May 2004

Typed Intermediate Languages

Stephen Tse
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/20
For more information, please contact repository@pobox.upenn.edu.
Typed Intermediate Languages

Abstract
Programs written in a typed language are guaranteed to satisfy the safety properties of the type system without runtime checks. A type system for an intermediate language allows static verification of safety properties independent of source languages, and opens up opportunities for advanced compiler optimizations.

This paper surveys three major intermediate languages: Java bytecode, typed assembly language and proof-carrying code. Java bytecode requires minimal type annotation but sophisticated verification algorithms. Typed assembly language permits low-level constructs such as registers and instruction blocks, yet still enforces control-flow safety and memory safety. Proof-carrying code provides a general framework for any safety properties definable in a meta-logical framework.

We motivate the use of typed intermediate languages, illustrate the type systems of the three languages mentioned above with examples, and compare their tradeoffs of expressiveness versus complexity. Additionally, we assess the impact of the three languages and identify research directions for future work.

Comments

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/20
Typed Intermediate Languages

Stephen Tse

Abstract

Programs written in a typed language are guaranteed to satisfy the safety properties of the type system without runtime checks. A type system for an intermediate language allows static verification of safety properties independent of source languages, and opens up opportunities for advanced compiler optimizations.

This paper surveys three major intermediate languages: Java bytecode, typed assembly language and proof-carrying code. Java bytecode requires minimal type annotation but sophisticated verification algorithms. Typed assembly language permits low-level constructs such as registers and instruction blocks, yet still enforces control-flow safety and memory safety. Proof-carrying code provides a general framework for any safety properties definable in a meta-logical framework.

We motivate the use of typed intermediate languages, illustrate the type systems of the three languages mentioned above with examples, and compare their tradeoffs of expressiveness versus complexity. Additionally, we assess the impact of the three languages and identify research directions for future work.

1 Introduction

Performance, safety, and extensibility are three basic requirements for most software systems. The design and implementation of a programming language must consider these requirements so that programs can be executed at the maximal speed, programmers can express the safety properties to be enforced, and extensions can be seamlessly integrated.

Safety, or security in general, is particularly important for distributed computing: when users download a browser applet or use foreign function interfaces, the extension should not crash the host computer. An intermediate language for specifying these extensions should remain general, allowing implementations from different languages and compilers and at the same time permitting low-level optimizations for performance. That is, the design for an intermediate language must also take expressiveness into account.
Hardware protection has been the traditional approach to ensure safety: memory spaces are isolated between processes and an illegal memory access generates a segmentation fault. Although hardware can make use of the precise state and the dynamic context of the program for protection, programmers can specify protection only per memory page and hardware must switch kernel contexts while running multiple processes. This safety policy is far too coarse-grained and the context switch is too expensive for high-level programming. Instead, software-based fault isolation [98] inserts runtime checks into the binary code and allows program extensions run in the same process space of the main program. Hardware is then free to schedule and execute instructions of both the main program and the extensions in the same kernel context, while this sandbox model still guarantees the safety of the main program from an illegal memory access of the extensions.

Typed languages Modern software engineering demands static guarantee of program safety. A buggy program for spacecraft should be rejected by a compiler, instead of causing a runtime segmentation fault in space. A type system for a programming language is a tractable formal method for proving the absence of certain program behaviors by approximating the runtime values of expressions [78]. That is, unlike hardware or software protection at run time, a type system is used as a program analysis to guarantee safety properties at compile time.

Since programs that are statically verified do not need as many runtime checks, typed languages allow code to achieve the maximal performance. High-level abstractions such as idioms in functional and object-oriented programming can be expressed as type systems, rather than as restrictions on hardware registers. Furthermore, there are security policies such as information flow [96, 86] that can be enforced only by program analysis but not by execution monitoring [85].

Intermediate languages Programs written in typed languages obey the safety properties of the type system—as long as the compiler is correct. In 1962, McCarthy described compiler correctness as “one of the most interesting and useful goals for the mathematical science of computation” [53]. The SPIN operating system [13], for example, employs a single trusted compiler along with cryptographic signatures to ensure safety of kernel extensions. However, as Necula and Lee [69] conclude, the technology to prove the correctness of an optimizing compiler is still lacking.

Instead of a certified compiler that produces only correct code, program safety can still be achieved with a certifying compiler that produces a proof certifying the safety of generated code [17, 69]. During linking, the code and the certificate are verified against the safety policies, and hence the code will run without type-errors. The motivation is that the complexity of proof verification is in general lower than that of theorem proving [65]. Compared to a realistic optimizing compiler, a verifier can be designed to be simple and small. Note that, as long as the certified code passes the verifier and the verifier is correct, the correctness of a certifying compiler no longer needs to be trusted. This leads to our last criterion for language design: minimal trust of computing base [86].

2
A typed intermediate language exploits the idea of certifying compilation such that code from each intermediate stage of the compiler can be verified with a type system [59]. If types are preserved during compilation, the typing derivation of the program in the last stage can serve as a proof that the program is safe. A typed intermediate language also allows different source languages to share the same compiler backend or the same virtual machine. Furthermore, compilers can use types to perform more aggressive optimizations such as loop transformations and array bounds-check elimination [89, 93, 69].

Overview

This paper studies the tradeoffs of different intermediate languages in terms of complexity and expressiveness of their type systems. For the following three intermediate languages, we survey the related papers and explain their verification algorithms or typing rules: Java bytecode, typed assembly language, and proof-carrying code.

Java bytecode is designed to be compact for distributed computing and yet supports high-level constructs such as threads, garbage collection, and object-oriented programming. In "Java bytecode verification: algorithms and formalizations" [50], Leroy reviews bytecode verification algorithms and puts them into a common framework of dataflow analysis. Section 2 summarizes Leroy's paper and gradually introduces the concepts of formal methods, from abstract interpretation, dataflow analysis, to model checking.

Typed assembly language (TAL) has a RISC-style instruction set but a high-level type system. In "From System F to typed assembly language" [62], Morrisett et al. develop such a type system that guarantees control-flow and memory safety. In addition, they show that TAL's type system is expressive enough to preserve types during translation of the polymorphic lambda calculus (System F) and yet permits low-level optimizations such as register allocation, instruction selection, and instruction scheduling. Section 3 formally shows how the type system enforces the control-flow safety and illustrates its expressiveness by translating a program in System F to TAL.

Proof-carrying code (PCC) allows binary code to carry proofs of arbitrary policies. In "Proof-carrying code" [66], Necula proposes such a new architecture of typed intermediate language in which a meta-logical framework is used to specify safety properties as well as to represent the safety proof in the code. Section 4 discusses safety properties as enforceable security policies and implements control-flow safety in a logical framework as an example of PCC.

Section 5 and 6 concludes with a high-level comparison of the three languages in terms of the design criteria above. Historically, TAL was developed after PCC, but in this paper we present PCC after TAL so that we can describe the general framework from concrete examples.

2 Java bytecode

"Formal methods will never have any impact until they can be used by people who don't understand them." — Tom Melham
Java had been designed to be a programming language for writing browser applets but soon became a popular language for general purposes [7]. The language has brought many important language designs such as employing a strong type system for the Java source language and for its virtual machine bytecodes into the mainstream. Programmers, who do not need to understand the soundness theorem or the verification algorithm, can depend on the type system to ensure safe execution of Java programs. The success of the language in industry consolidates the application of type systems as a lightweight formal method in diverse programming domains, ranging from smartcards to servlets [50].

The Java Virtual Machine (JVM) is a stack-based abstract machine with registers for accessing method parameters and local variables [51]. An important feature of the JVM is its specification of the following safety properties of bytecode execution:

1. Type safety: instructions must receive arguments of expected types.
2. Stack safety: the stack must not underflow or overflow.
3. Register safety: indices to method parameters and local variables must be valid.
4. Address safety: branch and jump addresses must be valid.
5. Initialization safety: registers and objects must be initialized before use.

