "]"
              Malloc ::= "malloc" Register "[" ClassOrInterface "]"
                 Mov ::= "mov" Register "," TypedValue
               Print ::= "print" Value
                  St ::= "st" Register "[" IntegerLiteral "]" "," Register
                 Sub ::= "sub" Register "," Register "," Value
                 Jmp ::= "jmp" Value
                Halt ::= "halt"
              Unpack ::= "unpack" Register
               Value ::= Register
                      |   Label
                      |   JunkValue
                      |   IntegerLiteral
            Register ::= "r" IntegerLiteral
               Label ::= <IDENTIFIER>
           JunkValue ::= "?" Type
      IntegerLiteral ::= <INTEGER_LITERAL>
          TypedValue ::= Register
                      |   Label
                      |   JunkValue
                      |   TypedIntegerLiteral
 TypedIntegerLiteral ::= <INTEGER_LITERAL> "[" Type "]"
```

Figure 18: Draft of Grammar for JavaTAL version 1(continued)

C JavaTALv1 Type Rules

The following type rules represent our current draft for the type system of JavaTAL version 1. At the time of this writing, the required proof for type-soundness has not yet been completed. We present these draft rules to give the reader a flavor of what we believe the type system will be like for JavaTAL version 1 when completed.

These rules are based heavily upon Morrisett et al.'s TAL type system [MWCG97], which has been proven type-sound. They also draw upon Abadi and Cardelli's typed object lambda calculus variants [AC96], which are also type-sound.

Judgement	Meaning
$\vdash \Gamma_1 \equiv \Gamma_2$ codetype	Γ_1 and Γ_2 are the same, except possibly for r0.
$\vdash \tau_1 \sqsubseteq \tau_2$ jvector	τ_1 is a jump vector subtype of τ_2
$l \vdash \Gamma$ r0type	Γ contains r_0 of type l
$\Lambda; \Omega \vdash D$	D is a well-formed declaration of class hierarchy information.
$\Lambda \vdash \Gamma_1 \leq \Gamma_2$ rftype	Γ_1 is a register file subtype of Γ_2
$\Lambda \vdash \tau$ type	τ is a well-formed type
$\Lambda \vdash \Psi$ htype	Ψ is a well-formed heap type
$\Lambda \vdash \Gamma$ rftype	Γ is a well-formed register file type
$\Lambda; \Omega; \Psi \vdash w : \tau$ wval	w is a well-formed word value of type τ
$\Lambda; \Omega; \Psi \vdash w : \tau^\varphi$ fwval	w is a well-formed word value of flagged type τ^φ
	(i.e., w has type τ, or w is $?\tau$ and φ is 0)
$\Lambda; \Omega; \Psi \vdash tw : \tau$ twval	tw is a well-formed typed word value of type τ
$\Lambda; \Omega; \Psi \vdash tw : \tau^\varphi$ ftwval	tw is a well-formed typed word value of flagged type τ^φ
	(i.e., tw has type τ, or tw is $?\tau$ and φ is 0)
$\Lambda; \Omega; \Psi; \Gamma \vdash v : \tau$ val	v is a well-formed small value of type τ
$\Lambda; \Omega; \Psi; \Gamma \vdash tv : \tau$ tval	tv is a well-formed typed small value of type τ
$\Lambda; \Omega; \Psi \vdash h : \tau$ hval	h is a well-formed heap value of type τ
$\Lambda; \Omega; \vdash H : \Psi$ heap	H is a well-formed heap of heap type Ψ
$\Lambda; \Omega; \Psi \vdash R : \Gamma$ regfile	R is a well-formed register file of register file type Γ
$\Lambda; \Omega; \Psi; \Gamma \vdash S$	S is a well-formed instruction sequence
$\vdash P$	P is a well-formed program

Figure 19: Static semantic judgments