A JVM implementation can enforce these properties by dynamic checks during interpretation. For performance and static guarantees, although some checks such as array bound checks, null pointer checks, access control checks are still necessary, most realistic JVMs perform bytecode verification to statically ensure safety once and for all before execution.

In *Java bytecode verification: algorithms and formalizations* [50], Leroy reviews different algorithms and puts them into a common framework of dataflow analysis. This section summarizes Leroy’s paper and describes the algorithms based on abstract interpretation, dataflow analysis, and model checking.

### 2.1 Abstract interpretation

Instead of values, a type-level abstract interpreter uses types to simulate the execution of instructions at a coarser level. Given the types of JVM instructions and those of class methods, such an abstract interpreter can quickly approximate the runtime behavior of a program without executing it. The motivation behind abstract interpretation is its *dynamic correctness*: if a program satisfies a safety property during abstract interpretation, so will it during the actual execution.

Let us explain the concepts of abstract interpretation with the program in Figure 1. On the left is a complete Java program for the factorial function $f(x) = x \times f(x - 1)$ with $f(0) = 1$. On the right is the fragment of the compiled bytecode for $f$, where the first column indicates the program location (method offset) and the second column contains JVM instructions and their arguments. We annotate the instructions after the % symbol.
class Main {
    static int f (int x) {
        if (x == 0) {
            return 1;
        } else {
            return x * f(x-1);
        }
    }
}

Method int f(int)
    0 iload_0 % x
    1 ifne 6 % if (x!=0)
    4 iconst_1 % 1
    5 ireturn % return 1
    6 iload_0 % x
    7 iload_0 % x
    8 iconst_1 % 1
    9 isub % x-1
   10 invokestatic <int f(int)>
   13 imul % x*f(x-1)
   14 ireturn % return x*f(x-1)

Figure 1: Factorial function in Java source and bytecode

The instruction *iload 0* loads an integer value from register 0 (which is *x* here) to the operand stack, and *iconst 1* pushes an integer constant 1 to the operand stack. The instruction *ifne 6* branches to location 6 if the operand is zero and *ireturn* finishes the method with an integer. *isub* and *imul* are for integer subtraction and multiplication. The instruction *invokestatic <int f(int)>* invokes the static method *f*, whose both argument and return have type *int*.

We can specify the dynamic semantics of the instructions above in term of changes to the operand stack *S* and the register file *R* as follows:

iconst n : S # R → S, n # R
iload r : S # R, r → n → S, n # R, r → n
isub : S, n₁, n₂ # R → S, n₁ - n₂ # R
imul : S, n₁, n₂ # R → S, n₁ × n₂ # R
invokestatic <t m(t₁, . . . , tₙ)> : S, v₁, . . . , vₙ # R → S, v # R if v = m(v₁, . . . , vₙ)

Here *S* is a linear list of values and we use *S, n* to represent the new stack of *S* with the value *n* pushed on top. *R* is finite mapping from indices to values and we use *R, r → n* to specify that *r* is a valid register of value *n*. We write *ϵ* for the empty stack or the empty register file. We use this loose notation for definitions just to explain abstract interpretation; in Section 3.1, we will define the syntax and the semantics more formally for a similar system. The semantics of control transfer instructions (*ifne* and *ireturn*) are described in the next subsection.

Now we can compare the rules above for the actual execution with the following rules for abstract interpretation:

iconst n : S # R → S, int # R
iload r : S # R, r:int → S, int # R, r:int
isub : S, int, int # R → S, int # R
imul : S, int, int # R → S, int # R
invokestatic <t m(t₁,...,tₙ)> : S, t₁',...,tₙ' # R → S, t # R if t₁' <: t₁,...,tₙ' <: tₙ

For abstract interpretation, S becomes a list of types and R a finite map from indices to
types, instead of values. From the types of the instruction, the result type of iconst, iload
and isub must be int. Similarly, the parameter and return types of method invocation can
be determined from the instruction arguments (<t m(t₁,...,tₙ)> ) without executing the
method.

The rules above specify the following safety properties of execution: (1) for iload r,
register r must be an integer (R, r : int), (2) for isub and imul, the top two operands
must both be integers (S, int, int), and (3) for invokestatic, the actual argument types
(t₁',...,tₙ') from the operand stack must be subtypes (t₁' <: t₁,...,tₙ' <: tₙ) of the formal
argument types (t₁,...,tₙ) from the method signature. In addition, the register indices
must be valid and the stack must not underflow.

Note that we specify type errors by the absence of rules such that if a program is in a
state with no rule to apply, then the program does not pass the bytecode verification and will
be rejected. By the dynamic correctness of abstract interpretation, a program that passes
the verification will satisfy the safety properties above in the actual execution.

2.2 Dataflow analysis

The abstract interpretation checks the safety properties only for straight-line codes. To
account for control transfer instructions, however, we need to use dataflow analysis to model
non-linear executions in the control-flow graph.

When the execution is linear, the input state (stack and register file) of an instruction
is the output state of the preceding instruction. The initial state of a method, written
as ϵ ≠ t₁,...,tₙ, ⊤,...,⊤, is an empty stack (ϵ) and register file with types of method
parameters (t₁,...,tₙ) and with the top type ⊤ for uninitialized local variables.

When the execution is non-linear, we must consider the output states of all possible
predecessors of an instruction, not just that of the immediate preceding instruction. The
idea of dataflow analysis is to merge all the output states by taking the least upper bound
(lub) of their types: for example, if both classes C₁ and C₂ extend C₀, and if a register
from one predecessor instruction has type C₁ and that from another has C₂, then the merged
output state has type C₀ = lub(C₁, C₂) in that register. We merge two stacks pointwise and
check if both stacks have the same size, and similarly for the register file.

More formally, a dataflow analysis is specified by a set of dataflow equations. We write
I(ℓ) for the instruction at program location ℓ, write IN(ℓ) for the input state of the instruc-
tion at ℓ, and write OUT(ℓ) for the output state. In particular, IN(0) specifies the input
state at the beginning of the method. The dataflow equations for bytecode verification are:

I(ℓ) : IN(ℓ) → OUT(ℓ)
class Main {
    static void f () {
        int x, y; 
        try {
            x = 0;
        } finally {
            y = x;
        }
    }
}

Method void f() with [(0,4,Exception,8)]

0 icost_0 % 0
1 istore_0 % x=0
2 jsr 14
5 goto 17
8 astore_2 % catch (Exception e)
12 aload_2 % Exception e
13 athrow % throw e
14 astore_3 % return address = 5
15 ret 3
17 ilod_0 % x
18 istore_1 % y = x
19 return

Figure 2: Exception handling in Java source and bytecode

\[
\begin{align*}
\text{IN}(\ell) &= \text{lub} \{ \text{OUT}(\ell') \mid \ell' \in \text{PRED}(\ell) \} \\
\text{IN}(0) &= \epsilon \# t_1, \ldots, t_n, \top, \ldots, \top
\end{align*}
\]

where \( \text{PRED}(\ell) \) is the set of predecessors of instruction at \( \ell \). We can readily compute \( \text{PRED}(\ell) \) from the target addresses of control transfer instructions. The equations can then be solved by fixpoint iteration [63], and the verification succeeds if there exists some solution.

Subroutines The dataflow analysis above, however, does not work for programs with subroutine instructions \textit{jsr} (jump to subroutine) and \textit{ret} (return from subroutine). A subroutine is similar to a method invocation, but the stack and register file of the caller are shared with the callee. Exception handling \textit{try...finally} in Java is compiled into subroutines because the exception table of the method is set up with target locations inside the same method.

Consider the program in Figure 2. This example illustrates a typical compilation of exception handling into subroutines. Here we have four blocks in the bytecode that correspond to the \textit{try} block, the implicit \textit{catch} block, the \textit{finally} block, and the last statement in the Java source. The key point is to illustrate that the \textit{try} block and the \textit{catch} block share the same subroutine for the \textit{finally} block but the two blocks do not initialize the same set of variables.

Let us first explain the detail of the compilation. Instructions at 0-1 correspond to the \textit{try} block, while those at 14-15 correspond to the \textit{finally} block. The instruction \textit{jsr 14} pushes the return address, which is 5 in this example, to the stack and jumps to 14. Inside the \textit{finally} block, the return address is first saved to register 3 so that the instruction \textit{ret} will use it later for returning control. Instructions at 8-13 correspond to the
implicit catch block that first saves the exception object into register 2 (astore.2), calls the finally block (jsr 14), and re-throws the exception (aload.2 and athrow). Each method has an exception table enumerating the scope and the type of exceptions: in our example, 
\[(0, 4, \text{Exception}, 8)\]
indicates that if an exception is raised between 0-4 and the exception is a subclass of Exception (that is, any exception), then the control is transferred to the catch block at 8.

The first challenge in analyzing programs with a subroutine is to determine the successor instructions of the subroutine. Since the return address for instructions jsr and ret is stored in the stack and register file as a first-class value, there seems to be no syntactic way to associate the return address with the jsr/ret pair. The second challenge is that both the try block and the catch blocks call the subroutine for the finally block, leading to precision loss of stack and register file types after merging. In the example, \(x\) is initialized in the try block \((R(0) = \text{int})\) but not in the catch block \((R(0) = \top)\) and thus their merged type is uninitialized \((R(0) = \top)\). Even though register 0 is not used in the finally block, the type information is lost when the control flows to the finally block and comes back: the instruction iload.0 at 17 for \(y = x\) does not pass the verification because \(R(0) = \top\).

**Polyvariant analysis**

One solution is to extend the dataflow analysis for subroutines to be context-sensitive such that instructions inside the subroutines are analyzed differently per call site, without merging the states of the callers. This approach is also called polyvariant bytecode verification as the state \(S \not= R\) at each program location is now parameterized by the subroutine call stack \(C\), which is also called a contour. A dataflow equation at \(\ell\) becomes
\[
I(\ell) : \text{IN}(\ell; C) \rightarrow \text{OUT}(\ell; C').
\]

In our example, we will analyze instructions at 14-15 in the call stack \(C = 5\) and again in \(C = 12\), where 5 and 12 are the return addresses after jsr, giving the output state \(\text{OUT}(15; 5)\) with \(R(0) = \text{int}\) and another output state \(\text{OUT}(15; 12)\) with \(R(0) = \top\). That is, we do not merge the two states to have \(R(0) = \top\) as before, and allow the instructions after 5 to use the initialized register 0.

More formally, for the instruction jsr \(\ell\) at \(\ell_1\) followed by an instruction at \(\ell_2\) in the call stack \(C\), the equation is:
\[
\text{IN}(\ell_2; C, \ell) = S, \text{RA}(\ell_2) \not= R \quad \text{if} \quad \text{OUT}(\ell_1; C) = S \not= R
\]

The equation says that the analysis pushes \(\ell_2\) to \(C\) and pushes \(\text{RA}(\ell_2)\) to the operand stack \(S\), where \(\text{RA}(\ell_2)\) is the type representing the return address at \(\ell_2\). Note that \(\text{RA}(\ell_2)\) is a singleton type, which contains the same amount of information at the type level as at the value level [8]. On the other hand, for the instruction ret \(n\) at \(\ell_1\) in \(C\), the equation is:
\[
\text{IN}(\ell_2; C) = \text{OUT}(\ell_1; C', \ell_2, C) = S \not= R, n: \text{RA}(\ell_2)
\]

The equation says that, from the return address \(\ell_2\) at register \(n\), the analysis pops the call stack \(C', \ell_2, C\) at \(\ell_1\) until the address \(\ell_2\) to get \(C\), and then propagate the output state at \(\ell_1\) back to the input state at \(\ell_2\) in \(C\).
2.3 Model checking

The contour-based polyvariant algorithm above successfully verifies subroutines without knowing their structures, but the algorithm may keep too many states per program location. In fact, the termination of the algorithm is not guaranteed and there exist programs that cause the verification to loop [50].

Based on model checking, the last algorithm here handles subroutines yet remains decidable. The intuition behind model checking is to explore all reachable states of the abstract interpreter. We first define the successor relation \( \ell|S \# R \rightarrow \ell'|S' \# R' \) for the transition function \( I(\ell): S \# R \rightarrow S' \# R' \) for the transition function in Section 2.1:

\[
\begin{align*}
&\ell|S \# R \rightarrow \ell'|S',R' \quad \text{if} \quad I(\ell): S \# R \rightarrow S' \# R' \quad \text{and} \quad \ell' \in \text{SUCC}(\ell) \\
&\ell|S \# R \rightarrow \bot \quad \text{if} \quad I(\ell): S \# R \not\rightarrow
\end{align*}
\]

where \( \text{SUCC}(\ell) \) is the set of successors of instruction at \( \ell \) and where \( \bot \) represents the “stuck” state in abstract interpretation. We then compute all reachable states by fixpoint iteration of the successor relation with the initial state \( 0|\epsilon \# t_1,\ldots,t_n,\top,\ldots,\top \) (similar to the \( \text{IN}(0) \) in Section 2.2). The verification succeeds if the closure of reachable states does not include \( \bot \).

For instance, the successor relation for instructions \( \text{jsr} \) and \( \text{ret} \) are:

\[
\begin{align*}
&\ell|S \# R \rightarrow \ell'|S,{RA}(\ell+3) \# R \quad \text{if} \quad I(\ell) = \text{jsr} \ell' \\
&\ell|S \# R \rightarrow \ell'|S \# R \quad \text{if} \quad I(\ell) = \text{ret} r \quad \text{and} \quad R(r) = {RA}(\ell') \\
&\ell|S \# R \rightarrow \bot \quad \text{if} \quad I(\ell) = \text{ret} r \quad \text{and} \quad R(r) \neq {RA}(\ell')
\end{align*}
\]

because the size of instruction \( \text{jsr} \ell' \) is always 3. Applying the algorithm on the example in Figure 2, the closure will include the following transitions and states. We can then deduce that the instruction \( y = x \) at 18 will not reach \( \bot \).

\[
\begin{align*}
2|\epsilon \# 0:\text{int} & \rightarrow 14|{RA}(5) \# 0:\text{int} \\
9|\epsilon \# 2:\text{Exception} & \rightarrow 14|{RA}(12) \# 2:\text{Exception} \\
14|\epsilon \# 0:\text{int},3:RA(5) & \rightarrow 5|\epsilon \# 0:\text{int},3:RA(5) \\
14|\epsilon \# 2:\text{Exception},3:RA(12) & \rightarrow 12|\epsilon \# 2:\text{Exception},3:RA(12)
\end{align*}
\]

Note that the number of reachable states are finite since the number of program locations \( \ell \), the sizes of the stack \( S \) and register file \( R \) with distinct types \( t \) are all fixed. Therefore, the algorithm always terminates.

2.4 Discussion

Schmidt [84] formalizes the idea that a dataflow analysis is a model checking of abstract interpretations, providing a common framework for bytecode verification algorithms. Leroy [50] also surveys many other variants of dataflow analyses for bytecode verification [35, 83, 18, 49, 72, 33, 90, 46, 11] and their formalizations in computer proof systems such as Coq [28] and
Isabell [74]. Bytecode verification can also be formulated as a type-checking problem such that the set of equations of a dataflow analysis algorithm corresponds to a type inference algorithm [91, 32, 33].

Coglio [18] claims that the algorithm based on model checking in Section 2.3 is the most precise static analysis as it considers all execution paths but does not compute expression values. To improve the exponential time complexity of the algorithm, Leroy [50] uses widening functions to merge equivalent states into the same class and proves that the widening algorithm is sound and complete with respect to the original algorithm.