$\boxed{\vdash \Gamma_1 \equiv \Gamma_2 \text{ codetype}}$

$$\vdash \{r_0 : \tau_0, r_1 : \tau_1, \ldots, r_n : \tau_n\} \equiv \{r_0 : \tau_0', r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ codetype}$$

$\boxed{\vdash \tau_1 \sqsubseteq \tau_2 \text{ jvector}}$

$$\frac{\Gamma_i \equiv \Gamma_i^1 \text{ codetype}}{\vdash \langle \Gamma_0 \ldots \Gamma_m \rangle \sqsubseteq \langle \Gamma_0^1 \ldots \Gamma_n^1 \rangle \text{ jvector}} \quad (0 \leq i \leq n) \quad (m \geq n)$$

$\boxed{l \vdash \Gamma \text{ r0type}}$

$$l \vdash \{r_0 : l, r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ r0type}$$

$\boxed{\Lambda; \Omega \vdash D}$

$$\frac{\begin{array}{l}\Lambda\{l_1 : \langle\langle\rangle\rangle\} \vdash \Gamma_i \text{ rftype} \\ l_1 \vdash \Gamma_i \text{ r0type} \\ \Lambda\{l : \langle\langle\Gamma_0, ..., \Gamma_n\rangle\rangle\}; \Omega \vdash D\end{array}}{\Lambda; \Omega \vdash \ \texttt{interface}\ l <<\Gamma_0, ..., \Gamma_n>>; D} \quad (0 \leq i \leq n)$$

$$\frac{\begin{array}{l}\Lambda \vdash l_2 :<< \Gamma_0^1, ..., \Gamma_m^1 >> \\ \vdash \langle\Gamma_0, ..., \Gamma_n\rangle \sqsubseteq \langle\Gamma_0^1, ..., \Gamma_m^1\rangle \text{ jvector} \\ \Lambda\{l_1 : \langle\langle\rangle\rangle\} \vdash \Gamma_i \text{ rftype} \\ l_1 \vdash \Gamma_i \text{ r0type} \\ \Lambda\{l_1 : \langle\langle\Gamma_0, ..., \Gamma_n\rangle\rangle\}; \Omega\{l_1 \leq l_2\} \vdash D\end{array}}{\Lambda; \Omega \vdash \ \texttt{interface}\ l_1\ \texttt{extends}\ l_2 <<\Gamma_0, ..., \Gamma_n>>; D} \quad (m \leq n), \quad (0 \leq i \leq m)$$

$$\frac{\begin{array}{l}\Lambda \vdash l_2 :<< \Gamma_0^1, ..., \Gamma_m^1 >> \\ \vdash \langle\Gamma_0, ..., \Gamma_n\rangle \sqsubseteq \langle\Gamma_0^1, ..., \Gamma_m^1\rangle \text{ jvector} \\ \Lambda\{l_1 : \langle\langle\rangle\rangle\} \vdash \Gamma_i \text{ rftype} \\ \Lambda\{l_1 : \langle\langle\rangle\rangle\} \vdash \tau_j \text{ type} \\ l_1 \vdash \Gamma_i \text{ r0type} \\ \Lambda\{l_1 : \langle\langle\Gamma_0, ..., \Gamma_n\rangle\tau_1, ...\tau_p\rangle\}; \Omega\{l_1 \leq l_2\} \vdash D\end{array}}{\Lambda; \Omega \vdash \ \texttt{class}\ l_1\ \texttt{implements}\ l_2 <<\Gamma_0, ..., \Gamma_n>\tau_1, ...\tau_p>; D} \quad \begin{array}{l}(m \leq n), \\ (0 \leq i \leq m), (0 \leq j \leq p)\end{array}$$

$$\frac{\begin{array}{l}\Lambda \vdash t :<< \Gamma_0, ..., \Gamma_n >> \\ \Lambda\{l_0 : \Gamma_0, ..., l_n : \Gamma_n\}; \Omega \vdash D\end{array}}{\Lambda; \Omega \vdash t\ \texttt{:<}\ l_0, ..., l_n>; D}$$

$\boxed{\Lambda \vdash \Gamma_1 \leq \Gamma_2 \text{ rftype}}$

$$\frac{\Lambda \vdash \tau_i \text{ type} \quad (1 \leq i \leq m)}{\Lambda \vdash \{r_1 : \tau_1, \ldots, r_m : \tau_m\} \leq \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ rftype}} \quad (m \geq n)$$

$\boxed{\Lambda \vdash \tau \text{ type}}$

$$\frac{\Lambda \vdash l : \tau' \quad (\text{for all } l \text{ occurring in } \tau)}{\Lambda \vdash \tau \text{ type}}$$

$\boxed{\Lambda \vdash \Psi \text{ htype}}$

$$\frac{\Lambda \vdash \tau_i \text{ type}}{\Lambda \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \text{ htype}} \quad (0 \leq i \leq n)$$

$\boxed{\Lambda \vdash \Gamma \text{ rftype}}$