The difficulty and complexity of Java bytecode verification comes as a price from the design of the type system and the desire to keep the bytecode compact. The size requirement is critical in distributed computing and small devices such as Java smartcards, but the requirement limits on the amount of type information in the bytecode. Leroy [49] argues that, by using off-card code transformations to normalize branch structures and register allocations in bytecode, his on-card verification algorithm simplifies the fixpoint equations and takes much less working memory.

Future research of Java bytecode verification aims to establish more advanced static properties such as resource bounds on memory usage or running time of applets [25, 41]. Moreover, the JVM checks access control dynamically with stack inspection [99]; recent work [42, 80, 9, 31, 10] employs type systems to statically verify such information flow. Another direction is to adapt the results to the closely related technologies, C♯ and CIL [55, 56].

3 Typed assembly language

“When bad languages do good types...” — Anonymous [79]

Traditionally, type systems have been designed for high-level languages such as ML [47, 57] and Haskell [44] to provide programmers with abstraction and type safety. Low-level languages such as C and assembly languages are considered “bad” in the sense that their type systems, if any, are not strong enough to guarantee even basic safety properties. New type systems with high-level safety properties, however, have been retrofitted into these low-level languages. In From System F to typed assembly language [62], Morrisett et al. develop a typed assembly language (TAL) that guarantees control-flow and memory safety.

Furthermore, TAL’s type system is expressive enough to preserve types during the translation of the polymorphic lambda calculus (System F). That is, the type system at such low level is still enforcing high-level language abstractions such as functions and polymorphism. With the type information during compilation, compilers can perform many aggressive optimizations such as continuation passing [26], closure conversion [58], unboxing [48], subsumption elimination [21], and region inference [14]. Also, experience [93, 69] shows that typed intermediate languages are useful in debugging complicated optimizations.

TAL’s instruction set is based on RISC so that primitives can be used together to support different programming paradigms and to permit low-level optimizations such as reg-
ister allocation, instruction selection, and instruction scheduling. In contrast, the CISC-
style instruction set of Java bytecode presumes a Java-like source language with heavy-
weight object-oriented and threading constructs. For instance, a JVM does not have tail-
recursive calls, polymorphism, or lightweight closures to efficiently support functional pro-
gramming [12, 54, 87]. On the other hand, many JVM instructions are very complex and
prohibit optimizations at the bytecode level: the instruction \texttt{invokevirtual} in Java byte-
code needs to load appropriate classes, dispatch the virtual method, set up a call frame
(new stack and register file), install appropriate exception handlers, and restore the envi-
ronment upon return. Most virtual machines need to employ just-in-time compilation to
optimize bytecode into native machine code, but such compilation is not type-preserving or
verified [1, 92].

In this section, we formally define the syntax and semantics of a subset of TAL and show
how control-flow safety is enforced. We then illustrate the expressiveness of its type system
by translating a program in System F to TAL while keeping type information along all steps.
The presentation and the examples are taken from Morrisett’s papers [62, 60].

3.1 TAL-0 and control-flow safety

Control-flow safety, like address safety in Section 2, ensures that a program does not jump
to arbitrary machine addresses but only to well-defined entry points. Control-flow safety
additionally enforces the type safety of the stack and register file at those entry points. We
will present the language TAL-0 [60], a subset of TAL without memory management, and
show how its type system enforces control-flow safety.

The following is the formal syntax of TAL-0 in BNF:

\[
\begin{align*}
\text{Operands} & : \quad v ::= r \mid n \mid ℓ \\
\text{Instructions} & : \quad i ::= \text{mov} r, v \mid \text{add} r, r, v \mid \text{bz} r, v \\
\text{Blocks} & : \quad I ::= \text{jmp} v \mid i; I \\
\text{Heap} & : \quad H ::= \epsilon \mid H, ℓ \mapsto I \\
\text{Register file} & : \quad R ::= \epsilon \mid R, r \mapsto n \mid R, r \mapsto ℓ
\end{align*}
\]

To simplify the presentation, only three instructions are supported: (1) \texttt{mov} \( r \ v \) moves

---

**Figure 3: Product function in TAL-0**

\[
\begin{align*}
\text{prod:} & \quad \text{mov} r_3, 0; \quad \% \text{initialize result} \\
& \quad \text{jmp} \text{loop} \quad \% \\
\text{loop:} & \quad \text{bz} r_1, \text{done} \quad \% \text{branch if} \ r_1 \ == 0 \\
& \quad \text{add} r_3, r_3, r_2 \quad \% \ r_3 = r_3 + r_2 \\
& \quad \text{add} r_1, r_1, -1 \quad \% \ r_1 = r_1 - 1 \\
& \quad \text{jmp} \text{loop} \quad \% \\
\text{done:} & \quad \text{jmp} r_4 \quad \% \text{return}
\end{align*}
\]
operand \( v \) into register \( r \), (2) \( \text{add } r_1, r_2, v \) adds the value of \( r_2 \) and \( v \), and puts the result into \( r_1 \), and (3) \( \text{bz } r \) \( v \) branches to operand \( v \) if \( r \) is zero. An operand can be a register \( r \), an integer \( n \), or a program location \( \ell \). A special instruction \( \text{jmp } v \) unconditionally jumps to operand \( v \) and is used to delimit an instruction block. Similar to the stack and register file in Section 2.1, heap \( H \) maps locations to instruction blocks while register file \( R \) maps indices to integers or locations. We write \( I \mid H \# R \) for the machine state with instruction block \( I \), heap \( H \) and register file \( R \).

As an example, Figure 3 shows the product function written in TAL-0. The code assumes that the inputs are in registers \( r_1 \) and \( r_2 \), the output in \( r_3 \), and the continuation in \( r_4 \). In other words, \( r_4 \) holds the location to jump to when the program finishes computing \( r_3 = r_1 \times r_2 \).

The dynamic semantics of TAL-0 are specified by the evaluation rules of the form \( I \mid H \# R \rightarrow I' \mid H' \# R' \), as shown in Figure 4, which says that machine state \( I \mid H \# R \) steps to state \( I' \mid H' \# R' \). Note that not all possible machine states have an evaluation rule: for example, there is no rule for \( \text{jmp } n; \ I \mid H \# R \) (jumping to an integer) or for \( \text{add } r_1, r_2, \ell \) (adding locations). We will next use a type system to rule out these “stuck” states in the dynamic semantics such that a well-typed program always progresses with some evaluation rule until halting.

The static semantics of TAL-0 are specified by the types and typing rules in Figure 5. We use \( \text{int} \) as the base type such that the typing rule \( H \# R \vdash n : \text{int} \) (T-Int) says that, under any heap type and any register file type, the integer literal \( n \) has the integer type \( \text{int} \). Here \( H \) and \( R \) are type contexts that map location \( \ell \) or register \( r \) to type \( t \) (T-Lab and T-Reg). Note that we use the meta-variable \( H \) for both the actual heap of code during evaluation and the heap type context during typing, and similarly \( R \) for register file. When values and types are used in the same rule, we use the convention that \( H_0, R_0 \) refer to values and \( H, R \) to types (T-Heap, T-Regs and T-State).