$$\frac{\Lambda \vdash \tau_i \text{ type}}{\Lambda \vdash \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ rftype}} \quad (0 \leq i \leq n)$$

$\boxed{\Lambda;\Omega;\Psi \vdash w : \tau \text{ wval}}$

$$\Lambda;\Omega;\Psi \vdash i : \mathsf{Int} \text{ wval} \qquad \frac{\Lambda;\Omega \vdash \Psi(l) \leq \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash l : \tau \text{ wval}} \qquad \frac{\Lambda;\Omega \vdash \Psi(l) \leq \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash l : \exists(\beta \leq \tau).\beta \text{ wval}}$$

$\boxed{\Lambda;\Omega;\Psi \vdash w : \tau^{\varphi} \text{ fwval}}$

$$\frac{\Lambda;\Omega;\Psi \vdash w : \tau \text{ wval}}{\Lambda;\Omega;\Psi \vdash w : \tau^{\varphi} \text{ fwval}} \qquad \frac{\Lambda \vdash \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash ?\tau : \tau^{0} \text{ fwval}}$$

$\boxed{\Lambda;\Omega;\Psi \vdash tw : \tau \text{ twval}}$

$$\Lambda;\Omega;\Psi \vdash i[\tau] : \tau \text{ twval} \qquad \frac{\Lambda;\Omega \vdash \Psi(l) \leq \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash l : \tau \text{ twval}} \qquad \frac{\Lambda;\Omega \vdash \Psi(l) \leq \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash l : \exists(\beta \leq \tau).\beta \text{ twval}}$$

$\boxed{\Lambda;\Omega;\Psi \vdash tw : \tau^{\varphi} \text{ ftwval}}$

$$\frac{\Lambda;\Omega;\Psi \vdash tw : \tau \text{ wval}}{\Lambda;\Omega;\Psi \vdash tw : \tau^{\varphi} \text{ ftwval}} \qquad \frac{\Lambda;\Omega \vdash \tau \text{ type}}{\Lambda;\Omega;\Psi \vdash ?\tau : \tau^{0} \text{ ftwval}}$$

$\boxed{\Lambda;\Omega;\Psi;\Gamma \vdash v : \tau \text{ val}}$

$$\Lambda;\Omega;\Psi;\Gamma \vdash r : \tau \text{ val} \quad (\Gamma(r) = \tau) \qquad \frac{\Lambda;\Omega;\Psi \vdash w : \tau \text{ wval}}{\Lambda;\Omega;\Psi;\Gamma \vdash w : \tau \text{ val}}$$

$\boxed{\Lambda;\Omega;\Psi;\Gamma \vdash tv : \tau \text{ tval}}$

$$\Lambda;\Omega;\Psi;\Gamma \vdash r : \tau \text{ tval} \quad (\Gamma(r) = \tau) \qquad \frac{\Lambda;\Omega;\Psi \vdash tw : \tau \text{ twval}}{\Lambda;\Omega;\Psi;\Gamma \vdash tw : \tau \text{ tval}}$$

$\boxed{\Lambda;\Omega;\Psi \vdash h : \tau \text{ hval}}$

$$\frac{\Lambda;\Omega;\Psi \vdash w_i : \tau_i^{\varphi_i} \text{ fwval}}{\Lambda;\Omega;\Psi \vdash \langle w_0, \ldots, w_{n-1} \rangle : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \text{ hval}}$$

$$\frac{\Lambda \vdash l : \Gamma \quad \Lambda;\Omega \vdash \Gamma \text{ rftype} \quad \Lambda;\Omega;\Psi;\Gamma \vdash S}{\Lambda;\Omega;\Psi \vdash \text{“}l \ \mathsf{:code} \ \Gamma \ S\text{ ”} : \Gamma \text{ hval}}$$

$\boxed{\Lambda;\Omega \vdash H : \Psi \text{ heap}}$

$$\frac{\Lambda;\Omega \vdash \Psi \text{ htype} \quad \Lambda;\Omega;\Psi \vdash h_i : \tau_i \text{ hval}}{\Lambda;\Omega \vdash \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} : \Psi \text{ heap}} \quad (\Psi = \{l_1 : \tau_1, \ldots, l_n : \tau_n\})$$

$\boxed{\Lambda;\Omega;\Psi \vdash R : \Gamma \text{ regfile}}$

$$\frac{\Lambda;\Omega;\Psi \vdash w_i : \tau_i \text{ wval}}{\Lambda;\Omega;\Psi \vdash \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ regfile}}$$