For instruction \( i \), we use its precondition on register file type \( R_1 \) and its postcondition on register file type \( R_2 \) under the heap type \( H \) to assign the type \( R_1 \rightarrow R_2 \) (T-Mov, T-Add and

---

\[
\begin{align*}
\text{jmp } \ell & \mid H \# R \rightarrow H(\ell) \mid H \# R \quad \text{(E-JmpL)} \\
\text{jmp } r & \mid H \# R \rightarrow H(R(r)) \mid H \# R \quad \text{(E-JmpR)} \\
\text{mov } r_1, \ell; & I \mid H \# R \rightarrow I \mid H \# R, r_1 \mapsto \ell \quad \text{(E-MovL)} \\
\text{mov } r_1, n; & I \mid H \# R \rightarrow I \mid H \# R, r_1 \mapsto n \quad \text{(E-MovN)} \\
\text{mov } r_1, r_2; & I \mid H \# R \rightarrow I \mid H \# R, r_1 \mapsto R(r_2) \quad \text{(E-MovR)} \\
\text{add } r_1, r_2, n; & I \mid H \# R \rightarrow I \mid H \# R, r_1 \mapsto R(r_2) + n \quad \text{(E-AddN)} \\
\text{add } r_1, r_2, r_3; & I \mid H \# R \rightarrow I \mid H \# R, r_1 \mapsto R(r_2) + R(r_3) \quad \text{(E-AddR)} \\
\text{bz } r_1, \ell; & I \mid H \# R \rightarrow H(\ell) \mid H \# R \quad \text{if } R(r_1) = 0 \quad \text{(E-BzL)} \\
\text{bz } r_1, r_2; & I \mid H \# R \rightarrow H(R(r_2)) \mid H \# R \quad \text{if } R(r_1) = 0 \quad \text{(E-BzR)} \\
\text{bz } r_1, v; & I \mid H \# R \rightarrow I \mid H \# R \quad \text{if } R(r_1) \neq 0 \quad \text{(E-Bnz)}
\end{align*}
\]

Figure 4: Evaluation rules for TAL-0
Operand types  \( t ::= \text{int} \mid R \mid \alpha \mid \forall \alpha. t \)

Heap types  \( H ::= \epsilon \mid H, \ell : t \)

Register file types  \( R ::= \epsilon \mid R, r : t \)

Operand typing  \( H \# R \vdash v : t \)

Instruction typing  \( H \vdash i : (R \rightarrow R) \)

Block typing  \( H \vdash I : t \)

Heap typing  \( \vdash H_0 : H \)

Register file typing  \( H \vdash R_0 : R \)

State typing  \( \vdash (I | H_0 \# R_0) : H \# R \)

\[
\begin{align*}
H \# R \vdash n : \text{int} & \quad \text{(T-Int)} \\
H \# R \vdash \ell : H(\ell) & \quad \text{(T-Lab)} \\
H \# R \vdash r : R(r) & \quad \text{(T-Reg)} \\
H \# R \vdash v : \forall \alpha. t_1 & \quad \text{(T-Inst)} \\
H \# R \vdash v : t_1[\alpha \mapsto t_2] & \\
H \# R \vdash v : t & \quad \text{(T-Mov)} \\
H \vdash \text{mov} r, v : (R \rightarrow R, r : t) & \\
H \# R \vdash r_2 : \text{int} & \quad \text{(T-Add)} \\
H \# R \vdash v : \text{int} & \quad \text{(T-Bz)} \\
H \vdash \text{add} r_1, r_2, v : (R \rightarrow R, r_1 : \text{int}) & \\
H \# R \vdash v : R & \\
H \# R \vdash R_0 & \quad \text{(T-Regs)} \\
H \vdash R_0 & \quad \text{(T-State)} \\
H \# R \vdash v : R & \quad \text{(T-Jmp)} \\
H \vdash \text{jmp} v : R & \quad \text{(T-Seq)} \\
H \vdash i : (R_1 \rightarrow R_2) & \quad \text{(T-Gen)} \\
H \vdash I : t & \quad \text{(T-Heap)} \\
\forall \ell \in \text{dom}(H). H \vdash H_0(\ell) : H(\ell) & \quad \text{(T-Heap)} \\
\forall r \in \text{dom}(R). H \# \epsilon \vdash H_0 & \quad \text{(T-Regs)} \\
H \vdash (I | H_0 \# R_0) : H \# R & \quad \text{(T-State)}
\end{align*}
\]

Figure 5: Types and typing rules for TAL-0
T-Bz). For instruction block jmp v or i; I, we check if the postcondition R₂ of an instruction of type R₁ → R₂ matches the precondition R₁ of the next instruction (T-Jmp and T-Seq). A machine state I | H₀; R₀ is well-typed if its instruction block I, heap H₀, and register file R₀ are all well-typed (T-State). We type-check heap and register file pointwise for locations or registers in their typing domains (T-Heap and T-Reg). Note that a heap is checked under the assumption of its own heap type because its instruction blocks can be mutually recursive (that is, code at ℓ₁ can jump to ℓ₂, which may jump back to ℓ₁).

Polymorphic types There remain two types (α and ∀α.t) and two typing rules (T-Gen and T-Inst) to be explained. They are for universal quantification, or parametric polymorphism. We will use an example to give the intuition behind how polymorphic types are important in type-checking the control transfer of instructions blocks.

Consider a program for computing 2 × 3 written as jmp prod | r₁ → 2, r₂ → 3, r₃ → 0, r₄ → halt # H₀ where halt is the location of code to halt the execution and H₀ is the product function in Figure 3. We want to show that the program type-checks under the following heap and register file types:

\[
H = \text{prod:R, loop:R, done:R} \\
R = r₁: \text{int, r₂: int, r₃: int, r₄: (∀α. r₁: int, r₂: int, r₃: int, r₄: α)}
\]

For example, the typing derivation for H ⊢ bz r₁, done : (R→R) is

\[
\begin{align*}
H &\not\vdash_\text{T-Reg} r₁: \text{int} & H &\not\vdash_\text{T-Reg} \text{done:R} \\
\frac{H \vdash bz r₁, done : (R→R)}{H \not\vdash_\text{T-Bz} r₁, done : (R→R)}
\end{align*}
\]

as R(r₁) = int and H(done) = R. Derivations for other parts of the program are similar, except that for the last instruction jmp r₄. We need to prove that H # R ⊢ r₄ : R (T-Jmp). But this requires proving that R(r₄) = R, which has no solution in a simple type system. With polymorphic types, we can generalize the type of r₄ to be a type variable α such that H # R ⊢ r₄ : (∀α. r₁: int, r₂: int, r₃: int, r₄: α) (T-Gen). Only at the instruction jmp r₄ is the polymorphic type instantiated to be (∀α. r₁: int, r₂: int, r₃: int, r₄: α)[α ↦ R] = r₁: int, r₂: int, r₃: int, r₄: R (T-Inst).

The following soundness theorem [60] guarantees that if a program type-checks statically, its execution will never get “stuck”.

**Theorem 1 (TAL-0 soundness)**

1. **Preservation:** If ⊢ (I | H₀ # R₀) : H # R and I | H₀ # R₀ → I’ | H₀ # R₀, then ⊢ (I’ | H₀ # R₀) : H # R. That is, a well-typed program keeps its type during evaluation.

2. **Progress:** If ⊢ (I | H₀ # R₀) : H # R, then either I = jmp halt or I | H₀ # R₀ → I’ | H₀’ # R₀’ for some I’,H₀’,R₀’. That is, a well-typed program progresses with some evaluation rule until halting.
3.2 Translation from System F

Polymorphic lambda calculus (System F) [34, 81] is a formalism for functional programming languages with universal quantification. The type system of System F is expressive enough to encode the pure subset of modern languages such as ML and Haskell. In order to illustrate the expressiveness of TAL’s type system, we will show how a source program in System F is translated into TAL while the type information is preserved along the translation.

Let us rewrite the factorial function in Figure 1 to compute the factorial of 6 in System F:

\[
(\lambda f \ (x : \text{int}) \ . \ \text{ifz} \ x \ 1 \ (x \times (f \ (x - 1)))) \ 6
\]

We write \( \lambda f \ (x_1 : t_1, \ldots, x_n : t_n) \ . \ e \) for a recursive function named \( f \) with typed parameters \( x_1 : t_1, \ldots, x_n : t_n \) and body expression \( e \). The expression \( \text{ifz} \ e_1 \ e_2 \ e_3 \) branches to \( e_2 \) or \( e_3 \) depending whether if \( e_1 \) is zero. A function of type \( t_1 \to \ldots \to t_n \to t \) takes arguments of type \( t_1, \ldots, t_n \) and returns a result of type \( t \). The factorial function, for example, has type \( \text{int} \to \text{int} \). We write function applications such as \( f \ (x - 1) \) and \( (\lambda f \ldots) \ 6 \) by juxtaposition.

The intuition behind the translation is to decompose System F’s primitives such as function applications, local functions and automatic memory management into TAL’s primitives such as jumps, heap and register file.

**Continuation passing** The first step is to make all continuations explicit by passing continuation as an argument to the function and, instead of returning to the outer context, by calling the continuation when the function is done. For example, the factorial function above becomes

\[
(\lambda f \ (x : \text{int}, k : \text{int} \to \text{void}) \ . \\
  \text{ifz} \ x \ (k \ 1) \\
  \text{let} \ x_0 = x - 1 \text{ in} \\
  f \ x_0 \ (\lambda f_1 \ (x_1 : \text{int}) \ . \ \text{let} \ x_2 = x \times x_1 \text{ in} \ k \ x_2) \\
6 \ (\lambda f_2 \ (x_2 : \text{int}) \ . \ \text{halt} \ x_2)
\]

We introduce a primitive \( \text{halt} \ e \) that stops the program with the result \( e \) (which is \( x_2 \) here). The factorial function \( f \) now takes continuation \( k \) as an additional argument and calls \( k \ 1 \) for the base case when \( x = 0 \). When \( x \neq 0 \), we recursively call \( f \) with \( x - 1 \) and the continuation \( f_1 \) that multiplies the result so far, \( x_1 \), with the input \( x \).

We also introduce \( \text{let} \)-bindings to specify the order of computation. Compared to \( f \ (x - 1) \ldots \) in System F, the expression \( \text{let} \ x_0 = x - 1 \text{ in} \ f \ x_0 \ldots \) first computes \( x - 1 \) and calls \( f \) with this value. We will see how a sequence of \( \text{let} \)-bindings and function calls can be readily translated into assembly code. Furthermore, we write the type of \( f_2 \) to be \( \text{int} \to \text{void} \) to emphasize that the function does not return any value. Similarly, \( f \) now has the type \( \text{int} \to (\text{int} \to \text{void}) \to \text{void} \).
let \( f = \lambda (e::<>, x:\text{int}, k:t_0). \)
   \[
   \begin{align*}
   \text{if} \ z \ x \\
   (\text{let} \ (\alpha, kk) = \text{unpack} \ k \ \text{in} \\
   \text{let} \ e_0 = \text{prj} \ 1 \ kk \ \text{in} \\
   \text{let} \ k_0 = \text{prj} \ 2 \ kk \ \text{in} \\
   \text{let} \ x_0 = 1 \ \text{in} \\
   k_0 \ e_0 \ x_0) \\
   (\text{let} \ e_0 = <x, k> \ \text{in} \\
   \text{let} \ kk_0 = <e_0, f_1> \ \text{in} \\
   \text{let} \ k_0 = \text{pack} \ (<\text{int}, t_0>, kk_0):t_0 \ \text{in} \\
   \text{let} \ x_0 = x - 1 \ \text{in} \\
   f \ e_0 \ x_0 \ k_0 \\
   )
   \end{align*}
   \]
   \[
   \text{let} \ f_2 = \lambda (e::<>, x:\text{int}). \)
   \[
   \text{halt} \ x
   \]
   \]

Figure 6: Factorial function after closure passing (where \( t_0 = \exists \alpha. \alpha \rightarrow \text{int} \rightarrow \text{void} \))

**Closure passing**  The second step is to lift local functions that access lexical variables from their enclosing functions to be global functions. In our example, after continuation passing, the factorial function contains the local function \( \lambda f_1 \ (x_1:\text{int}). \text{let} \ x_2 = x \times x_1 \ \text{in} \ k \ x_2, \) which accesses \( x \) and \( k \) in the enclosing function \( f. \) The intuition behind closure passing (or, closure conversion [6]) is to translate a local function such that it takes an additional parameter \( e \) as an environment of lexical variables and accesses those variables explicitly through the environment.

Figure 6 shows the factorial function after closure passing. The function \( f \) on the left column puts \( x \) and \( k \) into environment \( e_0 \) as a tuple of values \( e_0 = <x, k> \) such that \( f_2 \) on the right column will project out \( x_0 \) and \( k_0 \) as needed. Since all functions are global definitions now, we can define them with top-level \texttt{let}-bindings. We also write the computation \( f \ 6 \ f_2 \) explicitly as the function \texttt{main}.

Separating lexical variables from code, however, complicate the types of continuations. Before closure passing, both \( f_1 \) and \( f_2 \) have type \( \text{int} \rightarrow \text{void} \) and thus the continuation parameter \( k \) of \( f \) has type \( \text{int} \rightarrow \text{void} \). Now, with the extra environment parameter, \( f_1 \) has type \( <\text{int}, t_0> \rightarrow \text{int} \rightarrow \text{void} \) while \( f_2 \) has type \( <> \rightarrow \text{int} \rightarrow \text{void} \), where \( t_0 \) the type for the continuation. Similar to using universal quantification in the last subsection, we need to abstract the environment type from the continuation type. But we also need to pack the additional type information \(<\text{int}, t_0>\) into \( t_0 \), so that we can type check with the actual environment type after unpacking the continuation.

The solution is to use *existential quantification* that encapsulates extra type information
to be inspected at the call site. In Figure 6, f packs the type information \(<\text{int}, t_0>\) with the
continuation \(kk_0 = <e_0, f_1>\) for the recursive call while main packs \(<\text{int}>\) with \(kk_0 = <e_0, f_2>\).
To use the continuation, we unpack to find out its type information \(\alpha\) in additional to its
value \(kk\) by the new let-binding expression \(\text{let } (\alpha, kk) = \text{unpack } k \text{ in}\). The continuation
parameter \(k\) of \(f\) can now have the existential type \(t_0 = \exists \alpha. \alpha \rightarrow \text{int} \rightarrow \text{void}\).

**Memory allocation**  The remaining step of translating the factorial function into TAL
is to do register and heap allocation. We assume that an integer or a location fits into
a register, but a tuple (such as a continuation or an environment) requires heap storage
through explicit allocation.

Figure 7 shows the complete code for the factorial function in TAL, annotated with
corresponding lines of code in Figure 6. To support automatic memory management for
memory safety, TAL introduces an instruction \(\text{malloc } r : t\) to allocate heap space large
enough for type \(t\) and store the pointer into register \(r\). Memory safety ensures that locations
to heap are always valid. \(\text{malloc}\) can be implemented by linking to a conservative garbage
collector [15]. For example, the instruction block \(f_0\) (the first branch of ifz) contains the
translation of \(\text{let } e_0 = <x, k>\) in: we first allocate space with \(\text{malloc } r_1 : <\text{int}, t_1>\), and then
initialize fields with \(\text{store } r_1[0], r_2\) and \(\text{store } r_1[1], r_3\), assuming that \(r_2 = x\) and \(r_3 = k\).

Two additional instructions \(\text{pack } r_1, (t_1, r_2) : t_2\) and \(\text{unpack } (\alpha, r_1), r_2\) are for direct
translation of \(\text{let } x_1 = \text{pack } (t_1, x_2) : t_2\) and \(\text{let } (\alpha, x_1) = x_2\). These two instructions
simply annotate existential types in type-checking and can be implemented as \(\text{mov}\).

**3.3 Discussion**

Typed assembly language is a form of proof-carrying code (see the next section), but it
provides a fully automatic procedure for generating certified code. TAL starts with a well-
typed program in a high-level language such as System F and transforms its types as a proof
of safety, instead of re-constructing it as in bytecode verification of Java or theorem proving
of PCC. TAL’s semantics is so close to the machine code that TAL may as well be called a
typed target language, instead of a typed intermediate language.

Morrisett et al. [62] formally define the semantics of the intermediate language after
continuation passing and after closure passing, allowing compilers to aggressively optimize
between any of the translation steps. Furthermore, their simplified typing rule for polymor-
phic closure conversion is a significant contribution over the previous approach [58].