$\boxed{\Lambda; \Omega; \Psi; \Gamma \vdash S}$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r_s : \texttt{int}\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma \vdash v : \texttt{int}\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma\{r_d : \texttt{int}\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{op}\ r_d, r_s, v\ \ S} \quad \texttt{op} \in \{\texttt{add}, \texttt{sub}\}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r : \texttt{bool}\ \mathrm{val} \ \ \Lambda; \Omega; \Psi; \Gamma \vdash v : \Gamma'\ \mathrm{val} \ \ \Lambda; \Omega \vdash \Gamma \leq \Gamma'\ \mathrm{rftype} \ \ \Lambda; \Omega; \Psi; \Gamma\{r_d : \texttt{int}\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{bne}\ r, v\ \ S}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r : \texttt{int}\ \mathrm{val} \ \ \Lambda; \Omega; \Psi; \Gamma \vdash v : \Gamma'\ \mathrm{val} \ \ \Lambda; \Omega \vdash \Gamma \leq \Gamma'\ \mathrm{rftype} \ \ \Lambda; \Omega; \Psi; \Gamma\{r_d : \texttt{int}\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{bnz}\ r, v\ \ S}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r_s : \tau\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma \vdash v : \tau\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma\{r_d : \texttt{bool}\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{cmp}\ r_d, r_s, v\ \ S}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma\{r_d : \tau_i\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{ld}\ r_d, r_s[i]\ \ S} \quad (\varphi_i = 1), \quad (0 \leq i < n)$$

$$\frac{\Lambda \vdash l_C : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \quad \Lambda; \Omega; \Psi; \Gamma\{r_d : \langle \tau_0^0, \ldots, \tau_{n-1}^0 \rangle\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{malloc}\ r_d, [l_C]\ \ S}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash tv : \tau\ \mathrm{tval} \quad \Lambda; \Omega; \Psi; \Gamma\{r_d : \tau\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{mov}\ r_d, tv\ \ S}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash v : \texttt{int}\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{print}\ v\ \ S}$$

$$\frac{\begin{array}{l}\Lambda; \Omega; \Psi; \Gamma \vdash r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\ \mathrm{val} \quad \Lambda; \Omega; \Psi; \Gamma \vdash r_s : \tau_i\ \mathrm{val} \\ \Lambda; \Omega; \Psi; \Gamma\{r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\} \vdash S\end{array}}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{st}\ r_d[i], r_s\ \ S} \quad (0 \leq i < n)$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash v : \Gamma'\ \mathrm{val} \quad \Lambda; \Omega \vdash \Gamma \leq \Gamma'\ \mathrm{rftype}}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{jmp}\ v}$$

$$\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{halt}$$

$$\frac{\Lambda; \Omega; \Psi; \Gamma \vdash r : \exists(\beta \leq \tau).\beta\ \mathrm{val} \quad \Lambda\{\alpha : \Lambda(\tau)[r0 : \alpha \backslash r0 : \tau]\}; \Omega\{\alpha \leq I\}; \Psi; \Gamma\{r : \alpha\} \vdash S}{\Lambda; \Omega; \Psi; \Gamma \vdash \texttt{unpack}\ r\ \ S}$$

$$(\alpha \notin \Lambda)$$

$\boxed{\vdash P}$

$$\frac{\Lambda; \Omega \vdash D \quad \Lambda; \Omega \vdash H : \Psi\ \mathrm{heap} \quad \Lambda; \Omega; \Psi \vdash R : \Gamma\ \mathrm{regfile} \quad \Lambda; \Omega; \Psi; \Gamma \vdash S}{\vdash (D, H, R, S)}$$

D Type Systems Explicated for Non-Theoriticians

For computer scientists such as myself, who have not come from a strong theory background, the preceeding Appendices can appear to have no data content. Indeed, unless one is accustomed to the notation commonly used by theorists in the language-typing community, the equations appear very difficult to decipher. In reality, it is very easy to transform type-judgements into a form readily understood by mainstream computer scientists and software engineers.

The very highest level overview of this project is that we want to create two important new things: 1) a platform-independant, assembly-like language for compiling programs from any object-oriented language, transporting them from a source machine to a destination machine; 2) a program that quickly and flawlessly reads in one of our mobile code files, and decides it is "safe" according to a certain set of definitions.

The equations in the previous Appendix are a high-level description of the second item – a totally reliable verification program. Each equation, called a "type judgement," describes a boolean function in the verifier. For example, the top-level function is described by the equation:

$$(prog)\quad \frac{\Lambda;\Omega \vdash D \quad \Lambda;\Omega \vdash H : \Psi \text{ heap} \quad \Lambda;\Omega;\Psi \vdash R : \Gamma \text{ regfile} \quad \Lambda;\Omega;\Psi;\Gamma \vdash S}{\vdash (D, H, R, S)}$$

What this says is that we have a boolean function that takes four parameters. We will call it "Prog", as this particular type judgement defines well-formed programs. The four parameters to the program are defined below the line, "(D, H, R, S)". Information elsewhere in the paper reveals that "D" is all of the declarations in the mobile code file. "H" describes the heap memory – all of the heap-allocated data structures, and the code blocks. "R" describes the set of registers on our virtual processor. "S" is the sequence of instructions to be executed. The four bunches of symbols above the line describe the simple body of the Prog function. Prog returns true if and only if all four of the boolean functions it calls return true. Each of those four functions are defined by other type judgements earlier in the Appendix. They are:

$\Lambda;\Omega \vdash D$	Are the class declarations complete and correct?
$\Lambda;\Omega \vdash H : \Psi$ heap	Is the description of the Heap sensible?
$\Lambda;\Omega;\Psi \vdash R : \Gamma$ regfile	Is the description of the registers sensible?
$\Lambda;\Omega;\Psi;\Gamma \vdash S$	Does the instruction sequence make sense?

Naturally, there are very rigorous definitions of "sensible" in this context, but that is the general idea. Thus, this entire rule would look something like the following in C:

```
BOOLEAN prog (struct decls D, struct heaps H, struct regs R, struct code S)
 { return (DeclsOK(D) & HeapsOK(H) & RegsOK(R) & SequenceOK(S));}
```

Close inspection of the rules for the verifier should convince the reader that this is, indeed, a description of a program that detects and rejects any malformed JavaTAL file. However, we seek to prove the verifier is completely trustworthy with mathematical rigor, and that proof has not yet been constructed for this particular system.