The idea of assigning polymorphic types to continuations is also used in type-checking
subroutines in Java bytecode [73]. Other than solving the recursive type equations of contin-
uations and register files, polymorphism provides a least upper bound for register file types
at merge points of a control-flow graph (see Section 2.2) [37, 60]. Alternatives are subtyping
or recursive types, but Morrisett et al. [60] argues that polymorphism has other advantages
such as specifying calling convention for registers. For example, an instruction block of type

\[
\forall \alpha_1. (r_1: \alpha_1, r_2: (\exists \alpha_2. (\alpha_2, (r_1: \alpha_1, r_2: \alpha_2))))
\]

17
Figure 7: Factorial function in TAL (where \( t_0 = \exists \alpha. \alpha \rightarrow \text{int} \rightarrow \text{void}, t_1 = \exists \alpha. (\alpha, r_1: \alpha, r_2: \text{int})), \) and \( t_2 = \langle \text{int}, t_1>, (r_1: \langle \text{int}, t_1>, r_2: \text{int}) \rangle \)
where $\alpha_1$ and $\alpha_2$ are fresh and $r_2$ contains the continuation, must be parametric in $\alpha_1$ [97]. This means that the instruction block can use register $r_1$ for holding other values, but the block must save and restore the register upon return (callee-saves registers).

Extended type systems based on TAL’s have been developed to guarantee secure information flow [16], or power consumption in grid computing [95]. Other research directions include using dependent types or refinement types [102, 100, 101] to expose array-bound checks for optimizations, or formalizing memory management [24, 75, 94] with a typed garbage collection.

4 Proof-carrying code

“The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.” — Mark Mannasse

Both Java bytecode and typed assembly language enforce control-flow safety and memory safety, but each has a predefined type system for a fixed set of safety properties. Every quest for a richer type system to allow more optimizations or security guarantees requires new definitions of syntax, semantics, and policies as well as new proofs of soundness theorems and verification algorithms.

In Proof-carrying code (PCC) [66], Necula proposes a new architecture of typed intermediate language in which a meta-logical framework is used to specify safety properties as well as to represent the safety proof in the code. The motivation is that, with this architecture, (1) arbitrary safety properties can be expressed in a systematic manner, (2) different theorem proving techniques can be employed depending on applications, and (3) a single, simple verification algorithm can be used for all systems.

This section discusses PCC’s architecture of policies, provers and verifiers, and illustrates how control-flow safety can be expressed in a logical framework.

4.1 Policies, provers and verifiers

Throughout the paper we have informally described various policies and properties such as control-flow safety and memory safety. Schneider [85] formally defines them as follows:

- security policy: a predicate on all executions (e.g., information flow).
- security property: a predicate on one execution.
- safety property: a predicate for the absence of an event on a finite prefix of one execution (e.g., control-flow, memory access).
- liveness property: a predicate for the *presence* of an event on a finite prefix of one execution (e.g., termination, resource release)

Schneider also proves that only safety properties can be enforced by execution monitoring and security policies in general must be verified by static analysis [85]. Alpern and Schneider [2] on the other hand formally prove that the combination of safety property and liveness property is equivalent to security property.

Necula [66] observes that arbitrary safety properties can be formalized in the first-order logic as preconditions and postcondition on functions. Automatic theorem provers such as Floyd-style [30] verification condition generators [66], symbolic evaluations [69, 64], or type systems [62] can then express the static guarantee for the safety properties of the program as a proof in the first-order logic. Finally, safety verification amounts to proof checking in the same logic.

This setup places no restrictions on memory management or array bounds-checking unlike Java bytecode or TAL. On the other hand, there does not exist a complete algorithm for constructing proofs of arbitrary properties in PCC.

4.2 Logical framework

We will now give a concrete example of PCC architecture in which the safety properties are expressed as typing rules, proofs expressed as typing derivations, and verification expressed as type-checking.

The Edinburgh Logical Framework (LF) [39] is a meta-language for high-level specifications of programming languages and logics. LF can express target languages in the first-order predicate logic. Also, through the higher-order abstract syntax [76], a target language can use LF’s alpha-equivalence and beta-reduction instead of implementing its own parser and substitution function.

Let us consider again the control-flow safety of TAL-0. Figure 8 lists its specification in Twelf (a modern LF implementation [77]), which corresponds closely to its formal description in Section 3.1. A type in the target language is represented by a type constant of kind type in Twelf: registers (r), integers (n), labels (l), operands (v), instructions (i), blocks (I), heaps (H0), and register files (R0). We then define instances of operands (vr, vn, v1), instructions (mov, add, bz), blocks (jmp, seq), heaps (H0nil, H0cons), and register files (R0nil, R0cons, R0cons1). In Twelf we need to use explicit tagging names (vr r for register operands instead of simply r) and unique tagging names (H0nil and R0nil instead of simply ϵ). Similarly, we define types (t), heap types (H), and register file types (R) and their instances (int, tr, all, Hnil, Hcons, Rnil, Rcons). There is no type variable α because we use Twelf’s variable bindings and substitutions for the polymorphic types, which are now written as all: (t → t) → t.

A typing rule of the target language is represented by a type constructor of higher kind in Twelf: operand typing (vt), instruction typing (it), block typing (It), heap typing (Ht), register file typing (Rt), and machine state typing (St). Each typing rule is specified in the style of logic programming and corresponds closely to the rule in Figure 5, except Ht which
Figure 8: Control-flow safety in Twelf
now takes an additional argument of the original heap so that instruction blocks can be checked mutual-recursively. Also, we need to use explicit list lookup (H₀find and R₀find instead of simply H(ℓ) or R(r)) and to use capital letters for logical variables (such as H₁ and R₁ in rule tint).

We can now type-check H₁ | bz r₁, done : R₁ → R₁ in Twelf, as we have done manually in Section 3.1, by this additional code:

```twelf
prod : l. loop : l. done : l.
r1 : r. r2 : r. r3 : r. r4 : r.
R₁ = Rcons (Rcons (Rcons (Rcons Rnil r1 int) r2 int) r3 int) r4 (all [T] tr
       (Rcons (Rcons (Rcons Rnil r1 int) r2 int) r3 int) r4 T)).
H₁ = Hcons (Hcons (Hcons Hnil prod (tr R₁)) loop (tr R₁)) done (tr R₁).
%query 1 1 it H₁ (bz r₁ (vl done)) R₁ R₁.
```

Here we first instantiate prod, loop, done as labels and r₁, r₂, r₃, r₄ as registers. We need to cons up the list of heap and register file types (H₁ and R₁) in such a verbose way as there is no polymorphism in Twelf to define syntactic sugar (ε and comma) for the list operator [5]. The command %query 1 1 it H₁ (bz r₁ (vl done)) R₁ R₁ asks Twelf to verify that there is exactly one derivation for the instruction typing. The first argument of the command means at least one and the second argument means at most one derivation.

The complete program of product function in Figure 3 can be encoded in a similar fashion¹. However, type checking for the whole program will not terminate in Twelf because the typing rules are not syntax-directed [60]. In particular, we need an explicit type instantiation for operands v[t] in T-Inst; but we will skip the technical development here.

The following theorem [39] justifies the use of LF type checking of terms as a validity checking of proofs in the target language.

**Theorem 2 (LF representation adequacy)** There are bijections between terms, types and typings in the target language and those in LF. Hence a typing derivation in LF implies a valid proof in the target language.

### 4.3 Discussion

Necula [66] lists safe packet filters [67] as an compelling application of PCC. Previous approaches for such a kernel extension either use interpreters with restricted policies and expensive context switches, or insert run-time checks with no static guarantee. Moreover, PCC’s speed is impressive [67], outperforming BSD Packet Filter architecture [52] by 10 times, safe packets in SPIN operating system [13] by 2 times, and software-based fault isolation [98] by 30%. Other applications of PCC include certifying compilers for Java [20, 19] and for a safe subset of C [69]. The most serious drawback of PCC is the proof size [65], but recent work has led to efficient representation and validation of proofs [68, 71].

¹http://www.cis.upenn.edu/~stse/til/main.elf
Twelf can also express higher-order predicate logic and provides checks for output argument mode, induction termination, and case totality [82, 77]. With these facilities, meta-theorems of the target languages can be encoded in Twelf, including determinacy, uniqueness, termination, progress and preservation. Appel et al. [5] propose such a foundational approach of including meta-theorems as well as safety proofs along with the binary. Foundational proof-carrying code [4, 38] and foundational typed assembly languages [22] reduce the trusted computing base to a simple LF verifier with direct mappings of semantics to concrete machine architectures.

5 Comparison

In previous sections, we have discussed and compared the technical details of Java bytecode, typed assembly language, and proof-carrying code. Here we summarize the discussion by giving a high-level comparison of the three languages in term of the tradeoffs of complexity and expressiveness as well as other design criteria. We also briefly address the impact and the future work of these languages.

Complexity and performance  Java bytecode verification and TAL’s type checking are both decidable. The decidability of PCC’s type checking depends on the decidability of the type system being encoded. But we are also concerned with the time and the space complexities of the checking algorithms because they add overheads to the running time of the program. In some applications, the code size dominates other design criteria, especially when we need to distribute the code over network or download the code in a smartcard with severe memory constraint. Another important factor is the language complexity: if a language is loosely defined and is packed with nonorthogonal features, we must be more cautious in implementing and proving soundness of the language.

For Java bytecode, its static semantics is informally defined in The Java Virtual Machine Specification and only later reformulated in more formal ways by researchers [50]. Proving the safety of bytecode amounts to solving dataflow equations in the verification algorithm. Solving such equations may take exponential time. Therefore, even though the bytecode is very compact for distribution, bytecode verification can be expensive. Since Java bytecode requires only the basic type annotations for method parameters and register files, it is relatively easy for a Java compiler to generate Java bytecode. The complexity, however, is shifted to the bytecode verifier in the JVM, which has to rediscover the structures and the invariants of subroutines. Java bytecode relies heavily on just-in-time compilation for good execution performance, but the overhead of compilation as well as the inflexibility of bytecode make it difficult to achieve the fast speed of generated code in an assembly language.

TAL, in contrast, is formally defined with a set of typing and evaluation rules. These rules are syntax-directed and hence we can easily check the safety of TAL code in linear time. The checking algorithm is as simple as recursively matching the code with the corresponding rules. Moreover, TAL’s type system employs well-studied constructs such as universal types and existential types, leading to a high confidence in the soundness of the language. TAL’s
instructions are very close to the machine instructions and thus, other than overhead of the garbage collector, TAL can enjoy the raw performance of the machine. For instance, TAL allows multiple calling conventions to be specified so that programmers can tailor the code for speed in a particular machine architecture. But all basic blocks in TAL requires type annotations for the register files. This requirement may lead to large code for programs with lots of branches and loops. Extended type checking algorithms for TAL allow some type annotations to be omitted at the cost of longer verification time to reconstruct them before verification [36].

PCC’s meta-language is formally defined and proved to be sound. Verifying PCC code may involve higher-order unification for reconstructing terms and types. We can require PCC to be fully annotated so that the checking will be syntax-directed and can be done in linear time. PCC achieves as good execution performance as TAL. Type systems encoded in PCC can easily be tailored to express low-level policies such as memory layout and array-bounds information in order to utilize specific optimizations of a particular machine. Nevertheless, as a result of making the language, the code, and the proof all expressible in one framework, PCC faces many engineering challenges of representing and validating the proofs with a low space overhead.

**Expressiveness and extensibility**  On the other side of the tradeoff, we want an intermediate language to be expressive in a way that it has enough primitives to efficiently encode different high-level languages. At the same time, the type system of the language must enforce the abstraction in the source language such that type safety in the intermediate language implies type safety in the source.

Java bytecode has many modern language features such as objects, threads and memory management to facilitate high-level programming in an object-oriented style. Despite the claim of language independence of JVM, the bytecode language has limited support for other programming paradigms. For example, Java bytecode does not have primitives for efficient encodings of tail-recursive calls, polymorphism, or closures for functional programming.

TAL strikes a balance between expressiveness and complexity by allowing low-level primitives such as registers and jumps while still keeping its type system simple. We can readily translate high-level language like System F into TAL and we can perform register allocation and instruction scheduling with the fully annotated instruction blocks. However, TAL puts the burden on programmers to declare types, making TAL more suitable as a common intermediate language of compilers rather than a programmer-friendly source language. Also, it is worth investigating to see if adding objects or exceptions to TAL will significantly complicate its type system [61].

PCC is a highly extensible framework for specifying other intermediate languages. Each language in PCC can specify its own type system, unlike the fixed type systems of Java bytecode and TAL. Achieving the maximal expressiveness, PCC allows arbitrary policies to be specified and requires only a single verifier. It delegates all the work of proving safety to an external theorem prover.
Safety and minimal trust All of the three intermediate languages enforce the basic safety properties including type, control-flow, and memory safety. They all require a garbage collector at runtime for safe memory management.

Java bytecode specifies these policies in the transition function of abstract interpretation and in the successor relation of model checking. A real JVM also checks for stack safety to prevent stack underflow or overflow and for initialization safety to ensure that registers and objects must be initialized before use. However, a just-in-time compiler inside the JVM may bypass these checks or generate unsafe native code for performance. This means that the trust of the computing base for Java bytecode comprises the safety policies, the bytecode verifier as well as the just-in-time compiler. Note that, compared to the simple verifiers and the runtime systems of TAL and PCC, the verifier and the just-in-time compiler are much more complex systems.

TAL specifies the safety policies in the typing rules, which may also be extended to check for initialization safety. PCC allows encodings of type systems that express the same set of safety policies, as well as low-level policies such as memory-layout safety and array-bounds safety. The trust of the computing base for TAL and PCC includes only the policies and the verifiers. The verifiers for both TAL and PCC can be made very simple at the cost of verbose type annotations. As discussed earlier, tradeoffs can be made between space for annotation and time for verification by doing type reconstructions in TAL and PCC. But the type reconstruction algorithms also increase the trust of the computing base. Depending on applications, users can pick their spot in the tradeoff spectrum of performance versus trust in TAL and PCC.

Impact and future works Java brings the modern language technologies into mainstream, fundamentally changing the programming practice in industry. TAL and PCC put together results from logics, type theory, formal methods, and compiler techniques, to lay the foundation for further research of intermediate languages. They also energize the field of language design and implementation for more secure programs. In particular, there are lots of refinements and applications of the basic PCC and TAL including Foundational proof-carrying code [4], Enforcing high-level protocols in low-level software [27, 29], Cyclone: a safe dialect of C [43], CCured: type-safe retrofitting of legacy code [70], and many ongoing research projects [40, 88, 3, 23, 45].

There may not be an immediate commercial demand for TAL or PCC, but the two intermediate languages can readily be used as a formal interface for compiler backends or virtual machines. For example, most JVMs need to employ just-in-time compilation for performance, but such compilation is not type-preserving or verified. A commercial JVM can be compiled to TAL to guarantee type safety while enjoying TAL’s high performance. In the extreme, PCC can also be used, minimizing the trust of the computing base from an optimizing JVM, to TAL’s type checker, to an unified verifier of PCC.
6 Conclusion

In this paper, we have motivated the use of typed intermediate languages, explained the type systems of three influential languages, and compared their tradeoffs of expressiveness versus complexity. We emphasize performance, safety, extensibility, expressiveness, static guarantee, and minimal trust as the important design criteria of typed intermediate languages. The main contribution of this paper is presenting the intuitions behind these type systems with examples and summarizing the technical details of the original papers. Additionally, we have assessed the impact of the three languages and identified research directions for future work.

References


